Разбор параметров командной строки в Python

https://jenyay.net/Programming/Argparse

Введение

Как вы, наверное, знаете, все программы можно условно разделить на консольные и использующие графический интерфейс (сервисы под Windows и демоны под Linux не будем брать в расчет). Параметры запуска, задаваемые через командную строку, чаще всего используют консольные программы, хотя программы с графическим интерфейсом тоже не брезгуют этой возможностью. Наверняка в жизни каждого программиста была ситуация, когда приходилось разбирать параметры командной строки, как правило, это не самая интересная часть программы, но без нее не обойтись. Эта статья посвящена тому, как Python облегчает жизнь программистам при решении этой задачи благодаря своей стандартной библиотеке argparse.

Примеры без использования argparse

Путь для начала у нас есть простейший скрипт на Python. Для определенности назовем скрипт *coolprogram.py*, это будет классический Hello World, над которым мы будем работать в дальнейшем.

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
if __name__ == "__main__":
    print ("Привет, мир!")
```

Мы завершили эту сложнейшую программу и отдали ее заказчику, он доволен, но просит добавить в нее возможность указывать имя того, кого приветствуем, причем этот параметр может быть не обязательным. Т.е. программа может использоваться двумя путями:

```
$ python coolprogram.py
```

или

```
$ python coolprogram.py Вася
```

Первое, что приходит на ум при решении такой задачи, просто воспользоваться переменной *argv* из модуля *sys*. Напомню, что *sys.argv* содержит список параметров, переданных программе через командную строку, причем нулевой элемент списка - это имя нашего скрипта. Т.е. если у нас есть следующий скрипт с именем *params.py*:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import sys

if __name__ == "__main__":
    for param in sys.argv:
        print (param)
```

и мы запускаем его с помощью команды

```
python params.py
```

то в консоль будет выведена единственная строка:

```
params.py
```

Если же мы добавим несколько параметров,

```
python params.py param1 param2 param3
```

то эти параметры мы увидим в списке *sys.argv*, начиная с первого элемента:

```
params.py
param1
param2
param3
```

Здесь можно обратить внимание на то, что ссылка на интерпретатор Python в список этих параметров не входит, хотя он также присутствует в строке вызова нашего скрипта.

Вернемся к нашей задаче. Погрузившись в код на неделю, мы могли бы выдать заказчику следующий скрипт:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import sys

if __name__ == "__main__":
    if len (sys.argv) > 1:
        print ("Привет, {}!".format (sys.argv[1] ) )
    else:
        print ("Привет, мир!")
```

Теперь, если программа вызывается с помощью команды
`python coolprogram.py`

то результат будет прежний
`Привет, мир!`

а если мы добавим параметр:
`python coolprogram.py Вася`

то программа поприветствует некоего Васю:
`Привет, Вася!`

Пока все легко и никаких проблем не возникает. Теперь предположим, что требования заказчика вновь изменились, и на этот раз он хочет, чтобы имя приветствуемого человека передавалось после именованного параметра *--name* или *-n*, причем нужно следить, что в командной строке передано только одно имя. С этого момента у нас начнется вермишель из конструкций *if*.

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import sys

if __name__ == "__main__":
    if len (sys.argv) == 1:
        print ("Привет, мир!")
    else:
```

```
        if len (sys.argv) < 3:
            print ("Ошибка. Слишком мало параметров.")
            sys.exit (1)

        if len (sys.argv) > 3:
            print ("Ошибка. Слишком много параметров.")
            sys.exit (1)

        param_name = sys.argv[1]
        param_value = sys.argv[2]

        if (param_name == "--name" or
                param_name == "-n"):
            print ("Привет, {}!".format (param_value) )
        else:
            print ("Ошибка. Неизвестный параметр
'{}'".format (param_name) )
            sys.exit (1)
```

Здесь мы проверяем ситуацию, что мы вообще не передали ни одного параметра, потом проверяем, что дополнительных параметров у нас ровно два, что они называются именно *--name* или *-n*, и, если нас все устраивает, выводим приветствие.

Как видите, код превратился в тихий ужас. Изменить логику работы в нем в дальнейшем будет очень сложно, а при увеличении количества параметров нужно будет срочно применять объектно-ориентированные меры по отделению логики работы программы от разбора командной строки. Разбор командной строки мы могли бы выделить в отдельный класс (или классы), но мы этого здесь делать не будем, поскольку все уже сделано в стандартной библиотеке Python, которая называется **argparse**.

Использование библиотеки argparse

Первый пример

Как как было сказано выше, стандартная библиотека *argparse* предназначена для облегчения разбора командной строки. На нее можно возложить проверку переданных параметров: их количество и обозначения, а уже после того, как эта проверка будет выполнена автоматически, использовать полученные параметры в логике своей программы.

Основой работы с командной строкой в библиотеке *argparse* является класс *ArgumentParser*. У его конструктора и методов довольно много параметров, все их рассматривать не будем, поэтому в дальнейшем рассмотрим работу этого класса на примерах, попутно обсуждая различные параметры.

Простейший принцип работы с *argparse* следующий:

1. Создаем экземпляр класса *ArgumentParser*.
2. Добавляем в него информацию об ожидаемых параметрах с помощью метода *add_argument* (по одному вызову на каждый параметр).
3. Разбираем командную строку с помощью метода *parse_args*, передавая ему полученные параметры командной строки (кроме нулевого элемента списка *sys.argv*).
4. Начинаем использовать полученные параметры.

Для начала перепишем программу *coolprogram.py* с единственным параметром так, чтобы она использовала библиотеку *argparse*. Напомню, что в данном случае мы ожидаем следующий синтаксис параметров:

```
python coolprogram.py [Имя]
```

Здесь [Имя] является необязательным параметром.

Наша программа с использованием *argparse* может выглядеть следующим образом:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import sys
import argparse


def createParser ():
    parser = argparse.ArgumentParser()
    parser.add_argument ('name', nargs='?')

    return parser


if __name__ == '__main__':
    parser = createParser()
    namespace = parser.parse_args()

    # print (namespace)

    if namespace.name:
        print ("Привет, {}!".format (namespace.name) )
    else:
        print ("Привет, мир!")
```

На первый взгляд эта программа работает точно так же, как и раньше, хотя есть отличия, но мы их рассмотрим чуть позже. Пока разберемся с тем, что мы понаписали в программе.

Создание парсера вынесено в отдельную функцию, поскольку эта часть программы в будущем будет сильно изменяться и разрастаться. На строке 9 мы создали экземпляр класса *ArgumentParser* с параметрами по умолчанию. Что это за параметры, опять же, поговорим чуть позже.

На строке 10 мы добавили ожидаемый параметр в командной строке с помощью метода *add_argument*. При этом такой параметр будет считаться позиционным, т.е. он должен стоять именно на этом месте и у него не будет никаких предварительных обозначений (мы их добавим позже в виде '-n' или '--name'). Если бы мы не добавили именованный параметр *nargs='?'*, то этот параметр был бы обязательным. *nargs* может принимать различные значения. Если бы мы ему присвоили целочисленное значение больше 0, то это бы означало, что мы ожидаем ровно такое же количество передаваемых параметров (точнее, считалось бы, что первый параметр ожидал бы список из N элементов, разделенных пробелами, этот случай мы рассмотрим позже). Также этот параметр может принимать значение '?', '+', '*' и *argparse.REMAINDER*. Мы их не будем рассматривать, поскольку они важны в сочетании с необязательными именованными параметрами, которые могут располагаться как до, так и после нашего позиционного параметра. Тогда этот параметр будет показывать как интерпретировать список параметров, где будет заканчиваться один список параметров и начинаться другой.

Итак, мы создали парсер, после чего можно вызвать его метод *parse_args* для разбора командной строки. Если мы не укажем никакого параметра, это будет равносильно тому, что мы передадим в него все параметры из *sys.argv* кроме нулевого, который содержит имя нашей программы. т.е.

```
parser.parse_args (sys.argv[1:])
```

В качестве результата мы получим экземпляр класса *Namespace*, который будет содержать в качестве члена имя нашего параметра. Теперь можно раскомментировать строку 19 в приведенном выше примере, чтобы посмотреть, чему же равны наши параметры.

Если мы это сделаем и запустим программу с переданным параметром

```
python coolprogram.py Вася
```

то увидим его в пространстве имен.

