

Модуль argparse

argparse - это модуль для обработки аргументов командной строки. Примеры того, что позволяет делать модуль:

- создавать аргументы и опции, с которыми может вызываться скрипт
- указывать типы аргументов, значения по умолчанию
- указывать, какие действия соответствуют аргументам
- выполнять вызов функции при указании аргумента
- отображать сообщения с подсказками по использованию скрипта

argparse не единственный модуль для обработки аргументов командной строки. И даже не единственный такой модуль в стандартной библиотеке.

В книге рассматривается только argparse, но кроме него стоит обратить внимание на те модули, которые не входят в стандартную библиотеку Python.

Например, [click](#).

Примечание

[Хорошая статья](#), которая сравнивает разные модули обработки аргументов командной строки (рассматриваются argparse, click и docopt).

Пример скрипта ping_function.py:

```
import subprocess
import argparse

def ping_ip(ip_address, count):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding="utf-8",
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout + reply.stderr

parser = argparse.ArgumentParser(description="Ping script")
```

```

parser.add_argument("-a", dest="ip", required=True)
parser.add_argument("-c", dest="count", default=2, type=int)

args = parser.parse_args()
print(args)

rc, message = ping_ip(args.ip, args.count)
print(message)

```

Создание парсера:

- `parser = argparse.ArgumentParser(description='Ping script')`

Добавление аргументов:

- `parser.add_argument('-a', dest="ip")`
 - аргумент, который передается после опции `-a`, сохранится в переменную `ip`
- `parser.add_argument('-c', dest="count", default=2, type=int)`
 - аргумент, который передается после опции `-c`, будет сохранен в переменную `count`, но прежде будет конвертирован в число. Если аргумент не был указан, по умолчанию будет значение 2

Строка `args = parser.parse_args()` указывается после того, как определены все аргументы. После её выполнения в переменной `args` содержатся все аргументы, которые были переданы скрипту. К ним можно обращаться, используя синтаксис `args.ip`.

Попробуем вызвать скрипт с разными аргументами. Если переданы оба аргумента:

```

$ python ping_function.py -a 8.8.8.8 -c 5
Namespace(count=5, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.673 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.902 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=48 time=48.696 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=48 time=50.040 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=48 time=48.831 ms

--- 8.8.8.8 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.673/49.228/50.040/0.610 ms

Namespace - это объект, который возвращает метод parse\_args()

```

Передаем только IP-адрес:

```

$ python ping_function.py -a 8.8.8.8
Namespace(count=2, ip='8.8.8.8')
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=48.563 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=49.616 ms

```

```
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 48.563/49.090/49.616/0.526 ms
```

Вызов скрипта без аргументов:

```
$ python ping_function.py
Namespace(count=2, ip=None)
Traceback (most recent call last):
  File "ping_function.py", line 31, in <module>
    rc, message = ping_ip( args.ip, args.count )
  File "ping_function.py", line 16, in ping_ip
    stderr=temp)
  File "/usr/local/lib/python3.6/subprocess.py", line 336, in check_output
    **kwargs).stdout
  File "/usr/local/lib/python3.6/subprocess.py", line 403, in run
    with Popen(*popenargs, **kwargs) as process:
  File "/usr/local/lib/python3.6/subprocess.py", line 707, in __init__
    restore_signals, start_new_session)
  File "/usr/local/lib/python3.6/subprocess.py", line 1260, in _execute_child
    restore_signals, start_new_session, preexec_fn)
TypeError: expected str, bytes or os.PathLike object, not NoneType
```

Если бы функция была вызвана без аргументов, когда не используется `argparse`, возникла бы ошибка, что не все аргументы указаны.

Из-за `argparse`, фактически аргумент передается, только он равен `None`. Это видно в строке `Namespace(count=2, ip=None)`.

В таком скрипте IP-адрес необходимо указывать всегда. И в `argparse` можно указать, что аргумент является обязательным. Для этого надо изменить опцию `-a`: добавить в конце `required=True`:

```
parser.add_argument('-a', dest="ip", required=True)
```

Теперь, если вызвать скрипт без аргументов, вывод будет таким:

```
$ python ping_function.py
usage: ping_function.py [-h] -a IP [-c COUNT]
ping_function.py: error: the following arguments are required: -a
```

Теперь отображается понятное сообщение, что надо указать обязательный аргумент, и подсказка `usage`.

Также, благодаря `argparse`, доступен `help`:

```
$ python ping_function.py -h
usage: ping_function.py [-h] -a IP [-c COUNT]

Ping script

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

Обратите внимание, что в сообщении все опции находятся в секции `optional arguments`. `argparse` сам определяет, что указаны опции, так как они начинаются с `-` и в имени только одна буква.

Зададим IP-адрес как позиционный аргумент. Файл `ping_function_ver2.py`:

```
import subprocess
import argparse

def ping_ip(ip_address, count):
    """
    Ping IP address and return tuple:
    On success: (return code = 0, command output)
    On failure: (return code, error output (stderr))
    """
    reply = subprocess.run(
        f"ping -c {count} -n {ip_address}",
        shell=True,
        stdout=subprocess.PIPE,
        stderr=subprocess.PIPE,
        encoding="utf-8",
    )
    if reply.returncode == 0:
        return True, reply.stdout
    else:
        return False, reply.stdout + reply.stderr

parser = argparse.ArgumentParser(description="Ping script")

parser.add_argument("host", help="IP or name to ping")
parser.add_argument("-c", dest="count", default=2, type=int, help="Number of packets")

args = parser.parse_args()
print(args)

rc, message = ping_ip(args.host, args.count)
print(message)
```

Теперь вместо указания опции `-a`, можно просто передать IP-адрес. Он будет автоматически сохранен в переменной `host`. И автоматически считается обязательным. То есть, теперь не нужно указывать `required=True` и `dest="ip"`.

Кроме того, в скрипте указаны сообщения, которые будут выводиться при вызове `help`. Теперь вызов скрипта выглядит так:

```
$ python ping_function_ver2.py 8.8.8.8 -c 2
Namespace(host='8.8.8.8', count=2)
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=48 time=49.203 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=48 time=51.764 ms

--- 8.8.8.8 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 49.203/50.484/51.764/1.280 ms
```

А сообщение `help` так:

```
$ python ping_function_ver2.py -h
usage: ping_function_ver2.py [-h] [-c COUNT] host
```

Ping script

positional arguments:

host IP or name to ping

optional arguments:

-h, --help show this help message and exit

-c COUNT Number of packets

Вложенные парсеры

Рассмотрим один из способов организации более сложной иерархии аргументов.

Примечание

Этот пример покажет больше возможностей argparse, но они этим не ограничиваются, поэтому, если вы будете использовать argparse, обязательно посмотрите [документацию модуля](#) или [статью на PyMOTW](#).

Файл parse_dhcp_snooping.py:

```
# -*- coding: utf-8 -*-
import argparse

# Default values:
DFLT_DB_NAME = 'dhcp_snooping.db'
DFLT_DB_SCHEMA = 'dhcp_snooping_schema.sql'

def create(args):
    print(f"Creating DB {args.name} with DB schema {args.schema}")

def add(args):
    if args.sw_true:
        print("Adding switch data to database")
    else:
        print(f"Reading info from file(s) \n{' '.join(args.filename)}")
        print(f"\nAdding data to db {args.db_file}")

def get(args):
    if args.key and args.value:
        print(f"Geting data from DB: {args.db_file}")
        print(f"Request data for host(s) with {args.key} {args.value}")
    elif args.key or args.value:
        print("Please give two or zero args\n")
        print(show_subparser_help('get'))
    else:
        print(f"Showing {args.db_file} content...")

parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(title='subcommands',
                                   description='valid subcommands',
                                   help='description')

create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
```

```

create_parser.set_defaults(func=create)

add_parser = subparsers.add_parser('add', help='add data to db')
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
add_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db name')
add_parser.add_argument('-s', dest='sw_true', action='store_true',
                        help='add switch data if set, else add normal data')
add_parser.set_defaults(func=add)

get_parser = subparsers.add_parser('get', help='get data from db')
get_parser.add_argument('--db', dest='db_file', default=DFLT_DB_NAME, help='db name')
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
get_parser.add_argument('-v', dest="value", help='value of key')
get_parser.add_argument('-a', action='store_true', help='show db content')
get_parser.set_defaults(func=get)

if __name__ == '__main__':
    args = parser.parse_args()
    if not vars(args):
        parser.print_usage()
    else:
        args.func(args)

```

Теперь создается не только парсер, как в прошлом примере, но и вложенные парсеры. Вложенные парсеры будут отображаться как команды. Фактически, они будут использоваться как обязательные аргументы.

С помощью вложенных парсеров создается иерархия аргументов и опций. Аргументы, которые добавлены во вложенный парсер, будут доступны как аргументы этого парсера. Например, в этой части создан вложенный парсер `create_db`, и к нему добавлена опция `-n`:

```

create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', dest='name', default=DFLT_DB_NAME,
                        help='db filename')

```

Синтаксис создания вложенных парсеров и добавления к ним аргументов одинаков:

```

create_parser = subparsers.add_parser('create_db', help='create new db')
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                        default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                        help='db schema filename')
create_parser.set_defaults(func=create)

```

Метод `add_argument` добавляет аргумент. Тут синтаксис точно такой же, как и без использования вложенных парсеров.

В строке `create_parser.set_defaults(func=create)` указывается, что при вызове парсера `create_parser` будет вызвана функция `create`.

Функция `create` получает как аргумент все аргументы, которые были переданы. И внутри функции можно обращаться к нужным:

```
def create(args):
    print("Creating DB {} with DB schema {}".format((args.name, args.schema)))
```

Если вызвать `help` для этого скрипта, вывод будет таким:

```
$ python parse_dhcp_snooping.py -h
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...
```

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

```
subcommands:
  valid subcommands
```

{create_db,add,get}	description
create_db	create new db
add	add data to db
get	get data from db

Обратите внимание, что каждый вложенный парсер, который создан в скрипте, отображается как команда в подсказке `usage`:

```
usage: parse_dhcp_snooping.py [-h] {create_db,add,get} ...
```

У каждого вложенного парсера теперь есть свой `help`:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]
```

```
optional arguments:
  -h, --help            show this help message and exit
  -n db-filename        db filename
  -s SCHEMA             db schema filename
```

Кроме вложенных парсеров, в этом примере также есть несколько новых возможностей `argparse`.

metavar

В парсере `create_parser` используется новый аргумент - `metavar`:

```
create_parser.add_argument('-n', metavar='db-filename', dest='name',
                           default=DFLT_DB_NAME, help='db filename')
create_parser.add_argument('-s', dest='schema', default=DFLT_DB_SCHEMA,
                           help='db schema filename')
```

Аргумент `metavar` позволяет указывать имя аргумента для вывода в сообщении `usage` и `help`:

```
$ python parse_dhcp_snooping.py create_db -h
usage: parse_dhcp_snooping.py create_db [-h] [-n db-filename] [-s SCHEMA]
```

```
optional arguments:
  -h, --help            show this help message and exit
  -n db-filename        db filename
  -s SCHEMA             db schema filename
```

Посмотрите на разницу между опциями `-n` и `-s`:

- после опции `-n` и в `usage`, и в `help` указывается имя, которое указано в параметре `metavar`
- после опции `-s` указывается имя переменной, в которую сохраняется значение

`nargs`

В парсере `add_parser` используется `nargs`:

```
add_parser.add_argument('filename', nargs='+', help='file(s) to add to db')
```

Параметр `nargs` позволяет указать, что в этот аргумент должно попасть определенное количество элементов. В этом случае все аргументы, которые были переданы скрипту после имени аргумента `filename`, попадут в список `nargs`, но должен быть передан хотя бы один аргумент.

