Лабораторная работа 2.17. Разработка приложений с интерфейсом командной строки (CLI) в Python3

Цель работы: *приобретение построения приложений с интерфейсом командной строки с помощью языка программирования Python версии 3.x.*

Ход работы

Операционная среда или **окружение** *environment* — интерфейс, предоставляемый пользователю или программе операционной системой. В частности, пользовательский интерфейс является частью операционной среды.

Командная строка *command line* — принцип организации пользовательского интерфейса на основе ввода текстовых команд с клавиатуры и текстового вывода результатов на экран. Интерфейс на основе командной строки — *command line interface, CLI*.

Оболочка командной строки или просто **оболочка** *shell* — программное обеспечение, отвечающее за поддержку командной строки (обычно это компонент ОС, но может быть и сторонним ПО). Примеры: cmd.exe и Powershell в Windows, sh, csh, bash, ksh и др. в Unix-подобных системах. Оболочка командной строки предоставляет собственное окружение: “переменные среды” *environment variables* (глобальные и локальные для текущего сеанса) и интерпретатор текстовых команд.

Пакетный файл *batch file* или **сценарий** — содержащий команды оболочки файл, который можно запустить на исполнение как исполняемый файл.

Терминал (от лат. *terminus* — граница) — устройство или ПО, выступающее посредником между человеком и вычислительной системой. Обычно данный термин используется, когда точка доступа к системе вынесена в отдельное физическое устройство и предоставляет свой пользовательский интерфейс на основе внутреннего интерфейса (например, сетевых протоколов).

Консоль *console* — исторически реализация терминала с клавиатурой и текстовым дисплеем. В настоящее время это слово часто используется как синоним сеанса работы или окна оболочки командной строки. В том же смысле иногда применяется и слово “терминал”.

Консольное приложение *console application* — вид ПО, разработанный с расчётом на работу внутри оболочки командной строки, т.е. опирающийся на текстовый ввод-вывод.

Ранние компьютеры представляли собой громоздкие и дорогие устройства, их рабочее время распределялось между многими пользователями. При этом пользователь обычно не мог работать непосредственно с вычислительным устройством, а использовал терминал, который выполнялся в виде рабочего места с устройствами ввода-вывода, подключенными к компьютеру. Первые устройства такого рода использовали бумажные носители: перфокарты или перфоленту для ввода (символы задавались группами отверстий, пробитых или не пробитых в нужных позициях) и бумажную ленту для вывода (вывод компьютера печатался на ней устройством, аналогичным телетайпу).

Затем бумагу убрали, но представление диалога пользователя и компьютера в виде последовательности текстовых запросов и сообщений осталось: для ввода использовалась клавиатура, а для отображения и ввода и вывода — монитор. Равно остались и терминалы, с помощью которых множество пользователей могло работать с одним дорогим и высокопроизводительным компьютером (серийно выпускаемые большие компьютеры стали называть “мэйнфреймами”).

Впоследствии миниатюризация вычислительных устройств позволила сделать схему “ЭВМ ↔ сеть ↔ терминалы” необязательной, совместив ЭВМ и терминал в одном устройстве — персональном компьютере. Тем не менее, старый текстовый интерфейс продолжил существование ввиду своей исключительной простоты. Все современные операционные системы, несмотря на наличие во многих случаях развитого графического интерфейса пользователя и новых устройств ввода (мыши, тачскрина и т. д.), могут работать и в режиме текстового ввода-вывода (“консоли”).

Обычно операционные системы предоставляют специальную программу — оболочку командной строки, работающую в текстовом режиме и принимающую некоторый относительно стандартизованный набор команд, с помощью которых пользователь может управлять работой ОС, запускать другие программы. Эту программу “по старой памяти” и называют “терминалом” или “консолью”.

Поскольку Python является таким популярным языком программирования, а также поддерживает большинство операционных систем, он стал широко использоваться для создания инструментов командной строки для многих целей. Эти инструменты могут варьироваться от простых приложений CLI до более сложных, таких как инструмент AWS awscli.

Такие сложные инструменты, как этот, обычно управляются пользователем с помощью аргументов командной строки, что позволяет пользователю использовать определенные команды, устанавливать параметры и многое другое. Например, эти параметры могут указывать инструменту на вывод дополнительной информации, чтение данных из указанного источника или отправку вывода в определенное место.

Как правило, аргументы передаются инструментам CLI по-разному, в зависимости от вашей операционной системы:

- Подобно Unix: – за которым следует буква, например -h, или — за которым следует слово, например —help
- Windows:/, за которым следует буква или слово, например /help.

