parsing command line arguments

Parsing command line arguments

when we called -range.

Let's load that into memory.

You see we overwrote the file,

so we can run it with the four again.

We expect that to work like it

did previously and it does.

And then, we're going to use the -r this option.

And it takes two arguments,

the starting and ending points of the range.

And now, I get items

bigger than 500 but smaller than a 1000.

You can see I can call it with the --range as well.

It's the same, they're interchangeable.

And I can pick more items too,

because I still have my Count.

When I run with nothing,

7:18 / 11:44

1.0x HD

Video

Download video file

Transcripts

Download SubRip (.srt) file

Download Text (.txt) file

Concepts

Like any other program, Python scripts can be executed from a terminal. They can also be executed with command line arguments and options. These arguments and options can be used within the script to control the flow of the program. The arguments and options are captured by an environment variable `argv`, which can be accessed using the `sys` module.

The `argv` captures the command line arguments and options as a list. The first element `argv[0]` is always the command itself, which is the name of the script file. The rest of the list elements are the arguments and options. It is possible to process these arguments and options by writing code around the `argv` list; however, it is a daunting and tedious task. Command line arguments are used by many applications; therefore, Python standard library provides an `argparse` module that is much more robust and versatile and will make parsing command line argument very easy.

You will explore how to use the `argparse` module in the following examples, where you will develop a script `rand.py` that will generate random integers according to the arguments and options passed from the command line.

Examples

Change working directory to `command_line`

Necessary so all generated files are saved in this directory, the cell will generate an error message if you are already in the `command_line` directory.

```
%cd command_line
```

`argv` environment variable

```
%%writefile command_line.py

import sys

# number of arguments
argc = len(sys.argv)
print(argc, "arguments and options were passed")

# list of arguments and options
print("The arguments and options passed are: ")
for i in range(argc):
    print("argv[{:d}] = {}".format(i, sys.argv[i]))
```

Running the `command_line.py` script will generate:

```
%%bash

python3 command_line.py arg1 arg2 -option1 -option2
```

Generating random numbers

In the following examples, you will build a program `rand.py` to:

- print out a random integer between 0 and 10
- print out a number of random integers between 0 and 10, where the number is passed as a command line argument
- print out a number or random integers in a specific range; the number of random numbers and the range limits are passed as command line arguments
- print out a number or random integers in a specific range with an optional message; the number of random numbers, the range limits, and the option to print the message are all passed as command line arguments

NOTE: In the following examples, the `bash` executions must be run after the code segments that precede them, changing the order will result in errors and undesired output

`argparse` module

print out a random integer between 0 and 10

In this program, the `argparse` module is imported to define an object of type `argparse.ArgumentParser`, then parse the command line arguments using `parse_args()`

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# parse command line arguments
args = parser.parse_args()

# program
print(randint(0, 10))
```

Running the script from a terminal will generate:

```
%%bash

python3 rand.py
```

The program prints out a random number between 0 and 10 as expected. However, if we pass an unrecognized argument to the script, the `argparse` module will generate an appropriate usage message and build a help page automatically.

```
%%bash

python3 rand.py -i
```

```
%%bash

python3 rand.py -h
```

Adding arguments (`add_argument`)

print out a number of random integers between 0 and 10, where the number is passed as a command line argument

In this program, a `count` argument is added using the `add_argument` method. The `count` argument holds the number of random int to print.

- If `count` is not provided by the user, the script won't work and a user will be presented with a usage message
- The help is updated accordingly
- The argument passed is stored in `args.count`
- When passing 4 as an argument, the script generates 4 random numbers as expected
- The `add_argument` method takes several parameters:
  - Name of argument, and it is also the name of the variable storing the count.
  - 'type' of the argument, if not specified the default will be string
  - 'help' message to be displayed when a user requests the help message by the `-h` option
- The `add_argument` takes more optional parameters depending on the way you want to capture the arguments, we will explore a few more in the next examples.

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random i

# parse command line arguments
args = parser.parse_args()

# program
for i in range(args.count):
    print(randint(0, 10))
```

Running the script from a terminal will generate:

```
%%bash

python3 rand.py
```

```
%%bash

python3 rand.py -h
```

```
%%bash

python3 rand.py 4
```

`add_argument` parameters

print out a number or random integers in a specific range; the number of random numbers and the range limits are passed as command line arguments

The argument (`count`) is a positional argument because it is required and its position depends on the command itself. In the following example, we add an optional argument to let the user decide the range from which the random integers will be chosen.

- ‘-r’ is the short notation of the new argument, while '–range' is the long notation and you can use them interchangeably.
- `metavar` is the name that will be used in the help message
- `nargs` is the number of expected options after `-r` or `--range`, use `'*'` for unlimited. In this example, it will be 2, the lower and upper integer range limits
- `type` is the expected type (string by default)
- `default` is the default range when not specifying a range
- you can access the range options using `args.range[0]` and `args.range[1]`, if `nargs` was larger you can use the appropriate index to access the numbers passed

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random i

# Add optional arguments
parser.add_argument('-r', '--range', metavar = 'number', nargs = 2

# parse command line arguments
args = parser.parse_args()

# program
for i in range(args.count):
    print(randint(args.range[0], args.range[1]))
```

Running the script from a terminal will generate:

```
%%bash

python3 rand.py 4
```

```
%%bash

python3 rand.py 4 -r 500 1000
```

```
%%bash

python3 rand.py 4 --range 500 1000
```

```
%%bash

python3 rand.py 10 -r 500 1000
```

```
%%bash

python3 rand.py
```

```
%%bash

python3 rand.py -h
```

More about `metavar`

In the previous example, the number of expected arguments after `-r` (or `--range`) was `nargs = 2`. The help message illustrated that by `-r number number` (or `--range number number`). The word `number` was specified using the `metavar` parameter. The `metavar` was repeated 2 times to account for the 2 required arguments. It is also possible to specify different names for each of the required arguments by putting them in a tuple. In this example, the numbers passed to `-r` are renamed to `lower` and `upper` by assigning a tuple to `metavar`.

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random i

# Add optional arguments
parser.add_argument('-r', '--range', metavar = ('lower', 'upper'),

# parse command line arguments
args = parser.parse_args()

# program
for i in range(args.count):
    print(randint(args.range[0], args.range[1]))
```

```
%%bash

python3 rand.py -h
```

The `metavar` parameter can also be used with positional arguments to use an alternative name in the help message. However, only the displayed name is changed, the name of the attribute on the parse_args() is still under the original name. In this example, you will see how the `count` argument name is changed in the help message using `metavar`

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', metavar = 'rands', type = int, help =

# Add optional arguments
parser.add_argument('-r', '--range', metavar = ('lower', 'upper'),

# parse command line arguments
args = parser.parse_args()

# program
for i in range(args.count): # still accessed as args.count (not ar
    print(randint(args.range[0], args.range[1]))
```

```
%%bash

python3 rand.py -h
```

More `add_argument` parameters

print out a number or random integers in a specific range with an optional message; the number of random numbers, the range limits, and the option to print the message are all passed as command line arguments

In the following program, the `rand.py` script is updated so that it includes an optional `verbose` flag. When selected, the `verbose` flag will print out general messages about the currently selected options and arguments

- Since the value of `--verbose` should be `True` or `False`, the `action = 'store_true'` was used
- You can access `verbose` as `args.verbose`
- Rest of the argument is almost the same as for `--range` (or `-r`) argument

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', type = int, help = 'Count of random i

# Add optional arguments
parser.add_argument('-r', '--range', metavar = ('lower', 'upper'),

parser.add_argument('-v', '--verbose', action = 'store_true', help

# parse command line arguments
args = parser.parse_args()

# program
if args.verbose:
    print("Generating {:d} random integer in the range [{:d}, {:d}

for i in range(args.count):
    print(randint(args.range[0], args.range[1]))
```

```
%%bash

python3 rand.py 4 --range 500 1000 -v
```

```
%%bash

python3 rand.py -h
```

More about `action`

In the previous example, the `action = 'store_true'` was used to make `-v` (or `--verbose`) a Boolean flag that can be set to True or False. Python supports other actions:

- `store`: the default action for all arguments and it simply stores the value passed on the command line to the argument
- `store_true` and `store_false`: make an argument a Boolean flag and set it to True or False when entered by the user
- `store_const`: store a value specified by the keyword `const` in the argument. This is a more general form of `store_true` where you can store non-Boolean values in the argument
- `count`: the number of times an argument is used by the user

The following example show how these actions behave.

```
%%writefile rand.py

import argparse
from random import randint

# define an argument parser object
parser = argparse.ArgumentParser()

# Add positional arguments
parser.add_argument('count', action = 'store', type = int, help =

# Add optional arguments
parser.add_argument('-r', '--range', metavar = ('lower', 'upper'),

parser.add_argument('-c', '--const', action = 'store_const', const

parser.add_argument('-m', '--multiply', action = 'count', help = '

parser.add_argument('-v', '--verbose', action = 'store_true', help

# parse command line arguments
args = parser.parse_args()

# program

# if args.const is used, add 10 to the count entered by the user
num_of_rands = (args.count + args.const)

# when args.multiply is not used, its value is None
if (args.multiply != None):
    num_of_rands = num_of_rands * args.multiply

if args.verbose:
    print("Generating {:d} random integer in the range [{:d}, {:d}

for i in range(num_of_rands):
    print(randint(args.range[0], args.range[1]))
```

```
%%bash

python3 rand.py 4 --range 500 1000 -v -c
```

```
%%bash

python3 rand.py 4 --range 500 1000 -v -mmm
```

```
%%bash

python3 rand.py -h
```

NOTE: The Python Documentation site has more information about the parameters and capabilities of the `add_argument` method at https://docs.python.org/3/library/argparse.html

---

Task 2

Parsing command line arguments

Day of the week

```
%%writefile day_finder.py

# [ ] write a program that reads a date (month, day, year) as comm
# The program then prints out the week's day for that date
# If an optional flag (-c or --complete) is specified, the program

# help message should look like:
'''
usage: day_finder.py [-h] [-c] month day year

positional arguments:
  month           Month as a number (1, 12)
  day             Day as a number (1, 31) depending on the month
  year            Year as a 4 digits number (2018)

optional arguments:
  -h, --help      show this help message and exit
  -c, --complete  Show complete formatted date
'''

# HINT: Use a date object with strftime
```

```
%%bash

python3 day_finder.py 12 31 2017 -c
```

```
%%bash

python3 day_finder.py 12 31 2017
```

Sorting numbers

```
%%writefile sort_numbers.py

# [ ] Write a program that reads an unspecified number of integers
# then prints out the numbers in an ascending order
# the program should also have an optional argument to save the so

# help message should look like:
'''
usage: sort_numbers.py [-h] [-s] [numbers [numbers ...]]

positional arguments:
  numbers     int to be sorted

optional arguments:
  -h, --help  show this help message and exit
  -s, --save  save the sorted numbers on a file (sorted_numbers.tx
'''

#HINT: use nargs = '*' in an add_argument method
```

```
%%bash

python3 sort_numbers.py 23 49 5 300 43 582 58 29 62 69 320 60
```

Learn About Verified Certificates