

learnbyexample

Practice Python Projects



Sundeeep Agarwal

Table of contents

Preface	3
Prerequisites	3
Conventions	3
Acknowledgements	4
Feedback and Errata	4
Author info	4
License	4
Book version	5
CLI Calculator	6
Project summary	6
Real world influence	6
Bash shortcuts	6
Python CLI options	7
Python REPL	7
Bash function	7
Accepting stdin	8
Python CLI application	9
sys.argv	9
argparse	10
argparse initialization	10
Accepting an input expression	11
Adding optional flags	12
Accepting stdin	14
Shortcuts	16
Exercises	16
Further Reading	17
Poll Data Analysis	18
Project summary	18
Real world influence	18
Getting Reddit comments using PRAW	18
Installation	19
Reddit app	19
Extracting comments	19
API secrets	20
Data cleansing	20
Collecting data	21
Data inconsistencies	22
Extracting author names	24
Data similarity	25
Examples	25
Top authors	26
Exercises	27
Further Reading	27

Preface

Beginners who've finished a basic programming book or a course often wonder what they should do next. This article titled [I know how to program, but I don't know what to program](#) succinctly captures the feeling.

After solving exercises that test your understanding of syntax and common logical problems, working on projects is often recommended as the next step in the programming journey.

Working on projects that'll help you solve real world use cases would be ideal. You'll have enough incentive to push through difficulties instead of abandoning the project.

Sometimes though, you just don't know what to work on. Or, you have ideas, but not sure how to implement them, how to break down the project into manageable parts and so on. In such cases, a learning resource focused on projects can help.

This book presents five beginner to intermediate level projects inspired by real world use cases:

- [Enhance your CLI experience with a custom Python calculator](#)
- [Analyzing poll data from a Reddit comment thread](#)
- [Finding typos in plain text and Markdown files](#)
- [Creating a GUI for evaluating multiple choice questions](#)
- [Square Tic Tac Toe — creating a GUI game with AI logic](#)

To test your understanding and to make it more interesting, you'll also be presented with exercises at the end of each project. Resources for further exploration are also mentioned throughout the book.

Prerequisites

You should be comfortable with Python syntax and familiar with beginner-to-intermediate level programming concepts. For example, you should know how to use data types like `list`, `tuple`, `dict`, `set`, etc. Features like exceptions, file processing, sorting, comprehensions, generator expressions, etc. Classes, string methods and regular expressions will also be used in this book.

You are also expected to get comfortable with reading manuals, searching online, visiting external links provided for further reading, tinkering with the illustrated examples, asking for help when you are stuck and so on. In other words, be proactive and curious instead of just consuming the content passively.

If you are new to programming or Python, I'd highly recommend my [comprehensive curated list on Python](#) to get started.

Conventions

- The examples presented here have been tested with **Python version 3.13.3** and **GNU bash version 5.2.21**
- Code snippets that are copy pasted from the Python REPL shell have been modified for presentation purposes. For example, comments to provide context and explanations, blank lines to improve readability and so on.
- A comment with filename will be shown as the first line for program files.
- External links are provided for further exploration throughout the book. They have been chosen with care to provide more detailed resources on those topics as well as resources on related topics.

- The [practice_python_projects repo](#) has all the programs and related example files presented in this book, organized by project for convenience.

Acknowledgements

- [Python documentation](#) — manuals and tutorials
- [/r/learnpython/](#) and [/r/Python/](#) — helpful forums for Python programmers
- [stackoverflow](#) and [unix.stackexchange](#) — for getting answers on Python, Bash and other pertinent questions
- [tex.stackexchange](#) — for help on [pandoc](#) and `tex` related questions
- [canva](#) — cover image
- [oxipng](#), [pngquant](#) and [svgcleaner](#) — optimizing images
- [Warning](#) and [Info](#) icons by [Amada44](#)

Feedback and Errata

I would highly appreciate it if you'd let me know how you felt about this book. It could be anything from a simple thank you, pointing out a typo, mistakes in code snippets, which aspects of the book worked for you (or didn't!) and so on. Reader feedback is essential and especially so for self-published authors.