```
Namespace(name='Вася')
```

Если же теперь мы запустим программу без дополнительных параметров, то это значение будет равно None:

```
Namespace(name=None)
```

Мы можем изменить значение по умолчанию, что позволит нам несколько сократить программу. Пусть по умолчанию используется слово 'мир', ведь мы его приветствуем, если параметры не переданы. Для этого воспользуемся дополнительным именованным параметром *default* в методе *add_argument*.

```
1.    #!/usr/bin/python
2.    # -*- coding: UTF-8 -*-
3.
4.    import sys
5.    import argparse
6.
7.
8.    def createParser ():
9.        parser = argparse.ArgumentParser()
10.       parser.add_argument ('name', nargs='?', default='мир')
11.
12.       return parser
13.
14.
15.   if __name__ == '__main__':
16.       parser = createParser()
17.       namespace = parser.parse_args (sys.argv[1:])
18.
19.       # print (namespace)
20.
21.       print ("Привет, {}!".format (namespace.name) )
```

Программа продолжает работать точно также, как и раньше. Вы, наверное, заметили, что в предыдущем примере в метод *parse_args* на строке 17 передаются параметры командной строки из *sys.argv*. Это сделано для того, чтобы показать, что список параметров мы можем передавать явно, при необходимости мы его можем предварительно обработать, хотя это вряд ли понадобится, ведь почти всю обработку можно возложить на плечи библиотеки *argparse*.

Добавляем именованные параметры

Теперь снова переделаем нашу программу таким образом, чтобы использовать именованные параметры. Напомню, что согласно последнему желанию (в смысле, для данной программы) заказчика имя приветствуемого человека должно передаваться после параметра *--name* или *-n*. С помощью *pyparse* сделать это проще простого - достаточно в качестве первых двух параметров метода *add_argument* передать эти имена параметров.

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
```

```
7.
8.  def createParser ():
9.      parser = argparse.ArgumentParser()
10.     parser.add_argument ('-n', '--name', default='мир')
11.
12.     return parser
13.
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args(sys.argv[1:])
18.
19.     # print (namespace)
20.
21.     print ("Привет, {}!".format (namespace.name) )
```

Теперь, если мы запустим программу без параметров, то увидим знакомое "Привет, мир!", а если мы запустим программу с помощью команды

```
python coolprogram.py -n Вася
```

или

```
python coolprogram.py --name Вася
```

То приветствовать программа будет Васю. Обратите внимание, что теперь в методе *add_argument* мы убрали параметр *nargs='?'* , поскольку **все именованные параметры считаются необязательными**. А если они не обязательные, то возникает вопрос, как поведет себя argparse, если этот параметр не передан? Для этого уберем параметр *default* в *add_argument*.

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('-n', '--name')
10.
11.     return parser
12.
13.
14. if __name__ == '__main__':
15.     parser = createParser()
16.     namespace = parser.parse_args(sys.argv[1:])
17.
18.     print ("Привет, {}!".format (namespace.name) )
```

Если теперь запустить программу без параметров, то увидим приветствие великого *None*:

```
Привет, None!
```

Таким образом, если значение по умолчанию не указано, то оно считается равным *None*.

Раньше мы на это не обращали внимание, но нужно все-таки сказать, что *argparse* по умолчанию заточен для создания интерфейса командной строки именно в стиле UNIX, где именованные параметры начинаются с символов "-" или "--". Если такой параметр попытаться называть в стиле Windows (например, "/name"), то будет брошено исключение:

```
ValueError: invalid option string '/name': must start
with a character '-'
```

На самом деле буквально с помощью одного параметра мы можем заставить работать *argparse* в стиле Windows, но это выходит за рамки статьи, поэтому мы везде будем использовать стиль UNIX.

До этого мы задавали два имени для одного и того же параметра: длинное имя, начинающееся с "--" (*--name*) и короткое сокращение, начинающееся с "-" (*-n*). При этом получение значение параметра из пространства имен осуществляется по длинному имени:

```
print ("Привет, {}!".format (namespace.name) )
```

Если мы не зададим длинное имя, то придется обращаться к параметру через его короткое имя (*n*):

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('-n')
10.
11.     return parser
12.
13.
14. if __name__ == '__main__':
15.     parser = createParser()
16.     namespace = parser.parse_args(sys.argv[1:])
17.
18.     print (namespace)
19.
20.     print ("Привет, {}!".format (namespace.n) )
```

При этом пространство имен будет выглядеть как:
```
Namespace(n='Вася')
```

Хорошо, с уменьшением количества имен параметров разобрались, но мы можем еще и увеличить количество имен, например, мы можем добавить для того же параметра еще новое имя *--username*, для этого достаточно его добавить следующим параметром метода *add_argument*:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     parser.add_argument ('-n', '--name', '--username')
10.
11.    return parser
12.
13.
14. if __name__ == '__main__':
15.    parser = createParser()
16.    namespace = parser.parse_args(sys.argv[1:])
17.
18.    print (namespace)
19.
20.    print ("Привет, {}!".format (namespace.name) )
```

Теперь мы можем использовать три варианта передачи параметров:
```
python coolprogram.py -n Вася
python coolprogram.py --name Вася
python coolprogram.py --username Вася
```

Все три варианта равнозначны, при этом надо обратить внимание, что при получении значения этого параметра используется первое длинное имя, т.е. *name*. Пространство имен при использовании всех трех вариантов вызова программы будет выглядеть одинаково:
```
Namespace(name='Вася')
```

Параметры как списки

До сих пор мы ожидали, что в качестве значения параметра выступает строка, но бывают ситуации, когда необходимо, чтобы в качестве значения параметра принимался список строк. Например, пусть нам теперь нужно изменить программу так, чтобы мы могли приветствовать нескольких

человек. Первое, что приходит на ум - это вручную разобрать строку после имени параметра, т.е.

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     parser.add_argument ('-n', '--name', default='мир')
10.
11.    return parser
12.
13.
14. if __name__ == '__main__':
15.    parser = createParser()
16.    namespace = parser.parse_args(sys.argv[1:])
17.
18.    print (namespace)
19.
20.    for name in namespace.name.split():
21.        print ("Привет, {}!".format (name) )
```

Но это не самое лучшее решение, поскольку если пользователь захочет передать несколько параметров после *--name*, то ему придется оборачивать список имен в кавычки:

```
python coolprogram.py --name "Вася Оля Петя"
```

Поскольку, если написать просто

```
python coolprogram.py --name Вася Оля Петя
```

, то *argparse* решит, что "Оля Петя" - это отдельные параметры, имя для которых не задано, и напишет ошибку:

```
error: unrecognized arguments: Оля Петя
```

Чтобы указать библиотеке *argparse*, что значений параметров у нас может быть несколько, вспомним уже используемый нами параметр *nargs*. Но если в прошлый раз в качестве его значения мы использовали значение "?", обозначающее, что у нас может быть 0 или 1 значение, то теперь мы будем использовать значение *nargs='+'*, обозначающее, что мы ожидаем одно или более значение (по аналогии с регулярными выражениями).

