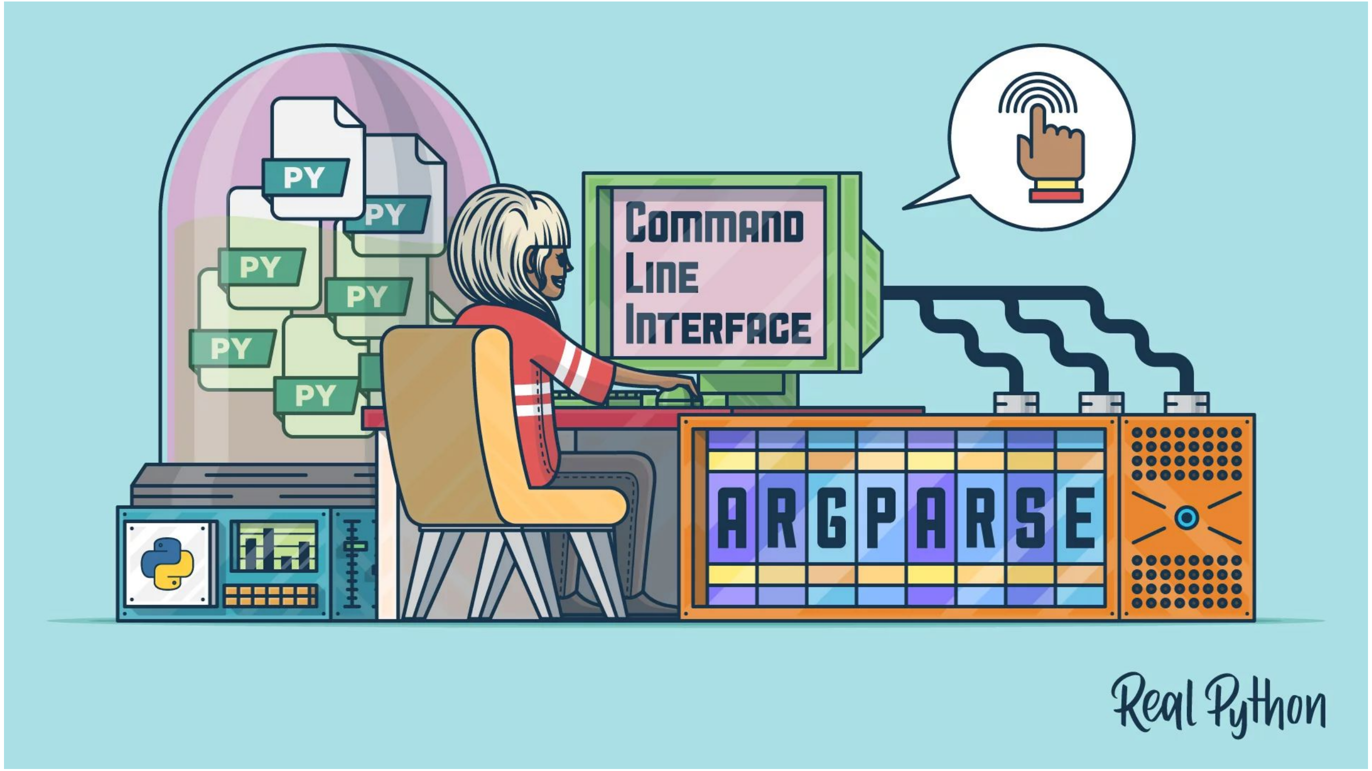
argparse



What is it?

Today we are gonna build command-line interface (CLI)!

Most of our programs have CLI - python, unix utilites (ls, ssh, mkdir), fastqc, bowtie and so on

Fraction of programs has graphic-user interface (GUI) in addition

And some programs have only GUI

`argparse` is a library for creating this interface

Interface

Interface is a border between our program and user. With an interface user can use our program without knowing python via the terminal or graphics

Furthermore, usually all (at least most) important functionality is shown via the interface, so users don't have to search through your code to find functions for their tasks

Now, question - why programs should have CLI even if they have GUI?

Primitive way

CLI creation can be done easily in python via `sys` module. This approach has ton of drawbacks

```python
import sys

# This variable store list with program launch arguments - everything after python
print(sys.argv)
```

Invocation

```
$ python draft.py
['draft.py']
```

```
$ python draft.py True 42 my_arg
['draft.py', 'True', '42', '1.41421357']
```

```
$ python draft.py True 42-my_arg "cli is my element!" back\ slash
['draft.py', 'True', '42-my_arg', 'cli is my element!', 'back slash']
```

Observations

1st value is script name, all others are strings parsed by shell

```
import sys


# Our awesome summator
print(sum(map(float, sys.argv[1:])))

$ python draft.py 32 2.5
34.5
```

What's wrong with this approach

As you have already seen lion's share of tools has keyword options to specify program behaviour. Also good programs have help message

With sys you will have to write all this stuff, but we have an argparse - cool successor of optparse!