Эти разные подходы существуют по историческим причинам. Многие программы в Unix-подобных системах поддерживают как одинарное, так и двойное тире. Обозначение одинарного тире в основном используется с однобуквенными параметрами, а двойное тире представляет собой более читаемый список параметров, что особенно полезно для сложных параметров, которые должны быть более явными.

> В этой лабораторной работе мы сосредоточимся исключительно на Unix-подобном формате.

Имейте в виду, что и имя, и значение аргумента специфичны для программы – нет общего определения, кроме нескольких общих соглашений, таких как —help для получения дополнительной информации об использовании инструмента. Как разработчик сценария Python вы решаете, какие аргументы предоставить вызывающей стороне и что они будут делать. Это требует правильной оценки.

По мере роста вашего списка доступных аргументов ваш код станет более сложным в попытках их точного анализа. К счастью, в Python есть ряд библиотек, которые помогут вам в этом. Мы рассмотрим несколько наиболее распространенных решений, от «сделай сам» с sys.argv до подхода «сделай за тебя» с argparse.

Обработка аргументов командной строки

Python 3 поддерживает несколько различных способов обработки аргументов командной строки. Встроенный способ – использовать модуль `sys`. С точки зрения имен и использования, он имеет прямое отношение к библиотеке C (libc). Второй способ – это модуль `getopt`, который обрабатывает как короткие, так и длинные параметры, включая оценку значений параметров.

Кроме того, существуют два других общих метода. Это модуль `argparse`, производный от модуля optparse, доступного до Python 2.7. Другой метод – использование модуля `docopt`, доступного на GitHub. У каждого из этих способов есть свои плюсы и минусы, поэтому стоит оценить каждый, чтобы увидеть, какой из них лучше всего соответствует вашим потребностям.

Запустим команду **ls** (для Linux использовать bash для Windows – PowerShell) и посмотрим, какие у нее есть аргументы:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

- Можно запускать команду без параметров, она покажет содержимое текущей директории.
- Позиционный аргумент - имя директории, которую мы просматриваем. Программа решает что делать с аргументом на основе того, *где* он появился в командной строке. Так в команде `cp src dst` первый аргумент - что копируем, второй аргумент - куда копируем.
- `-l` может появиться, а может и нет. Опциональный аргумент.
    - Возможна склейка аргументов, т.е. либо вызов `ls -l -a`, либо `ls -la`.
    - опциональные аргументы возможно записать в короткой `-s` или полной `--size` форме.
- есть хелп

Модуль `sys`

Это базовый модуль, который с самого начала поставлялся с Python. Он использует подход, очень похожий на библиотеку C, с использованием `argc` и `argv` для доступа к аргументам. Модуль `sys` реализует аргументы командной строки в простой структуре списка с именем `sys.argv`.

Каждый элемент списка представляет собой единственный аргумент. Первый элемент в списке `sys.argv [0]` – это имя скрипта Python. Остальные элементы списка, от `sys.argv [1]` до `sys.argv [n]`, являются аргументами командной строки с 2 по n. В качестве разделителя между аргументами используется пробел. Значения аргументов, содержащие пробел, должны быть заключены в кавычки, чтобы их правильно проанализировал `sys`.

Эквивалент `argc` – это просто количество элементов в списке. Чтобы получить это значение, используйте оператор `len()`. Позже мы покажем это на примере кода.

Модуль `getopt`

Как вы могли заметить ранее, модуль `sys` разбивает строку командной строки только на отдельные фасеты. Модуль `getopt` в Python идет немного дальше и расширяет разделение входной строки проверкой параметров. Основанный на функции C `getopt`, он позволяет использовать как короткие, так и длинные варианты, включая присвоение значений.

На практике для правильной обработки входных данных требуется модуль `sys`. Для этого необходимо заранее загрузить как модуль `sys`, так и модуль `getopt`. Затем из списка входных параметров мы удаляем первый элемент списка (см. код ниже) и сохраняем оставшийся список аргументов командной строки в переменной с именем `arguments_list`.

```
# Include standard modules
import getopt, sys

# Get full command-line arguments
full_cmd_arguments = sys.argv

# Keep all but the first
argument_list = full_cmd_arguments[1:]
print(argument_list)
```

Аргументы в списке аргументов теперь можно анализировать с помощью метода `getopts()`. Но перед этим нам нужно сообщить `getopts()` о том, какие параметры допустимы. Они определены так:

```
short_options = "ho:v"
long_options = ["help", "output=", "verbose"]
```

Это означает, что эти аргументы мы считаем действительными, а также некоторую дополнительную информацию:

```
-----------------------------------------
long argument short argument with value
-----------------------------------------
--help        -h    no
--output      -o    yes
--verbose     -v    no
-----------------------------------------
```

Вы могли заметить, что после опции o short ставится двоеточие (:). Это сообщает `getopt`, что этой опции следует присвоить значение. Теперь это позволяет нам обрабатывать список аргументов. Для метода `getopt()` необходимо настроить три параметра – список фактических аргументов из argv, а также допустимые короткие и длинные параметры (показаны в

предыдущем фрагменте кода).

