РАЗБОР АРГУМЕНТОВ КОМАНДНОЙ СТРОКИ НА PYTHON

Иванов А.В., 7 июня 2010 г.
aivanov(злая собака)keldysh(жирная точка)ru
Институт прикладной математики им. М.В. Келдыша РАН

Можно сделать защиту от дурака, но только от не изобретательного.

Закон Нейсдра

Содержание

1 Введение

1.1 Ода командной строке (можно не читать)

Если Вы программируете под *nix[1], рано или поздно Вы столкнетесь с необходимостью разбора аргументов командной строки в своих приложениях. Командная строка (точнее ее унылое подобие) есть даже в MS Windows.

[1] Таким образом мы будем обозначать все семейство Unix и Linux систем.

У командной строки есть масса преимуществ перед форточками, хотя для человека непривычного она выглядит чудовищно сложно. Конечно форточки легче осваивать — вроде все на виду, только вози мышью по экрану. Но когда Вы осваиваетесь с приложением, форточки начинают мешать быстро работать (если это не какой нибудь графический редактор конечно). Каждый раз, когда Вы тянете руку к мыши, Вы теряете время (именно поэтому я лично так люблю текстовый редактор Emacs, там мышь вообще не нужна). Конечно и в форточках можно задать кучу комбинаций горячих клавиш, что позволит отчасти обходиться без мыши — собственно граждане, вынужденные профессионально работать с оконными интерфейсами, так и делают.

Но главное — командная строка позволяет Вам использовать всю мощь концепции Unix Way. Например представьте, что Вам необходимо единообразно обработать несколько сот файлов при помощи оконного интерфейса. Если интерфейс не предоставляет средств для пакетной обработки, каждый файл придётся обрабатывать отдельно (несколько кликов мышью для каждого файла в лучшем случае). Средства пакетной обработки у оконных интерфейсов если и есть, то весьма специфические и способны решать лишь очень ограниченный круг задач. Некоторые приложения предлагают пользователю писать макросы на каком то своём внутреннем языке, типа Visual Baisic, но куда проще выучить shell.

Приложение, имеющее интерфейс командной строки, гарантированно можно будет использовать в комплексе со всеми остальными утилитами *nix. Фактически *nix — это огромный ящик разнообразных деталек наподобие конструктора Лего. Если вдруг оказывается, что какой то детальки там нет (чаще Вы просто не можете ее найти), Вы можете добавить деталь от себя. При этом командная строка заменяет стандартные пупырышки в конструкторе Лего — какой бы формы деталь не была, ее удастся сцепить с остальными. Концепция Unix Way в этом и состоит — хотите построить дом, стройте его по возможности из стандартных кирпичей, добавляя от себя авторские витражные стекла и балконы с кованной решёткой. Вы можете сосредоточиться на общей архитектуре здания, на проработке завитушек балконных решёток или скажем убранстве спален — для всего остального есть готовые решения таких же сумасшедших энтузиастов, делавших соответствующие детали из соображений максимального удобства и функциональности.

В отличии от Unix Way, традиционная «постройка дома» под MS Windows подразумевает возведение монументального сооружения из монолитного железобетона с нуля. Вы конечно можете взять стандартный проект, но джакузи туда не влезет никак, а крыша из глиняной черепицы будет смотреться странно. Авторский проект соседа с джакузи Вам поможет слабо — придётся сносить взрывчаткой несущие стены, бурить наискось потолки, и получившееся в итоге сооружение (пусть и украшенное фантастической мозаикой) развалиться от первого же залётного дятла.

1.2 Общие замечания

Для тех кто, никогда командной строки не видел и задается вопросом «О чем вообще разговор?» — командная строка это такое приложение (т.н. терминал или shell), в котором с клавиатуры вводятся команды, т.е. Вы пишете машине инструкции, что нужно делать. Каждая команда состоит обычно из имени запускаемого приложения и аргументов командной строки (тех слов, которые вводятся после имени приложения). Скажем
$ cp file1 file2
здесь cp — команда, file1 и file2 её аргументы, а вся строка предписывает компьютеру скопировать file1 в file2, подробнее читайте в [1, 2].

Для облегчения ввода опций и файлов с длинными описательными именами в *nix системах применяется библиотека readline, поддерживающая контекстное дополнение буфера ввода. При вводе имени файла или команды достаточно ввести первые несколько букв и нажать клавишу Tab — библиотека автоматически завершит имя команды или файла (директории) если это возможно однозначно сделать, или покажет список возможных вариантов завершения. Кроме того поддерживается история команд (клавиши Up и Down) и поиск в истории команд (комбинация Ctrl+r). Благодаря библиотеке readline можно давать командам и файлам (каталогам) длинные описательные имена, и при этом ничего не терять в скорости работы, поскольку полностью эти имена набирать не обязательно, readline сделает это за Вас. Зачастую для поиска нужной команды достаточно ввести первые буквы «по наитию» — например можно предположить, что команда для просмотра фотографий в raw–формате начинается со слов show или view. Вводя show нажимаем Tab и выбираем из списка возможных завершений команду showfoto.

Важным свойством приложения с интерфейсом командной строки является справка (обычно вызывается опциями -h или --help, некоторые приложения выводят справку при вызове без аргументов или при некор-

ректном задании аргументов). Поскольку никаких кнопочек не видно[2] необходим лёгкий доступ к справке. Недокументированные приложения с интерфейсом командной строки — это глумление над пользователем почище, чем MS Windows. Хорошо, если приложение простое, и единственным его пользователем является тот, кто приложение написал — он может посмотреть исходный код и что то там понять. Во всех остальных случаях справка **обязательно** должна быть.

При большом количестве опций и параметров приложения, хорошим тоном считается задание некоторой наиболее часто употребляемой конфигурации по умолчанию, в качестве базовой. Все остальные конфигурации получаются на ее основе при помощи «точечного» изменения отдельных параметров.

У каждого запущенного приложения есть стандартные потоки ввода, вывода и ошибок (stdin, stdout и stderr). Потоки могут перенаправляться из приложения (файлов) в приложения (файлы) через т.н. pipe (трубопроводы), что является ещё одним стандартным и очень мощным средством взаимодействия приложений. Многие сложные приложения (например gnuplot, grub, интерпретатор языка Python) имеют свой собственный язык и способны как принимать инструкции через аргументы командной строки, так и читать их со стандартного ввода, в том числе и в интерактивном режиме. Чтение аргументов со стандартного ввода может быть особенно удобно, когда аргументов у приложения много и они не влезают в одну строку.