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
```

```
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('-n', '--name', nargs='+', default=['мир'])
10.
11.     return parser
12.
13.
14. if __name__ == '__main__':
15.     parser = createParser()
16.     namespace = parser.parse_args(sys.argv[1:])
17.
18.     print (namespace)
19.
20.     for name in namespace.name:
21.         print ("Привет, {}!".format (name) )
```

Обратите внимание, что здесь в качестве значения по умолчанию используется **список** из одного слова.

Теперь, если мы выполним эту программу с помощью команды

```
python coolprogram.py --name Вася Оля Петя
```

То программа нам выведет:

```
Namespace(name=['Вася', 'Оля', 'Петя'])
Привет, Вася!
Привет, Оля!
Привет, Петя!
```

Таким образом, мы возложили задачу разделения параметров на плечи *argparse*. Напомню, что *nargs* может принимать и другие значения, в том числе целое число *N*, обозначающее, что мы ожидаем ровно *N* значений.

Выбор из вариантов

Представим себе еще один случай. Пусть у нас есть консольная программа, наподобие ImageMagick, то есть работающая с графикой. Наверняка эта программа будет поддерживать ограниченное количество форматов графических файлов. В этом случае может понадобиться ограничить значения некоторого параметра заранее заданным списком значений (например, gif, png, jpg). Разумеется, такую несложную проверку мы могли бы сделать и сами, но лучше пусть этим займется *argparse*. Для этого достаточно в метод *add_argument* передать еще один параметр - *choices*, который принимает список возможных значений параметра.

Чтобы не удаляться от наших предыдущих примеров, изменим последнюю программу таким образом, чтобы она могла приветствовать только лишь заранее очерченный круг лиц (неких Васю, Олю или Петю). Сделать это абсолютно несложно:

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('-n', '--name', choices=['Вася', 'Оля', 'Петя'], default='Оля')
10.
11.     return parser
12.
13.
14. if __name__ == '__main__':
15.     parser = createParser()
16.     namespace = parser.parse_args(sys.argv[1:])
17.
18.     print (namespace)
19.
20.     print ("Привет, {}!".format (namespace.name) )
```

Теперь, если мы передадим в качестве параметра одно из перечисленных имен, то программа будет работать, как и прежде, однако незнакомых людей она откажется приветствовать:

```
$ python coolprogram.py --name Вася

Namespace(name='Вася')
Привет, Вася!

$ python coolprogram.py --name Иван

usage: coolprogram.py [-h] [-n {Вася,Оля,Петя}]
coolprogram.py: error: argument -n/--name: invalid
choice: 'Иван' (choose from 'Вася', 'Оля', 'Петя')
```

Иногда может быть полезным в качестве значения параметра *choices* использовать список целых чисел, полученных с помощью стандартной функции *range*.

Указание типов параметров

До сих пор мы в качестве входных параметров использовали строки, что логично, поскольку на самом нижнем уровне все входные параметры считаются строками. Однако часто на практике в качестве параметров хотелось бы использовать другие типы: целые числа или числа с плавающей точкой. Конечно, не проблема вручную преобразовать строку в нужный тип или написать ошибку в случае, если этого сделать невозможно. Однако, даже такую мелочь можно переложить на *argparse*. Для этого достаточно добавить

еще один именованный параметр в метод *add_argument*, а именно параметр *type*, который будет указывать ожидаемый тип значения параметра. Изменим пример таким образом, чтобы в качестве входного параметра с именем *-c* или *--count* принималось целое число, которое обозначает, сколько раз нужно вывести строку "Привет, мир!" (временно забудем про параметры *-n* и *--name*).

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('-c', '--count', default=1, type=int)
10.
11.     return parser
12.
13.
14. if __name__ == '__main__':
15.     parser = createParser()
16.     namespace = parser.parse_args(sys.argv[1:])
17.
18.     print (namespace)
19.
20.     for _ in range (namespace.count):
21.         print ("Привет, мир!")
```

Нас здесь больше всего интересует 9-я строка, где мы указали ожидаемый тип параметра. Обратите внимание, что в качестве значения параметра *type* мы передали не строку, а стандартную функцию преобразования в целое число. В остальном там нет никаких особенностей.

В следующем блоке показан пример работы этой программы:

```
$ python coolprogram.py --count 5

Namespace(count=5)
Привет, мир!
Привет, мир!
Привет, мир!
Привет, мир!
Привет, мир!
```

Здесь мы видим, что в пространстве имен параметр *count* сразу записан как целое число (без кавычек). Если же мы попытаемся передать в качестве параметра какую-то строку, которую преобразовать в целое число невозможно, то мы получим ошибку:

```
$ python coolprogram.py --count Абырвалг

usage: coolprogram.py [-h] [-c COUNT]
coolprogram.py: error: argument -c/--count: invalid int
value: 'Абырвалг'
```

Имена файлов как параметры

Но это не все возможности параметра *type*. Во многих программах в качестве входных параметров используются имена файлов, которые нужно прочитать. При этом имена файлов передают обычно не просто так, эти файлы или читают, или в них что-то пишут, а, как известно, открытие файла - это лотерея, может быть он откроется, а может и нет, здесь всегда надо ловить исключения. А раз это такая частая операция, то *argparse* позволяет и это автоматизировать.

Для примера напишем программу, которая просто выводит содержимое текстового файла, имя которого задается после именованного параметра *-n* или *--name* Для работы программы понадобится дополнительный файл "text.txt" для простоты в кодировке UTF-8.

Для того, чтобы указать *argparse*, что в качестве входного параметра мы ожидаем файл, который должен быть открыт для чтения, в метод *add_argument* нужно передать параметр *type*, равный *open* (опять же, это стандартная функция для открытия файла, а не строка). Использование этого параметра показано ниже:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     parser.add_argument ('-n', '--name', type=open)
10.
11.    return parser
12.
13.
14. if __name__ == '__main__':
15.    parser = createParser()
16.    namespace = parser.parse_args(sys.argv[1:])
17.
18.    print (namespace)
19.
20.    text = namespace.name.read()
21.
22.    print (text)
```

Если вы запустите этот пример, то кроме содержимого файла, переданного в качестве параметра *--name*, увидите, что пространство имен *namespace* будет содержать уже открытый файл:

```
$ python coolprogram.py --name text.txt

Namespace(name=<_io.TextIOWrapper name='text.txt'
mode='r' encoding='UTF-8'>)
Содержимое текстового файла
```

Если же мы укажем имя несуществующего файла, то получим исключение:

```
1. $ python coolprogram.py --name invalid.txt
2.
3. Traceback (most recent call last):
4.   File "coolprogram.py", line 16, in <module>
5.     namespace = parser.parse_args(sys.argv[1:])
6.   File "/usr/lib/python3.3/argparse.py", line 1714, in parse_args
7.     args, argv = self.parse_known_args(args, namespace)
8.   File "/usr/lib/python3.3/argparse.py", line 1746, in parse_known_args
9.     namespace, args = self._parse_known_args(args, namespace)
10.  File "/usr/lib/python3.3/argparse.py", line 1952, in _parse_known_args
11.    start_index = consume_optional(start_index)
12.  File "/usr/lib/python3.3/argparse.py", line 1892, in consume_optional
13.    take_action(action, args, option_string)
14.  File "/usr/lib/python3.3/argparse.py", line 1804, in take_action
15.    argument_values = self._get_values(action, argument_strings)
16.  File "/usr/lib/python3.3/argparse.py", line 2247, in _get_values
17.    value = self._get_value(action, arg_string)
18.  File "/usr/lib/python3.3/argparse.py", line 2276, in _get_value
19.    result = type_func(arg_string)
20. FileNotFoundError: [Errno 2] No such file or directory: 'invalid.txt'
```

Как видно из строк 4-5, исключение будет брошено при попытке разобрать параметры с помощью метода *parse_args*. Сначала кажется странным, что argparse не обрабатывает такую простую ситуацию, бросая такое некрасивое исключение. Казалось бы, почему бы библиотеке самой его не обработать и не вывести ошибку, как это было сделано при использовании параметра *type=int*?

Несмотря на то, что использование параметра *type=open* описано в документации, на мой взгляд его лучше не использовать в таком виде, поскольку *argparse* предлагает более красивое решение этой проблемы. Достаточно в качестве параметра *type* передать не функцию *open*, а экземпляр класса, полученного с помощью функции *argparse.FileType*, предназначенной для безопасной попытки открытия файла.

Функция *argparse.FileType* выглядит следующим образом и напоминает упрощенную версию функции *open*:

```
1. argparse.FileType(mode='r', bufsize=-1, encoding=None, errors=None)
```

Здесь первый параметр задает режим открытия файла ('r' - чтение, 'w' - запись и т.д. по аналогии с функцией *open*), второй задает размер буфера, если нужно использовать буферизированое чтение, третий параметр задает кодировку открываемого файла, а последний - действие в случае ошибки декодирования файла. Эта функция создает экземпляр класса, предназначенного для работы с файлами.

