Running Python Programs from the Command-line

When you call your Python programs from the command-line, you can pass arguments to it too. In the most basic form this looks like:

```
$ python your_program.py arg1 arg2
```

Passing CLI arguments is common praxis for running your program in various configurations.

In this session, we will learn how to parse CLI arguments and options so that you can run your programs parametrized from the command-line.

Parsing CLI Arguments

Arguments are given -separated by spaces- after the name of your program on the CLI. Within your code, you can access them via the `argv` in the `sys` module. Here `argv[0]` is the script pathname (if known) and all the following elements of that list are the arguments given to your program.

```
import sys


print('Script is: ' + sys.argv[0])
print('Arguments given: ' + str(sys.argv[1:]))
```

Your turn

Clear Mu and type this in:

```
import sys



print(__file__)
print(sys.argv[0])
```

- What do you think the two arguments mean (`__file__` and `sys.argv[0]`)?
- Why do you think it prints what it prints?
- Remove the `[0]` from the second line. Do you get what you expect?

Your program now looks like this:

```
import sys



print(__file__)
print(sys.argv[0])
```

- Save it somewhere you can find it
- Open a terminal, and navigate to the place where you saved the file
- Run it with `python my_program.py`, where `my_program.py` is the name of your file
- Do you get what you expect?

Exercise

Write a small command-line program that prints all the command-line arguments (one per line), regardless of how many arguments you give it!

Executing

```
python printer.py faithful and friendly with stories to shaaaaare
```

Shall produce:

```
faithful
and
friendly
with
stories
to
shaaaaare
```

Exercise:

Write a small command-line program that can create lines of Scatman songs. For example, the following call

```
python scatman.py ba-da 2 ba-be 1 bop 2 bodda 1 bope 1
```

Shall produce the 13th line of the song: `ba-daba-da ba-be bopbop bodda bope`

The arguments for your program shall be an even length list of string integer pairs, where the string denotes the *scatter* and the integer denotes how often it appears in that line.

See the Scatman lyrics for more possible lines to produce: https://www.azlyrics.com/lyrics/scatmanjohn/scatmanskibabopbadopbop.html (https://www.azlyrics.com/lyrics/scatmanjohn/scatmanskibabopbadopbop.html) And the song: https://www.youtube.com/watch?v=Hy8kmNEo1i8 (https://www.youtube.com/watch?v=Hy8kmNEo1i8)

CLI options

So we saw how you can give a program arguments that it **requires** before it can run. But what about optional arguments? Or flags?

To provide something to a CLI program that is **optional** the convention is to use a single dash and a single character (`-h`) or double dashed and a full word (`--help`).

Try that out with `python` : `python -h` and `python --help`

Parsing CLI options and arguments

You can of course parse all the arguments and options yourself, but there exist a module for exactly this: `argparse` :

```
import argparse


parser = argparse.ArgumentParser()
parser.add_argument('person', help='A required argument!')
parser.add_argument('-f', '--foo', help='An optional argument')
arguments = parser.parse_args()

print("Person: " + arguments.person)
print("Arguments: " + arguments.foo)
```

Program pipes

Just like your program here in your notebooks can use `input` to read input from the user, so can your CLI program. But instead of calling `input()` we have to call a special function in the `sys.stdin` module called `read`:

```
import sys


user_input = sys.stdin.read()
print(user_input)
```

Program pipes

`stdin` stands for **standard input**, which is the common way to put something **into** a program.

As you probably guessed, there is also a way to get something **out** of your program. And, yes, that's called `stdout`. Only, you don't have to specify that you write to `stdout`. Whenever you `print`, that is done automatically.

Piping Arguments to Your Program

Here is another example (see `cli_reverse.py`), that reverses all lines that are piped to it. That is, it is called via for example `cat your.txt | python cli_reverse.py`. The program writes it's output again to `stdout`.

```
import sys


input_lines = sys.stdin.read().split('\n')
output_lines = reversed(input_lines)
output_str = '\n'.join(output_lines)
sys.stdout.write(output_str)
```

Pipe inception!

Wait!, you think. Does that mean that we can take the output of a program and feed that to another program? Well, yes it does.

You can combine CLI arguments and piping support, as the following example illustrates. `cli_replace.py` replaces a match for a regular expression line by line with a replacement text. The regular expression and the replacement text are given as arguments to the program. This is done with the 'pipe' character `|` .

OBS: When writing the following commands into your command line, use only the actual command never the first line saying `%%bash` ! This is unfortuantely a technicality needed here for presentation.

In [ ]:

```
%%bash
echo 'Hello world' | rev
```

In [ ]:

```
%%bash
echo 'Hello world' | wc -c
```

In [ ]:

```
%%bash
cat his_last_bow.txt | wc -c
```

In [ ]:

```
%%bash
cat his_last_bow.txt | rev
```