В этой статье мы будем рассматривать разбор аргументов командной строки в приложениях на языке Python [3]. Это один из лучших языков на сегодняшний день, скорость разработки на Python на порядок и более выше, чем скорость разработки на C++, а код куда лаконичнее и читабельнее. К недостаткам можно отнести сложность отладки больших проектов из за отсутствия контроля типов на этапе компиляции в байт-код и низкую (по сравнению с C++ на три–четыре порядка) производительность. Однако, огромные приложения противоречат Unix Way, а от приложений реализующих интерфейс высокой производительности и не требуется.

2 Ручной разбор аргументов

2.1 Простейший пример — разбор позиционных аргументов

Давайте напишем простую программу для решения квадратного уравнения вида

$$ax^2 + bx + c = 0.$$

Как известно, $x_{1,2} = (-b \pm \sqrt{b^2 - 4ac})/2a$ при $b^2 - 4ac \geq 0$. Наше приложение sq-eq1.py будет принимать коэффициенты a, b, c в виде трёх аргументов командной строки, выводить значения корней и выводить справку при некорректном задании аргументов.

file examples/sq-eq1.py:

```
     1 #!/usr/bin/python
     2 'usage: sq-eq1.py a b c --> x1 x2'
     3 import sys
     4 try: a, b, c = map( float, sys.argv[1:] )
     5 except : print __doc__; sys.exit()
     6 D = b*b-4*a*c
     7 if D<0 : print 'nan nan'
     8 else : print (-b-D**.5)/(2.*a), (-b+D**.5)/(2.*a)
```

В первой строке мы указываем, что данный файл должен выполняться в интерпретаторе python расположенном в /usr/bin/python[3].

Вторая строка это т.н. строка документации [3], замечательная особенность Python.

В третьей строке мы импортируем модуль sys, содержащий кроме всего прочего список аргументов командной строки sys.argv. Первый элемент списка (с нулевым индексом) это имя запущенного файла, остальные — собственно аргументы командной строки.

[2] Что на мой взгляд скорее достоинство, поскольку опций может быть чудовищно много, и, если на каждую повесить кнопочку или пункт меню, то места на экране уже не останется. Скажем в LaTeX более трёх тысяч опций и команд, я помню меньше сотни, а остальные мне просто не нужны (при необходимости я лезу в справочник). У MS Word по сравнению с LaTeX возможности куда скромнее, но поскольку на каждую опцию есть кнопка/пункт меню, продраться к чему то, кроме изменения стиля шрифта, является архисложной задачей.

[3] В *nix системах программа для выполнения файла определяется не по расширению а по первой строчке файла

В четвёртой строке мы пытаемся разобрать аргументы — привести их к типу float и присвоить переменным a, b, c. Если по каким то причинам это не удается сделать, возбуждается исключительная ситуация, которая перехватывается в пятой строке. при этом выводиться строка документации и приложение завершает работу.

Дальше мы рассчитываем дискриминант, проверяем его знак и выводим корни. Вот результаты работы:

```
1 examples> ./sq-eq1.py
2 usage: sq-eq1.py a b c --> x1 x2
3 examples> ./sq-eq1.py 1 2 -3
4 -3.0 1.0
5 examples> ./sq-eq1.py 1 2 3
6 nan nan
```

2.2 Разбираем именованные аргументы при помощи функции exec

В предыдущем примере мы использовали позиционное задание аргументов, т.е. a должно было быть первым, b вторым, c третьим. Такой подход не всегда удобен — представьте что аргументов не три а десять! При большом количестве аргументов удобно при задании использовать их имена, не фиксируя порядок задания аргументов.

file examples/sq-eq2.py:

```
1 #!/usr/bin/python
2 'usage: sq-eq2.py [a=val] [b=val] [c=val] [-h] [--help] --> x1 x2'
3 import sys
4 a, b, c = 0., 0., 0.
5 if len(sys.argv)==1 or '-h' in sys.argv or '--help' in sys.argv : print __doc__;
sys.exit()
6 for arg in sys.argv[1:] : exec arg
7 D = b*b-4*a*c
8 print 'Solved %s*x**2 + %s*x + %s = 0, D=%s'%( a, b, c, D )
9 if D<0 : print 'nan nan'
10 else : print (-b-D**.5)/(2.*a), (-b+D**.5)/(2.*a)
```

В четвёртой строке мы задаем значения аргументов по умолчанию. В пятой проверяем, не была ли запрошена справка — справка выводиться при вызове приложения без аргументов или опциями -h или –help.

Разбор аргументов производиться в шестой строке, где для каждого аргумента вызывается функция exec. Такой подход отличается простотой и фантастической гибкостью — фактически мы используем всю мощь интерпретатора Python.

```
1 examples> ./sq-eq2.py --help
2 usage: sq-eq2.py [a=val] [b=val] [c=val] [-h] [--help] --> x1 x2
3 examples> ./sq-eq2.py a=1
4 Solved 1*x**2 + 0.0*x + 0.0 = 0, D=0.0
5 -0.0 0.0
6 examples> ./sq-eq2.py c=1 a=2*c b=c-a
7 Solved 2*x**2 + -1*x + 1 = 0, D=-7
8 nan nan
9 examples> ./sq-eq2.py a=1 b=a*2 c=-b-a**2
10 Solved 1*x**2 + 2*x + -3 = 0, D=16
11 -3.0 1.0
```

Но такой подход имеет два существенных недостатка.

Во первых, Python очень гибкий, открытый и демократичный язык — пользователь имеет возможность залезть куда угодно и сделать что угодно. Оборотной стороной такой открытости является то, что пользователь по ошибке или по злому умыслу может влезть куда то не туда и сделать что то не то. Например, задав в качестве аргументов приложения sq-eq2.py команды 'import os' 'os.system("rm -rf ~/")', пользователь способен уничтожить свой домашний каталог (зачем ему это понадобиться — другой вопрос, но лучше не доводить дело до его выяснения).