Сам вызов метода хранится в инструкции `try-catch`, чтобы скрыть ошибки во время оценки. Исключение возникает, если обнаруживается аргумент, который не является частью списка, как определено ранее. Скрипт в Python выведет сообщение об ошибке на экран и выйдет с кодом ошибки 2.

```
try:
    arguments, values = getopt.getopt(argument_list, short_options,
long_options)
except getopt.error as err:
    # Output error, and return with an error code
    print(str(err))
    sys.exit(2)
```

Наконец, аргументы с соответствующими значениями сохраняются в двух переменных с именами arguments и values. Теперь вы можете легко оценить эти переменные в своем коде. Мы можем использовать цикл `for` для перебора списка распознанных аргументов, одна запись за другой.

```
# Evaluate given options
for current_argument, current_value in arguments:
    if current_argument in ("-v", "--verbose"):
        print("Enabling verbose mode")
    elif current_argument in ("-h", "--help"):
        print("Displaying help")
    elif current_argument in ("-o", "--output"):
        print(f"Enabling special output mode ({current_value})")
```

Ниже вы можете увидеть результат выполнения этого кода. Далее показано, как программа реагирует как на допустимые, так и на недопустимые программные аргументы:

```
$ python arguments-getopt.py -h
Displaying help
$ python arguments-getopt.py --help
Displaying help
$ python arguments-getopt.py --output=green --help -v
Enabling special output mode (green)
Displaying help
Enabling verbose mode
$ python arguments-getopt.py -verbose
option -e not recognized
```

Последний вызов нашей программы поначалу может показаться немного запутанным. Чтобы понять это, вам нужно знать, что сокращенные параметры (иногда также называемые флагами) могут использоваться вместе с одним тире. Это позволяет вашему инструменту легче воспринимать множество вариантов.

Модуль `argparse`

Начиная с версий Python 2.7 и Python 3.2, в набор стандартных библиотек была включена библиотека `argparse` для обработки аргументов (параметров, ключей) командной строки. Хотелось бы остановить на ней Ваше внимание.

Для начала рассмотрим, что интересного предлагает `argparse`:

- анализ аргументов `sys.argv`;
- конвертирование строковых аргументов в объекты Вашей программы и работа с ними;
- форматирование и вывод информативных подсказок.

Одним из аргументов противников включения argparse в Python был довод о том, что в стандартных модулях и без этого содержится две библиотеки для семантической обработки (парсинга) параметров командной строки. Однако, как заявляют разработчики `argparse`, библиотеки `getopt` и `optparse` уступают argparse по нескольким причинам:

- обладая всей полнотой действий с обычными параметрами командной строки, они не умеют обрабатывать позиционные аргументы (positional arguments). Позиционные аргументы — это аргументы, влияющие на работу программы, в зависимости от порядка, в котором они в эту программу передаются. Простейший пример — программа cp, имеющая минимум 2 таких аргумента *(«cp source destination»)*.
- `argparse` дает на выходе более качественные сообщения о подсказке при минимуме затрат (в этом плане при работе с `optparse` часто можно наблюдать некоторую избыточность кода);
- `argparse` дает возможность программисту устанавливать для себя, какие символы являются параметрами, а какие нет. В отличие от него, `optparse` считает опции с синтаксисом наподобие "-pf, -file, +rgb, /f и т.п. «внутренне противоречивыми» и «не поддерживается `optpars`'ом и никогда не будет»;
- `argparse` даст Вам возможность использовать несколько значений переменных у одного аргумента командной строки (nargs);
- `argparse` поддерживает субкоманды (subcommands). Это когда основной парсер отсылает к другому (субпарсеру), в зависимости от аргументов на входе.

Для начала работы с `argparse` необходимо задать парсер:

```
import argparse
parser = argparse.ArgumentParser(description='Great Description To Be Here')
```

Далее, парсеру стоит указать, какие объекты Вы от него ждете. В частном случае, это может выглядеть так:

```
parser.add_argument('-n', action='store', dest='n', help='Simple value')
```

Если действие (action) для данного аргумента не задано, то по умолчанию он будет сохраняться (store) в namespace, причем мы также можем указать тип этого аргумента (int, boolean и тд). Если имя возвращаемого аргумента (dest) задано, его значение будет сохранено в соответствующем атрибуте namespace.