You can reach me via:

- Issue Manager: https://github.com/learnbyexample/practice_python_projects/issues
- E-mail: learnbyexample.net@gmail.com
- Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a lazy being who prefers to work just enough to support his modest lifestyle. He accumulated vast wealth working as a Design Engineer at Analog Devices and retired from the corporate world at the ripe age of twenty-eight. Unfortunately, he squandered his savings within a few years and had to scramble trying to earn a living. Against all odds, selling programming ebooks saved his lazy self from having to look for a job again. He can now afford all the fantasy ebooks he wants to read and spends unhealthy amount of time browsing the internet.

When the creative muse strikes, he can be found working on yet another programming ebook (which invariably ends up having at least one example with regular expressions). Researching materials for his ebooks and everyday social media usage drowned his bookmarks, so he maintains curated resource lists for sanity sake. He is thankful for free learning resources and open source tools. His own contributions can be found at <https://github.com/learnbyexample>.

List of books: <https://learnbyexample.github.io/books/>

License

This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

Code snippets are available under [MIT License](#).

Resources mentioned in the Acknowledgements section above are available under original licenses.

Book version

1.5

See [Version_changes.md](#) to track changes across book versions.

CLI Calculator

In this project, you'll learn to write a tool that can be used from a command line interface (CLI). First, you'll see how to directly pass Python code from the command line and create shell shortcuts to simplify the invocation. Next, you'll learn how to create a custom CLI application with Python. Finally, you'll be given exercises to test your understanding and resource links for further exploration. The links to these sections are given below:

- [Bash shortcuts](#)
- [Python CLI application](#)
- [Exercises](#)



If you are on Windows, you can still follow along most of this project by skipping the `bash` specific portions. The CLI tool creation using `argparse` isn't tied to a specific operating system. Use `py` instead of `python3.13` for program execution. See [docs.python: Windows command-line](#) and the rest of that page for more details. Alternatively, you can use [Windows Subsystem for Linux](#).

Project summary

- Execute Python instructions from the command line
- Use shell shortcuts to simplify typing on the command line
- Evaluate string content as Python code
- Create user friendly command line interfaces
- Allow `stdin` as a source for user input

The following modules and concepts will be utilized in this project:

- [docs.python: sys](#)
- [docs.python: argparse](#)
- [docs.python: eval](#)
- [docs.python: Modules](#)
- [docs.python: Exception handling](#)

Real world influence

I had two main reasons to implement this project:

- learn how to write a CLI application with Python
- a simple CLI calculator for personal use

There are powerful tools like `bc` but I wanted easier syntax without fiddling with settings like `scale`. Instead of writing a shell script to customize `bc` for my use cases, I went with Python since I wanted to learn about the `argparse` module too.


Bash shortcuts

In this section, you'll see how to execute Python instructions from the command line and use shell shortcuts to create simple CLI applications. This project uses `bash` as the shell to showcase examples.

Python CLI options

Passing a file to the interpreter from the command line is one of the ways to execute a Python program. You can also use the `-c` option to directly pass instructions to be executed as an argument. This is suitable for small programs, like getting the result of a mathematical expression. Here's an example:

```
# use py instead of python3.13 for Windows
$ python3.13 -c 'print(5 ** 2)'
25
```

 Use `python3.13 -h` to see all the available options. See [docs.python: Command line and environment](#) for documentation.

Python REPL

If you call the interpreter without passing instructions to be executed, you'll get an interactive shell known as REPL (stands for **R**ead **E**valuate **P**rint **L**oop). This is typically used to execute instructions for learning and debugging purposes. REPL is well suited to act as a calculator too. Since the result of an expression is automatically printed, you don't need to explicitly call the `print()` function. A special variable `_` holds the result of the last executed expression. Here are some examples:

```
$ python3.13 -q
>>> 2 * 31 - 3
59
>>> _ * 2
118
>>> exit()
```

See also:

- [docs.python: Using the Python Interpreter](#)
- [docs.python: Using Python as a Calculator](#)
- [IPython](#) — an alternate feature-rich Python REPL

Bash function

Calling the `print()` function via the `-c` option from the command line is simple enough. But you could further simplify by creating a CLI application using a `bash` function as shown below:

```
# bash_func.sh
pc() { python3.13 -c 'print("$1")' ; }
```