Во вторых, обработка каждого аргумента функцией exec подразумевает, что аргументы задаются с синтаксисом Python, который далеко не всегда удобен в shell. В частности, скобки и кавычки требуют экранирования, поскольку shell на них реагирует по своему. Например, что бы задать строковый параметр (допустим имя файла), необходимо записать filename=\'path-to-file\' — не самый удобный вариант, особенно если Вы используете readline для автоматического дополнения имени файла. Про задание набора файлов при помощи масок (символы ? * и т.д.) рядовому пользователю в этом случае можно вообще забыть.

Тем не менее, в ряде случаев использование функции exec может быть весьма полезно, особенно в комбинации с разбором позиционных аргументов. Единственным ограничением здесь является лишь Ваша фантазия.

3 Модуль getopt — каноническое решение

Поскольку командная строка существует даже дольше чем Unix, канонический синтаксис задания аргументов команд определён уже давно. Не обязательно строго его придерживаться, многие команды имеют несколько иной синтаксис.

В каноническом синтаксисе опции могут быть короткими (однобуквенными), начинающиеся при задании с одного знака минус или длинными (многобуквенными), начинающимися с двух знаков минус. Некоторые опции могут принимать один аргумент. Все аргументы командной строки делятся на список опций и концевой срез последовательности аргументов. Короткие могут указываться слитно, например -abc эквивалентно -a -b -c. Аргумент коротких опций может указываться сразу после опции без пробела, -O value эквивалентно -Ovalue. Длинные опции с аргументом могут использоваться в виде --option=value, что эквивалентно --option value.

Ветераном разбора аргументов командной строки безусловно является модуль getopt, реализованный почти для всех языков программирования. В Python модуль getopt предоставляет функцию
getopt(args, short_options [, long_options])
где args — список разбираемых аргументов, short_options — строка описывающая короткие опции (после опций принимающий аргумент должно стоять двоеточие), long_options — список длинных опций (после опций принимающий аргумент должен стоять знак равенства). Функция возвращает кортеж из списка разобранных опций и концевого среза списка аргументов. Список разобранных опций состоит из пар вида (опция,аргумент опции или пустая строка). Подробное описание см. в [3].

Пример использования модуля getopt приведён ниже. Кроме ввода значений переменных a, b, c и вывода справки мы добавили возможность вывода только первого (опции -f, --first) или только второго (опции -s, --second) корней уравнения.

file examples/sq-eq3.py:

```
     1 #!/usr/bin/python
     2 'usage: sq-eq3.py [-a val] [-b val] [-c val] [-h] [--help] [-f|--first|-s|--second] -->
x1 x2'
     3 import sys, getopt
     4 opts, args = getopt.getopt( sys.argv[1:], 'a:b:c:hsf', 'help first second'.split() )
     5 opts = dict(opts)
     6 if len(sys.argv)==1 or '-h' in opts or '--help' in opts : print __doc__; sys.exit()
     7 a, b, c = map( float, map( opts.get, '-a -b -c'.split(), (0, 0, 0) ) )
     8 D = b*b-4*a*c
     9 print 'Solved %s*x**2 + %s*x + %s = 0, D=%s'%( a, b, c, D )
    10 if '-f' in opts or '--first' in opts : print (-b-D**.5)/(2.*a) if D>=0 else 'nan'
    11 elif '-s' in opts or '--second' in opts : print (-b+D**.5)/(2.*a) if D>=0 else 'nan'
    12 else : print '%s %s'%( (-b-D**.5)/(2.*a), (-b+D**.5)/(2.*a) ) if D>=0 else 'nan nan'
```

Разбор опций происходит в четвёртой строке. В пятой строке список пар (опция,параметр) преобразуется в словарь для упрощения дальнейшей работы. В седьмой строке средствами функционального программирования из словаря извлекаются и преобразуются к типу float значения для переменных a, b, c. В строках 10–12 выводятся значения корней в зависимости от наличия опций -f, --first, -s, --second. Результаты работы:

```
     1 examples> ./sq-eq3.py -a 1
     2 Solved 1.0*x**2 + 0.0*x + 0.0 = 0, D=0.0
     3 -0.0 0.0
```

```
 4 examples> ./sq-eq3.py -a 1 -a2
 5 Solved 2.0*x**2 + 0.0*x + 0.0 = 0, D=0.0
 6 -0.0 0.0
 7 examples> ./sq-eq3.py -a 1
 8 Solved 1.0*x**2 + 0.0*x + 0.0 = 0, D=0.0
 9 -0.0 0.0
10 examples> ./sq-eq3.py -a 1 -s
11 Solved 1.0*x**2 + 0.0*x + 0.0 = 0, D=0.0
12 0.0
13 examples> ./sq-eq3.py -a 1 --first
14 Solved 1.0*x**2 + 0.0*x + 0.0 = 0, D=0.0
15 -0.0
```

Обратите внимание, что при нескольких вызовах одной опции -a с разными значениями параметров актуальным является последний — такое поведение обеспечивается конвертацией результата работы функции getopt.getopt в словарь.

Канонический синтаксис оптимизирован с точки зрения стабильности и минимума вводимых символов, но для решения многих задач он оказывается слишком примитивным.

4 Модуль IGL — идеальное решение

4.1 Общая концепция

Описанные выше способы разбора аргументов командной строки подходят лишь для сравнительно несложных приложений. К недостаткам следует отнести

- необходимость ручного приведения типа параметров опций (все аргументы командной строки естественно имеют строковый тип);
- отсутствие автоматической генерации справки по опциям, что усложняет модернизацию приложений (зачастую появление новых опций или изменение поведения старых не сразу отражается в документации);
- сложность реализации задания параметров в древовидных структурах данных;
- отсутствие автоматических сокращений имён опций;
- отсутствие автоматического режима чтения опций со стандартного ввода в интерактивном режиме.

Кроме того, сама реализация разбора опций, как вручную так и через модуль getopt, является довольно низкоуровневой, требует значительных усилий и сопровождается большим количеством ошибок. Между тем, Python позволяет изящно решить все вышеперечисленные проблемы.

Интерфейс командной строки можно рассматривать как возможность менять из shell значения некоторых переменных и запускать из shell некоторые методы с аргументами или без. Фактически речь идёт о создании специфичного для конкретной задачи микроязыка с синтаксисом удобным для использования в shell.