Таким образом, мы можем исправить предыдущий пример:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     parser.add_argument ('-n', '--name', type=argparse.FileType())
10.
11.    return parser
12.
13.
14. if __name__ == '__main__':
15.    parser = createParser()
16.    namespace = parser.parse_args(sys.argv[1:])
17.
18.    print (namespace)
19.
20.    text = namespace.name.read()
21.
22.    print (text)
```

Теперь, если пользователь попытается передать имя несуществующего файла, то он увидит не большой страшный листинг исключения, а внятную ошибку:

```
$ python coolprogram.py --name invalid.txt

usage: coolprogram.py [-h] [-n NAME]
coolprogram.py: error: argument -n/--name: can't open
'invalid.txt': [Errno 2] No such file or directory:
'invalid.txt'
```

Аналогично мы можем принимать имя файла для записи, при этом только надо не забыть передать в фукнцию *argparse.FileType* в качестве первого параметра строку 'w'.

Обязательные именованные параметры

До сих пор мы говорили, что позиционные параметры у нас обязательные. Напомню, что позиционными считаются те параметры, имена которых не начинаются с символов "-" или "--". Почему они называются позиционными, поговорим чуть позже, когда перейдем к разделу про разбор нескольких параметров.

Т.е. следующая программа у нас должна обязательно получить один параметр:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7.
8. def createParser ():
9.     parser = argparse.ArgumentParser()
10.    parser.add_argument ('name')
11.
12.    return parser
13.
14.
15. if __name__ == '__main__':
16.    parser = createParser()
17.    namespace = parser.parse_args()
18.
19.    print (namespace)
20.
21.    print ("Привет, {}!".format (namespace.name) )
```

А поскольку параметр у нас не именованный (не начинается с символов "-" или "--"), то мы не должны в командной строке указывать имя параметра. Если же мы не укажем обязательный позиционный параметр, то *argparse* нам выведет понятную ошибку, а не будет бросать исключения. В следующем блоке показаны два случая использования этого скрипта: с переданным параметром и без него.

```
$ python coolprogram.py Вася

Namespace(name='Вася')
Привет, Вася!


$ python coolprogram.py

usage: coolprogram.py [-h] name
```

```
coolprogram.py: error: the following arguments are
required: name
```

Теперь вернемся к примеру, где мы этот параметр сделали именованным:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7.
8. def createParser ():
9.     parser = argparse.ArgumentParser()
10.    parser.add_argument ('-n', '--name')
11.
12.    return parser
13.
14.
15. if __name__ == '__main__':
16.    parser = createParser()
17.    namespace = parser.parse_args()
18.
19.    print (namespace)
20.
21.    print ("Привет, {}!".format (namespace.name) )
```

Именованные параметры по умолчанию считаются необязательными, если мы его не укажем, то получим в качестве результата работы:

```
Namespace(name=None)
Привет, None!
```

Если же нас такое поведение не устраивает, мы можем указать, что этот параметр является обязательным, для этого достаточно в метод *add_argument* добавить параметр *required=True*

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7.
8. def createParser ():
9.     parser = argparse.ArgumentParser()
10.    parser.add_argument ('-n', '--name', required=True)
11.
12.    return parser
13.
```

```
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args()
18.
19.     print (namespace)
20.
21.     print ("Привет, {}!".format (namespace.name) )
```

Теперь при попытке запустить программу без параметров пользователь увидит ошибку:

```
$ python coolprogram.py

usage: coolprogram.py [-h] -n NAME
coolprogram.py: error: the following arguments are
required: -n/--name
```

Параметры как флаги

Иногда может возникнуть желание добавить параметры, которые должны работать как флаги, т.е. они или были указаны, или нет. Например, пусть у нас есть простейшая программа "Hello World", но если мы ей укажем параметр "--goodbye" или "-g", то программа не только поздоровается, но и попрощается с миром. :)

```
$ python coolprogram.py

Привет, мир!


$ python coolprogram.py --goodbye

Привет, мир!
Прощай, мир!
```

Для того, чтобы смоделировать такое поведение нам понадобится еще один параметр метода *add_argument*, который мы до этого не использовали. Этот аргумент называется *action*, который предназначен для выполнения некоторых действий над значениями переданного параметра. Мы не будем подробно рассматривать все возможные действия, поскольку многие из них довольно специфические и требуют подробного рассмотрения, и в двух словах их не объяснишь, поэтому рассмотрим только один из вариантов использования этого параметра.

С формальной точки зрения, если мы явно не указываем значения параметра *action*, то он равен строке *"store"*, это означает, что парсер должен просто хранить полученные значения переменных, это происходило во всех приведенных выше примерах.

Для начала мы воспользуемся другим значением параметра *action*, а именно строкой *"store_const"*, которая обозначает, что если данный параметр указан, то он всегда будет принимать значение, указанное в другом параметре метода *add_argument* - *const*. Если этот параметр указан не будет, то его значение будет равно *None*.

Напишем наш новый Hello/goodbye World:

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.
8.  def createParser ():
9.      parser = argparse.ArgumentParser()
10.     parser.add_argument ('-g', '--goodbye', action='store_const', const=True)
11.
12.     return parser
13.
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args()
18.
19.     print (namespace)
20.
21.     print ("Привет, мир!")
22.     if namespace.goodbye:
23.         print ("Прощай, мир!")
```

Теперь посмотрим, как он работает, и проследим за значением параметра *goodbye*:

```
$ python coolprogram.py

Namespace(goodbye=None)
Привет, мир!


$ python coolprogram.py --goodbye

Namespace(goodbye=True)
Привет, мир!
Прощай, мир!
```

Если нас смущает, что без указания параметра *--goodbye* его значение равно *None*, а не *False*, то мы можем воспользоваться уже знакомым нам параметром *default*:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7.
8. def createParser ():
9.     parser = argparse.ArgumentParser()
10.    parser.add_argument ('-g', '--goodbye', action='store_const', const=True, default=False)
11.
12.    return parser
13.
14.
15. if __name__ == '__main__':
16.    parser = createParser()
17.    namespace = parser.parse_args()
18.
19.    print (namespace)
20.
21.    print ("Привет, мир!")
22.    if namespace.goodbye:
23.        print ("Прощай, мир!")
```

Теперь, если этот параметр не указан, то он будет равен *False*, в остальном поведение не изменилось:

```
$ python coolprogram.py

Namespace(goodbye=False)
Привет, мир!


$ python coolprogram.py --goodbye

Namespace(goodbye=True)
Привет, мир!
Прощай, мир!
```

Может возникнуть вопрос, почему бы просто не использовать параметр *default*, зачем нужен еще *action*? Но одним *default* мы не обойдемся, поскольку *action* со значением *store* (значение по умолчанию для него) подразумевает, что если параметр (в данном случае *--goodbye*) указан, то

после него должно идти какое-то значение, а параметр *default* позволит указать значение этого параметра только при отсутствии *--goodbye*.

Поскольку используемый выше прием для создания флагов довольно часто используется, предусмотрено значение параметра *action*, позволяющее не указывать значение *const* для булевых значений: *store_true* и *store_false*. При использовании этих значений указанный параметр командной строки будет принимать соответственно *True* или *False*, если он указан. В данном случае мы можем воспользоваться значением *action="store_true"*:

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7.
8. def createParser ():
9.     parser = argparse.ArgumentParser()
10.    parser.add_argument ('-g', '--goodbye', action='store_true', default=False)
11.
12.    return parser
13.
14.
15. if __name__ == '__main__':
16.    parser = createParser()
17.    namespace = parser.parse_args()
18.
19.    print (namespace)
20.
21.    print ("Привет, мир!")
22.    if namespace.goodbye:
23.        print ("Прощай, мир!")
```

Результат работы скрипта не изменится.

Использование нескольких параметров

До сих пор во всех примерах мы задавали лишь один параметр, обязательный или необязательный, поскольку нас интересовало в первую очередь то, какие свойства могут быть у параметров. В этом разделе мы рассмотрим случаи, когда программа ожидает несколько параметров, что чаще всего и бывает.

Для добавления второго, третьего и т.д. параметра необходимо повторно вызвать метод *add_argument* класса *ArgumentParser*, описав с помощью переданных значений свойства ожидаемого параметра.

Например, следующий скрипт ожидает два позиционных параметра - имя приветствуемого человека и число, обозначающее, сколько раз его нужно поприветствовать. В данном случае оба позиционных параметра будут являться обязательными. Причем, первый параметр всегда задает имя, а

второй - количество, в данном примере поменять местами мы их не можем, отсюда и название таких параметров - "позиционные".

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('name')
10.     parser.add_argument ('count', type=int)
11.
12.     return parser
13.
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args(sys.argv[1:])
18.
19.     print (namespace)
20.
21.     for _ in range (namespace.count):
22.         print ("Привет, {}!".format (namespace.name) )
```

Различные способы запуска такой программы показаны ниже. В пером случае все задано корректно, а в остальных случаях есть ошибки: сначала аргументы перепутаны местами, вследствие чего второй параметр не удалось преобразовать в целое число, затем забыли передать один из параметров:

```
$ python coolprogram.py Петя 3

Namespace(count=3, name='Петя')
Привет, Петя!
Привет, Петя!
Привет, Петя!


$ python coolprogram.py 3 Петя

usage: coolprogram.py [-h] name count
coolprogram.py: error: argument count: invalid int
value: 'Петя'


$ python coolprogram.py Петя
```

```
usage: coolprogram.py [-h] name count
coolprogram.py: error: the following arguments are
required: count
```

Лично я недолюбливаю позиционные параметры как раз из-за того, что надо запоминать, в каком порядке они идут. Давайте сделаем их оба именованными, в этом случае порядок их передачи будет не важен, правда, при этом придется больше вводить символов, поскольку нужно будет вводить имена параметров.