You can type that on your current active terminal or add it your `.bashrc` file so that the shortcut is always available for use (assuming `pc` isn't an existing command). The function is named `pc` (short for Python Calculator). The first argument passed to `pc` in turn is passed along as the argument for Python's `print()` function. To see how `bash` processes this user defined function, you can use `set -x` as shown below. See [unix.stackexchange: How to debug a bash script?](#) for more details.

```
$ set -x
$ pc '40 + 2'
```

```
+ pc '40 + 2'
+ python3.13 -c 'print(40 + 2)'
42

# don't forget to quote your argument, otherwise spaces
# and other shell metacharacters will cause issues like this example
$ pc 40 + 2
+ pc 40 + 2
+ python3.13 -c 'print(40)'
40

$ set +x
+ set +x
```

Here are some more examples of using `pc` as a handy calculator from the command line.

```
$ pc '2 * 31 - 3'
59

$ pc '0xfe'
254

$ pc '76 / 13'
5.846153846153846
$ pc '76 // 13'
5
```



See also [unix.stackexchange: when to use alias, functions and scripts](https://unix.stackexchange.com/questions/111111/when-to-use-alias-functions-and-scripts).

Accepting stdin

Many CLI applications allow you to pass `stdin` data as input. To add that functionality, you can use an `if` statement to read a line from standard input if the number of arguments is zero or the `-` character is passed as the argument. The modified `pc` function is shown below:

```
# bash_func_stdin.sh
pc()
{
    ip_expr="$1"
    if [[ $# -eq 0 || $1 = '-' ]]; then
        read -r ip_expr
    fi
    python3.13 -c 'print("$ip_expr")'
}
```

Here are some examples. Use `set -x` if you wish to see how the function gets evaluated for these examples.

```
$ source bash_func_stdin.sh

$ echo '97 + 232' | pc
```


329

```
$ echo '97 + 232' | pc -
```

329

```
$ pc '32 ** 12'
```

1152921504606846976



See [wooledge: Bash Guide](#) and [ryanstutorial: Bash scripting tutorial](#) if you'd like to learn more about `bash` shell scripting. See also [shellcheck](#), a linting tool to avoid common mistakes and improve your script.

Python CLI application

In this section, you'll see how to implement a CLI application using Python features, instead of relying on the shell. First, you'll learn how to work with command line arguments using the `sys` module. Followed by the `argparse` module, which is specifically designed for creating CLI applications.

`sys.argv`

Command line arguments passed when executing a Python program can be accessed using the `sys.argv` list. The first element (index `0`) contains the name of the Python script or `-c` or an empty string, depending on how the interpreter was called. See [docs.python: sys.argv](#) for details.

Rest of the elements will have the command line arguments, if any were passed along the script to be executed. The data type of the `sys.argv` elements is `str` class. The `eval()` function allows you to execute a string as a Python instruction. Here's an example:

```
$ python3.13 -c 'import sys; print(eval(sys.argv[1]))' '23 ** 2'
```

529

```
# bash shortcut
```

```
$ pc() { python3.13 -c 'import sys; print(eval(sys.argv[1]))' "$1" ; }
```

```
$ pc '23 ** 2'
```

529

```
$ pc '0x2F'
```

47



Using the `eval()` function isn't recommended if the input passed to it isn't under your control, for example an input typed by a user from a website application. The arbitrary code execution issue would apply to the `bash` shortcuts seen in the previous section as well, because the input argument is interpreted without any sanity check.

However, for this calculator project used for personal purposes, such security considerations aren't applicable. See [stackoverflow: evaluating a mathematical expression](#) for more details about the dangers of using the `eval()` function and alternate ways to evaluate a string as code.

argparse

Quoting from [docs.python: argparse](#):

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. The module will also issue errors when users give the program invalid arguments.

argparse initialization

If this is your first time using the `argparse` module, it is recommended to understand the initialization instructions and effects provided by default. Quoting from [docs.python: argparse](#):

The `ArgumentParser` object will hold all the information necessary to parse the command line into Python data types.

`ArgumentParser` parses arguments through the `parse_args()` method. This will inspect the command line, convert each argument to the appropriate type and then invoke the appropriate action.

```
# arg_help.py
import argparse

parser = argparse.ArgumentParser()
args = parser.parse_args()
```

The documentation for the CLI application is generated automatically based on the information passed to the parser. You can use the help option (which is automatically added) to view the content, as shown below:

```
$ python3.13 arg_help.py -h
usage: arg_help.py [-h]

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


In addition, options or arguments that are not defined will generate an error.