Идеальным решением в Python является группировка всех параметров и методов, к которым должен быть обеспечен доступ из shell, в одном пользовательском классе. Тогда:

- При изменении значений параметров строка с новым значением может быть автоматически приведена к типу старого значения (случай с изменением типа, когда например старое значение было числом а новое должно быть строкой представляется весьма экзотическим).
- На основе строки документации класса, описания сигнатур методов[4] и строк документации методов можно автоматически генерировать справку. Отслеживать актуальность документации становиться намного проще — строки документации расположены в коде в теле методов, а давая методам и их аргументам описательные имена можно в ряде случаев вообще обойтись без строк документации.

[4] Python позволяет получить полную информацию о сигнатуре метода, т.е. имя метода, имена аргументов, значения аргументов по умолчанию и т.д.

- Для доступа к древовидным структурам данных достаточно в качестве поля класса задать аналогичный класс обеспечивающий доступ к поддереву.
- Сокращение имён опций и чтение аргументов со стандартного ввода или из файла может быть реализовано автоматически.

Все эти чудесные возможности реализуются в одном модуле IGL (Interface Generation Library — библиотека генерации интерфейса). Модуль предоставляет пользователю класс BASE, содержащий конструктор (позволяет сразу устанавливать произвольное количество полей класса через список именованных аргументов), методы __call__ (перегруженные скобки '()', занимается разбором аргументов командной строки) и __help__ (автоматически генерирует справку). При создании своего класса пользователю достаточно отнаследовать BASE и добавить в него свои методы и поля, доступ к которым из shell и генерацию справки обеспечивают методы __call__ и __help__.

4.2 Простейший пример использования

Использование модуля IGL для разбора аргументов при решении квадратного уравнения сильно смахивает на стрельбу из пушки по воробьям, тем не менее мы приведем один из возможных вариантов:

file examples/sq-eq4.py:
```
 1 #!/usr/bin/python
 2 import sys, IGL
 3 class sqeq(IGL.BASE) :
 4     def solve( self, a=0., b=0., c=0., first=True, second=True ) :
 5         '> x1 x2'
 6         D = b*b-4*a*c
 7         print 'Solved %s*x**2 + %s*x + %s = 0, D=%s'%( a, b, c, D )
 8         x1, x2 = ( (-b-D**.5)/(2.*a), (-b+D**.5)/(2.*a) ) if D>=0 else ( 'nan', 'nan' )
 9         print x1 if first else '', x2 if second else ''
10 sqeq()( ['solve']+sys.argv[1:] )
```

В десятой строке при запуске разбора аргументов специально добавляется аргумент 'solve' инициализирующий одноимённую опцию. Пример работы:

```
 1 examples> ./sq-eq4.py ?
 2     solve( a=0.0, b=0.0, c=0.0, first=True, second=True ) --- > x1 x2
 3 examples> ./sq-eq4.py 1 2 3
 4 Solved 1.0*x**2 + 2.0*x + 3.0 = 0, D=-8.0
 5 nan nan
 6 examples> ./sq-eq4.py 1 2
 7 Solved 1.0*x**2 + 2.0*x + 0.0 = 0, D=4.0
 8 -2.0 0.0
 9 examples> ./sq-eq4.py 1 2 second=off
10 Solved 1.0*x**2 + 2.0*x + 0.0 = 0, D=4.0
11 -2.0
12 examples> ./sq-eq4.py 2 c=3 --s -4 5 6 first=off
13 Solved 2.0*x**2 + 0.0*x + 3.0 = 0, D=-24.0
14 nan nan
15 Solved -4.0*x**2 + 5.0*x + 6.0 = 0, D=121.0
16  -0.75
```

Обратите внимание — в рамках одного запуска приложения можно решить несколько уравнений.

4.3 Описание синтаксиса

4.3.1 Используемая терминология

Под **аргументами** мы будем понимать аргументы командной строки, подлежащие разбору. Т.е. аргумент — это некоторое строковое значение, переданное в приложение из shell или прочитанное из файла (со стандартного

ввода либо откуда то ещё). Важно, что аргументы — это то, что на входе приложения.

Под **опциями** мы будем понимать методы/поля пользовательского класса–наследника класса BASE, доступные из shell для вызова/изменения.

Методы пользовательского класса могут иметь аргументы. Как и в Python, аргументы метода могут задаваться как в **позиционном** виде (т.е. вводимое значение ассоциируется с соответствующим аргументом метода на основе его позиции в общем списке), так и в **именованном** виде (т.е. явно указывается имя-аргумента=вводимое-значение). Изменение поля пользовательского класса можно рассматривать как вызов одноимённой опции с единственным аргументом — новым значением поля[5]. Во избежании дальнейшей путаницы (поскольку аргументами называются как вводимые из командной строки значения, так и аргументы опций), в дальнейшем аргументы опций мы будем называть параметрами опций, или просто **параметрами**.

Под **терминированием** опции мы будем понимать запуск опции (соответствующего метода пользовательского класса) с накопленными ранее аргументами.

4.3.2 Базовые принципы

Синтаксис IGL близок к каноническому, но есть существенные отличия.

Имена опций Однобуквенные (в каноническом смысле) опции не рассматриваются вообще, т.е. число символов в имени опции не имеет значения, опции с именами из одной буквы и из нескольких букв с точки зрения IGL равноправны и обрабатываются единообразно.

Доступ к полям и методам пользовательского класса, с именами начинающимися со знака подчёркивания блокируется, т.е. имя опции не может начинаться со значка подчёркивания. В остальном требования к именам опций такие же как и к идентификаторам Python — в имени могут встречаться буквы, цифры и знак подчёркивания, строчные и прописные буквы различаются, первой должна быть буква.

Сокращение имён опций При вводе опция может начинаться с '--', или писаться целиком без '--'. Имя опции начатой с '--' может быть указанно полностью или сокращено по одному из двух вариантов:

1. указываются лишь несколько первых букв имени опции;
2. из имени опции выбрасываются произвольные буквы (кроме первой), порядок оставшихся букв должен сохраняться.