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     parser.add_argument ('-n', '--name')
10.    parser.add_argument ('-c', '--count', type=int, default=1)
11.
12.    return parser
13.
14.
15. if __name__ == '__main__':
16.    parser = createParser()
17.    namespace = parser.parse_args(sys.argv[1:])
18.
19.    print (namespace)
20.
21.    for _ in range (namespace.count):
22.        print ("Привет, {}!".format (namespace.name) )
```

Теперь параметры можно менять местами, ведь имена параметров все-равно заданы.

```
$ python coolprogram.py --name Петя --count 3

Namespace(count=3, name='Петя')
Привет, Петя!
Привет, Петя!
Привет, Петя!


$ python coolprogram.py --count 3 --name Петя

Namespace(count=3, name='Петя')
Привет, Петя!
```

```
Привет, Петя!
Привет, Петя!
```

А как быть, если у нас имеются и позиционные, и необязательные именованные параметры?

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  def createParser ():
8.      parser = argparse.ArgumentParser()
9.      parser.add_argument ('name')
10.     parser.add_argument ('-c', '--count', type=int, default=1)
11.
12.     return parser
13.
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args(sys.argv[1:])
18.
19.     print (namespace)
20.
21.     for _ in range (namespace.count):
22.         print ("Привет, {}!".format (namespace.name) )
```

В каком порядке мы должны передавать параметры? Сначала позиционные параметры, а потом именованные или наоборот? На самом деле это не важно, оба варианта будут работать:

```
$ python coolprogram.py Петя --count 3

Namespace(count=3, name='Петя')
Привет, Петя!
Привет, Петя!
Привет, Петя!

$ python coolprogram.py --count 3 Петя

Namespace(count=3, name='Петя')
Привет, Петя!
Привет, Петя!
Привет, Петя!
```

Использование подпарсеров

Как вы уже поняли, библиотека *argparse* - достаточно мощная штука, выполняющая практически полностью разбор параметров командной строки и их проверку. Мы рассмотрели достаточно много примеров ее использования, но нельзя не упомянуть об еще одной возможности, а именно о подпарсерах (subparsers). Понять, что это такое проще на примере других программ.

Наверняка вы постоянно пользуетесь какой-нибудь системой контроля версий вроде svn, git, bzr или чем-то подобным. И хотя для них написано множество графических оболочек, надеюсь, вы хоть раз запускали их из консоли. Какие особенности у этих программ с точки зрения командной строки? А особенность заключается в том, что у них в качестве первого параметра всегда выступает имя команды, за которой следуют параметры, специфические именно для этой команды. Например:

```
$ git add -A
$ git commit -a -m "Текст коммита"
```

Причем для каждой такой команды свои параметры, они могут иметь одинаковые имена, но обозначать разные вещи. Вот такое разделение на команды и позволяют сделать подпарсеры.

Изменим нашу программу для вывода "Hello / goodbye world" таким образом, чтобы команда *hello* говорила программе, что нужно поприветствовать тех людей, список которых передан в параметре *--names* или *-n*, а команда *goodbye* указывала программе, что нужно попрощаться с миром столько раз, какое число передано с помощью параметра *--count* или *-c*. Поскольку у нас для каждой команды используется только один параметр, возможно, стоило бы его сделать позиционным, чтобы не набирать его имя, но именованные параметры более наглядны, особенно для демонстрации.

Чтобы реализовать такое поведение нужно выполнить несколько шагов:

1. Создать экземпляр класса *ArgumentParser* (назовем его корневым парсером).
2. Создать хранилище подпарсеров с помощью метода *add_subparsers* класса *ArgumentParser* (результатом будет экземпляр внутреннего класса *_SubParsersAction*).
3. Создать вложенный парсер с помощью метода *add_parser* класса хранилища парсеров. Результатом вызова каждого метода *add_parser* будет экземпляр класса *ArgumentParser*, уже знакомый нам.
4. Для каждого созданного вложенного парсера заполнить информацию об ожидаемых параметрах с помощью все того же метода *add_argument*.
5. Получить пространство имен с помощью метода *parse_args* класса *ArgumentParser* (корневого парсера).

6. Использовать полученное пространство имен так же, как и раньше с той лишь разницей, что в нем будет храниться имя переданной команды и параметры, специфические для нее.

Вот как выглядит текст программы для описанного чуть выше "Hello / goodbye world":

```
1. #!/usr/bin/python
2. # -*- coding: UTF-8 -*-
3.
4. import sys
5. import argparse
6.
7. def createParser ():
8.     parser = argparse.ArgumentParser()
9.     subparsers = parser.add_subparsers (dest='command')
10.
11.    hello_parser = subparsers.add_parser ('hello')
12.    hello_parser.add_argument ('--names', '-n', nargs='+', default=['мир'])
13.
14.    goodbye_parser = subparsers.add_parser ('goodbye')
15.    goodbye_parser.add_argument ('-c', '--count', type=int, default=1)
16.
17.    return parser
18.
19.
20. def run_hello (namespace):
21.    """
22.    Выполнение команды hello
23.    """
24.    for name in namespace.names:
25.        print ("Привет, {}!".format (name) )
26.
27.
28.
29. def run_goodbye (namespace):
30.    """
31.    Выполнение команды goodbye
32.    """
33.    for _ in range (namespace.count):
34.        print ("Прощай, мир!")
35.
36.
37. if __name__ == '__main__':
38.    parser = createParser()
39.    namespace = parser.parse_args(sys.argv[1:])
40.
41.    print (namespace)
42.
43.    if namespace.command == "hello":
44.        run_hello (namespace)
```

```
45.    elif namespace.command == "goodbye":
46.        run_goodbye (namespace)
47.    else:
48.        print ("Что-то пошло не так...")
```

В реальной программе каждую команду неплохо было бы обернуть в отдельный класс, но здесь для простоты мы обойдемся обычной функцией.

Думаю, что код достаточно лаконичный, чтобы по нему не было особых вопросов. Единственное, на что хочется обратить внимание, это то, что при создании хранилища подпарсеров с помощью метода *add_subparsers* мы использовали именованный параметр *dest = 'command'*, который указывает, что имя переданной команды в пространстве имен будет храниться под именем *command*. В принципе, этот параметр является необязательным, и если его не указать, то имя переданной команды не попадет в пространство имен, но в этом случае получить имя команды будет затруднительно.