```
$ python3.13 arg_help.py -c
usage: arg_help.py [-h]
arg_help.py: error: unrecognized arguments: -c

$ python3.13 arg_help.py '2 + 3'
usage: arg_help.py [-h]
arg_help.py: error: unrecognized arguments: 2 + 3
```

A required argument wasn't declared in this program, so there's no error for the below usage:

```
$ python3.13 arg_help.py
```

 See also [docs.python: Argparse Tutorial](#).

Accepting an input expression

```
# single_arg.py
import argparse, sys

parser = argparse.ArgumentParser()
parser.add_argument('ip_expr',
                    help="input expression to be evaluated")
args = parser.parse_args()

try:
    result = eval(args.ip_expr)
    print(result)
except (NameError, SyntaxError):
    sys.exit("Error: Not a valid input expression")
```

The `add_argument()` method allows you to add details about an option/argument for the CLI application. The first parameter names an argument or options (starts with `-`). The optional `help` parameter lets you add documentation for that particular option/argument. See [docs.python: add_argument](#) for documentation and details about other parameters.

The value for `ip_expr` passed by the user will be available as an attribute of `args`, which stores the object returned by the `parse_args()` method. The default data type for arguments is `str`, which is good enough here for `eval()`.

The help documentation for this script is shown below:

```
$ python3.13 single_arg.py -h
usage: single_arg.py [-h] ip_expr

positional arguments:
  ip_expr      input expression to be evaluated

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

Note that the script uses a `try-except` block to give user friendly feedback for some of the common issues. Passing a string to `sys.exit()` gets printed to the `stderr` stream and sets the exit status as `1` to indicate something has gone wrong. See [docs.python: sys.exit](#) for documentation. Here are some usage examples:

```
$ python3.13 single_arg.py '40 + 2'
42

# if no argument is passed to the script
$ python3.13 single_arg.py
usage: single_arg.py [-h] ip_expr
single_arg.py: error: the following arguments are required: ip_expr
$ echo $?
```

2

```
# SyntaxError
$ python3.13 single_arg.py '5 \ 2'
Error: Not a valid input expression
$ echo $?
1

# NameError
$ python3.13 single_arg.py '5 + num'
Error: Not a valid input expression
```

Adding optional flags

To add an option, use `-<char>` for short options and `--<name>` for long options. You can add both as well, for example `'-v', '--verbose'`. If you use both short and long options, the attribute name will be whichever option is the latest. For the CLI application, five short options have been added, as shown below.

```
# options.py
import argparse, sys

parser = argparse.ArgumentParser()
parser.add_argument('ip_expr',
                    help="input expression to be evaluated")
parser.add_argument('-f', type=int,
                    help="specify floating point output precision")
parser.add_argument('-b', action="store_true",
                    help="output in binary format")
parser.add_argument('-o', action="store_true",
                    help="output in octal format")
parser.add_argument('-x', action="store_true",
                    help="output in hexadecimal format")
parser.add_argument('-v', action="store_true",
                    help="verbose mode, shows both input and output")
args = parser.parse_args()

try:
    result = eval(args.ip_expr)

    if args.f:
        result = f'{result:.{args.f}f}'
    elif args.b:
        result = f'{int(result):#b}'
    elif args.o:
        result = f'{int(result):#o}'
    elif args.x:
        result = f'{int(result):#x}'

    if args.v:
```

```

        print(f'{args.ip_expr} = {result}')
    else:
        print(result)
except (NameError, SyntaxError):
    sys.exit("Error: Not a valid input expression")

```

The `type` parameter for the `add_argument()` method allows you to specify what data type should be applied for that option. The `-f` option is used here to set the precision for floating-point output. The code doesn't actually check if the output is a floating-point value — that is left as an exercise for you.

The `-b`, `-o`, `-x` and `-v` options are intended as boolean data types. Using `action="store_true"` indicates that the associated attribute should be set to `False` as their default value. When the option is used from the command line, their value will be set to `True`. The `-b`, `-o` and `-x` options are used here to get the output in binary, octal and hexadecimal formats respectively. The `-v` option will print both the input expression and the evaluated result.