Сокращение должно позволять однозначно разрешить имя опции. Если опция не найдена сразу, производиться попытка разрешить имя опция на основе первого варианта, затем на основе второго, если обе попытки были неудачны возбуждается исключение.

Для разрешения сокращения имен опции не начинающихся с '--', необходимо задать в пользовательском классе список _always_use_abbrv. Для разрешения сокращения опций класса, в список необходимо добавить эллемент '' (пустая строка). Для разрешения сокращений при задании именованных аргументов опции необходимо добавить в список паттерн, для которого проверке имени опции функцией fnamtch.fnmatch вернет истину.

Специальные управляющие последовательности Среди аргументов могут встречаться специальные управляющие последовательности, не приводящие к вызову каких–либо опций, но изменяющие ход разбора аргументов. К ним относятся:

- ';' — явно терминировать текущую опцию;
- '--' — читать аргументы со стандартного ввода в интерактивном режиме;
- read <FILENAME> — читать аргументы из файла <FILENAME>;
- '\' — рассматривать следующую строку как продолжение предыдущей (при чтении аргументов со стандартного ввода или из файла);
- '{' — открыть новый уровень при работе с древовидной структурой опций, см. раздел 4.4.6;

[5]Собственно в Python для этого и используется специальный метод __setattr__

- '}' — закрыть текущий уровень при работе с древовидной структурой опций, см. раздел 4.4.6;
- '?' — вывести справку.

Большая часть из этих управляющих последовательностей может быть изменена программным способом в том числе и в процессе разбора аргументов (см. раздел 4.4.7), что приведет к соответствующим изменениям синтаксиса.

Ещё раз подчеркнем, что специальные управляющие последовательности должны вводиться как **отдельные** аргументы, т.е. должны быть выделены пробелами, символами табуляции или конца строки. Ввод аргумента вида 'myopt?' приведет скорее всего к ошибке, в худшем случае 'myopt?' будет воспринят как параметр предыдущей опции или неявной опции, но справка по опции 'myopt' гарантированно **НЕ** будет выведена.

Поскольку shell предварительно обрабатывает все аргументы командной строки (а все аргументы, которые читаются со стандартного ввода или из файла, пропускаются через shell, см. раздел 4.4.5), необходимо предварять специальные управляющие последовательности '?', ';' и '\' символом \.

4.3.3 Изменение полей класса

При изменении значения заранее заданного поля класса допустимы варианты name value или name=value, где name — имя поля (опция), указанное полностью или начатое с '--' и сокращённое по указанному выше принципу, value — новое значение (параметр опции).

Вызов опции–метода класса с одним аргументом в формате name=value не допускается, т.е. опции–методы и опции–поля не вполне равноправны.

При изменении значения поля класса, новое значение приводиться к типу старого значения, см. раздел 4.4.1.

4.3.4 Вызов опций–методов класса

В Python методы могут вызваться с позиционными или именованными аргументами, могут иметь произвольное количество позиционных и/или именованных аргументов [3]. Модуль IGL поддерживает все эти возможности при вызове опций. Позиционные аргументы метода указываются как есть, через пробел или знак табуляции. Именованные аргументы метода вводятся в виде argname=value где argname — указанное полностью или начатое с '--' и возможно сокращённое имя аргумента метода.

В отличии от Python, при вызове опции позиционные и именованные параметры могут задаваться в произвольным порядке, но ни один параметр не может быть задан дважды. Не допускается задание позиционных параметров, позиция которых согласно сигнатуре была изменена за счёт задания именованных параметров. Т.е. если первый (согласно сигнатуре) параметр опции был задан как именованный, то все остальные параметры (кроме позиционных параметров из списка произвольной длины, передаваемых в метод как *args) тоже должны быть заданны как именованные.

Если параметры опции имели значения по умолчанию (поскольку соответствующие аргументы метода имели значения по умолчанию), производиться приведение типа нового значения к типу старого значения (см. раздел 4.4.1).

Опция терминируется (вызывается соответствующий метод пользовательского класса):

- при явном задании всех параметров (в том числе и параметров, имеющих значение по умолчанию);
- при вызове следующей опции;
- специальным терминирующим символом '\;'.

Таким образом, если опция принимает произвольный список именованных параметров, ее можно терминировать лишь при команды помощи '\;' или закончив строку в стандартном вводе или файле.

4.3.5 Обращение к справке

Справка может быть вызвана при помощи '\?'. Если ни одна из опций не активирована (т.е. не ожидает аргументов) выводиться вся справка по пользовательскому классу. При этом, если заданна неявная опция, в конце выводиться сообщение
"Set '%s' as implicit option"%_implicit_opt.

Если '\?' указан после инициализированной опции, выводиться справка по опции.

4.4 Алгоритм разбора аргументов

Метод __call__(agrs, arg_stack=None) построен по принципу машины Тьюринга. Аргументы из последовательности args разбираются один за другим. Аргументы воспринимаются либо как имя опции, либо как параметр опции, либо как специальная управляющая последовательность. Аргументы вида <KEY>=<VAL> воспринимаются либо как изменение поля класса либо как задание именованного параметра опции.

В большинстве случаев алгоритм разбора адекватно определяет как следует воспринимать аргумент. Приоритетным является восприятия аргумента как параметра текущей опции, если опция была инициализирована. Параметры опции накапливаются по мере обработки аргументов и передаются в соответствующий метод пользовательского класса при терминировании опции. Во избежании ошибок, связанных с тем, что имя опции указанное без '--' воспримается как аргумент предыдущей опции, необходимо либо явно терминировать опции при помощи '\;' либо конца строки, либо начинать имя новой опции с '--'.

Если аргумент имеет вид <KEY>=<VAL> и текущая опция не имеет параметра с именем <KEY>, аргумент трактуется как изменение значения поля класса <KEY> и приводит в частности к терминированию текущей опции.

При разборе аргументов в цикле, пока список аргументов не опустеет, выполняются следующие действия:

- очередной аргумент извлекается из списка; если аргумент является специальной управляющей последовательностью он обрабатывается соответствующим образом и итерация цикла завершается;
- в зависимости от состояния парсера и вида аргумента производятся действия, описанные в таблице 1.