Давайте теперь посмотрим, как работает приведенная выше программа, особенно обратите внимание на содержимое пространства имен:

```
$ python coolprogram.py hello --names Татьяна Александр
Иван

Namespace(command='hello', names=['Татьяна',
'Александр', 'Иван'])
Привет, Татьяна!
Привет, Александр!
Привет, Иван!


$ python coolprogram.py goodbye --count 3

Namespace(command='goodbye', count=3)
Прощай, мир!
Прощай, мир!
Прощай, мир!


$ python coolprogram.py

Namespace(command=None)
Что-то пошло не так...
```

Оформление справки

До сих пор мы старались как можно больше работы возложить на *argparse* и при этом радовались, что эта стандартная библиотека может сама проверять переданные параметры на ошибки, и в случае их возникновения

выдавать нам сообщения о них. Но, честно говоря, эти сообщения по умолчанию не особо информативны для конечного пользователя. К тому же он не всегда может помнить о требуемых параметрах. В этом случае правильный пользователь вызывает справку по программе. Так уж исторически сложилось, что в UNIX-подобных операционных системах для вызова справки по программе помимо знаменитой программы *man* используется специальный параметр командной строки *--help* или *-h*, причем эти справки как правило выглядят примерно одинаково по оформлению. Чтобы было понятно, о чем я говорю, вызовем такую справку для программы *htop*:

```
$ htop -h

htop 1.0.2 - (C) 2004-2011 Hisham Muhammad
Released under the GNU GPL.

-C --no-color               Use a monochrome color scheme
-d --delay=DELAY            Set the delay between updates, in
tenths of seconds
-h --help                   Print this help screen
-s --sort-key=COLUMN        Sort by COLUMN (try --sort-key=help
for a list)
-u --user=USERNAME          Show only processes of a given user
-p --pid=PID,[,PID,PID...]  Show only the given PIDs
-v --version          Print version info

Long options may be passed with a single dash.

Press F1 inside htop for online help.
See 'man htop' for more information.
```

В эту справку входит название программы, ее версия, копирайт, возможно, описание того, что она делает, и список ожидаемых параметров командной строки - обязательные и не очень.

Нам, как программистам, будет приятно узнать, что библиотека *argparse* для любой программы, ее использующей, автоматически создает такую справку, правда, по умолчанию она выглядит не самым впечатляющим образом, но ее можно (нужно) доработать напильником.

Далее мы будем заниматься оформлением нашего последнего скрипта. Даже если мы ничего не будем для этого предпринимать, то справка все-равно будет создана, чтобы ее увидеть запустим скрипт с параметром *--help* (можно и просто *-h*):

```
$ python coolprogram.py --help

usage: coolprogram.py [-h] {hello,goodbye} ...

positional arguments:
  {hello,goodbye}
```

```
optional arguments:
  -h, --help       show this help message and exit
```

Мы также можем получить справку о каждой команде (подпарсере):

```
$ python coolprogram.py hello --help

usage: coolprogram.py hello [-h] [--names NAMES [NAMES ...]]

optional arguments:
  -h, --help            show this help message and exit
  --names NAMES [NAMES ...], -n NAMES [NAMES ...]


$ python coolprogram.py goodbye --help

usage: coolprogram.py goodbye [-h] [-c COUNT]

optional arguments:
  -h, --help            show this help message and exit
  -c COUNT, --count COUNT
```

Для того, чтобы повлиять на внешний вид выводимой справки, используются дополнительные параметры конструкторе *ArgumentParser*, а также в методах *add_argument*, *add_subparsers* и *add_parser*.

Прежде чем браться за код, сведем в один список основные параметры, влияющие на внешний вид справки.

- Чтобы изменить выводимое имя программы, используется параметр *prog* конструктора класса *ArgumentParser*. Если этот параметр не указан, в качестве имени программы берется значение *sys.argv[0]*.
- Чтобы добавить краткое описание программы, используется параметр *description* конструктора класса *ArgumentParser*.
- Чтобы добавить пояснения после списка всех возможных параметров, используется параметр *epilog* конструктора класса *ArgumentParser*.
- Чтобы добавить описание параметра, используется параметр *help* метода *add_argument*.
- Чтобы изменить имя аргумента параметра (не имя самого параметра!), используется параметр *metavar* метода *add_argument*.
- Чтобы добавить описание к группе подпарсеров (мы до сих пор использовали только одну группу подпарсеров, создаваемых с помощью метода *add_subparsers*, но, в принципе, их может быть несколько), используется параметр *description* метода *add_subparsers*.
- Чтобы изменить заголовок группы подпарсеров (по умолчанию используется имя "subcommands"), используется параметр *title* метода *add_subparsers*.
- Чтобы изменить оформление списка возможных команд подпарсера, используется параметр *metavar* метода *add_subparsers*. По умолчанию

список команд выводится в фигурных скобках: {command1, command2, ...}

Это еще не полный список того, как можно изменять внешний вид справки, но это основные моменты, которые понадобятся в первую очередь.

Давайте постепенно изменять нашу программу и смотреть на полученный результат. Сначала добавим описание для программы в целом.

Чтобы не отвлекаться на реализацию отдельных команд, далее приведен только код создания парсера, остальная часть программы осталась неизменной.

```
1.  def createParser ():
2.      parser = argparse.ArgumentParser(
3.              prog = 'coolprogram',
4.              description = '''Это очень полезная программа,
5.  которая позволяет поприветствовать нужных людей,
6.  или попрощаться... со всеми.''',
7.              epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
8.  не несет никакой ответственности ни за что.'''
9.              )
10.     subparsers = parser.add_subparsers (dest='command')
11.
12.     hello_parser = subparsers.add_parser ('hello')
13.     hello_parser.add_argument ('--names', '-n', nargs='+', default=['мир'])
14.
15.     goodbye_parser = subparsers.add_parser ('goodbye')
16.     goodbye_parser.add_argument ('-c', '--count', type=int, default=1)
17.
18.     return parser
```

Посмотрим, как теперь выглядит справка по программе:

```
$ python coolprogram.py -h
usage: coolprogram [-h] {hello,goodbye} ...

Это очень нужная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.

positional arguments:
  {hello,goodbye}

optional arguments:
  -h, --help       show this help message and exit

(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.
```

Оставим в стороне философский вопрос о том, на каком языке писать справку по программе, для простоты и наглядности мы будем везде использовать русский язык. Еще можно обратить внимание на то, что при выводе описаний были проигнорированы все переносы строк внутри многострочной конструкции '"..."', так что можно смело разбивать описания на несколько строк, а не писать их в одну строку в тысячу сто символов.

Теперь добавим описания команд и их параметров.

```
1.  def createParser ():
2.      parser = argparse.ArgumentParser(
3.              prog = 'coolprogram',
4.              description = '"Это очень полезная программа,
5.  которая позволяет поприветствовать нужных людей,
6.  или попрощаться... со всеми."',
7.              epilog = '"(c) Jenyay 2014. Автор программы, как всегда,
8.  не несет никакой ответственности ни за что."'
9.              )
10.     subparsers = parser.add_subparsers (dest = 'command',
11.             title = 'Возможные команды',
12.             description = 'Команды, которые должны быть в качестве первого параметра
    %(prog)s')
13.
14.     hello_parser = subparsers.add_parser ('hello')
15.     hello_parser.add_argument ('--names', '-n', nargs='+', default=['мир'],
16.             help = 'Список приветствуемых людей')
17.
18.     goodbye_parser = subparsers.add_parser ('goodbye')
19.     goodbye_parser.add_argument ('-c', '--count', type=int, default=1,
20.             help = 'Сколько раз попрощаться с миром')
21.
22.     return parser
```

Теперь справка по программе и по ее командам выглядит следующим образом:

```
$ python coolprogram.py -h
usage: coolprogram [-h] {hello,goodbye} ...

Это очень полезная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.

optional arguments:
  -h, --help       show this help message and exit

Возможные команды:
  Команды, которые должны быть в качестве первого параметра
coolprogram

  {hello,goodbye}
```

```
(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.

$ python coolprogram.py hello -h
usage: coolprogram hello [-h] [--names NAMES [NAMES ...]]

optional arguments:
  -h, --help            show this help message and exit
  --names NAMES [NAMES ...], -n NAMES [NAMES ...]
                        Список приветствуемых людей

$ python coolprogram.py goodbye -h
usage: coolprogram goodbye [-h] [-c COUNT]

optional arguments:
  -h, --help            show this help message and exit
  -c COUNT, --count COUNT
                        Сколько раз попрощаться с миром
```

В коде выше мы использовали специальный символ подстановки *%(prog)s*, который заменяется в тексте на имя программы, указанное с помощью параметра *prog* конструктора *ArgumentParser*.