Сообщение `help` в таком случае выглядит так:

```
$ python parse_dhcp_snooping.py add -h
usage: parse_dhcp_snooping.py add [-h] [--db DB_FILE] [-s]
                                filename [filename ...]

positional arguments:
  filename              file(s) to add to db

optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          db name
  -s                    add switch data if set, else add normal data
```

Если передать несколько файлов, они попадут в список. А так как функция `add` просто выводит имена файлов, вывод получится таким:

```
$ python parse_dhcp_snooping.py add filename test1.txt test2.txt
Reading info from file(s)
filename, test1.txt, test2.txt
```

```
Adding data to db dhcp_snooping.db
```

`nargs` поддерживает такие значения:

- `N` - должно быть указанное количество аргументов. Аргументы будут в списке (даже если указан 1)
- `?` - 0 или 1 аргумент
- `*` - все аргументы попадут в список
- `+` - все аргументы попадут в список, но должен быть передан хотя бы один аргумент

`choices`

В парсере `get_parser` используется `choices`:

```
get_parser.add_argument('-k', dest="key",
                        choices=['mac', 'ip', 'vlan', 'interface', 'switch'],
                        help='host key (parameter) to search')
```

Для некоторых аргументов важно, чтобы значение было выбрано только из определенных вариантов. В таких случаях можно указывать `choices`.

Для этого парсера `help` выглядит так:

```
$ python parse_dhcp_snooping.py get -h
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                [-k {mac,ip,vlan,interface,switch}]
                                [-v VALUE] [-a]
```

```
optional arguments:
  -h, --help            show this help message and exit
  --db DB_FILE          db name
  -k {mac,ip,vlan,interface,switch}
                        host key (parameter) to search
  -v VALUE              value of key
  -a                    show db content
```

А если выбрать неправильный вариант:

```
$ python parse_dhcp_snooping.py get -k test
usage: parse_dhcp_snooping.py get [-h] [--db DB_FILE]
                                [-k {mac,ip,vlan,interface,switch}]
                                [-v VALUE] [-a]

parse_dhcp_snooping.py get: error: argument -k: invalid choice: 'test' (choose from 'mac', 'ip', 'vlan', 'interface', 'switch')
```

В данном примере важно указать варианты на выбор, так как затем на основании выбранного варианта генерируется SQL-запрос. И, благодаря ``choices``, нет возможности указать какой-то параметр, кроме разрешенных.

Импорт парсера

В файле `parse_dhcp_snooping.py` последние две строки будут выполняться только в том случае, если скрипт был вызван как основной.

```
if __name__ == '__main__':
    args = parser.parse_args()
    args.func(args)
```

А значит, если импортировать файл, эти строки не будут вызваны.

Попробуем импортировать парсер в другой файл (файл `call_pds.py`):

```
from parse_dhcp_snooping import parser

args = parser.parse_args()
args.func(args)
```

Вызов сообщения `help`:

```
$ python call_pds.py -h
usage: call_pds.py [-h] {create_db,add,get} ...
```

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

subcommands:
  valid subcommands

  {create_db,add,get}  description
  create_db           create new db
  add                 add data to db
  get                 get data from db
```

Вызов аргумента:

```
$ python call_pds.py add test.txt test2.txt
Reading info from file(s)
test.txt, test2.txt
```

```
Adding data to db dhcp_snooping.db
```

Всё работает без проблем.

Передача аргументов вручную

Последняя особенность argparse - возможность передавать аргументы вручную.

Аргументы можно передать как список при вызове метода `parse_args()` (файл `call_pds2.py`):

```
from parse_dhcp_snooping import parser, get

args = parser.parse_args('add test.txt test2.txt'.split())
args.func(args)
```

Необходимо использовать метод `split()`, так как метод `parse_args()` ожидает список аргументов.

Результат будет таким же, как если бы скрипт был вызван с аргументами:

```
$ python call_pds2.py
Reading info from file(s)
test.txt, test2.txt
```

```
Adding data to db dhcp_snooping.db
```

[Next](#) [Previous](#)

© 2015-2020, [Natasha Samoylenko](#). Обновлено: февр. 23, 2021. [Скачать книгу](#).

Built with [Sphinx](#) using a [theme](#) provided by [Read the Docs](#).

This work is licensed under a [CC-BY-SA-4.0](#).

Разбор параметров командной строки в Python

Содержание

- [Введение](#)
- [Примеры без использования argparse](#)
- [Использование библиотеки argparse](#)
 - [Первый пример](#)
 - [Добавляем именованные параметры](#)
 - [Параметры как списки](#)
 - [Выбор из вариантов](#)
 - [Указание типов параметров](#)
 - [Имена файлов как параметры](#)
 - [Обязательные именованные параметры](#)
 - [Параметры как флаги](#)
 - [Использование нескольких параметров](#)
 - [Использование подпарсеров](#)
 - [Оформление справки](#)
- [Заключение](#)
- [Комментарии](#)

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 брезгают 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.

В дальнейшем все примеры я буду приводить с расчетом на Python 3.3, но практически все то же самое можно использовать и в Python 2.7, хотя для запуска на Python 2.7 примеры придется исправить, в основном добавить символ "u" перед началом строк.

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!")
```

[Исходник](#)

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
```

[Исходник](#)

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)
```

[Исходник](#)

и мы запускаем его с помощью команды

```
python params.py
```

[Исходник](#)

то в консоль будет выведена единственная строка:

```
params.py
```

[Исходник](#)

Если же мы добавим несколько параметров,

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

[Исходник](#)

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

```
params.py
param1
```

```
param2  
param3
```

[Исходник](#)

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!")
```

[Исходник](#)

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

[Исходник](#)

то результат будет прежний
Привет, мир!

[Исходник](#)

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

[Исходник](#)

то программа поприветствует некоего Васю:
Привет, Вася!

[Исходник](#)

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)

```

[Исходник](#)

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]
```

[Исходник](#)

Здесь [Имя] является необязательным параметром.

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

```
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='?')
11.
12.     return parser
13.
14.
15. if __name__ == '__main__':
16.     parser = createParser()
17.     namespace = parser.parse_args()
18.
19.     # print (namespace)
20.
21.     if namespace.name:
22.         print ("Привет, {}".format (namespace.name) )
23.     else:
24.         print ("Привет, мир!")
```

[Исходник](#)

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:])
```

[Исходник](#)

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
```

[Исходник](#)

, то увидим его в пространстве имен.