состояние парсера/ вид аргумента	нет активной опции	активная опция ожидает обязательный параметр	активная опция ожидает необязательный параметр	все параметры активной опции заданны, опция не принимает список именованных параметров	все параметры активной опции заданны, опция не принимает список неименованных параметров
--OPT	AO[E.0,E.11]	[E.9]	TO[E.4,E.5], RA	TO[E.4,E.5], RA	TO[E.4,E.5], RA
OPT	AO[E.11] или AIO, RA	+P[E.1,E.2]	+P[E.1,E.2]	+P[E.1,E.2]	TO[E.4,E.5], RA
[--]KEY=VAL	K=V[E.0,E.12] или AIO, RA	+K=V [E.3,E.10]	+K=V [E.3,E.10]	TO[E.4,E.5], RA	+K=V[E.3,E.10]
OPT.SUBOPT...	OPT(...)[E.13] или AIO, RA	+P[E.1,E.2]	+P[E.1,E.2]	+P[E.1,E.2]	TO[E.4,E.5], RA
--OPT.SUBOPT...	OPT(...) [E.0,E.13]	[E.9]	TO[E.4,E.5], RA	TO[E.4,E.5], RA	TO[E.4,E.5], RA
@ARG	AIO, RA	+P[E.1,E.2]	+P[E.1,E.2]	+P[E.1,E.2]	TO[E.4,E.5], RA
иное	AIO, RA	+P[E.1,E.2]	+P[E.1,E.2]	+P[E.1,E.2]	TO[E.4,E.5], RA

Таблица 1: действия, производимые в зависимости от состояния парсера и вида очередного аргумента. Здесь AO — активация новой опции, AIO — активация неявной опции, TO — терминирование активной опции, RA — повторная обработка аргумента, +P — добавление в активную опцию неименованного параметра, K=V — изменение поля пользовательского класса, +K=V — добавление в активную опцию именованного параметра, OPT(...) — запуск разбора аргументов в подопции на одну инструкцию. Кроме того указаны исключения [E.XX], которые могут возбуждаться в том или ином случае, см. таблицу 3.

4.4.1 Преобразование типов параметров и ввод булевых значений

Как уже отмечалось, тип всех вводимых параметров изначально строковый. Тип вводимых параметров опций приводится к типу значений аргументов методов по умолчанию (если они были заданны), тип новых значений полей класса приводится к типу старых значений полей класса.

У такого поведения есть два исключения. Если старое значение None, тип параметра не изменяется (остается строкой). Если старый тип bool, для задания значения True можно вводить 'y', 'Y', 'yes', 'Yes', 'YES', 'on', 'On', 'ON', 'true', 'True' или 'TRUE'; для задания значения False можно вводить 'n', 'N', 'no', 'No', 'NO', 'off', 'Off', 'OFF', 'false', 'False' или 'FALSE'. Во всех остальных случаях будет сгенерировано исключение E.1 (см. таблицу 3).

4.4.2 Неявные опции

Для ввода параметров опции необходимо сначала инициализировать (указать в последовательности аргументов) какую либо опцию. Если параметры указываются без опции, вообще говоря это приводит к ошибке. Такое поведение не всегда удобно — зачастую в shell требуется ввести список файлов в качестве концевого среза последовательности аргументов командной строки (см. описание модуля getopt), или сразу после команды ввести какие то параметры, ожидаемые по умолчанию (например при типовом поведении приложения первым аргументом указывается каталог, если каталог не указан в качестве каталога используется текущая директория). В таких случаях обязательное указание опции оказывается излишним, т.е. опция должна быть заданна неявно.

Если встречается параметр a без опции, перед возбуждением исключения E.15 (см. таблицу 3) проверяется значение необязательного поля _implicit_opt в экземпляре пользовательского класса. Для инициализации неявной опции поле должно содержать имя опции (строку). Имя опции добавляется в последовательность аргументов перед аргументом a и обрабатывается стандартным образом. Инициализируемая опция должна принимать хотя бы один параметр или изменять значение поля _implicit_opt — в противном случае произойдет зацикливание программы.

Неявная опция не может быть инициализирована при помощи аргумента вида <KEY>=<VAL>, такой аргумент будет воспринят как попытка изменения значения поля класса a не как задание именованного параметра неявной опции. Т.е. для инициализации неявной опции первый параметр должен быть обязательно позиционным, остальные параметры могут быть позиционными или именованными, как и для обычной опции.

Значение поля _implicit_opt может изменяться любой опцией:

file examples/igl-smpl1.py:

```
1 #!/usr/bin/python
2 import sys, IGL
3 class sample(IGL.BASE) :
4     def f1( self ) : print 'f1'; self._implicit_opt='f2'
5     def f2( self, x ) : print 'f2', x; self._implicit_opt='p'
6 sample( p=0, _implicit_opt='f1' )( sys.argv[1:] )
```

Пример работы:

```
1 examples> ./igl-smpl1.py 1 2 3 p ?
2 f1
3 f2 1
4 p=3
5 examples> ./igl-smpl1.py p=4 p ?
6 p=4
```

Обратите внимание, что при первом запуске неявной опцией в итоге становится поле p, для которого сначала задается значение 2 (но это никак не отображается), затем задается значение 3. Для того, что бы узнать значение поля, необходимо ввести имя поля (как опцию) и обратиться к справке.

4.4.3 Синонимы для имён опций

Может возникнуть необходимость дать одной опции несколько различных имён. В частности, некоторые имена опций, корректные с точки зрения модуля IGL (т.е. состоящие из букв, цифр и знаков подчёркивания и начинающиеся с буквы), являются ключевыми словами Python и не могут использоваться в качестве имён методов или полей пользовательского класса. Для этих целей необходимо задать поле _aliases пользовательского класса в виде словаря, содержащего записи вида '<REAL-OPT-NAME>' : ['<ALIASE1>', <ALIASE2>,...] где <REAL-OPT-NAME> — настоящее имя опции, <ALIASE1>, <ALIASE2>, ... — альтернативные имена. При этом настоящее имя опции может начинаться с символа подчёркивания, т.е. опция может быть доступна лишь через альтернативные имена.

Информация об альтернативных именах выводиться в справке по каждой опции.