The help documentation for this script is shown below. By default, uppercase of the option name will be used to describe the value expected for that option. Which is why you see `-f F` here. You can use `metavar='precision'` to change it to `-f precision` instead.

```

$ python3.13 options.py -h
usage: options.py [-h] [-f F] [-b] [-o] [-x] [-v] ip_expr

positional arguments:
  ip_expr      input expression to be evaluated

optional arguments:
  -h, --help  show this help message and exit
  -f F        specify floating point output precision
  -b          output in binary format
  -o          output in octal format
  -x          output in hexadecimal format
  -v          verbose mode, shows both input and output

```

Here are some usage examples:

```

$ python3.13 options.py '22 / 7'
3.142857142857143
$ python3.13 options.py -f3 '22 / 7'
3.143
$ python3.13 options.py -f2 '32 ** 2'
1024.00

$ python3.13 options.py -bv '543 * 2'
543 * 2 = 0b10000111110
$ python3.13 options.py -x '0x1F * 0xA'
0x136
$ python3.13 options.py -o '0xdeadbeef'
0o33653337357

```

Since the `-f` option expects an `int` value, you'll get an error if you don't pass a value or if

the value passed isn't a valid integer.

```
$ python3.13 options.py -fa '22 / 7'
usage: options.py [-h] [-f F] [-b] [-o] [-x] [-v] ip_expr
options.py: error: argument -f: invalid int value: 'a'

$ python3.13 options.py -f
usage: options.py [-h] [-f F] [-b] [-o] [-x] [-v] ip_expr
options.py: error: argument -f: expected one argument

$ python3.13 options.py -f '22 / 7'
usage: options.py [-h] [-f F] [-b] [-o] [-x] [-v] ip_expr
options.py: error: argument -f: invalid int value: '22 / 7'

$ python3.13 options.py -f '22'
usage: options.py [-h] [-f F] [-b] [-o] [-x] [-v] ip_expr
options.py: error: the following arguments are required: ip_expr
```

Accepting stdin

The final feature to be added is the ability to accept `stdin` as the input expression. The `sys.stdin` filehandle can be used to read `stdin` data. The modified script is shown below.

```
# py_calc.py
import argparse, sys

parser = argparse.ArgumentParser()
parser.add_argument('ip_expr', nargs='?',
                    help="input expression to be evaluated")
parser.add_argument('-f', type=int,
                    help="specify floating point output precision")
parser.add_argument('-b', action="store_true",
                    help="output in binary format")
parser.add_argument('-o', action="store_true",
                    help="output in octal format")
parser.add_argument('-x', action="store_true",
                    help="output in hexadecimal format")
parser.add_argument('-v', action="store_true",
                    help="verbose mode, shows both input and output")
args = parser.parse_args()

if args.ip_expr in (None, '-'):
    args.ip_expr = sys.stdin.readline().strip()

try:
    result = eval(args.ip_expr)

    if args.f:
        result = f'{result:.{args.f}f}'
    elif args.b:
        result = f'{int(result):#b}'
```

```

elif args.o:
    result = f'{int(result):#o}'
elif args.x:
    result = f'{int(result):#x}'

if args.v:
    print(f'{args.ip_expr} = {result}')
else:
    print(result)
except (NameError, SyntaxError):
    sys.exit("Error: Not a valid input expression")

```

The `nargs` parameter allows to specify how many arguments can be accepted with a single action. You can use an integer value to get that many arguments as a list or use specific regular expression like metacharacters to indicate varying number of arguments. The `ip_expr` argument is made optional here by setting `nargs` to `?`.

If `ip_expr` isn't passed as an argument by the user, the attribute will get `None` as the value. The `-` character is often used to indicate `stdin` as the input data. So, if `ip_expr` is `None` or `-`, the code will try to read a line from `stdin` as the input expression. The `strip()` string method is applied to the `stdin` data mainly to prevent newline from messing up the output for the `-v` option. Rest of the code is the same as seen before.