```
Namespace (name='Вася')
```

[Исходник](#)

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

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

[Исходник](#)

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) )
```

[Исходник](#)

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 *ruparse* 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) )

```

[Исходник](#)

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
```

[Исходник](#)

или

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

[Исходник](#)

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) )

```

[Исходник](#)

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

Привет, None!

[Исходник](#)

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 '-'

```

[Исходник](#)

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 значения параметра из пространства имен осуществляется по длинному имени:

```

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

```

[Исходник](#)

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) )

```

[Исходник](#)

При этом пространство имен будет выглядеть как:

```
Namespace(n='Вася')
```

[Исходник](#)

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) )

```

[Исходник](#)

Теперь мы можем использовать три варианта передачи параметров:

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

[Исходник](#)

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')
```

[Исходник](#)

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) )
```

[Исходник](#)

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"
```

[Исходник](#)

Поскольку, если написать просто

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

[Исходник](#)

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

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

[Исходник](#)

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) )
```

[Исходник](#)

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
```

[Исходник](#)

То программа нам выведет:

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

[Исходник](#)

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) )
```

[Исходник](#)

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')
```

[Исходник](#)

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!" (времененно 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!")

```

[Исходник](#)

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!

```

[Исходник](#)

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'

```

[Исходник](#)

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)
```

[Исходник](#)

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

[Исходник](#)

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'

```

[Исходник](#)

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)

```

[Исходник](#)

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 буферизированное 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)
```

[Исходник](#)

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'
```

[Исходник](#)

Аналогично мы можем принимать имя файла для записи, при этом только надо не забыть передать в функцию *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) )
```

[Исходник](#)

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

[Исходник](#)

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) )
```

[Исходник](#)

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

Namespace(name=None)

Привет, None!

[Исходник](#)

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) )

```

[Исходник](#)

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
```

[Исходник](#)

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!
```

[Исходник](#)

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!")
```

[Исходник](#)

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!
```

[Исходник](#)

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!")
```

[Исходник](#)

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!
```

[Исходник](#)

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 первом 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
```

[Исходник](#)

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) )
```

[Исходник](#)

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!
```

[Исходник](#)

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) )
```

[Исходник](#)

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!
```

[Исходник](#)

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"
```

[Исходник](#)

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...")

```

[Исходник](#)

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',
'Iван'])
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...
```

[Исходник](#)

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.
```

[Исходник](#)

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
```

[Исходник](#)

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
```

[Исходник](#)

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 = """(с) 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'])
```

Исходник

Исходник

[illegible]

```

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

```

[Исходник](#)

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}

```

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

[Исходник](#)

```

$ 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

```

[Исходник](#)

```

$ 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
```

[Исходник](#)

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) Jenuay 2014. Автор программы, как всегда,
8. не несет никакой ответственности ни за что."'
9.     )
10.
11.     subparsers = parser.add_subparsers (dest = 'command',
12.                                         title = 'Возможные команды',
13.                                         description = 'Команды, которые должны быть в
14. качестве первого параметра %(prog)s')
15.
16.     hello_parser = subparsers.add_parser ('hello')
17.     hello_parser.add_argument ('--names', '-
18. n', nargs='+', default=['мир'],
19.                                help = 'Список приветствуемых людей',
20.                                metavar = 'ИМЯ')
21.
22.     goodbye_parser = subparsers.add_parser ('goodbye')
23.     goodbye_parser.add_argument ('-c', '--
24. count', type=int, default=1,
25.                                help = 'Сколько раз попрощаться с миром',
26.                                metavar = 'КОЛИЧЕСТВО')
27.
28.     return parser
```

[Исходник](#)

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
```

[Исходник](#)

```
$ 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
```

[Исходник](#)

Однако, что-то справку все-таки портит. Мы практически все русифицировали, остались непереуведенными 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) Jenuay 2014. Автор программы, как всегда,
8. не несет никакой ответственности ни за что.""")
9.
10.
11.     subparsers = parser.add_subparsers (dest = 'command',
12.         title = 'Возможные команды',
13.         description = 'Команды, которые должны быть в
14. качестве первого параметра %(prog)s')
```

```

14.         hello_parser = subparsers.add_parser ('hello')
15.
16.         hello_group = hello_parser.add_argument_group (title='Параметры')
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.
23.         goodbye_group = goodbye_parser.add_argument_group (title='Параметры')
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

```

[Исходник](#)

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
```

[Исходник](#)

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) Jenuay 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,
14.    # ведь у него тоже должен быть параметр --help / -h
15.
16.    parent_group = parser.add_argument_group (title='Параметры')
17.
18.    parent_group.add_argument ('--help', '-
19. h', action='help', help='Справка')
20.
21.    # Создаем группу подпарсеров
22.
23.    subparsers = parser.add_subparsers (dest = 'command',
24.        title = 'Возможные команды',
25.        description = 'Команды, которые должны быть в
26. качестве первого параметра %(prog)s')
27.
28.    # Создаем парсер для команды hello
29.
30.    hello_parser = subparsers.add_parser ('hello',
31.        add_help = False)
32.
33.    # Создаем новую группу параметров
34.
35.    hello_group = hello_parser.add_argument_group (title='Парамет
36. ры')
37.
38.    # Добавляем параметры
39.
40.    hello_group.add_argument ('--names', '-
41. n', nargs='+', default=['мир'],
```

```

34.             help = 'Список приветствуемых людей',
35.             metavar = 'ИМЯ')
36.
37.         hello_group.add_argument ('--help', '-
h', action='help', help='Справка')
38.
39.
40.         # Создаем парсер для команды goodbye
41.         goodbye_parser = subparsers.add_parser ('goodbye', add_help =
False)
42.
43.         # Создаем новую группу параметров
44.         goodbye_group = goodbye_parser.add_argument_group (title='Па
раметры')
45.
46.         # Добавляем параметры
47.         goodbye_group.add_argument ('-c', '--
count', type=int, default=1,
48.             help = 'Сколько раз попрощаться с миром',
49.             metavar = 'КОЛИЧЕСТВО')
50.
51.         goodbye_group.add_argument ('--help', '-
h', action='help', help='Справка')
52.
53.         return parser

