Digital Career Institute

Python Course - Introduction

Digital Career Institute

Command Line Interface

Topics

- Read parameters in CLI context
- input() function
- cmd
- sys
- getopt

Digital Career Institute

Command Line Interface (CLI)

- CLI provides a way for a user to **interact** with a program running in a text-based shell interpreter.
- Some examples of shell interpreters are Bash on Linux or Command Prompt on Windows.
- A command line interface is enabled by the shell interpreter that exposes a command prompt.

Command prompt

- A **command prompt** (or just *prompt*) is a sequence of (one or more) characters used in a command-line interface to indicate **readiness** to accept commands.

- It literally prompts the user to take action.

- A prompt usually ends with one of the characters $, %, #, : , > or - and often includes other information, such as the path of the current working directory and the hostname.

Command prompt

- **Command prompt** can be characterized by the following elements:
  - A **command** or program
  - Zero or more command line **arguments**
  - An **output** representing the result of the command
  - Textual documentation referred to as **usage** or **help**
  - Not every command line interface may provide all these elements

Command prompt - example no. 1

- In this example, the Python interpreter takes option -c for **command**, which says to execute the Python command line arguments following the option -c as a Python program.

```
artur@artur-MSI:~$ python3 -c "print('Welcome to DCI course')"
Welcome to DCI course
```

Command prompt - example no. 1

- This example shows how to invoke Python with -h to display the help:

```
artur@artur-MSI:~$ python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
```

Command Line Arguments

- The arguments that are given after the name of the program in the command line shell of the operating system are known as **Command Line Arguments**. Python provides various ways of dealing with these types of arguments. The three most common are:
  - sys.argv
  - getopt module
  - argparse module

Python Command Line Arguments

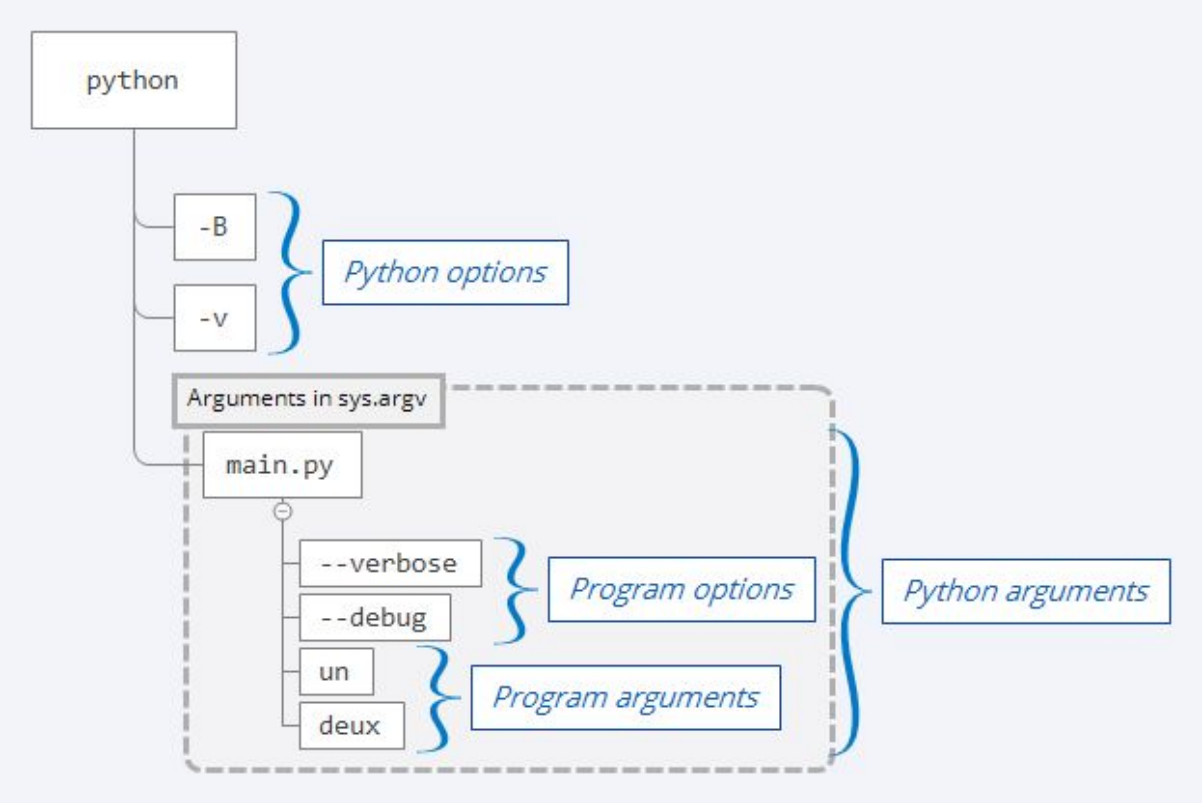
Python command line arguments are a subset of the command line interface. They can be composed of different types of arguments:

1. **Options** modify the behavior of a particular command or program.

2. **Arguments** represent the source or destination to be processed.

3. **Subcommands** allow a program to define more than one command with the respective set of options and arguments.

Python Command Line Arguments



sys module

- The **sys module** in Python provides various functions and variables that are used to manipulate different parts of the Python runtime environment.

- It allows operating on the interpreter as it provides access to the variables and functions that interact strongly with the interpreter.

sys.argv

- **argv** is a variable provided by the **sys** module which holds a list of all the arguments passed to the command line (including the script name).

- So even if you don't pass any arguments to your script. The argv variable always contains **at least one** element i.e the script name.

- The arguments in argv are always parsed as **string**. So be careful if you are expecting your input to be of any other type. You may need to cast or convert the elements according to your requirements.

sys.argv

- The sys module exposes an array named **argv** that includes the following:
    - **argv[0]** contains the name of the current Python program.
    - **argv[1:]**, the rest of the list, contains any and all Python command line arguments passed to the program.

sys.argv - example

```
1 # argv.py
2 import sys
3
4 print(f"Name of the script      : {sys.argv[0]=}")
5 print(f"Arguments of the script : {sys.argv[1:]=}")
```

- **Line 2** imports the internal Python module sys.
- **Line 4** extracts the name of the program by accessing the first element of the list sys.argv.
- **Line 5** displays the Python command line arguments by fetching all the remaining elements of the list sys.argv.

sys.argv - example

- After execution of code in file argv.py :

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv.py some arguments here
Name of the script      : sys.argv[0]='argv.py'
Arguments of the script : sys.argv[1:]=['some', 'arguments', 'here']
```

sys.argv - summing arguments (code)

```python
import sys

# total arguments
n = len(sys.argv)
print("Total arguments passed:", n)

# Arguments passed
print("Name of Python script:", sys.argv[0])

# Addition of numbers
sum_of_arguments = 0

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

print("Result:", sum_of_arguments)

```

sys.argv - summing arguments (results)

```
artur@artur-MSI:~/Desktop/DCI$ python3 argv-summing.py 1 2 3 4
Total arguments passed: 5
Name of Python script: argv-summing.py
Result: 10
```

sys.argv

- sys.argv is of the type **<list>** so you can access the elements just as you would from any other list. For example, **sys.argv[1]**

- It does not provide any inherent mechanism to make any of the arguments as required or optional and we also cannot limit the number of arguments supplied to our script.

- Sys.argv can be more than sufficient if your problem definition is simple enough. But if your requirements are a bit more advanced than just adding two numbers, you may need to use **getopt** or **argparse**.

getopt module

- **getopt** provides us with features that make it **easier** to process command line arguments in Python.
- **getopt** is a module that comes bundled with any standard python installation and therefore you **need not** install it explicitly.
- A major advantage of getopt over just using sys.argv is getopt supports **switch style options** (for example: -s or --sum).
- Hence getopt supported options are **position-independent**. The example $ ls -li works the same as $ ls -il

getopt module

- The options are of two types:
    - Options that need a value to be passed with them. These are defined by the option name suffixed with **=** (for example: **num1=**)
    - Options that behave as a flag and do not need a value. These are defined by passing the option name **without the suffix =** (for example: --subtract)

getopt module

- The options can have two variations:
    - **shortopts** are one letter options, denoted by prefixing a single - to an option name (for example, $ ls -l)
    - **longopts** are a more descriptive representation of an option, denoted by prefixing two – to an option name (for example, $ ls --long-list)

getopt (usage)

- getopt module provides a **getopt**(args, shortopts, longopts=[]) function which we can use to define our options:

- Code of getopt() **function** usage:

```
(opts, args) = getopt.getopt(sys.argv[1:], 'ha:b:s', ['help','num1=', 'num2=', 'subtract'])
```

- sys.argv holds the unformatted list of all the arguments passed to a python script.

getopt (usage)

- There are two variables used (**opts**, **args**) because getopt.getopt function returns two elements:
    - one containing a <list> of **options**
    - second has a <list> of **arguments** that are not specified in our getopt initialization.

getopt (usage)

- You can specify **shortopts** as a colon(:) separated single letter characters.
- You can specify **longopts** as a comma-separated list of words with the suffix **=**
- **longopts** without the suffix = are considered as **a flag** and they should be passed without any value.
- Now, to use the options passed to our program we can just iterate over the opts variable like any other list.

getopt (usage)

- Now, to use the options passed to our program we can just iterate over the opts variable like any other list:

```
for (o, a) in opts:
```

- Here **o** will hold our option name and **a** will have any value assigned to the option.

- Also notice that as **--subtract** is being used as a flag it will not have any value. This flag is used to decide whether to print the sum of the inputs **or** the difference in them.

getopt (usage)

```
artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7

 Sum of two numbers is : 13

artur@artur-MSI:~/Desktop/DCI$ python3 get_opt.py -a 6 -b 7 --subtract

 Difference in two numbers is : -1
```

- Full example is available under the link in documentation!

argparse module

- The argparse module makes it easy to write user-friendly command-line interfaces.
- The program defines what arguments it requires, and argparse will figure out how to parse those out of sys.argv.
- The argparse module also automatically generates help and usage messages and issues errors when users give the program invalid arguments.

argparse (example)

- The following code is a Python program that takes a list of integers and produces either the sum or the max:

```
import argparse

parser = argparse.ArgumentParser(description='Process some integers.')
parser.add_argument('integers', metavar='N', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate', action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the max)')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

argparse (example)

- Assuming the Python code above is saved into a file called prog.py, it can be run at the command line and provides useful help messages:

```
$ python prog.py -h
usage: prog.py [-h] [--sum] N [N ...]

Process some integers.

positional arguments:
 N           an integer for the accumulator

optional arguments:
 -h, --help  show this help message and exit
 --sum       sum the integers (default: find the max)
```

argparse (example)

- When run with the appropriate arguments, it prints either the sum or the max of the command-line integers:

```
$ python prog.py 1 2 3 4
4

$ python prog.py 1 2 3 4 --sum
10
```

argparse (example)

- If invalid arguments are passed in, it will issue an error:

```
$ python prog.py a b c
usage: prog.py [-h] [--sum] N [N ...]
prog.py: error: argument N: invalid int value: 'a'
```

- You can find more information about argparse module in the official tutorial (link in the **documentation**)

input() function

- Programs usually request for some user **input** to serve its function (e.g. calculators asking for what numbers to use, to add/subtract etc.).

- In Python, we request user input using the **input()** function.

```
>>> number = input("Type your first number: ")
Type your first number: 123
>>> print(number)
123
>>> print(type(number))
<class 'str'>
```

input() function

- This shown code is requesting for user input, and will store it in the **number** variable.
- **Note**: Inputs are automatically saved as **strings**.
- Therefore, always convert (cast) to proper type before doing any math operators like addition / subtraction.

At the core of the lesson

Digital Career Institute

Lesson learned:

- We know the idea of command line interface (CLI) and command line arguments
- We know usage of the sys.argv variable
- We know usage of the getopt module
- We know usage of the argparse module
- We know usage of the input() function