The help documentation for this script is shown below. The only difference is that the input expression is now optional as indicated by `[ip_expr]`.

```

$ python3.13 py_calc.py -h
usage: py_calc.py [-h] [-f F] [-b] [-o] [-x] [-v] [ip_expr]

```

```

positional arguments:
  ip_expr      input expression to be evaluated

```

```

optional arguments:
  -h, --help  show this help message and exit
  -f F        specify floating point output precision
  -b          output in binary format
  -o          output in octal format
  -x          output in hexadecimal format
  -v          verbose mode, shows both input and output

```

Here are some usage examples:

```

# stdin from the output of another command
$ echo '40 + 2' | python3.13 py_calc.py
42

# providing stdin data manually after pressing the Enter key
$ python3.13 py_calc.py
43 / 5
8.6

# strip() will remove whitespace from the start/end of strings

```

```
$ echo ' 0b101 + 3' | python3.13 py_calc.py -vx
0b101 + 3 = 0x8

$ echo '0b101 + 3' | python3.13 py_calc.py -vx -
0b101 + 3 = 0x8

# expression passed as an argument, works the same as seen before
$ python3.13 py_calc.py '5 % 2'
1
```

Shortcuts

To simplify calling the Python CLI calculator, you can create an alias or an executable Python script.

Use the absolute path of the script to create the alias and add it to `.bashrc`, so that it will work from any working directory. The path used below would differ for you.

```
alias pc='python3.13 /home/learnbyexample/python_projs/py_calc.py'
```

To create an executable, you'll have to first add a [shebang](#) as the first line of the Python script. You can use the `type` built-in command to get the path of the Python interpreter.

```
$ type python3.13
python3.13 is /usr/local/bin/python3.13
```

So, the `shebang` for this case will be `#!/usr/local/bin/python3.13`. After adding the execute permission, copy the file to one of the `PATH` directories. I have `~/cbin/` as one of the paths. See [unix.stackexchange: How to correctly add a path to PATH?](#) for more details about the `PATH` environment variable.

```
$ chmod +x py_calc.py

$ cp py_calc.py ~/cbin/pc

$ pc '40 + 2'
42
```

With that, the lessons for this project comes to an end. Solve the practice problems given in the exercises section to test your understanding.

Exercises

Modify the scripts such that these additional features are also implemented.

- If the output is of the `float` data type, apply `.2f` precision by default. This should be overridden if a value is passed along with the `-f` option. Also, add a new option `-F` to turn off the default `.2f` precision.

```
$ pc '4 / 3'
1.33

$ pc -f3 '22 / 7'
3.143
```



```
$ pc -F '22 / 7'
3.142857142857143

# if output isn't float, .2f shouldn't be applied
$ pc '12 ** 12'
8916100448256
```

- Use the `math` module to allow mathematical methods and constants like `sin` , `pi` , etc.

```
$ pc 'sin(radians(90))'
1.00

$ pc 'pi * 2'
6.283185307179586

$ pc 'factorial(5)'
120
```

- If the input expression has a sequence of numbers followed by the `!` character, replace such a sequence with the factorial value. Assume that the input will not have `!` applied to negative or floating-point numbers. Alternatively, you can issue an error if such numbers are detected.

```
$ pc '2 + 5!'
122
```

- Go through [docs.python: ArgumentParser](#) and experiment with parameters like `description` , `epilog` , etc.

Further Reading

Python has a rich ecosystem in addition to the impressive standard library. You can find plenty of modules to choose for common tasks, including alternatives for the standard modules. Check out these projects for CLI related applications.

- [click](#) — Python package for creating beautiful command line interfaces in a composable way with as little code as necessary
- [rich](#) — library for *rich* text and beautiful formatting in the terminal
- [SimpleParsing](#) — Simple, Elegant, Typed Argument Parsing with `argparse`
- [Gooyey](#) — turn Python command line program into a full GUI application
- [CLI Guidelines](#) — an opinionated guide to help you write better CLI programs

Poll Data Analysis

In this project, you'll learn how to use an application programming interface (API) to fetch data. From this raw data, you'll extract data of interest and then apply heuristic rules to correct possible mistakes (at the cost of introducing new bugs).

- [Getting Reddit comments using PRAW](#)
- [Data cleansing](#)
- [Data similarity](#)
- [Exercises](#)

Project summary

- Get top level comments from Reddit threads
- Use regular expressions to explore data inconsistencies and extract author names
- Correct typos by comparing similarity between names

The following modules and concepts will be utilized in this project:

- [pypi: praw](#)
- [docs.python: json](#)
- [Data cleansing](#)
- [docs.python: re](#)
- [pypi: rapidfuzz](#)

Real world influence

I read a lot of fantasy novels and [/r/Fantasy/](#) is one of my favorite social forums. They conduct a few polls every year for best novels, novellas, standalones, etc. These polls help me pick new books to read.