```

[Исходник](#)

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

```

[Исходник](#)

```
$ 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}

(с) Jenuay 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 = "(с) Jenuay 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,
14.    # ведь у него тоже должен быть параметр --help / -h
15.    parent_group = parser.add_argument_group (title='Параметры')
16.
17.    parent_group.add_argument ('--help', '-
    h', action='help', help='Справка')
18.
```

```
19.
20.     # Создаем группу подпарсеров
21.     subparsers = parser.add_subparsers (dest = 'command',
22.                                         title = 'Возможные команды',
23.                                         description = 'Команды, которые должны быть в
        качестве первого параметра %(prog)s')
24.
25.     # Создаем парсер для команды hello
26.     hello_parser = subparsers.add_parser ('hello',
27.                                           add_help = False,
28.                                           help = 'Запуск в режиме "Hello, world!"',
29.                                           description = '"Запуск в режиме "Hello, world!".
30.       В этом режиме программа приветствует список людей, переданных в
        качестве параметра."')
31.
32.     # Создаем новую группу параметров
33.     hello_group = hello_parser.add_argument_group (title='Парамет
        ры')
34.
35.     # Добавляем параметры
36.     hello_group.add_argument ('--names', '-
        n', nargs='+', default=['мир'],
37.                               help = 'Список приветствуемых людей',
38.                               metavar = 'ИМЯ')
39.
40.     hello_group.add_argument ('--help', '-
        h', action='help', help='Справка')
41.
42.
43.     # Создаем парсер для команды goodbye
44.     goodbye_parser = subparsers.add_parser ('goodbye',
45.                                              add_help = False,
46.                                              help = 'Запуск в режиме "Goodbye, world!"',
47.                                              description = '"В этом режиме программа прощается с
        миром."')
48.
49.     # Создаем новую группу параметров
50.     goodbye_group = goodbye_parser.add_argument_group (title='Па
        раметры')
```

```

51.
52.         # Добавляем параметры
53.         goodbye_group.add_argument ('-c', '--
count', type=int, default=1,
54.                                     help = 'Сколько раз попрощаться с миром',
55.                                     metavar = 'КОЛИЧЕСТВО')
56.
57.         goodbye_group.add_argument ('--help', '-
h', action='help', help='Справка')
58.
59.         return parser

```

[Исходник](#)

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!"

```

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

[Исходник](#)

```
$ 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

```

[Исходник](#)

```
$ 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

[Исходник](#)

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
23.     # парсера,
24.     # ведь у него тоже должен быть параметр --help / -h
25.     parent_group = parser.add_argument_group (title='Параметры')
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))

```

[Исходник](#)

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, напомним, обозначает имя программы, после которого мы добавим номер версии. Воспользуемся этим параметром:

```
$ pythono coolprogram.py --version
```

```
coolprogram 1.2.1
```

[Исходник](#)

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!"

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

[Исходник](#)

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 никакой 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()

```

[Исходник](#)

В остальном программа 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!"

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

[Исходник](#)

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

Заключение

Наконец-то эта статья заканчивается. Интересно, много ли людей дочитали до этих строк, их хочется поблагодарить за терпение.

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.

Спасибо за внимание, жду ваших комментариев и оценок.

<https://zlukfo.bitbucket.io/articles/python-argparse/>

Python: приемы парсинга аргументов командной строки

Начиная с версии 2.7 стандартным модулем парсинга командной строки является argparse.

Создание объекта парсера

```
import argparse

parser = argparse.ArgumentParser()
```

Вообще argparse решает две основные задачи а) парсинг аргументов вызова скрипта и б) вывод справки об этих аргументах.

Справка по аргументам

Выводимая в консоль справка состоит из 4-х блоков

1. командная строка запуска скрипта (usage). По-умолчанию parse_args() анализирует все добавленные в наш объект аргументы и генерирует строку usage с учетом синтаксиса и типа этих аргументов. Значение usage можно изменить при создании объекта парсера, определив параметр usage, например так

```
parser=argparse.ArgumentParser(usage='Script options "%(prog)s"')
```

2. описание скрипта (description). По умолчанию не задано, задается при создании экземпляра парсера

```
parser=argparse.ArgumentParser(description='The "%(prog)s" is a test script')
```

3. Далее идет перечисление аргументов, добавленных в созданный экземпляр парсера, и их описание. При этом, при выводе аргументы группируются - сначала выводится справка о позиционных аргументах, а затем - по опциональным

4. Заключительный блок (epilog), например, с примерами использования ключей

```
parser=argparse.ArgumentParser(epilog='Sample: "%(prog)s -h" - view help')
```

Протестировать, как будет выводиться справочная информация, можно через метод экземпляра парсера `print_help()`

```
parser=argparse.ArgumentParser()  
parser.print_help()
```

Argparser имеет несколько встроенных форматов, позволяющих выводить справочную информацию в различном представлении и с различной степенью детализации. Для этого используются следующие классы

- `RawTextHelpFormatter` (класс по умолчанию)
- `ArgumentDefaultsHelpFormatter` - добавляет информацию о значениях аргументов по умолчанию
- `MetavarTypeHelpFormatter` - выводит тип значения аргумента

Чтобы настроить нужный формат вывода справки, создаем экземпляр парсера так

