Arguments parser

Command line arguments are those values that are passed during calling of program along with the calling statement.

General information

Almost every modern programming language is capable of accepting command line arguments. This is a very important feature because it allows dynamic input from users, whether they have written the program or not.

As you probably know, all programs can be conditionally divided into console programs and those using a graphical interface (we will not take into account services under Windows and daemons under Linux). Command line startup parameters are most often used by console programs, although GUI programs do not hesitate to do so either. Surely in the life of every programmer there was a situation when it was necessary to parse command line parameters, as a rule, this is not the most interesting part of the program, but one cannot do without it.

Standards

A few available standards provide some definitions and guidelines to promote consistency for implementing commands and their arguments. These are the main UNIX standards and references:

- POSIX Utility Conventions
- GNU Standards for Command Line Interfaces
- docopt

The standards above define guidelines and nomenclatures for anything related to programs and Python command line arguments.

Standards: POSIX

- A program or utility is followed by options, option-arguments, and operands.
- All options should be preceded with a hyphen or minus (-) delimiter character.
- Option-arguments should not be optional.

Standards: GNU

- All programs should support two standard options, which are --version and --help.
- Long-named options are equivalent to the single-letter Unix-style options. An example is --debug and -d.

Standards: docopt

- Short options can be stacked, meaning that -abc is equivalent to -a -b -c.
- Long options can have arguments specified after a space or the equals sign (=). The long option --input=ARG is equivalent to --input ARG.

Types of Python argument parser

Python contains several libraries that allow code to accept user input from a command:

- **sys.argv**
- **getopt**
- **argparse**

At the moment, argparse is the best and most common option.

1. sys.argv

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

Thus, the first element of the array sys.argv() is the name of the program itself. sys.argv() is an array for command line arguments in Python. To employ this module named “sys” is used. sys.argv is similar to an array and the values are also retrieved like Python array.

1. sys.argv

```
import sys

print("This is the name of the program:", sys.argv[0])

print("Argument List:", str(sys.argv))

# results

$ python3 test.py 1 2 3 4

This is the name of the program: test.py

Argument List: ['test.py', '1', '2', '3', '4']
```

The above program has been saved by the name “test.py” and thus has to be called in the following in command prompt

1. sys.argv

Functions that can be used with sys.argv

- len()- function is used to count the number of arguments passed to the command line. Since the iteration starts with 0, it also counts the name of the program as one argument. If one just wants to deal with other inputs they can use (len(sys.argv)-1).
- str()- this function is used to present the array as a string array. Makes displaying the command line array easier and better.

1. sys.argv

```
import sys

print("This is the name of the program:",
    sys.argv[0])
print("Number of elements including the name of the program:",
    len(sys.argv))
print("Number of elements excluding the name of the program:",
    (len(sys.argv)-1))
print("Argument List:",
    str(sys.argv))
```

Results:
$ python3 test.py 1 2 3 4

This is the name of the program: test.py

Number of elements including the name of the program: 5

Number of elements excluding the name of the program: 4

Argument List: ['test.py', '1', '2', '3', '4']

1. sys.argv

Example of usage:

```
import sys

add = 0.0

# Getting the length of command
# line arguments
n = len(sys.argv)

for i in range(1, n):
        add += float(sys.argv[i])

print ("the sum is :", add)
```

Results:

python3 test.py 1 2 3 4

the sum is : 10.0

1. sys.argv: Handling Errors

Python command line arguments are loose strings. Many things can go wrong, so it's a good idea to provide the users of your program with some guidance in the event they pass incorrect arguments at the command line. For example, reverse.py expects one argument, and if you omit it, then you get an error:

```
$ python reverse.py

Traceback (most recent call last):

  File "reverse.py", line 5, in <module>

    arg = sys.argv[1]

IndexError: list index out of range
```

1. sys.argv: Handling Errors

The Python exception IndexError is raised, and the corresponding traceback shows that the error is caused by the expression arg = sys.argv[1]. The message of the exception is list index out of range. You didn't pass an argument at the command line, so there's nothing in the list sys.argv at index 1.

This is a common pattern that can be addressed in a few different ways. For this initial example, you'll keep it brief by including the expression arg = sys.argv[1] in a try block. Modify the code as follows:

```
import sys

try:
   arg = sys.argv[1]
except IndexError:
   raise SystemExit(f"Usage: {sys.argv[0]} <string_to_reverse>")
print(arg[::-1])
```

2. getopt

The getopt module is a parser for command-line options based on the convention established by the Unix getopt() function. It is in general used for parsing an argument sequence such as sys.argv. In other words, this module helps scripts to parse command-line arguments in sys.argv. It works similar to the C getopt() function for parsing command-line parameters.

getopt.getopt() takes the following arguments:

- The usual arguments list minus the script name, sys.argv[1:]
- A string defining the short options
- A list of strings for the long options

Note that a short option followed by a colon (:) expects an option argument, and that a long option trailed with an equals sign (=) expects an option argument.