В нашем случае:

```
>>> print(parser.parse_args(['-n', '3']))

Namespace(n='3')

>>> print(parser.parse_args([]))

Namespace(n=None)

>>> print(parser.parse_args(['-a', '3']))
error: unrecognized arguments: -a 3
```

Простой пример программы, возводящей в квадрат значение позиционного аргумента (square) и формирующей вывод в зависимости от аргумента опционального (-v):

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square", type=int, help="display a square of a given number"
)
parser.add_argument(
    "-v", "--verbose", action="store_true", help="increase output verbosity"
)
args = parser.parse_args()

answer = args.square**2
if args.verbose:
    print("the square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

Остановимся на действиях (**actions**). Они могут быть следующими:

- **store:** возвращает в пространство имен значение (после необязательного приведения типа). Как уже говорилось, store — действие по умолчанию;
- **store_const:** в основном используется для флагов. Либо вернет Вам значение, указанное в const, либо (если ничего не указано), None. Пример:

```
>>> parser.add_argument('--LifetheUniverseandEverything', action='store_const',
const=42)
>>> print(parser.parse_args(['--LifetheUniverseandEverything']))

Namespace(LifetheUniverseandEverything=42)
```

- **store_true / store_false:** аналог `store_const`, но для булевых `True` и `False`;
- **append:** возвращает список путем добавления в него значений агрументов. Пример:

```
>>> parser.add_argument('--l', action='append')
>>> print(parser.parse_args('--l a --l b --l Y'.split()))

Namespace(l=['abY'])
```

- **append_const:** возвращение значения, определенного в спецификации аргумента, в список.

- **count:** как следует из названия, считает, сколько раз встречается значение данного аргумента. Пример:

```
>>> parser.add_argument('--verbose', '-v', action='count')
>>> print parser.parse_args('-vvv'.split())

Namespace(verbose=3)
```

В зависимости от переданного в конструктор парсера аргумента `add_help` (булевого типа), будет определяться, включать или не включать в стандартный вывод по ключам ['-h', '--help'] сообщения о помощи. То же самое будет иметь место с аргументом version (строкового типа), ключи по умолчанию: ['-v', '--version']. При запросе помощи или номера версии, дальнейшее выполнение прерывается.

```
parser = argparse.ArgumentParser(add_help=True, version='4.0')
```

Иногда необходимо определить некий набор параметров командной строки, который будет распространяться на все парсеры Вашей программы. В данном случае часто приводят пример необходимости авторизации:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--user', action="store")
>>> parent_parser.add_argument('--password', action="store")
>>> child_parser = argparse.ArgumentParser(parents=[parent_parser])
>>> child_parser.add_argument('--show_all', action="store_true")
>>> print(child_parser.parse_args(['--user', 'guest']))

Namespace(password=None, show_all=False, user='guest')
```

Обратите внимание, что родительский парсер создается с параметром `add_help=False`. Это сделано потому, что каждый парсер будет честно стараться добавить свой обработчик ключа '-h', чем вызовет конфликтную ситуацию. Отсюда возникает вопрос, что делать, если у вашего дочернего парсера имеются те же ключи, что и у родительского и при этом вы хотите их использовать без всяких конфликтов? Делается это простым добавлением аргумента `conflict_handler`:

```
>>> parent_parser = argparse.ArgumentParser(add_help=False)
>>> parent_parser.add_argument('--user', action="store")
>>> parent_parser.add_argument('--password', action="store")
>>> child_parser = argparse.ArgumentParser(parents=[parent_parser],
conflict_handler='resolve')
>>> child_parser.add_argument('--user', action="store", default="Guest")
>>> print(child_parser.parse_args())

Namespace(password=None, user='Guest')
```

Помимо вышеописанного подхода к обработке команд разного уровня, существует еще и альтернативный подход, позволяющий объединить обработку всех команд в одной программе, используя субпарсеры (subparsers).