4.4.4 Выбор поведения при вызове приложения без аргументов

Зачастую, вызов приложения без аргументов подразумевает какое то специфическое поведение — например переход в интерактивный режим работы или вывод справки по приложению. Модуль IGL не имеет встроенных средств для задания такого поведения, но пользователь легко может решить эту проблему самостоятельно, изменив соответствующим образом список аргументов, передаваемый в метод __call__ для разбора. Например

```
...( sys.argv[:1] if len(sys.argv)>1 else ['?'] )
```

обеспечит вывод справки при вызове приложения без аргументов, а

```
...( sys.argv[:1] if len(sys.argv)>1 else ['--'] )
```

обеспечит переход в интерактивный режим работы.

4.4.5 Задание аргументов через файлы или стандартный поток ввода в интерактивном режиме

Для запуска чтения аргументов из файла необходимо указать read <FILENAME>, т.е. два аргумента, первый из которых — специальная управляющая последовательность read, а второй — имя файла <FILENAME> с аргументами. Для чтения аргументов со стандартного ввода в интерактивном режиме необходимо указать '--' (два знака минус) в качестве отдельного аргумента (т.е. выражения, выделенного пробелами, знаками табуляции или символами конца строки).

При чтении аргументов из фалов или со стандартного ввода, аргументы читаются построчно. Каждая строка пропускается через shell при помощи команды Python

```
os.popen( 'for i in %s \ndo echo "$i" \ndone'%l ).readlines() )
```

т.е. строка будет разобрана на аргументы при помощи shell, при этом будут стандартным для shell способом обработаны содержащиеся в строке знаки комментариев, маски для задания имён файлов, циклы, условия и т.д. В частности, это приводит к игнорированию пустых строк и строк начинающихся со знака комментария '#'. Концы строк, отделённые знаком комментария, так же отбрасываются.

Чтение аргументов продолжается до окончания файла или закрытия стандартного потока ввода. Чтение аргументов из стандартного потока ввода может запускаться произвольное количество раз.

В процессе чтении аргументов из файла или со стандартного потока ввода может запускаться чтение аргументов из другого файла или со стандартного потока, количество уровней вложения неограниченно. Обрабатываемые файлы хранятся в отдельном стеке. При запуске чтения аргументов из того же файла возможно зацикливание, если в процессе работы файл не изменяется.

Достижение конца строки приводит к терминированию последней инициализированной опции, если строка не была завершена специальной управляющей последовательностью '\\' (два обратных слэша) — в этом случае следующая строка считается продолжением данной. Как при чтении аргументов из файла, так и при чтении аргументов со стандартного ввода, терминированные опции сразу выполняются.

При работе в интерактивном режиме выводится приглашение вида
classname>
где classname — имя класса, в котором запущен метод __call__. При вводе строки, следующей за строкой завешенной последовательностью '\\', выводиться приглашение ... (многоточие).

В интерактивном режиме работы все исключения перехватываются, выводятся в стандартный поток ошибок, и работа продолжается. Для завершения интерактивного сеанса необходимо нажать Ctrl-D (закрытие терминала). Нажатие Ctrl-C (прерывание с клавиатуры) приводит к завершению выполнения метода пользовательского класса, перехватывается в __call__ и интерактивный сеанс продолжается.

4.4.6 Работа с древовидными структурами данных

Задание данных организованных в древовидные структуры из shell является нетрадиционной для shell задачей — обычно приложения с интерфейсом командной строки куда примитивней. Тем не менее такая необходимость эпизодически возникает, кроме того возможность организовывать данные в древовидные структуры делает приложение намного нагляднее.

Для работы с древовидными структурами данных, соответствующие поля класса должны быть заданны как другие пользовательские классы–наследники BASE, либо методы класса должны вызываться без аргументов и возвращать пользовательские классы–наследники BASE.

Если в опции opt есть подоопция subopt (т.е. в поле класса с именем opt, являющемся экземпляром пользовательского класса–наследника BASE, есть поле или метод с именем subopt), то доступ к нему может быть осуществлён либо как opt.subopt либо как opt { subopt ... }. Во втором случае, при открытии фигурной скобки для экземпляра класса opt.subopt вызывается метод __call__, который обрабатывает аргументы вплоть до закрытия фигурной скобки либо до окончания списка аргументов. При этом открывающая и закрывающая фигурные скобки могут находиться в разных файлах или интерактивных сессиях, поскольку стек файлов не зависит от стека вызовов методов __call__.

Количество закрывающих фигурных скобок не обязательно должно соответствовать количеству открывающих — каждая открывающая скобка запускает соответствующий метод __call__, но не закрытые явно методы будут все равно завершены после завершения разбора всех аргументов.

Несколько подопцитй могут указываться в цепочке вида opt.subopt.subsubopt..., при этом последняя опция обрабатывается по обычным правилам, т.е. допустимы выражения вида opt.subopt.subsubopt=<VAL> (изменение значения поля subsubopt) или opt.subopt.subsubopt <PAR1> <PAR2> ... (запуск опции subsubopt с параметрами <PAR1> <PAR2> ...).

Имена подопций в цепочке должны указываться полностью, либо начинаться с '--' и сокращаться по обычным правилам, например opt.--subo.--sso.

Не допускается использование открывающей фигурной скобки для уже запущенного метода __call__ (ошибка E.8, см. таблицу 3).

4.4.7 Параметры определяющие ключевые выражения управляющие синтаксисом

Все ключевые выражения управляющие синтаксисом не запрограммированы жёстко, а заданны как значения некоторых глобальных переменных модуля IGL (таблица 2). Изменяя эти параметры можно отчасти изменить предоставляемый модулем синтаксис. При этом алгоритм разбора аргументов останется неизменными.

параметр	значение	назначение параметра
_opt	'--'	явное указание опции (предваряет имя опции без пробела)
_arg	'@'	явное указание параметра опции (предваряет параметр без пробела)
_dot	'.'	разделитель между подопциями при спуске по дереву опций, см. раздел 4.4.6
_end	';'	явное терминирование опции
_bra	'{'	открытие уровня в дереве опций, см. раздел 4.4.6
_ket	'}'	закрытие уровня в дереве опций, см. раздел 4.4.6
_stdin	'--'	читать аргументы со стандартного ввода в интерактивном режиме
_file	'read'	читать аргументы из файла (следующим аргументом должно идти имя файла)
_help	'?'	обращение к справке

Таблица 2: Глобальные параметры модуля IGL, содержащие ключевые слова и последовательности

4.5 Вывод отладочной информации, исключения и сообщения об ошибках

Для вывода подробной информации о процессе разбора аргументов необходимо задать в поле пользовательского класса _verbose файловый объект, в который должен производиться вывод (по умолчанию поле _verbose задано как None). При этом будет выводиться информация об изменении состояния парсера с номерами строк модуля IGL.py.

