

Библиотека argparse. Задачи на создание скриптов с ее помощью

План урока

- 1 Повторение
- 2 Библиотека argparse
- 3 Модули, импорт модулей из скриптов

Аннотация

*В этом уроке мы продолжим работать с командной строкой и разберём возможности библиотеки **argparse**.*

1. Повторение

На предыдущем занятии мы рассмотрели простой способ работы с данными, которые поступают в программу (скрипт) через параметры командной строки. Однако такой способ не даёт нужной гибкости, и в сложных случаях его применять не стоит.

Рассмотрим пример реализации программы, которая ожидает получить в параметрах список чисел в системе счисления с основанием, переданном в параметре `base` (если он не указан,

то подразумевается **двоичная** система счисления) и переводит их в десятичную.

Преобразованный список чисел выводится на экран.

В двоичной системе счисления есть только две цифры — 0 и 1, и запись чисел в ней получается существенно длиннее, чем в десятичной.

```
import sys

def print_help(msg=""):
    print("Usage: {} [-h] [--log LOG] [--base BASE] int [int ...]\n{}".format(
        sys.argv[0], msg))

def main(args):
    integers = []
    log_file = ''
    base = 2

    while (args):
        arg = args.pop(0)
        if arg == '-h':
            print_help()
            return None, None
        elif arg == '--base':
            try:
                base = int(args.pop(0))
            except ValueError:
                print_help("invalid base value: {}".format(arg))
                return None, None
        elif arg == '--log':
            log_file = args.pop(0)
        else:
            integers.append(arg)

    if not integers:
        print_help('No int args')
        return None, None

    try:
        return (list(map(lambda x: int(x, base), integers)), log_file)
    except ValueError as e:
        print_help("invalid value: {}".format(e))
        return None, None

numbers, log_file = main(sys.argv[1:])
```

```

if log_file is None:
    pass
elif log_file == "":
    print(*numbers)
else:
    with open(log_file, "wt") as output:
        print(*numbers, file=output)

```

Несколько слов о тексте выше. В функции мы последовательно читаем список переданных параметров и, если получаем служебное значение (-h, --log, --base), то анализируем следующий за ним параметр. Если же мы получаем число, то сохраняем его в списке. Когда список заполнен, его преобразуют **lambda**-функции.

Посмотрим на то, что у нас получилось.

Вызов без параметров:

```

python3 files/ex1.py

Usage: files/ex1.py [-h] [--log LOG] [--base BASE] int [int ...]
No int args

```

Вызов с правильными параметрами, но без указания основания системы счисления:

```

python3 files/ex1.py 110 1 1010

6 1 10

```

Вызов с ошибочными параметрами и указанием системы счисления:

```

python3 files/ex1.py 147 22 3 --base 3

Usage: files/ex1.py [-h] [--log LOG] [--base BASE] int [int ...]
invalid value: invalid literal for int() with base 3: '147'

```

Вызов с правильными параметрами и сохранением в файл:

```

python3 files/ex1.py 110 1 1010 --base 3 --log files/work.log

```

Вроде всё работает, но посмотрите, сколько текста мы написали ради трёх параметров! А если

параметры имеют псевдонимы? Например, для вывода справочной информации по команде можно задавать как параметр `-h`, так и параметр `--help`. Что же делать?

Выход из ситуации напрашивается сам собой. Если нужно сделать простую обработку параметров, то используйте список `argv` из библиотеки `sys`, но для сложных ситуаций придётся искать другой путь.

И в Python есть более лёгкий, быстрый и надёжный способ — это библиотека **argparse**.

2. Библиотека argparse

Попробуем решить нашу задачу иным способом:

```
import argparse
import sys

parser = argparse.ArgumentParser(
    description="convert integers to decimal system")
parser.add_argument('integers', metavar='integers', nargs='+',
                    type=str, help='integers to be converted')
parser.add_argument('--base', default=2, type=int,
                    help='default numeric system')
parser.add_argument('--log', default=sys.stdout, type=argparse.FileType('w'),
                    help='the file where converted data should be written')

args = parser.parse_args()
s = " ".join(map(lambda x: str(int(x, args.base)), args.integers))
args.log.write(s + '\n')
args.log.close()
```

```
python3 files/ex2.py -h
```

```
usage: ex2.py [-h] [--base BASE] [--log LOG] integers [integers ...]
           convert integers to decimal system
```

```
positional arguments:
  integers      integers to be converted
```

```
optional arguments:
  -h, --help    show this help message and exit
  --base BASE   default numeric system
```

```
--log LOG    the file where converted data should be written
```

```
python3 files/ex2.py 1 11 111
```

```
1 3 7
```

Интересно?

Давайте разбираться сначала.

Для парсинга аргументов с помощью **argparse** требуется импортировать саму библиотеку, создать экземпляр объекта **ArgumentParser** и запустить функцию парсинга **parse_args()**:

```
import argparse

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

Если запустить эту программу с ключом -h, то мы сразу получим справочную информацию.