Теперь обратим свой взор на имена значений параметров. По умолчанию они совпадают с именем параметра, но написаны заглавными буквами. В данной программе еще можно понять, что под этими названиями подразумевается, но лучше дать им более осмысленные имена. Для этого воспользуемся параметром *metavar*.

```
1.  def createParser ():
2.      parser = argparse.ArgumentParser(
3.              prog = 'coolprogram',
4.              description = '''Это очень полезная программа,
5.  которая позволяет поприветствовать нужных людей,
6.  или попрощаться... со всеми.''',
7.              epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
8.  не несет никакой ответственности ни за что.'''
9.              )
10.     subparsers = parser.add_subparsers (dest = 'command',
11.             title = 'Возможные команды',
12.             description = 'Команды, которые должны быть в качестве первого параметра
        %(prog)s')
13.
14.     hello_parser = subparsers.add_parser ('hello')
15.     hello_parser.add_argument ('--names', '-n', nargs='+', default=['мир'],
16.             help = 'Список приветствуемых людей',
17.             metavar = 'ИМЯ')
18.
19.     goodbye_parser = subparsers.add_parser ('goodbye')
20.     goodbye_parser.add_argument ('-c', '--count', type=int, default=1,
21.             help = 'Сколько раз попрощаться с миром',
```

```
22.         metavar = 'КОЛИЧЕСТВО')
23.
24.     return parser
```

Теперь справка стала чуть более симпатичной:

```
$ python coolprogram.py hello -h

usage: coolprogram hello [-h] [--names ИМЯ [ИМЯ ...]]

optional arguments:
  -h, --help            show this help message and exit
  --names ИМЯ [ИМЯ ...], -n ИМЯ [ИМЯ ...]
                        Список приветствуемых людей

$ python coolprogram.py goodbye -h

usage: coolprogram goodbye [-h] [-c КОЛИЧЕСТВО]

optional arguments:
  -h, --help            show this help message and exit
  -c КОЛИЧЕСТВО, --count КОЛИЧЕСТВО
                        Сколько раз попрощаться с миром
```

Однако, что-то справку все-таки портит. Мы практически все русифицировали, остались непереведенными только фразы "usage" и "optional arguments". Займемся сейчас второй фразой.

Ее перевести будет уже чуть сложнее, но не сильно. На самом деле все параметры по умолчанию не только принадлежат какому-то парсеру, но и могут объединяться по группам. Если группы явно не заданы, то параметры попадают в одну из групп: "positional arguments" или "optional arguments". Но мы можем создать и свою группу с помощью метода *add_argument_group*. Этот метод возвращает экземпляр класса группы (экземпляр внутреннего класса *_ArgumentGroup*), после чего мы должны, как и прежде, вызывать метод *add_argument* для добавления параметров, но уже не парсера, а группы. Это показано в следующем примере:

```
1.  def createParser ():
2.      parser = argparse.ArgumentParser(
3.              prog = 'coolprogram',
4.              description = '''Это очень полезная программа,
5.  которая позволяет поприветствовать нужных людей,
6.  или попрощаться... со всеми.''',
7.              epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
8.  не несет никакой ответственности ни за что.'''
9.              )
10.     subparsers = parser.add_subparsers (dest = 'command',
11.             title = 'Возможные команды',
12.             description = 'Команды, которые должны быть в качестве первого параметра
    %(prog)s')
```

```
13.
14.    hello_parser = subparsers.add_parser ('hello')
15.    hello_group = hello_parser.add_argument_group (title='Параметры')
16.
17.    hello_group.add_argument ('--names', '-n', nargs='+', default=['мир'],
18.          help = 'Список приветствуемых людей',
19.          metavar = 'ИМЯ')
20.
21.    goodbye_parser = subparsers.add_parser ('goodbye')
22.    goodbye_group = goodbye_parser.add_argument_group (title='Параметры')
23.
24.    goodbye_group.add_argument ('-c', '--count', type=int, default=1,
25.          help = 'Сколько раз попрощаться с миром',
26.          metavar = 'КОЛИЧЕСТВО')
27.
28.    return parser
```

Теперь справка по параметрам будет выглядеть следующим образом:

```
$ python coolprogram.py hello -h

usage: coolprogram hello [-h] [--names ИМЯ [ИМЯ ...]]

optional arguments:
  -h, --help            show this help message and exit

Параметры:
  --names ИМЯ [ИМЯ ...], -n ИМЯ [ИМЯ ...]
                        Список приветствуемых людей
```

Уже лучше, однако группа "optional arguments" никуда не делась, поскольку параметр *--help* / *-h* был добавлен библиотекой еще до того, как мы создали группу. Выход из этой ситуации простой. Во-первых, надо сказать классу *ArgumentParser*, что нам его встроенная справка не нужна, для этого в конструктор передадим еще один параметр *add_help=False*. Также надо не забыть добавить параметр *add_help = False* в метод *add_parser* для каждого подпарсера. Затем добавим параметр *--help* / *-h* вручную. Самим нам оформлять справку не придется, поскольку при добавлении этого параметра мы передадим в метод *add_argument* параметр *action='help*', который указывает, что при получении этого параметра нужно отобразить справку.

Теперь наш код создания парсера выглядит следующим образом:

```
1. def createParser ():
2.     # Создаем класс парсера
3.     parser = argparse.ArgumentParser(
4.           prog = 'coolprogram',
5.           description = '''Это очень полезная программа,
6. которая позволяет поприветствовать нужных людей,
7. или попрощаться... со всеми.''',
8.           epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
```

```
    не несет никакой ответственности ни за что.'",
            add_help = False
            )

    # Создаем группу параметров для родительского парсера,
    # ведь у него тоже должен быть параметр --help / -h
    parent_group = parser.add_argument_group (title='Параметры')

    parent_group.add_argument ('--help', '-h', action='help', help='Справка')


    # Создаем группу подпарсеров
    subparsers = parser.add_subparsers (dest = 'command',
            title = 'Возможные команды',
            description = 'Команды, которые должны быть в качестве первого параметра
        %(prog)s')

    # Создаем парсер для команды hello
    hello_parser = subparsers.add_parser ('hello',
            add_help = False)

    # Создаем новую группу параметров
    hello_group = hello_parser.add_argument_group (title='Параметры')

    # Добавляем параметры
    hello_group.add_argument ('--names', '-n', nargs='+', default=['мир'],
            help = 'Список приветствуемых людей',
            metavar = 'ИМЯ')

    hello_group.add_argument ('--help', '-h', action='help', help='Справка')


    # Создаем парсер для команды goodbye
    goodbye_parser = subparsers.add_parser ('goodbye', add_help = False)

    # Создаем новую группу параметров
    goodbye_group = goodbye_parser.add_argument_group (title='Параметры')

    # Добавляем параметры
    goodbye_group.add_argument ('-c', '--count', type=int, default=1,
            help = 'Сколько раз попрощаться с миром',
            metavar = 'КОЛИЧЕСТВО')

    goodbye_group.add_argument ('--help', '-h', action='help', help='Справка')

    return parser
```

Выглядит почти идеально:

```
$ python coolprogram.py hello -h

usage: coolprogram hello [--names ИМЯ [ИМЯ ...]] [--help]
```

```
Параметры:
  --names ИМЯ [ИМЯ ...], -n ИМЯ [ИМЯ ...]
                        Список приветствуемых людей
  --help, -h            Справка

$ py3 coolprogram.py goodbye -h

usage: coolprogram goodbye [-c КОЛИЧЕСТВО] [--help]

Параметры:
  -c КОЛИЧЕСТВО, --count КОЛИЧЕСТВО
                        Сколько раз попрощаться с миром
  --help, -h            Справка
```

Не забыли мы и про справку для корневого парсера:

```
$ python coolprogram.py -h
usage: coolprogram [--help] {hello,goodbye} ...

Это очень полезная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.

Параметры:
  --help, -h       Справка

Возможные команды:
  Команды, которые должны быть в качестве первого параметра
coolprogram

  {hello,goodbye}

(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.
```