```
parser=argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)  
parser.print_help()
```

Добавление аргументов

По умолчанию, в экземпляр созданный `ArgumentParser()` сразу добавляется аргумент `'-h'` - вывод справочной информации о скрипте и аргументах его запуска. Отключить добавление ключа `'-h'` можно так

```
parser=ArgumentParser(add_help=False)
```

Остальные аргументы добавляются вызовом метода `add_argument()`. Аргументы бывают позиционные (обязательные) и опциональные (необязательные)

Имена аргументов

Опциональные аргументы отличаются от позиционных наличием префикса. По умолчанию в созданном экземпляре парсера префикс - '-' (дефис). Но можно установить и другой, или несколько префиксов одновременно

```
parser=argparse.ArgumentParser(prefix_chars='-#+')
parser.add_argument('p')
parser.add_argument('-op')
parser.add_argument('+opt')
parser.add_argument('#optional')
parser.print_help()
```

Позиционный аргумент задается строкой - имя аргумента опциональный - строкой (которой предшествует префикс) или последовательностью строк (через запятую) - в этом случае перечисленная последовательность имен будет считаться синонимами одного аргумента. Теоретически argparse позволяет определить аргумент таким способом

```
parser=argparse.ArgumentParser(prefix_chars='-#+')
parser.add_argument('-o', '--option', '+opt', '+#-opti', action='store')
print parser.parse_args()
```

Все перечисленные комбинации префикс+имя являются синонимами одного аргумента, использование любой из них в командной строке будет давать один и тот же результат. Но вряд ли кому нужен такой зоопарк. Поэтому по принятому соглашению, опциональный аргумент задается кратким именем '-o' и (необязательно) полным '--option'

Псевдонимы аргументов

К добавленному аргументу необходимо так-то обращаться. Делается это через псевдоним аргумента. Для позиционного аргумента его псевдоним совпадает с именем. Для опционального по умолчанию псевдоним формируется так: отбрасывается префикс и выбирается (из одного или более) вариант имени, имеющий максимальную длину. Но мы можем самостоятельно задать псевдоним аргумента

```
parser=argparse.ArgumentParser()
parser.add_argument('-a', '--aaa', dest='b')
```

```
print parser.parse_args(['-a1'])  
print parser.parse_args(['--aaa', '1'])
```

Типы аргументов

Argparse позволяет определить тип значений аргумента, задаваемого в командной строке. При парсинге аргументов командной строки проверяется соответствие значения аргумента заданному типу. По умолчанию предполагается, что значение добавляемого аргумента имеет строковый тип (string). В качестве альтернативы мы можем задать а) целочисленный тип (int), б) число с плавающей точкой (float), в) дескриптор файла (open), или с более тонкой настройкой - argparse.FileType() (см. [документацию](#))

```
parser=argparse.ArgumentParser()  
parser.add_argument('-i', type=int)  
parser.add_argument('-f', type=float)  
parser.add_argument('--file', type=open)  
p=parser.parse_args('-i1 -f 1.6 --file test.txt'.split())  
print p  
for line in p.file:  
    print line
```

Но самое интересное - в качестве типа значения аргумента можно задать ссылку на функцию, которая будет принимать значение как строку, преобразовывать ее к нужному типу и возвращать результат в аргумент. Эту возможность удобно использовать, например, чтобы задать аргументу несколько именованных параметров и получить результат в аргумент в виде словаря

```
parser=argparse.ArgumentParser()  
parser.add_argument("--params",  
                    type=lambda kv: dict([i.split('=')for i in kv.split(" ")]))  
  
print parser.parse_args(["--params", 'foo=6 bar=baz'])
```

Существует еще один полезный тип значений аргумента - ограниченный список. Если заданное в командной строке значение не принадлежит этому

списку - будет сгенерировано исключение. Список допустимых значений задается в параметре `choices`, а параметр `type` задает тип значения каждого элемента списка (указывается, если все элементы принадлежат одному типу)

```
parser=argparse.ArgumentParser()
parser.add_argument('foo', type=int, choices=range(5, 10))
parser.add_argument('-a', choices=['string', 1])
print parser.parse_args(['6', '-a', 'string'])
```

Действия над аргументами

1. Параметр вызова `add_argument()` **'action'** указывает парсеру, как необходимо обрабатывать задаваемый аргумент и его значение. По умолчанию, значение аргумента просто сохраняется (`store`)
2. Если **значение аргумента всегда постоянно** (константа) - пользователю нет необходимости каждый раз указывать в командной строке это значение, достаточно просто указать аргумент. Чтобы парсер в данном случае корректно отработал и сохранил константу как значение аргумента, `'action'` присваивается значение `'store_const'` и дополнительно определяется константа

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='store_const', const=42)
parser.parse_args(['--foo'])
```

Если аргумент не был указан в командной строке - в пространстве имен его значение - `'None'`

Параметр `'const'` (и `'default'` тоже) в качестве значения могут принимать ссылки на функции. Это может быть полезным, вот пример

```
parser = argparse.ArgumentParser()
parser.add_argument('-a', nargs='+', type=int)
parser.add_argument('-s', action='store_const', const=sum, default=max)
p=parser.parse_args('-a 1 2 3'.split())
print p.s(p.a)
p=parser.parse_args('-a 1 2 3 -s'.split())
print p.s(p.a)
```

3. Если же аргумент по сути своей - **логическое значение** и нам важно только знать был ли он указан при запуске скрипта в командной строке или нет - для action задаем значение 'store_true' или 'store_false'. Для 'store_true': если аргумент был указан в командной строке - в пространстве имен ему присваивается значение 'True', если нет - 'False'. Для 'store_false' - соответственно все наоборот
4. Если в командной строке могут быть заданы несколько значений одного аргумента, которые необходимо объединить в список, сделать это можно задав для action значение 'append'

```
parser = argparse.ArgumentParser()
parser.add_argument('--foo', action='append')
parser.parse_args('--foo 1 --foo 2'.split())
```