```
python3 files/first_argparse.py -h

usage: first_argparse.py [-h]

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

Вот и всё, что нужно программе, чтобы начать обрабатывать приходящие в неё аргументы. Таким образом, приложив минимум усилий, у нас есть:

1. Документация (которую можно вызвать, передав опцию --help или -h).
2. Проверка валидности (корректности) аргументов.
3. Сообщения об ошибках при получении невалидных аргументов.

Но это далеко не всё, что может argparse. Скорее всего, раз мы решили им воспользоваться, то захотим получить значения других, в том числе и не совсем стандартных аргументов.

Чтобы сообщить парсеру о таких аргументах у **ArgumentParser**’а есть метод **add_argument**. Он принимает множество параметров, однако обязательным является только **название** создаваемого аргумента или флага (первый параметр).

```
ArgumentParser.add_argument(<name or flags> [, help][, metavar][, type]
                             [, nargs][, default][, action][, const]
                             [, choices][, required][, dest])
```

Парсер использует это имя для обозначений переменных, при выводе в справочной информации и т.д. Остальные модификаторы могут потребоваться для реализации более сложных конструкций. Рассмотрим некоторые из них.

Текст подсказки, который показывается при вызове справки, вводится с помощью параметра **help**. **metavar** отвечает за название параметра в подсказке (если его не указать, то берётся имя из первого аргумента name). **type** гарантирует, что параметр хранит значения только указанного типа (причём число можно представить в виде строки, но не любую строку можно привести к числу).

При описании свойств аргумента в `add_argument` можно обозначить ожидаемое количество таких аргументов с помощью **nargs**. Это число (как 3 или 7) или строка «+» (которая означает, что элементов должно быть 1 или больше), или «?» (может 1, а может и не быть), или «*» (любое количество от 0 до бесконечности).

Ещё одним полезным параметром является значение аргумента по умолчанию (программисты говорят **дефолтное** значение, от английского слова *default*), которое задается с помощью параметра **default**. Этот параметр может принимать в себя строку, массив, объект (к примеру, `sys.stdin`) и многое другое.

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("arg1")
parser.add_argument("arg2", help="echo this string")
parser.add_argument("int_args", metavar="N", type=int,
                    nargs='+', help="echo some integers")
args = parser.parse_args()

print(args.arg1)
print(args.arg2)
print(args.int_args)
```

```
python3 files/ex3.py -h
```

```
usage: ex3.py [-h] arg1 arg2 N [N ...]
```

```
positional arguments:
```

```
  arg1
```

```
  arg2          echo this string
```

```

N          echo some integers

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

-----

python3 files/ex3.py 'one'      'two' 3 4 17

one
two
[3, 4, 17]

```

Давайте рассмотрим и проанализируем ещё один пример:

```

import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--name")
parser.add_argument("-up", "--up_case", action="store_true",
                    help="convert name to upper register")
parser.add_argument("--number", choices=[0, 1, 2], type=int,
                    default=0, help="select number", required=True)
parser.add_argument("--no-name", action="store_const",
                    const="no", dest="name")
args = parser.parse_args()

name = args.name
if (args.up_case):
    name = name.upper()

print("The name is {}. And the number = {}".format(name, args.number))

```

Мы уже встречались с именованными аргументами, то есть такими, для которых расположение в командной строке при вызове программы не играет роли, а важно лишь то, чтобы значение аргумента предварялось его именем (ключом). Примером такого аргумента является, например, **--base** в самом первом примере этого урока.

Чтобы сообщить argparse о желании использовать именованный аргумент достаточно вставить знак «-» перед его названием (можно один, можно два, а можно и оба варианта сразу).

В примере выше определён именованный аргумент **name**. Для него мы не указали ни подсказки (для справки), ни каких-либо других параметров.

Ещё у нас есть аргумент **up_case** (или **up**), простой флаг, принимающий значение true, если он указан. Такое поведение достигается благодаря параметру **action** у метода **add_argument**.

Возможные значения этого параметра:

— store_true — установить значение **true**;

— store_false — установить значение **false**;

— store_const — установить значение, указанное в параметре **const**. При этом параметр **dest** хранит имя переменной, в которой сохраняется это значение.

Фраза `parser.add_argument("--no-name", action="store_const", const="no", dest="name")` расшифровывается так:

1. Создать именованный параметр **no-name**.

2. При его указании проинициализировать переменную с именем **name** (указано в параметре **dest**) значением **no**.

При добавлении аргумента **number** мы указали параметр **choice**, что позволило определить возможные значения для аргумента. Параметр **required = True** указывает на то, что аргумент является обязательным.

Попробуйте самостоятельно поработать с программой выше, поизменять параметры и посмотреть, что получается.

Несколько примеров её работы:

```
python3 files/ex4.py --help
```

```
usage: ex4.py [-h] [--name NAME] [-up] --number {0,1,2} [--no-name]
```

```
optional arguments:
```

```
-h, --help            show this help message and exit
--name NAME
-up, --up_case        convert name to upper register
--number {0,1,2}      select number
--no-name
```