```
import argparse


```

```
parser = argparse.ArgumentParser()
subparsers = parser.add_subparsers(help='List of commands')

# A list command
list_parser = subparsers.add_parser('list', help='List contents')
list_parser.add_argument('dirname', action='store', help='Directory to list')

# A create command
create_parser = subparsers.add_parser('create', help='Create a directory')
create_parser.add_argument('dirname', action='store', help='New directory to
create')
create_parser.add_argument(
    '--read-only',
    default=False,
    action='store_true',
    help='Set permissions to prevent writing to the directory',
)
```

Вот что выдаст программа с ключом '-h':

```
usage: ap.py [-h] {list,create} ...positional arguments:{list,create} list of
commandslist List contentscreate Create a directoryoptional arguments:-h, --help
show this help message and exit
```

В примере можно отметить следующие вещи:

1. *Позиционные* аргументы list, create, передаваемые программе — по сути субпарсеры.
2. Аргумент `--read-only` субпарсера `create_parser` — *опциональный*, `dir_name` — необходимый для обоих субпарсеров.
3. Предусмотрена справка (помощь) как для парсера, так и для каждго из субпарсеров.

Вообще, argparse — довольно мощная и легкая библиотека, предоставляющая очень удобный интерфейс для работы с параметрами командной строки. Далее будут рассмотрено решение наиболее распространенных практических задач с помощью модуля `argparse`.

Простой разбор аргументов

Почти ничего не сделали.

```
import argparse


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

что получили?

```
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py --verbose
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python3 prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

- Без аргументов ничего не делает.
- Есть помощь, которая запускается по ключам -h или --help
- Если задаем аргументы, которых нет, получаем сообщение об ошибке + usage.

Позиционные аргументы

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument("echo")

args = parser.parse_args()
print(args.echo)
```

запускаем:

```
$ python3 prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python3 prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python3 prog.py foo
foo
```

- функция `add_argument("echo")` добавила позиционный аргумент в список допустимых аргументов.
- `parse_args()` возвращает данные, в атрибуте *echo* этих данных находится позиционный аргумент.
- В помощь попадет без описания
- если он не задан, получаем сообщение об ошибке, какого именно аргумента нет.

Добавим помощь:

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")

args = parser.parse_args()
print(args.echo)
```

получим:

```
$ python3 prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

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

Пусть программа считает квадрат аргумента:

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")

args = parser.parse_args()
print(args.square ** 2)
```

получаем:

```
$ python3 prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print(args.square**2)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Очевидно, что аргументы разбираются как строки. А нужно число.

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square",
    help="display a square of a given number",
    type=int
)

args = parser.parse_args()
print(args.square**2)
```

получим:

```
$ python3 prog.py 4
16
$ python3 prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four
```

Заметьте, что диагностика изменилась. Аргумент есть, но не того типа - тоже проведет к сообщению об ошибке.

Опциональные аргументы

Добавляются той же функцией **add_argument**, начинаются с --.

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")

args = parser.parse_args()
if args.verbosity:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py --verbosity 1
verbosity turned on
$ python3 prog.py
$ python3 prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python3 prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

- Программа написана так, чтобы что-то печатать, когда задано --verbosity и ничего не печатать, когда не задано.
- аргумент не обязательный, можно запускать без него.
- если запустили без него, то значение переменной `args.verbosity` выставляется в `None` и проверка `if args.verbosity дает False`.
- добавлено его описание в секцию optional arguments
- требует параметра после --verbosity (любого типа), иначе не работает.

Но такое поведение удобно, например, для опционального задания конфиг-файла, `-c myconfig.ini`, но не для переменной, которая либо включена (повышенный уровень логирования), либо выключена.

Сделаем ее булевской переменной:

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "--verbose",
    help="increase output verbosity",
    action="store_true"
)

args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py --verbose
verbosity turned on
$ python3 prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python3 prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Теперь опциональный аргумент должен запускаться без параметров и его значение либо `True` (если указан), либо `False` (если не указан).

Помощь изменилась.

Короткие имена -v и --verbosity

Просто перечислите варианты задания аргумента в add_argument:

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "-v",
    "--verbose",
    help="increase output verbosity",
    action="store_true"
)

args = parser.parse_args()
if args.verbose:
    print("verbosity turned on")
```

получим:

```
$ python3 prog.py -v
verbosity turned on
$ python3 prog.py --help
usage: prog.py [-h] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Позиционные и опциональные аргументы вместе

В программу вычисления квадрата числа добавим опцию --verbosity.

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square",
    type=int,
    help="display a square of a given number"
)
parser.add_argument(
    "-v",
    "--verbose",
    action="store_true",
    help="increase output verbosity"
)

args = parser.parse_args()
answer = args.square ** 2
if args.verbose:
    print("the square of {} equals {}".format(args.square, answer))
else:
    print(answer)
```

Запустим программу:

```
$ python3 prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python3 prog.py 4
16
$ python3 prog.py 4 --verbose
the square of 4 equals 16
$ python3 prog.py --verbose 4
the square of 4 equals 16
```

- Вернули позиционный аргумент, значит получили ошибку, что его нет.
- порядок позиционного и опционального аргумента не имеет значения.

