

Поделитесь своим мнением о Яндекс.Лицее и помогите нам в его развитии

Пройти опрос →

Урок argparse

Библиотека argparse. Задачи на создание скриптов с ее помощью

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

Аннотация

В этом уроке мы продолжим работать с командной строкой и разберем 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 дает 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(f"Usage: {sys.argv[0]} [-h] [--log LOG] [--base BASE] int [int ...]\n{msg}")
```

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

    while (args):
        arg = args.pop(0)
        if arg == '-h':
            print_help()
            return
```



```

    print_help()
    return None, None
elif arg == '--base':
    try:
        base = int(args.pop(0))
    except ValueError:
        print_help(f"invalid base value: {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(f"invalid value: {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, 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 бы все работает, но посмотрите, сколько текста мы написали ради трех 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 придется искать другой путь.

И в Python есть более легкий, быстрый и надежный 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.

Для парсинга аргументов с помощью `argparse` требуется импортировать саму библиотеку (дополнительно устанавливать ее не нужно), 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`, 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 все, что нужно программе, чтобы начать обрабатывать приходящие в нее аргументы. Таким образом, 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 все, что может `argparse`. Скорее всего, раз мы решили им воспользоваться, 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 берется имя из первого аргумента `name`). `type` гарантирует, что параметр хранит значения только указанного типа (причем 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 параметра `nargs`. Это число (как 2 или 7) или строка '+' (которая означает, что элементов должно быть 1 или

Еще 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 еще 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(f"The name is {name}. And the number = {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 определен именованный аргумент `name`. Для него мы не указали ни подсказки (для справки), ни каких-либо других параметров.

Еще 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 ее 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 вопрос: где искать файл?

Давайте разберемся.

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.

Если ни по одному из путей файл не найден, мы получим ошибку.

Как вы уже знаете, Python позволяет импортировать те модули, которые вы разработали самостоятельно. Для этого (если импортируемый файл и файл, в который импортируем, лежат в одной директории) надо написать:

```
import имя_файла_без_.py
```

Например, для файла `module.py` импорт будет выглядеть так:

```
import module
```

Однако при импорте Python выполнит этот файл, как будто вы запустили его на исполнение как отдельную программу. Иногда именно это нам и нужно, но существует достаточно большой процент сценариев, когда такое

В любой программе на Python есть глобальная переменная `__name__`. Анализ ее 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. Мы уже встречались с подобной записью при изучении библиотеки PyQt

Рассмотрим еще один пример:

```
# 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()
```

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 произойдет.

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

4. Заключение

Мы немного отвлеклись от непосредственного изучения взаимодействия с различными API, чтобы изучить

Справка

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

[Пользовательское соглашение.](#)

© 2018 – 2021 ООО «Яндекс»