The poll results are manually tallied, since there can be typos, bad entries and so on. I wanted to see if this process can be automated and gave me an excuse to get familiar with using APIs and some of the third-party Python modules.

I learned a lot, especially about the challenges in data analysis. I hope you'll learn a lot too.

Getting Reddit comments using PRAW

In this section, you'll learn to use the `praw` module for extracting comments from a given Reddit thread. You'll also see how to fetch only the top level comments.

From [pypi: praw](#):

PRAW, an acronym for "Python Reddit API Wrapper", is a Python package that allows for simple access to Reddit's API. PRAW aims to be easy to use and internally follows all of Reddit's API rules. With PRAW there's no need to introduce `sleep` calls in your code. Give your client an appropriate user agent and you're set.

From [wikipedia: API](#):


An application programming interface (API) is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software. A document or standard that describes how to build such a connection or interface is called an API specification. A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation.

Installation

You can install `praw` using the following commands:

```
# virtual environment
$ pip install praw

# normal environment
# use py instead of python3.13 for Windows
$ python3.13 -m pip install --user praw
```

 I'd highly recommend using virtual environments to manage projects that use third party modules. See [Installing modules and Virtual environments](#) chapter from my Python introduction ebook if you are not familiar with installing modules.

Reddit app

First login to your Reddit account. Next, visit <https://www.reddit.com/prefs/apps/> and click the **are you a developer? create an app...** button.

For this project, using the **script** option is enough. Two of the fields are mandatory:

- name
- redirect uri

The redirect uri isn't needed for this particular project though. As mentioned in Reddit's [OAuth2 Quick Start Example](#) guide, `http://www.example.com/unused/redirect/uri` can be used instead.

After filling the details, you'll get a screen with details about the app, which you can update if needed. If applicable, you'll also get an email from Reddit.

Extracting comments

This section will give you an example of extracting comments from a particular discussion thread on Reddit. The code used is based on the [Comment Extraction and Parsing](#) tutorial from the documentation, which also informs that:

If you are only analyzing public comments, entering a username and password is optional.

The sample discussion thread used here is from [the /r/booksuggestions subreddit](#). You can use this URL in the code or just the `nsm98m` id.

From the app you created in the previous section, you need to copy the `client_id` and `client_secret` details. You'll find the **id** at the top of the app details (usually 14 characters)

and the **secret** field is clearly marked. With those details collected, here's how you can get all the comments:

```
>>> import praw

>>> reddit = praw.Reddit(
...     user_agent="Get Comments by /u/name", #change 'name' to your username
...     client_id="XXX",                     #change 'XXX' to your id
...     client_secret="XXX",                 #change 'XXX' to your secret
... )

# use the url keyword argument if you want to pass a link instead of id
>>> submission = reddit.submission(id='nsm98m')
>>> submission.comments.replace_more(limit=None)
[]
# all comments are saved in a list here for illustration purposes
>>> comments = submission.comments.list()
# content of the first comment
>>> print(comments[0].body)
The Murder of Roger Ackroyd by Agatha Christie still has the
best twist I've ever read.
# fourth comment and so on
>>> comments[3].body
'The Silent Patient'
```

Use `submission.comments` instead of `submission.comments.list()` to fetch only the top level comments.

API secrets

As mentioned in Reddit's [OAuth2 Quick Start Example](#) guide:

You should NEVER post your client secret (or your reddit password) in public. If you create a bot, you should take steps to ensure that the bot's password and the app's client secret are secured against digital theft.

To avoid accidentally revealing API secrets online (publishing your code on GitHub for example), one way is to store them in a secrets file locally. Such a secrets filename should be part of the `.gitignore` file so that it won't get committed to the GitHub repo.

Data cleansing

Now that you've seen a basic example with the `praw` module, you'll start this project by getting the top level comments from two Reddit threads. These threads were used to conduct a poll about favorite speculative fiction written by women. From the raw data so obtained, author names have to be extracted. But the data format isn't always as expected. You'll use regular expressions to explore inconsistencies, remove unwanted characters from the names and ignore entries that couldn't be parsed in the format required.

From [wikipedia: Data cleansing](#):

Data cleansing or data cleaning is the process of identifying and correcting (or removing) corrupt, inaccurate, or irrelevant records from a dataset, table, or database. It involves detecting incomplete, incorrect, or inaccurate parts of the data and then replacing, modifying, or deleting the affected data. Data cleansing can be performed interactively using data wrangling tools, or through batch processing often via scripts or a data quality firewall.