Опциональный аргумент с параметром + позиционный аргумент

Сделаем шаг назад. Пусть наш опциональный аргумент имеет параметр - какой уровень логирования будет использован (целое число, как и позиционный аргумент).

Как в этом случае задать и разобрать аргументы?

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square",
    type=int,
    help="display a square of a given number"
)
parser.add_argument(
    "-v",
    "--verbosity",
    type=int,
    help="increase output verbosity"
)

args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запускаем:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python3 prog.py 4 -v 1
4^2 == 16
$ python3 prog.py 4 -v 2
the square of 4 equals 16
$ python3 prog.py 4 -v 3
16
```

Выбор значения из заранее определенного списка

Не нравится, что при задании `--verbosity 3` мы получили 0 уровень. И никто не скажет, какие уровни доступны. Зададим допустимый набор аргументов:

```
import argparse


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

```
    square",
    type=int,
    help="display a square of a given number"
)
parser.add_argument(
    "-v",
    "--verbosity",
    type=int,
    choices=[0, 1, 2],
    help="increase output verbosity"
)

args = parser.parse_args()
answer = args.square ** 2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

optional arguments:
  -h, --help            show this help message and exit
  -v {0,1,2}, --verbosity {0,1,2}
                        increase output verbosity
```

Заметьте, изменились и хелп, и сообщение ошибке.

Подсчет количества заданных аргументов -v action='count'

Хотим определять уровень логирования по тому, сколько раз задали опцию -v в командной строке: ничего, -v, -vv.

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square",
    type=int,
    help="display the square of a given number"
)
parser.add_argument(
    "-v",
    "--verbosity",
```

```
    action="count",
    help="increase output verbosity"
)

args = parser.parse_args()
answer = args.square ** 2
if args.verbosity == 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity == 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запускаем:

```
$ python3 prog.py 4
16
$ python3 prog.py 4 -v
4^2 == 16
$ python3 prog.py 4 -vv
the square of 4 equals 16
$ python3 prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python3 prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python3 prog.py 4 -h
usage: prog.py [-h] [-v] square

positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python3 prog.py 4 -vvv
16
```

- Если не определен ни один флаг -v, то args.verbosity = None
- Без разницы - указыват короткую или полную форму флага -v или --verbosity.
- Слишком большое количество аргументов -vvv дает опять лаконичную форму логирования.

Поправим это, заменив `if args.verbosity == 2` на `if args.verbosity >= 2`.

Увы, если не задано -v, то переменная None и для нее не определено >=.

Добавим опцию **default**=0, чтобы когда аргумент не задан, значение переменной было не None, а 0.

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument(
    "square",
    type=int,
    help="display a square of a given number"
```

```
)
parser.add_argument(
    "-v",
    "--verbosity",
    action="count",
    default=0,
    help="increase output verbosity"
)

args = parser.parse_args()
answer = args.square ** 2
if args.verbosity >= 2:
    print("the square of {} equals {}".format(args.square, answer))
elif args.verbosity >= 1:
    print("{}^2 == {}".format(args.square, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py 4 -vvv
the square of 4 equals 16
$ python3 prog.py 4
16
```

Опциональный аргумент МОЖЕТ содержать параметр

```
>>> parser = argparse.ArgumentParser(prog='PROG')
>>> parser.add_argument('--foo', nargs='?', help='foo help')
>>> parser.add_argument('bar', nargs='+', help='bar help')
>>> parser.print_help()
usage: PROG [-h] [--foo [FOO]] bar [bar ...]

positional arguments:
 bar          bar help

optional arguments:
 -h, --help   show this help message and exit
 --foo [FOO]  foo help
```

Напишем программу, которая возводит не в квадрат, а в указанную степень:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)

args = parser.parse_args()
answer = args.x ** args.y
if args.verbosity >= 2:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
elif args.verbosity >= 1:
    print("{}^{} == {}".format(args.x, args.y, answer))
else:
    print(answer)
```

запустим:

```
$ python3 prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python3 prog.py -h
usage: prog.py [-h] [-v] x y

positional arguments:
  x                the base
  y                the exponent

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python3 prog.py 4 2 -v
4^2 == 16
```

Взаимоисключающие аргументы

Зададим два аргумента: -v - повысить уровень логирования, -q - отключить логирование.