В случае возникновения ситуаций с которыми не может справиться алгоритм разбора аргументов, возбуждается исключение типа ParseError, определённое в модуле IGL как наследник стандартного родительского класса исключений Exception. Все сообщения об ошибках имеют коды вида E.N (в настоящее время $0 \leq N < 16$) и перечислены в таблице 3.

Код ошибки	Сообщение	Описание ошибки
E.0	option "<S>" unrecognized (variants=<VARIANTS>)	заданная в сокращённом виде опция <S> не распознана, поскольку существует несколько вариантов <VARIANTS> ее распознавания
E.1	can't convert "<ARG>" to bool	значение <ARG> не может быть приведено к типу bool, поскольку допустимы лишь 'y', 'Y', 'yes', 'Yes', 'YES', 'on', 'On', 'ON', 'true', 'True', 'TRUE', 'n', 'N', 'no', 'No', 'NO', 'off', 'Off', 'OFF', 'false', 'False', 'FALSE'
E.2	<N>-st non-keyword param <KEY1>=<VAL1> after keyword param <KEY2>=<VAL2>	значение <VAL1> заданное для параметра <KEY1> под номером <N> как для позиционного параметра, введено после того, как было заданно значение <VAL2> для параметра <KEY2> как для именованного параметра, причём параметр <KEY2> находится в сигнатуре опции **раньше**, чем параметр <KEY1> (в этом случае после задания <KEY2> позиция <KEY1> изменяется, что потенциально является источником ошибок)
E.3	<OPT>(...) got multiple values for keyword parametr "<KEY>"	опция <OPT> получила несколько значений для именованного параметра <KEY>, т.е. пользователь произвел попытку несколько раз задать значения для одного и того же параметра опции
E.4	terminate "<NAME>=... but value expected!	попытка терминировать процесс изменения поля класса <NAME> без указания нового значения поля
E.5	terminate "<OPT>" without args <PAR1>, <PAR2>, ...	попытка терминировать опцию <OPT> с неполным набором обязательных параметров, не заданны параметры <ARG1>, <ARG2>, ...
E.6	type(<OPT>) in "<CHAIN>" is <CLASS>, but BASE subclass expected!	попытка указать в цепочке <CHAIN> вида OPT.SUBOPT.SUBSUBOPT... поле, являющееся экземпляром класса <CLASS>, который не является наследником класса BASE
E.7	<ARG_STACK> read : filename expected!	для стека <ARG_STACK> после ключевого слова read, переключающего разбор на чтение аргументов из файла, не было указанно имя файла
E.8	incorrect used "{"	некорректное использование открывающей скобки {
E.9	<OPT> <PAR> : got option, but parametr expected!	опция <OPT> ожидала параметр, однако по синтаксису введённое значение <PAR> явная опция (поскольку начинается с '--')
E.10	<OPT> <KEY>=<VAL> bad parametr name!	при задании именованного параметра <KEY>=<VAL> для опции <OPT> имя параметра <KEY> введено с ошибкой
E.11	option "<OPT>" unrecognized!	опция <OPT> не распознана (имя опции введено с ошибкой)
E.12	parametr "<KEY>" unrecognized!	в записи вида ...<KEY>=<VAL> указанно имя <KEY> несуществующего поля
E.13	sub-option "<OPT>" in "<CHAIN>" unrecognized!	подопция <OPT> в цепочке опций <CHAIN> не распознана (имя опции введено с ошибкой)
E.14	got explicit parametr "<PAR> but option expected!	по синтаксису параметр <PAR> не является именем опции, но ни обычная ни неявная опции не инициализированы, т.е. должна быть указанна опция а не параметр, см. раздел 4.4.2
E.15	unknown error for arg "<ARG>"	модуль не в состоянии не только корректно обработать аргумент <ARG>, но даже распознать возникшую ошибку

Таблица 3: Сообщения об ошибках модуля IGL

В интерактивном режиме работы все исключения, наследующие Exception перехватываются, выводятся в стандартный поток ошибок и работа продолжается. Исключение KeyboarInterrupt (возникающее при нажатии <Ctrl-C>) перехватывается и приводит к выходу из интерактивного режима.

В остальных случаях исключения приводят к завершению работы функции __call__ и могут перехватываться и обрабатываться пользователем.

5 Заключение

Разбор аргументов командной строки не ограничивается приведёнными здесь тремя способами — «у каждого додика своя методика». Тем не менее рассмотренные варианты достаточно характерны: канонический getopt, ленивый exec и сверхмощный IGL (или какие то аналогичные IGL модули, специфичные для конкретных задач).

Я постарался сделать описание IGL как можно более полным, тем не менее некоторые вопросы ещё не раскрыты. Я сознаю, что примеров могло быть побольше, а алгоритм разбора аргументов неплохо бы расписать пополнее и снабдить блок–схемой. Надеюсь, что эти недостатки удастся исправить в будущем.

Модуль IGL ещё сыроват, но имеет большие перспективы для развития. В частности, планируется подключить библиотеку readline, обеспечить вывод логов, и, возможно, автоматическую генерацию оконного интерфейса — в этом случае пользователь сможет самостоятельно выбирать между «форточками» и командной строкой.

Список литературы

[1] Григорий Строкин «BASH конспект» 1997. http://www.linux.org.ru/books/bash-conspect.html

[2] Mendel Cooper «Advanced Bash-Scripting Guide», перевод: Андрей Киселев «Искусство программирования на языке сценариев командной оболочки» http://www.opennet.ru/docs/RUS/bash_scripting_guide/

[3] Г. Россум, Ф.Л.Дж. Дрейк, Д.С. Откидач. Язык программирования Python. 2001 — 454С.

[4] Марк Лутц. Программирование на Python. С-Пб.: «Символ». 2002 — 1135С.