2. getopt

Result:
python3 test.py -f Anna -l Black
Anna Black
Here we have created a function full_name(), which prints the full name after getting first name and last name from the command line. We have also abbreviated first name as ‘f’ and last name as ‘l’.

```
try:
    opts, args = getopt.getopt(argv, "f:l:",
                    ["first_name =",
                     "last_name ="])
except:
    print("Error")
 for opt, arg in opts:
    if opt in ['-f', '--first_name']:
       first_name = arg
    elif opt in ['-l', '--last_name']:
       last_name = arg
```

```
import sys
import getopt

def full_name():
   first_name = None
   last_name = None

   argv = sys.argv[1:]
   try:
      opts, args = getopt.getopt(argv, "f:l:")
   except:
      print("Error")

   for opt, arg in opts:
      if opt in ['-f']:
         first_name = arg
      elif opt in ['-l']:
         last_name = arg
   print( first_name +" " + last_name)

full_name()
```

3. argparse

The argparse Python library was released as part of the standard library along with Python 3.2. Since this release, due to its popularity, it has been integrated into Python 2.7 and all future versions of Python, quickly becoming the gold standard for command-line arguments. It provides the following features:

- Ability to customize help messages and documentation for command-line arguments.
- Setting default values for arguments in a clean and readable way.
- The presence of a specified number of options for a specific command-line argument.
- Support for a variable number of parameters for one argument.
- The ability to automatically apply an action or function to input based on specific instructions.

3. argparse: Basic setup

```
import argparse
parser = argparse.ArgumentParser(description='A tutorial of argparse!')
args = parser.parse_args()
```

This code has 3 important components:

- Import argparse.
- Creation of an argument parser (with a description).
- Parsing command line arguments.

3. argparse: Basic setup

Now let's add a command line argument named “a” as shown in the code snippet below. To pass a command line argument to a Python script, run it with python3 argparse.py --a = 5:

```
import argparse
parser = argparse.ArgumentParser (description = 'A tutorial of argparse!')
parser.add_argument ("- a")
args = parser.parse_args ()
a = args.a
print (a)
```

Note the use of the .add_argument () function to pass "a" as a command line argument. To access the variable obtained from a, use args.a.

Note that without specifying a value for a on the command line, args.a will be None.

3. argparse

```
import argparse
parser = argparse.ArgumentParser (description = 'A tutorial of argparse!')
parser.add_argument ("- a", default = 1, help = "This is the 'a' variable")
args = parser.parse_args ()
a = args.a
print (a)
```

In this case, if we don't pass a value for a via the command line, then a will default to 1. By adding a line for the help variable, we can also print a more efficient description for each variable using --help:

```
usage: run.py [-h] [--a A]
A tutorial of argparse!
optional arguments:
  -h, --help show this help message and exit
  --a A This is the 'a' variable
```

3. argparse

We can provide information about the type of each variable so that the type conversion is performed directly on the input:

```
import argparse
parser = argparse.ArgumentParser (description = 'A tutorial of argparse!')
parser.add_argument ("- a", default = 1, type = int, help = "This is the 'a' variable")
parser.add_argument ("- name", required=True, type = str, help = "Your name")
args = parser.parse_args ()
```

The required keyword can be used to ensure that the user always passes a value for a specific argument. The set value True forces the user to enter data only for this value, otherwise the program will generate an error and stop.

3. argparse

We can also enter possible values for a specific command line argument using the choices argument. A useful feature, especially when you have a set of if-else statements in your code that perform certain operations on a single line basis. Example:

```
import argparse
parser = argparse.ArgumentParser (description = 'A tutorial of argparse!')
parser.add_argument ("- a", default = 1, type = int, help = "This is the 'a' variable")
parser.add_argument ("- education",
        choices = ["highschool", "college", "university", "other"],
        required = True, type = str, help = "Your name")
args = parser.parse_args ()
ed = args.education
if ed == "college" or ed == "university":
   print ("Has degree")
elif ed == "highschool":
 print ("Finished Highschool")
else:
   print ("Does not have degree")
```

Now, if you enter any value in the choices list, the code will work and will accept the education argument. However, if you enter something that is not on the list, such as the number 5, you will receive the following message, which prompts you to select options from the list:

```
usage: run.py [-h] [--a A] --education {highschool, college, university, other}
run.py: error: argument --education: invalid choice: '5' (choose from 'highschool', 'college', 'university', 'other')
```

3. argparse

The action argument can be used to specify the action to be taken by the argument parser. For example, an argument that is automatically set to the boolean value True if present or constant. The example below shows both cases.

```
import argparse
parser = argparse.ArgumentParser (description = 'A tutorial of argparse!')
parser.add_argument ("- a", action = "store_true", help = "This is the 'a' variable")
parser.add_argument ("- b", action = "store_const", const = 10,
        help = "This is the 'b' variable")
args = parser.parse_args ()
a = args.a
b = args.b
print (a)
print (b)
```

The above code says that if a argument is present on the command line, then it will be True, otherwise False. Likewise, if the b command line argument is present, it must be 10, otherwise None due to the lack of a default!