Добавим их в **add_mutually_exclusive_group()**

```
import argparse


parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")

args = parser.parse_args()
answer = args.x ** args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

запустим:

```
$ python3 prog.py 4 2
4^2 == 16
$ python3 prog.py 4 2 -q
16
$ python3 prog.py 4 2 -v
4 to the power 2 equals 16
$ python3 prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python3 prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

Описание программы

argparse.ArgumentParser(description="calculate X to the power of Y"**)**

```
import argparse


parser = argparse.ArgumentParser(
    description="calculate X to the power of Y"
)
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")

args = parser.parse_args()
answer = args.x ** args.y

if args.quiet:
    print(answer)
elif args.verbose:
    print("{} to the power {} equals {}".format(args.x, args.y, answer))
else:
    print("{}^{} == {}".format(args.x, args.y, answer))
```

запустим:

```
$ python3 prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

Пример 1. *Для примера 1 лабораторной работы 2.16 разработайте интерфейс командной строки.*

При построении интерфейса командной строки мы не можем постоянно держать данные в оперативной памяти как это было сделано в предыдущей лабораторной работе. Поэтому имя файла JSON с данными программы должно быть одним из обязательных позиционных аргументов командной строки.

Добавим также следующие подкоманды:

- add – добавление рабочего, имя которого задано в аргументе с параметром `--name`, должность – в аргументе с параметром `--post`, а год поступления – в аргументе с параметром `--year`.
- display – отображение списка всех работников.
- select – выбор и отображение требуемых работников, у которых заданный период передается через аргумент с параметром `--period`.

Напишем программу для решения поставленной задачи.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import argparse
import json
import os.path
import sys
from datetime import date


def add_worker(staff, name, post, year):
    """
    Добавить данные о работнике.
    """
    staff.append(
        {
            "name": name,
            "post": post,
            "year": year
        }
    )

    return staff


def display_workers(staff):
    """
    Отобразить список работников.
    """
    # Проверить, что список работников не пуст.
    if staff:
        # Заголовок таблицы.
        line = '+-{}-+-{}-+-{}-+-{}-+'.format(
            '-' * 4,
            '-' * 30,
            '-' * 20,
            '-' * 8
        )
```

```
        print(line)
        print(
            '| {:^4} | {:^30} | {:^20} | {:^8} |'.format(
                "№",
                "Ф.И.О.",
                "Должность",
                "Год"
            )
        )
        print(line)

        # Вывести данные о всех сотрудниках.
        for idx, worker in enumerate(staff, 1):
            print(
                '| {:>4} | {:<30} | {:<20} | {:>8} |'.format(
                    idx,
                    worker.get('name', ''),
                    worker.get('post', ''),
                    worker.get('year', 0)
                )
            )
            print(line)

    else:
        print("Список работников пуст.")


def select_workers(staff, period):
    """
    Выбрать работников с заданным стажем.
    """
    # Получить текущую дату.
    today = date.today()

    # Сформировать список работников.
    result = []
    for employee in staff:
        if today.year - employee.get('year', today.year) >= period:
            result.append(employee)

    # Возвратить список выбранных работников.
        return result


def save_workers(file_name, staff):
    """
    Сохранить всех работников в файл JSON.
    """
    # Открыть файл с заданным именем для записи.
    with open(file_name, "w", encoding="utf-8") as fout:
        # Выполнить сериализацию данных в формат JSON.
        # Для поддержки кирилицы установим ensure_ascii=False
        json.dump(staff, fout, ensure_ascii=False, indent=4)


def load_workers(file_name):
    """
    Загрузить всех работников из файла JSON.
```

```
    """
    # Открыть файл с заданным именем для чтения.
    with open(file_name, "r", encoding="utf-8") as fin:
        return json.load(fin)


def main(command_line=None):
    # Создать родительский парсер для определения имени файла.
    file_parser = argparse.ArgumentParser(add_help=False)
    file_parser.add_argument(
        "filename",
        action="store",
        help="The data file name"
    )

    # Создать основной парсер командной строки.
    parser = argparse.ArgumentParser("workers")
    parser.add_argument(
        "--version",
        action="version",
        version="%(prog)s 0.1.0"
    )

    subparsers = parser.add_subparsers(dest="command")

    # Создать субпарсер для добавления работника.
    add = subparsers.add_parser(
        "add",
        parents=[file_parser],
        help="Add a new worker"
    )
    add.add_argument(
        "-n",
        "--name",
        action="store",
        required=True,
        help="The worker's name"
    )
    add.add_argument(
        "-p",
        "--post",
        action="store",
        help="The worker's post"
    )
    add.add_argument(
        "-y",
        "--year",
        action="store",
        type=int,
        required=True,
        help="The year of hiring"
    )

    # Создать субпарсер для отображения всех работников.
    _ = subparsers.add_parser(
        "display",
        parents=[file_parser],
        help="Display all workers"
```

```
    )

    # Создать субпарсер для выбора работников.
    select = subparsers.add_parser(
        "select",
        parents=[file_parser],
        help="Select the workers"
    )
    select.add_argument(
        "-P",
        "--period",
        action="store",
        type=int,
        required=True,
        help="The required period"
    )

    # Выполнить разбор аргументов командной строки.
    args = parser.parse_args(command_line)

    # Загрузить всех работников из файла, если файл существует.
    is_dirty = False
    if os.path.exists(args.filename):
        workers = load_workers(args.filename)
    else:
        workers = []

    # Добавить работника.
    if args.command == "add":
        workers = add_worker(
            workers,
            args.name,
            args.post,
            args.year
        )
        is_dirty = True

    # Отобразить всех работников.
    elif args.command == "display":
        display_workers(workers)

    # Выбрать требуемых рааботников.
    elif args.command == "select":
        selected = select_workers(workers, args.period)
        display_workers(selected)

    # Сохранить данные в файл, если список работников был изменен.
    if is_dirty:
        save_workers(args.filename, workers)


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

Результаты работы программы:

1. Для добавления нового работника необходимо выполнить следующую команду в терминале:

```
$ python workers.py add data.json --name="Сидоров Сидор" -- post="Главный инженер" --year=2012
```

Данная команда добавляет нового работника *Сидоров Сидор* с должностью *Главный инженер*, годом начала работы *2012* и записывает результаты операции в файл *data.json*. Пример содержимого файла *data.json* после добавления работника:

```
[
    {
        "name": "Иванов Иван",
        "post": "Директор",
        "year": 2007
    },
    {
        "name": "Петров Петр",
        "post": "Бухгалтер",
        "year": 2010
    },
    {
        "name": "Сидоров Сидор",
        "post": "Главный инженер",
        "year": 2012
    }
]
```

2. Для отображения информации обо всех работниках необходимо выполнить команду:

```
$ python workers.py display data.json
+------+----------------------------------+----------------------+----------+
|  №   |             Ф.И.О.               |      Должность       |   Год    |
+------+----------------------------------+----------------------+----------+
|    1 | Иванов Иван                      | Директор             |     2007 |
+------+----------------------------------+----------------------+----------+
|    2 | Петров Петр                      | Бухгалтер            |     2010 |
+------+----------------------------------+----------------------+----------+
|    3 | Сидоров Сидор                    | Главный инженер      |     2012 |
+------+----------------------------------+----------------------+----------+
```

3. Для выбора работников с заданном периодом работы необходимо выполнить команду:

```
$ python workers.py select data.json --period=12
+------+----------------------------------+----------------------+----------+
|  №   |             Ф.И.О.               |      Должность       |   Год    |
+------+----------------------------------+----------------------+----------+
|    1 | Иванов Иван                      | Директор             |     2007 |
+------+----------------------------------+----------------------+----------+
```

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.

2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.
3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Проработайте примеры лабораторной работы. Создайте для них отдельные модули языка Python. Зафиксируйте изменения в репозитории.
8. Приведите в отчете скриншоты результатов выполнения примера при различных исходных данных вводимых с клавиатуры.
9. Зафиксируйте сделанные изменения в репозитории.
10. Приведите в отчете скриншоты работы программ решения индивидуальных заданий.
11. Зафиксируйте сделанные изменения в репозитории.
12. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
13. Выполните слияние ветки для разработки с веткой *master*/*main*.
14. Отправьте сделанные изменения на сервер GitHub.
15. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Индивидуальные задания

Задание

Для своего варианта лабораторной работы 2.16 необходимо дополнительно реализовать интерфейс командной строки (CLI).

Задание повышенной сложности

Самостоятельно изучите работу с пакетом `click` для построения интерфейса командной строки (CLI). Для своего варианта лабораторной работы 2.16 необходимо реализовать интерфейс командной строки с использованием пакета `click`.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется письменно в рабочей тетради, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. В чем отличие терминала и консоли?
2. Что такое консольное приложение?
3. Какие существуют средства языка программирования Python для построения приложений командной строки?
4. Какие особенности построение CLI с использованием модуля `sys`?
5. Какие особенности построение CLI с использованием модуля `getopt`?
6. Какие особенности построение CLI с использованием модуля `argparse`?