Реализовать такую обработку значений аргумента можно еще минимум двумя способами 1) определив функцию-обработчик в type добавляемого аргумента, 2) указав количество значений nargs для аргумента (см. описание соответствующих параметров)

5. Если существует аргумент, значение для которого не предусмотрено и необходимо посчитать сколько раз этот аргумент был указан в командной строке - используем значение 'count'

```
parser = argparse.ArgumentParser()
parser.add_argument('--verbose', '-v', action='count')
print parser.parse_args(['--verbose', '-v', '-vvv'])
```

Данное действие может быть полезно, если аргумент отождествляет, например, уровень. Тогда '-v' соответствует первому уровню, '-vv' - второму и так далее.

Существуют еще виды действий 'help' и 'version', посмотреть их описание можно в документации. Также, если описанных типов действия недостаточно для описания поведения аргумента, можно создать собственное действие, определив его через класс (см. документацию)

Количество значений опционального аргумента

По умолчанию количество значений аргумента равно 1. Но можно задать любое, определив их число через параметр `nargs` при добавлении аргумента.

Если количество значений для аргумента известно и фиксированно - в `nargs` указываем соответствующее число.

Если аргумент может быть а) указан в командной строке с **одним** значением, б) указан без значения, в) не указан вовсе - задаем `nargs='?'`

```
parser = argparse.ArgumentParser()
parser.add_argument('-f', nargs='?', const='c', default='d')
print parser.parse_args()
print parser.parse_args(['-f'])
print parser.parse_args(['-f7'])
```

Если случай аналогичен случаю выше, но количество аргументов может быть **один и более** - задаем `nargs='*'`. В данном случае применение `'const'` недопустимо.

Еще один случай - если число значений аргумента должно быть **не менее одного**. В этом случае используется значение `nargs='+'`

Указание обязательных аргументов

Параметр `'required=True'` - удобное дополнение, чтобы указать, что аргумент является обязательным для определения в командной строке. В принципе, реализовать данную проверку можно и без этого параметра, но он облегчает проверку

Получение аргументов и их значений

Чтобы при запуске скрипта получить набор заданных в командной строке аргументов и их значений, используется метод `parse_args()`. Ранее он неоднократно встречался в примерах.

Метод создает экземпляр класса `Namespace`, который включает все заданные в командной строке аргументы (позиционные и опциональные) и их значения.

Доступ к значению аргумента (если он был задан в командной строке) осуществляется по его псевдониму

```
parser=argparse.ArgumentParser()  
parser.add_argument('-a')  
p=parser.parse_args()  
print p.a
```

Получить значение всех аргументов (в том числе и не заданных в командной строке) можно через метод `_get_kwargs()`

```
parser=argparse.ArgumentParser()  
parser.add_argument('-a')  
parser.add_argument('-b')  
p=parser.parse_args()  
print p._get_kwargs()
```

Значения незадаанных аргументов будут 'None'.

Протестировать из скрипта как будут распознаваться заданные аргументы и их значения можно так

```
parser=argparse.ArgumentParser()  
parser.add_argument('-a')  
parser.add_argument('-b')  
parser.add_argument('c')  
p=parser.parse_args(['c', '-a', '1', '-bone'])  
print p
```

Здесь несколько важных моментов работы парсера

1. аргументы в командной строке можно задавать в любой последовательности
2. значения опциональных аргументов можно НЕ отделять от имени аргумента пробелом

И вообще, опциональные аргументы в командной строке можно определять несколькими способами

1. 'test.py -a 1'
2. 'test.py -a1'
3. 'test.py -a=1'

И еще метод `parse_args()` позволяет не полностью указывать имя аргумента, а только первую его часть. Если при этом не возникает конфликта имен. Например

```
parser = argparse.ArgumentParser()
parser.add_argument('-barcode')
parser.add_argument('-banner')
p=parser.parse_args()
print p
```

Вызов 'test.py -bar 1' правильно определит какой аргумент был задан в командной строке. А 'test.py -ba 1' приведет к ошибке конфликта имен

Еще Namespace может быть легко преобразован в словарь

```
parser = argparse.ArgumentParser()
parser.add_argument('-a', nargs='+', type=int)
p=parser.parse_args('-a 1 2 3'.split())
print vars(p)
```

Аргументы из файла

Argparse позволяет передавать скрипту значения аргументов из файлов. Это полезно когда а) при запуске скрипта в командную строку необходимо вводить большое количество аргументов, что неудобно, б) можно использовать как аналог конфигурационных файлов запуска скрипта с различными устойчивыми наборами параметров. Загрузка аргументов из файла выполняется так

```
parser = argparse.ArgumentParser(fromfile_prefix_chars='#')
parser.add_argument('-f', nargs=2)
parser.add_argument('--bar')
parser.add_argument('b')
p=parser.parse_args(['#arg.test'])
```

```
print p
```

При создании экземпляра парсера необходимо определить односимвольный префикс. Он будет указывать парсеру, что после этого префикса идет имя файла (а не очередной аргумент) из которого нужно получить набор аргументов для запуска скрипта.

Далее, в файле можно перечислять только те аргументы, которые добавлены в экземпляр парсера. правильно.

Правило составления файла с аргументами следующее: сначала идет псевдоним опционального аргумента (со всеми необходимыми префиксами), затем на следующей строке - его значение (если предусмотрено). Если аргумент предполагает несколько значений каждый указывается в новой строке. В остальном правила указания аргументов в файле (в том числе и их порядок) таакой же как и в командной строке.

Теперь при запуске скрипта достаточно только указать имя файла с аргументами.

