Python in the Real World Project Proposal - Docopt

Chengen Li
Kevin Williams
CIS5930

Project Description

Docopt is an existing Python program which makes it easy for users to create their own command line interfaces. To do so, users provide a description of their interface (usually in the docstring) containing usage patterns and option descriptions. By parsing the provided description, Docopt can create and fill out a dictionary containing all the arguments, options, and commands described in the docstring and the values for these arguments provided by the user upon any command line execution. When someone imports docopt, they can use the resulting dictionary to handle their own command line interfaces easily.

Our goal is to create our own version of Docopt from the ground up, barring a few advanced features. Using the Docopt source code as a reference, we can build a similar program for creating command line interfaces.

Usage of Docopt

- Takes 5 arguments
    - doc (str): contains text description of interface (usually within docstring, or __doc__)
    - Argv: optional argument vector to supply command line arguments
        - By default, argv takes input from command line, or sys.argv[1:]
        - Can supply a list of strings
    - Help: bool value which specifies whether the parser should automatically print the help message, defaults to True
        - Print whatever's specified in doc
    - Version: specifies version of program, defaults to None
    - Options_first: bool value specifying whether to allow mixing of options and positional arguments
        - If True, after first positional argument is found, all subsequent arguments will be interpreted as positional

Pattern Format

- Find substring of doc starting with "usage" (case insensitive) and ending with an empty line
- First word after "usage" interpreted as program name
    - Each line with program name signifies exclusive patterns
- Each pattern, or line, consists of the following:
    - Arguments: specified as either UPPER-CASE words or surrounded by <angular-brackets>

    - Options: started with a dash (‘-’ or ‘--’)
        - Can stack one-letter options
        - Must be lowercase
    - Commands: words that do not follow above conventions
        - Also includes special commands ‘-’ and ‘--’
- Specify patterns
    - Brackets ([ ]): optional elements
    - Parenthesis (( )): required elements
        - Elements not in [ ] are also required
    - Pipe (|): mutually exclusive elements
        - Group using () if one of the elements is required
        - Group using [] if both of the elements are optional
    - Ellipsis (...): One or more elements
        - Unary operator on the expression to the left
        - FILE … indicates one or more FILEs
        - [FILE …] indicates zero or more FILEs
    - [options] (case sensitive): shortcut for any options
        - Signifies that any of the options described below in the options description could be used in the usage pattern

To be Implemented

- Main function takes one required argument for parsing the docstring in the python program. Users can either specify the docstring in this parameter or use the default docstring from the python file. Three parts (name, Usage:, and Options:) will be separated by a new line. First line is always the name of the program “‘’name”. All tabs are case sensitive.
- Four optional arguments will be taken from the main function
    1. “Help” argument for whether the program should print the help messages automatically.
    2. “Version” argument for specifying the version of such a program.
    3. “Argv” argument for programmers to specify the parameters.
- Receive the docstring from python file or from the string passed in from the main function
    1. Print the usage example right after the program is executed if the programmer specifies help=true.
    2. Print all the options right after the program is executed if the programmer specifies help=true.
- Parse the usage strings
    1. Receive the name of the python program.
    2. Receive the required arguments for running the program.
    3. Receive the required arguments with options inside ‘()’ and specify the options by a ‘|’
    4. Receive the optional arguments inside ‘[]’.
    5. Receive the optional arguments with choices inside ‘[]’ and separating the options by a ‘|’.

6. The values that pass into the program will be surrounded by the '<value>'
7. From one or more values, the format would be '<value>...', and for one or more would be '[<value>...]'.

- Parse the options strings
    1. The format of arguments is either in the form "-o".
    2. The format of arguments with a value is in "-o=<value>".
    3. The format for multiple argument value pairs all pairs will be separated by a comma (-o=<value1>, -o=<value2>).
    4. The format for one argument with multiple values will be for one or more "-o=<value...>" and for zero or more "-o=<[value..]>"
- Output is in a JSON format to show users what argument has activities by the command line.
- If the programmer violates the format of the docstring, an error message will show.

Approach

1. Create main function docopt taking 1 required argument (doc) and 3 optional arguments (doc, argv, help, version)
2. Parse doc
    a. Find usage pattern as substring of doc starting with usage (case insensitive) and ending with an empty line
    b. Retrieve program name -> first word after "usage"
        i. Every subsequent line in the usage section must start with the program name
    c. For each line, retrieve arguments (UPPER or <>), options (starts with - or --), and commands (don't follow above conventions)
        i. Starting from an empty dictionary, add any new arguments, options, and commands you find (NO REPEATS)
        ii. By the end of parsing each line, we should have a comprehensive dict identifying all the possible command line arguments
3. Compare command line arguments with existing patterns
    a. The goal here is to find a matching pattern for the given command line arguments
    b. Use count of argv to eliminate any nonmatching patterns off the bat
    c. For efficiency, start by looking for keyword commands
        i. In the example provided by the source code, one such keyword is "tcp". Since argv contains "tcp", we know to try to match with the first usage pattern
    d. Next, look for options (tokens starting with '-' or '--')
        i. For any option found, look in the options description to see if the option has an argument and/or default value
            1. If so, the token after the option in argv is to be considered the option argument
    e. If no matching pattern is found, raise an error
4. Fill out dictionary with appropriate values
    a. Now that we have a matching usage pattern, it's time to fill in the arguments

    b. Different arguments have different types, so be cognisant of that
    c. For commands, like “tcp”, the value in the dict will be either True or False, depending on whether or not the command is present in argv
    d. Options can have either boolean values (no option argument) or string/float values (with option argument)
        i. Check option description section
    e. Positional arguments (<> or UPPER) have string/float values
        i. Ex: In the example within the source code, positional arguments <host> and <port> have values ‘127.0.0.1’ and ‘80’, respectively
5. Return completed dictionary to the user