Осталось сделать самую малость - добавить описания для команд. Для этого воспользуемся параметрами *help* и *description* метода *add_parser*.

```
1.  def createParser ():
2.      # Создаем класс парсера
3.      parser = argparse.ArgumentParser(
4.              prog = 'coolprogram',
5.              description = '''Это очень полезная программа,
6.  которая позволяет поприветствовать нужных людей,
7.  или попрощаться... со всеми.''',
8.              epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
9.  не несет никакой ответственности ни за что.''',
10.             add_help = False
11.             )
12.
13.     # Создаем группу параметров для родительского парсера,
```

```
    # ведь у него тоже должен быть параметр --help / -h
    parent_group = parser.add_argument_group (title='Параметры')

    parent_group.add_argument ('--help', '-h', action='help', help='Справка')


    # Создаем группу подпарсеров
    subparsers = parser.add_subparsers (dest = 'command',
        title = 'Возможные команды',
        description = 'Команды, которые должны быть в качестве первого параметра %(prog)s')

    # Создаем парсер для команды hello
    hello_parser = subparsers.add_parser ('hello',
        add_help = False,
        help = 'Запуск в режиме "Hello, world!"',
        description = '"Запуск в режиме "Hello, world!".
В этом режиме программа приветствует список людей, переданных в качестве параметра."')

    # Создаем новую группу параметров
    hello_group = hello_parser.add_argument_group (title='Параметры')

    # Добавляем параметры
    hello_group.add_argument ('--names', '-n', nargs='+', default=['мир'],
        help = 'Список приветствуемых людей',
        metavar = 'ИМЯ')

    hello_group.add_argument ('--help', '-h', action='help', help='Справка')


    # Создаем парсер для команды goodbye
    goodbye_parser = subparsers.add_parser ('goodbye',
        add_help = False,
        help = 'Запуск в режиме "Goodbye, world!"',
        description = '"В этом режиме программа прощается с миром."')

    # Создаем новую группу параметров
    goodbye_group = goodbye_parser.add_argument_group (title='Параметры')

    # Добавляем параметры
    goodbye_group.add_argument ('-c', '--count', type=int, default=1,
        help = 'Сколько раз попрощаться с миром',
        metavar = 'КОЛИЧЕСТВО')

    goodbye_group.add_argument ('--help', '-h', action='help', help='Справка')

    return parser
```

Теперь справка выглядит полностью законченной:

```
$ python coolprogram.py -h

usage: coolprogram [--help] {hello,goodbye} ...

Это очень полезная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.

Параметры:
  --help, -h        Справка

Возможные команды:
  Команды, которые должны быть в качестве первого параметра
coolprogram

  {hello,goodbye}
    hello           Запуск в режиме "Hello, world!"
    goodbye         Запуск в режиме "Goodbye, world!"

(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.

$ python coolprogram.py hello -h

usage: coolprogram hello [--names ИМЯ [ИМЯ ...]] [--help]

Запуск в режиме "Hello, world!". В этом режиме программа
приветствует список
людей, переданных в качестве параметра.

Параметры:
  --names ИМЯ [ИМЯ ...], -n ИМЯ [ИМЯ ...]
                        Список приветствуемых людей
  --help, -h            Справка

$ python coolprogram.py goodbye -h

usage: coolprogram goodbye [-c КОЛИЧЕСТВО] [--help]

В этом режиме программа прощается с миром.

Параметры:
  -c КОЛИЧЕСТВО, --count КОЛИЧЕСТВО
                        Сколько раз попрощаться с миром
  --help, -h            Справка
```

И чтобы окончательно навести порядок, не мешало бы добавить еще один параметр в родительский парсер, а именно параметр *--version* для показа текущей версии программы. Это легко реализуется благодаря параметру *action='version'* метода *add_argument*.

```
1.  #!/usr/bin/python
2.  # -*- coding: UTF-8 -*-
3.
4.  import sys
5.  import argparse
6.
7.  version = "1.2.1"
8.
9.
10. def createParser ():
11.     # Создаем класс парсера
12.     parser = argparse.ArgumentParser(
13.             prog = 'coolprogram',
14.             description = '''Это очень полезная программа,
15. которая позволяет поприветствовать нужных людей,
16. или попрощаться... со всеми.''',
17.             epilog = '''(c) Jenyay 2014. Автор программы, как всегда,
18. не несет никакой ответственности ни за что.''',
19.             add_help = False
20.             )
21.
22.     # Создаем группу параметров для родительского парсера,
23.     # ведь у него тоже должен быть параметр --help / -h
24.     parent_group = parser.add_argument_group (title='Параметры')
25.
26.     parent_group.add_argument ('--help', '-h', action='help', help='Справка')
27.
28.     parent_group.add_argument ('--version',
29.             action='version',
30.             help = 'Вывести номер версии',
31.             version='%(prog)s {}'.format (version))
```

Для наглядности номер версии будет храниться в глобальной переменной *version*, а воспользуемся мы ей в строках 28-31, где опять используется символ подстановки *%(prog)s*, который, напомню, обозначает имя программы, после которого мы добавим номер версии. Воспользуемся этим параметром:

```
$ pythono coolprogram.py --version

coolprogram 1.2.1

А справка по параметрам теперь выглядит следующим образом:
$ python coolprogram.py -h

usage: coolprogram [--help] [--version] {hello,goodbye} ...

Это очень полезная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.

Параметры:
```

```
  --help, -h        Справка
  --version         Вывести номер версии

Возможные команды:
  Команды, которые должны быть в качестве первого параметра
coolprogram

  {hello,goodbye}
    hello           Запуск в режиме "Hello, world!"
    goodbye         Запуск в режиме "Goodbye, world!"

(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.
```

И последнее, что нам осталось научиться делать со справкой - это принудительно ее выводить, например, при вводе пользователем каких-то неправильных комбинаций параметров (в том числе, когда он вводит взаимоисключающие параметры).

Сейчас у нас в случае, если пользователь не вводит никакую команду, выводится текст: "Что-то пошло не так...", но буквально одной строкой мы изменим это поведение таким образом, чтобы в этом случае выводилась справка. Для этого достаточно вызвать метод *print_help* класса *ArgumentParser*:

```
1.  if __name__ == '__main__':
2.      parser = createParser()
3.      namespace = parser.parse_args(sys.argv[1:])
4.
5.      print (namespace)
6.
7.      if namespace.command == "hello":
8.          run_hello (namespace)
9.      elif namespace.command == "goodbye":
10.         run_goodbye (namespace)
11.     else:
12.         parser.print_help()
```

В остальном програма осталась без изменений. Попробуем запустить ее без параметров:

```
$ python coolprogram.py

Namespace(command=None)
usage: coolprogram [--help] [--version] {hello,goodbye} ...

Это очень полезная программа, которая позволяет поприветствовать
нужных людей,
или попрощаться... со всеми.
```

```
Параметры:
  --help, -h        Справка
  --version         Вывести номер версии

Возможные команды:
  Команды, которые должны быть в качестве первого параметра
coolprogram

  {hello,goodbye}
    hello           Запуск в режиме "Hello, world!"
    goodbye         Запуск в режиме "Goodbye, world!"

(c) Jenyay 2014. Автор программы, как всегда, не несет никакой
ответственности
ни за что.
```

Как видите, оформление справки занимает довольно много места, но по сути в этом нет ничего сложного, нужно просто заполнить нужные параметры.

Заключение

В этой статье мы рассмотрели основные возможности стандартной Python-библиотеки *argparse*, посмотрели, как она упрощает жизнь по сравнению с ручным разбором параметров, научились создавать именованные и позиционные параметры, рассмотрели различные варианты их использования, в том числе в виде флагов, а также научились создавать команды и оформлять справку.

Несмотря на большой объем статьи, некоторые вопросы все-равно остались "за бортом". Например, мы не рассмотрели "склеивание" параметров (когда вместо git add -a -m "..." вы пишете просто git -am "..."). Проигнорировали классы, позволяющие еще более углубиться в оформление справки, не рассмотрели некоторые возможности параметра *action* метода *add_argument*, а также не поговорили о том, что на самом деле можно изменить формат параметров вида "-h" на формат вида "/h", принятый в Windows. Но все эти вопросы описаны в справке, в которой только надо найти описание нужной возможности.

Не мешало бы еще рассмотреть альтернативные нестандартные библиотеки для разбора командной строки, но это уже тема для отдельной статьи.

---

argparse - зачем такие сложности

argparse - зачем такие сложности, проще разобрать аргументы вручную sys.argv, чем использовать этот argparse:

```
import sys
```

```
for i in sys.argv:
    a=i.split("=")
    if a[0]=='--arg':
        print a[1]
```