Кстати ничто не мешает нам задавать аргументы параллельно в файле и командной строке. В данном случае парсер работает по следующему правилу: значение аргумента будет таким, который был задан последним. Тоест если сначала указать имя файла с аргументами, а затем определить аргумент в командной строке, то аргументу будет присвоено значение из командной строки. И наоборот

И еще пара моментов. Описанные правила указания аргументов в файле определяются работой парсера по умолчанию. Вместе с тем `argparse` позволяет создать собственный парсер, задающий другие правила. Соответственно и порядок указания аргументов в файле изменится. Подробнее - [в документации](#)

Чтобы избежать исключение парсера, вызванного тем, что в файле определены аргументы не добавленные в парсер, в скрипте можно предусмотреть проверку. Делается это при помощи метода `parse_known_args()`. Например так

```
parser = argparse.ArgumentParser(fromfile_prefix_chars='#')
```

```
parser.add_argument('-a')
parser.add_argument('-b')
p=parser.parse_known_args(['#arg.test'])
print p
```

Метод возвращает кортеж, первый элемент которого - Namespace распознанных (добавленных в парсер) аргументов и их значений, которые были заданы при вызове скрипта (в командной строке или в файле). Второй элемент - список нераспознанных аргументов и их значений

Команды

Их идея в следующем. Запускаемый скрипт может выполнять несколько задач, каждая из которых а) может быть описана отдельной командой, б) каждая команда имеет собственный набор аргументов. При этом, наборы аргументов различны, слабо пересекаются друг с другом или не пересекаются вовсе. Если это не так (все команды имеют одинаковой (почти) набор аргументов), то городить огород с выделением команд скрипта не имеет смысла. Приведем пример - скрипт работы с базой данных, который должен выполнять задачи:

1. добавление записи в таблицу, задача 'add'
2. просмотр записей таблицы по фильтру, задача 'view'.

Очевидно, что сначала нужно подключиться к бд. Значит необходим аргумент, задающий параметры подключения, назовем его '-p'. Этот аргумент необходим для всех задач.

Для задачи 'add' необходим аргумент '-d', задающий данные для полей добавляемой записи. А для задачи 'view' - аргумент, определяющий фильтр '-f'. Итого, для задач имеем набор общих аргументов ['-p'] и набор не пересекающихся ['-d'] и ['-f']. Реализация примерно такая

```
parser = argparse.ArgumentParser()
parser.add_argument('-p', required=True, help='Connection params (for all commands)')

# создаем подпарсер команд пользователя. Внимание !! в парсер можно добавить только
ОДИН подпарсер
user_subparsers = parser.add_subparsers(help='Script commands for user')
```

```
# добавляем команды и аргументы для каждой из них

parser_add = user_subparsers.add_parser('add', help="Discribe for command 'add'")
parser_add.add_argument('-d',help='add help')

parser_view = user_subparsers.add_parser('view', help="Discribe for command 'view'")
parser_view.add_argument('-f',help='view help')

parser.print_help()

print parser.parse_args(['-p', 'j', 'add'])
```

Команды являются **обязательными**. Определив в коде скрипта, что мы будем использовать команды, при вызове в командной строке мы обязаны указывать одну из них

Что нам дает применение команд - Более ясное понимание выполняемых действий. Если исключить команды у нас получится один общий набор атрибутов ['-p', '-d', '-f']. Причем для одного действия обязательной будет только часть команд. Для другого - другая часть. В итоге, у пользователя скрипта может возникнуть непонимание, какое сочетание аргументов обязательно использовать, чтобы выполнить конкретное дейсвие. Команды помогают сориентироваться в наборе необходимых аргументов. А вывод справки

```
print parser.parse_args(['add', '-h'])
```

помогает понять какие аргументы необходимы для выполнения действия.

В итоге, используя команды, мы можем упростить код нашего скрипта. Поясним, используя пример выше Если мы реализуем запуск скрипта без команд, то чтобы определить, какое дейсвие хочет выполнить пользователь, необходимо проверить (if...else), какие документы он задал, и уже после этого запускать соответствующий код.

Но если мы реализуем выполнение команд в скрипте, то для каждой из них можем задать функцию которая будет выполнять необходимый код, а задание проверки необходимых параметров возложить на argparse. Этот прием реализуется с помощью метода set_defaults()

Задание аргументов по умолчанию (set_defaults())

Метод задает для парсера (или команды) аргументы и их значения, которые будут добавлены в Namespace автоматически, без необходимости их определения пользователем. В качестве значения аргумента по умолчанию мы можем добавить и функцию, которая будет выполнять необходимые действия.

Таким образом определение и выполнение намерений пользователя с помощью команд может выглядеть так(в качестве примера напишем скрипт решающий две задачи: 1) возведение числа в степень и 2) получение натурального логарифма из числа. Для обеих задач существует общий обязательный аргумент - число. Кроме того, для задачи (1) должен быть определен дополнительный обязательный аргумент - степень (число с плавающей точкой). Для задачи (2) нужен аргумент основание логарифма).

```
def pow(args):  
    return args.d**args.s  
  
import math  
def log(args):  
    return math.log(args.d,args.b)  
  
if __name__=='__main__':  
    parser = argparse.ArgumentParser()  
    parser.add_argument('-d', required=True, type=int)  
    subparsers = parser.add_subparsers()  
  
    parser_pow = subparsers.add_parser('pow')  
    parser_pow.add_argument('-s', required=True, type=float)  
    parser_pow.set_defaults(f=pow)  
  
    parser_log = subparsers.add_parser('log')  
    parser_log.add_argument('-b', type=float)  
    parser_log.set_defaults(f=log)  
  
    p=parser.parse_args(['-d 2', 'pow', '-s 3'])
```

```
print p.f(p)

p=parser.parse_args(['-d 2', 'log', '-b 3'])

print p.f(p)
```

Т.е., применяя команды отпадает необходимость писать проверки на необходимые аргументы - эту задачу выполняет argparse

В статье рассмотрены основные (не все) возможности использования argparse. Подробнее - [в документации \(англ.\)](#)