Command-line Parser

--a brief introduction to argparse module

--Created by David Song

Contents

Introduction

This document will briefly describe the usage of the argparse module.

1. Command line input parameters in sys.argv

Python parameters are passed in through sys.argv. Suppose a python file named `my_app.py` which takes two input parameters, below is the running command:

```
python my_app.py 1 2
```

Inside the `my_app.py`, the sys.argv is a list of strings: `['my_app.py, '1', '2']`

Position `[0]` holds the program file name, the rest are the input parameters start from position 1.

Basically, we need to validate the input before continuing the execution. If the input is invalid, print the program's usage information.

Below is the `my_app.py` code:

```
def main(argv):

    if len(argv)<3:
        raise Exception("need more parameters")
        printUsage()

    p1 = argv[1]
    p2 = argv[2]


if __name__ == '__main__':

    print (sys.argv)
    main(sys.argv)
```

2. The argparse module

Use the argparse module, change above code in main() like below:

```
import argparse

def main(argv):

    parser = argparse.ArgumentParser()
    parser.add_argument("p1", help="parameter 1", type=int)

    parser.add_argument("p2", help="parameter 2", type=int)

    args = parser.parse_args()

    print('p1={}; p2={}'.format(args.p1, args.p2))
```

The above code tells it needs two mandatory parameters from command line, and converts them to int. If input parameters are not correct, it will throw exceptions, print out messages and exist.

The parse_args() takes sys.argv as default, you can pass in any string arrays (or list) into the function call as its parameter.

Inside the add_argument method, the first is the name of the parameter will be used, the help parameter is used to display usage when running python my_app.py --help; The type=int means please covert the input from string to an integer. Only the name parameter is required, others are optional.

```
print (args) -> Namespace(p1=1, p2=2)
```

The above code shows the usage of positional arguments, the sequence of add_argument statements defines the order of expected mandatory input parameters. The add_argument simply means “I am expecting an input parameter and gives it a name afterward “

Optional Arguments

Use -- (double dash) for an optional argument or - (single dash) for short.

```
parser = argparse.ArgumentParser()
parser.add_argument("-p1")
parser.add_argument("--p2")

args = parser.parse_args()
print (args)
Run command:

python my_app.py -p1 1 --p2 2
```

-> Namespace(p1=1, p2=2)

```
python my_app.py -p1 1
```

-> Namespace(p1='1', p2=None)

Without giving any optional arguments, it is still valid:

python my_app.py

-> Namespace(p1=None, p2=None)

Note: double dash (--) or single dash (-) needs to be synchronized in command line and add_argument() method, meaning if single dash (-) is given in add_argument() method, then single dash (-) needs to be used in the command line, and vice versa.

Mandatory option

```
parser.add_argument("-p1", required=True)
```

without the -p1 parameter in the command line will cause an error:

Run command: python my_app.py

Output: my_app.py: error: argument -p1 is required

Why using mandatory option? This is because sometimes developers want to formalize the input arguments using the dash (-) as prefix for an argument name, follows by the argument value; The purpose is to make the input arguments in command line easy to read, especially when there are multiple input arguments, the command line is long.

Action

```
parser.add_argument("-p1", required=True)
parser.add_argument("-p2", action="store_true")
parser.add_argument("-p3", action='store_const', const=3 )
```

Run command: python my_app.py -p1 1 -p2 -p3

```
-> Namespace(p1='1', p2=True, p3=3)
```

Action default value is store, which means to take the corresponding command line argument and store its value to the given name. When using `action="store_true"` or any words after store (`action="store_*"`) , then don't give any value in the command line after the argument name, as shown in the above example.

Limiting argument value

```
parser.add_argument('-p1', choices=[1,2,3], type=int)
```

Run command: python my_app.py -p1 1

```
-> Namespace(p1=1)
```

Giving any values other than those values specified in the choices to p1 will cause an error.

Repeating values

```
parser.add_argument('-p1', required=True, nargs='+', type=int )
```

Run command: python my_app.py -p1 1 2 3

```
-> Namespace(p1=[1, 2, 3])
```

You can use a specific number instead of using wild cards, e.g. `nargs='3'`.

Summary

This section will show an example to illustrate commonly used `argparse` features, for more information, please refer to https://docs.python.org/3/library/argparse.html

```
import sys
import argparse

def main(argv):

    parser = argparse.ArgumentParser(prog="my_app",
                                     description="demostrate how to use argparse")

    parser.add_argument('-p1', dest='p1', required=True,
                        choices=[1,2,3], type=int, help='parameter 1')

    parser.add_argument('-p2', action='store_true', dest='p2', help='parameter 2')

    parser.add_argument('-p3', action='store_const', const=3, dest='p3',
                        help='parameter 3')

    parser.add_argument('-p4', dest='p4', nargs=2, help='parameter 4')

    args = parser.parse_args()
    print (args)

if __name__ == '__main__':

    print (sys.argv)
    main(sys.argv)
```

Run command: python my_app.py -p1 1 -p2 -p3 -p4 4 5

```
->
['my_app.py', '-p1', '1', '-p2', '-p3', '-p4', '4', '5']
Namespace(p1=1, p2=True, p3=3, p4=['4', '5'])
```

Explaination:

1. Create a parser with the application name, and description
2. dest – store the argument name that will be used alternatively, default is the name after the dash. This is optional, normally we don't need it.

To show the usage of this application, run command: python my_app.py -h, will show:

```
usage: my_app [-h] [-p1 {1,2,3}] [-p2] [-p3] [-p4 P4 P4]
demostrate how to use argparse

optional arguments:
  -h, --help   show this help message and exit
  -p1 {1,2,3}  parameter 1
  -p2          parameter 2
  -p3          parameter 3
  -p4 P4 P4    parameter 4
```

Final words, there are so many ways or features to use the `argparse` module, but keep in mind don't make it too fancy to confuse yourself and other people, it will make your application error prone.