```
-----
```

```
python3 files/ex4.py -up --no-name
```

```
usage: ex4.py [-h] [--name NAME] [-up] --number {0,1,2} [--no-name]
```

```
ex4.py: error: the following arguments are required: --number
```

```
An exception has occurred, use %tb to see the full traceback.
```

```
SystemExit: 2
```


3. Модули, импорт модулей из скриптов

Настало время немного детальнее поговорить о модулях в Python. Мы уже неоднократно ими пользовались, но не упомянули о том, как же они устроены внутри. Давайте исправим этот недочет.

Когда в тексте нашей программы мы пишем команду **import**, Python пытается подключить (загрузить) файл, имя которого мы указали. Но для этого он должен ответить на 2 вопроса:

1. Где искать файл?
2. Что означает термин **подключить** файл?

Давайте разберёмся с этими вопросами.

В начале прошлого урока мы говорили про системную переменную **PATH** и поиск исполняемых файлов.

В языке Python работает похожая технология для модулей при выполнении команды `import`. Давайте узнаем, где же Python будет искать файл с модулем, когда получит соответствующую команду.

Для этого посмотрим на значение переменной **sys.path** из библиотеки (модуля) **sys**:

```
import sys
import pprint
pprint.pprint(sys.path)

-----

['',
 '/Users/anaconda/lib/python36.zip',
 '/Users/anaconda/lib/python3.6',
 '/Users/anaconda/lib/python3.6/lib-dynload',
 '/Users/anaconda/lib/python3.6/site-packages',
 '/Users/anaconda/lib/python3.6/site-packages/Sphinx-1.6.3-py3.6.egg',
 '/Users/anaconda/lib/python3.6/site-packages/aeosa',
 '/Users/anaconda/lib/python3.6/site-packages/setuptools-27.2.0-py3.6.egg',
 '/Users/anaconda/lib/python3.6/site-packages/IPython/extensions',
 '/Users/.ipython']
```

Как вы могли заметить, переменная **sys.path** хранит некоторый список путей. А на первой позиции в этом списке стоит **пустая строка**. Это означает, что при импорте модуля, поиск первым делом будет осуществляться в каталоге, где находится сама запускаемая программа.

Если в текущем каталоге модуль не найден, то Python попытается найти его последовательно во всех каталогах списка `sys.path`. Он может искать даже в zip-архивах.

Для того, чтобы ответить на второй вопрос, выполним следующие действия:

1. Создадим файл **my_module.py** с одной функцией **some_func** и основной частью.
2. Импортируем его в программу **my_script.py** и вызовем функцию **some_func**.
3. Модифицируем тексты программ и посмотрим на результат.

```
# my_module.py

def some_func():
    print("func is running")

print("main part of my_module.py")
some_func()
```

```
# my_script.py

from my_module import some_func

print("my_script is running")
some_func()
```

```
python3 files/my_script.py
```

```
-----

main part of my_module.py
func is running
my_script is running
func is running
```

Посмотрите, что получилось: по команде `import` Python **выполнил** программу `my_module.py`, то есть исполнил все инструкции основной программы и вызвал функцию `some_func`. А уже после этого управление передалось в `my_script`.

Таким образом, мы ответили на вопрос «Что означает подключить модуль?». Это означает **выполнить** этот модуль.

Но ведь, на самом деле, мы ожидали не совсем такого эффекта! Мы хотели выполнить только `my_script.py`, где использовать наработки из `my_module.py`. Как быть?

В любой программе на Python есть специальная переменная **name**. Анализ её значения и поможет нам корректно работать с модулями.

Правило такое: если в переменной **name** находится значение «**main**», это означает, что интерпретатор Python вызвал программу самостоятельно. Если этот файл импортируется с помощью команды `import`, переменной **name** будет присвоено имя модуля.

Воспользуемся знанием и немного переделаем программы (добавим в их название слово `good`):

```
# my_module_good.py

def some_func():
    print("func is running")
    if __name__ == "__main__":
        print("I was called without ##import##")

def main():
    print("Main part of my_module.py")
    some_func()

if __name__ == "__main__":
    main()
```

```
# my_script_good.py

from my_module_good import some_func

def main():
    print("My_script is running")
    some_func()

if __name__ == "__main__":
    main()
```

```
python3 files/my_module_good.py
```

```
Main part of my_module.py
```

```
func is running  
I was called without ##import##
```

```
python3 files/my_script_good.py
```

```
My_script is running  
func is running
```

Как мы видим, если вызвать напрямую `my_module_good.py`, то запустится функция `main`, откуда произойдет вызов функции `some_func`. А если мы просто подключаем модуль, то вызова функции `main` из файла `my_module_good.py` не произойдёт.

Такой способ оформления программы является общепринятой практикой, и мы призываем вас следовать ей в будущем.

[Помощь](#)

© 2018 – 2019 ООО «Яндекс»