What he will do for you

- parse user input
- convert types
- generate help message

```python
import argparse


# Initialize parser
parser = argparse.ArgumentParser()
# Add new option integers which can absorb 1 and more
# integers
parser.add_argument('integers', metavar='N',
       type=int, nargs='+',
       help='an integer for the accumulator')

# Parse all arguments
args = parser.parse_args()
print(args, args.__dict__, sep='\n')
```

```
$ python draft.py 1
Namespace(integers=[1])
{'integers': [1]}

$ python draft.py 1 2 3
Namespace(integers=[1, 2, 3])
{'integers': [1, 2, 3]}

$ python draft.py
usage: draft.py [-h] N [N ...]
draft.py: error: the following arguments are required: N

$ python draft.py 1 2 3.4
usage: draft.py [-h] N [N ...]
draft.py: error: argument N: invalid int value: '3.4'
```

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

positional arguments:
  N           an integer for the accumulator

optional arguments:
  -h, --help  show this help message and exit
```

Adding option

Just a reminder - combinations of these arguments can specify behaviour of your interface - so read the manual! I haven't covered const and action here

```
add_argument(short_name, long_name, nargs, default, required, choices,
type, metavar, help)
```

- `short_name` - by convention consists of 1 letter and `-` before it
- `long_name` - full name of option with `--` before it
- `nargs` - what's up with arguments? By default take 1 argument. Can be
    - `int` - will absorb specified number of values after the option and return list with them
    - `'?'` - 0 or 1
    - `'*'` - 0 to infinity
    - `'+'` - 1 to infinity

`add_argument(short_name, long_name, nargs, default, required, choices, type, metavar, help)`

- `default` - default value for this option if no argument was provided
- `required` - whether this option is obligatory
- `choices` - list with valid arguments
- `type` - type of the argument, argument(s) will be automatically casted to that type, can be vanilla `int`, `str`,... or your own in such form

```python
def is_valid(s):
    """Example of checking correct input type function"""
    if (not isinstance(s, str)) or s == 'a':
        raise ValueError
    return s
```

```
add_argument(short_name, long_name, nargs, default, required, choices,
type, metavar, help)
```

- `metavar` - how this argument is represented in the help message
- `help` - message for this option in the help message

```python
# Initialize parser
parser = argparse.ArgumentParser()

# Add new option integers which can absorb 1 and more
integers
parser.add_argument('-i', '--integers', default=0,
        required=False, metavar='number', type=int,
        nargs='+', help='an integer for the accumulator')

# Parse all arguments
args = parser.parse_args()
print(args, args.__dict__, sep='\n')
```

```
$ python draft.py -h
usage: draft.py [-h] [-i number [number ...]]

optional arguments:
  -h, --help            show this help message and exit
  -i number [number ...], --integers number [number ...]
                        an integer for the accumulator

$ python draft.py
Namespace(integers=0)
{'integers': 0}

$ python draft.py -i 1 2
Namespace(integers=[1, 2])
{'integers': [1, 2]}
```

```python
# Initialize parser
parser = argparse.ArgumentParser()
# Add new option integers which can absorb 1 and more
 integers
parser.add_argument('-i', '--integers', default=0,
 required=False, metavar='number', type=int, nargs='+',
 help='an integer for the accumulator')
# Parse all arguments
args = parser.parse_args()
args = args.__dict__

# Program logic itself
result = sum(args['integers'])
print(result)
```

ArgumentParser parameters

```
ArgumentParser(prog, usage, description=None, epilog=None)
```

- `prog` - name of program, `sys.argv[0]` by default
- `usage` - usage string, generated by argparse by default
- `description` - description string before option help, is absent by default
- `epilog` - string after option help, is absent by default

Advanced materials

There are also more complicated things like argument grouping, subparsers and so on, be aware of it!)

Structure

One of the ways to arrange it

- all argument parsing in one function
- program logic (in other files possibly)
- invocation of these 2 parts if the file was launched

Whether the file was launched can be inferred via `__name__` variable -

if its value `'__main__'` - this script was executed, not imported from the other place