3. argparse

There are several actions that are already defined and ready to be used. Let's analyze them in detail:

- store stores the input value to the Namespace object. (This is the default action.)
- store_const stores a constant value when the corresponding optional arguments are specified.
- store_true stores the Boolean value True when the corresponding optional argument is specified and stores a False elsewhere.
- store_false stores the Boolean value False when the corresponding optional argument is specified and stores True elsewhere.
- append stores a list, appending a value to the list each time the option is provided.
- append_const stores a list appending a constant value to the list each time the option is provided.
- count stores an int that is equal to the times the option has been provided.
- help shows a help text and exits.
- version shows the version of the program and exits.

3. argparse

```
import argparse

my_parser = argparse.ArgumentParser()
my_parser.version = '1.0'
my_parser.add_argument('-a', action='store')
my_parser.add_argument('-b', action='store_const', const=42)
my_parser.add_argument('-c', action='store_true')
my_parser.add_argument('-d', action='store_false')
my_parser.add_argument('-e', action='append')
my_parser.add_argument('-f', action='append_const', const=42)
my_parser.add_argument('-g', action='count')
my_parser.add_argument('-i', action='help')
my_parser.add_argument('-j', action='version')

args = my_parser.parse_args()

print(vars(args))
```

3. argparse

As you have already seen, when you add an argument to the parser, the value of this argument is stored in a property of the Namespace object. This property is named by default as the first argument passed to .add_argument() for the positional argument and as the long option string for optional arguments.

However, it's possible to specify the name of this property just by using the keyword dest when you're adding an argument to the parser:

```
import argparse

my_parser = argparse.ArgumentParser()
my_parser.add_argument('-v',
                       '--verbosity',
                       action='store',
                       type=int,
                       dest='my_verbosity_level')

args = my_parser.parse_args()
print(vars(args))
```

By running this program, you'll see that now the args variable contains a .my_verbosity_level property, even if by default the name of the property should have been .verbosity:

Serialization

Data serialization is the process of converting structured data to a format that allows sharing or storage of the data in a form that allows recovery of its original structure. In some cases, the secondary intention of data serialization is to minimize the data's size which then reduces disk space or bandwidth requirements.

Flat vs. Nested data

Before beginning to serialize data, it is important to identify or decide how the data should be structured during data serialization - flat or nested. The differences in the two styles are shown in the below examples.

Flat style:

```
{ "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }
```

Nested style:

```
{"A"
  { "field1": "value1", "field2": "value2", "field3": "value3" }}
```

Serializing Text

Simple file (flat data)
If the data to be serialized is located in a file and contains flat data, Python offers two methods to serialize data.
repr
The repr method in Python takes a single object parameter and returns a printable representation of the input:

```
# input as flat text
a =  { "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }

# the same input can also be read from a file
a = open('/tmp/file.py', 'r')

# returns a printable representation of the input;
# the output can be written to a file as well
print(repr(a))

# write content to files using repr
with open('/tmp/file.py') as f:f.write(repr(a))
```

Serializing Text

CSV file (flat data)
The CSV module in Python implements classes to read and write tabular data in CSV format.

Simple example for reading:

```
# Reading CSV content from a file
import csv
with open('/tmp/file.csv', newline='') as f:
  reader = csv.reader(f)
  for row in reader:
    print(row)
```

Simple example for writing:

```
# Writing CSV content to a file
import csv
with open('/temp/file.csv', 'w', newline='') as f:
  writer = csv.writer(f)
  writer.writerows(iterable)
```

Serializing Text

YAML (nested data)

There are many third party modules to parse and read/write YAML file structures in Python. One such example is below.

```
# Reading YAML content from a file using the load method
import yaml
with open('/tmp/file.yaml', 'r', newline='') as f:
  try:
    print(yaml.load(f))
  except yaml.YAMLError as ymlexcp:
    print(ymlexcp)
```

Serializing Text

JSON file (nested data)
Python's JSON module can be used to read and write JSON files. Example code is below.

Reading:

```
# Reading JSON content from a file
import json
with open('/tmp/file.json', 'r') as f:
  data = json.load(f)
```

Writing:

```
# Writing JSON content to a file using the dump method
import json
with open('/tmp/file.json', 'w') as f:
  json.dump(data, f, sort_keys=True)
```

Serializing Text

XML (nested data)

XML parsing in Python is possible using the xml package.

```
# reading XML content from a file
import xml.etree.ElementTree as ET
tree = ET.parse('country_data.xml')
root = tree.getroot()
```