Collecting data

The two poll threads being analyzed for this project are [2019](#) and [2021](#). The poll asked users to specify their favorite speculative fictional books written by women, with a maximum of 10 entries. The voting comment was restricted to contain only book titles and authors. Any other discussion had to be placed under those entries as comments.

The below program builds on the example shown earlier. A `tuple` object stores the voting thread `year` and the corresponding `id` values. And then a loop goes over each entry and writes only the top level comments to the respective output files.

```
# save_top_comments.py
import json
import praw

with open('.secrets/tokens.json') as f:
    secrets = json.load(f)

reddit = praw.Reddit(
    user_agent=secrets['user_agent'],
    client_id=secrets['client_id'],
    client_secret=secrets['client_secret'],
)

thread_details = (('2019', 'cib77j'), ('2021', 'm20rd1'))

for year, thread_id in thread_details:
    submission = reddit.submission(id=thread_id)
    submission.comments.replace_more(limit=None)

    op_file = f'top_comments_{year}.txt'
    with open(op_file, 'w') as f:
        for top_level_comment in submission.comments:
            f.write(top_level_comment.body + '\n')
```

The `tokens.json` file contains the information passed to the `praw.Reddit()` method. The data structure is shown below — you'll need to replace the values with your own valid information.

```
$ cat .secrets/tokens.json
{
  "user_agent": "Get Comments by /u/name",
  "client_id": "XXX",
  "client_secret": "XXX"
}
```

Data inconsistencies

As mentioned earlier, the poll asked users to specify their favorite speculative fictional books written by women, with a maximum of 10 entries. Users were also instructed to use only one entry per series, but series name or any individual book title can be specified. To analyze this data as intended, you'll have to find a way to collate all entries that fall under the same series. This is out of scope for this project. Instead, only author names will be used for the analysis, which is a significant deviation from the poll's intention.

Counting author names alone makes it easier to code this project, but you'll still come to appreciate why data cleansing is a very important step. Users were asked to write their entries as book title followed by hyphen or the word `by` and finally the author name. Assuming there is at least one whitespace character before and after the separators, here's a program that displays all the mismatching lines.

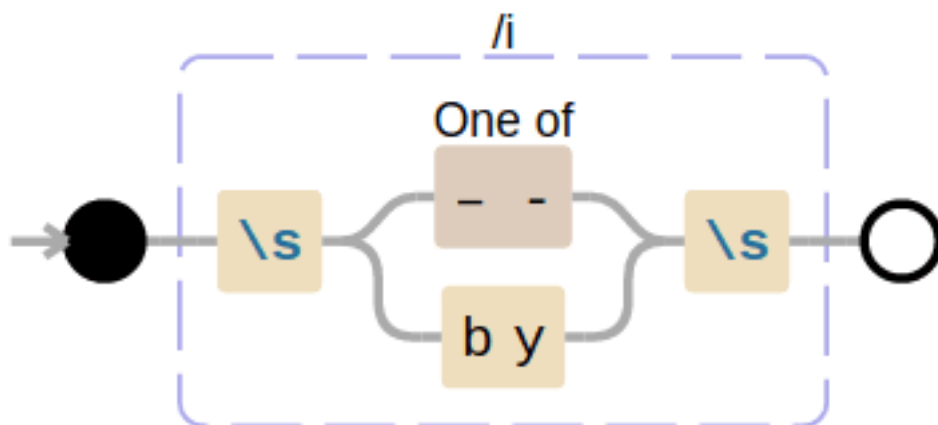
```
import re


file = 'top_comments_2019.txt'
pat = re.compile(r'\s(?:[--]|by)\s', flags=re.I)

with open(file) as f:
    for line in f:
        if re.fullmatch(r'\s+', line):
            continue
        elif not pat.search(line):
            print(line, end='')
```

The `re.fullmatch` regexp is used to ignore all lines containing only whitespaces. The next regexp checks if hyphen (or em dash) or `by` surrounded by whitespace characters is present in the line. Case is also ignored when `by` is matched. Matching the whitespace characters is important because the book titles or author names could contain `by` or hyphens. While this can still give false matches, the goal is to reduce errors as much as possible, not 100% accuracy. If a line doesn't match this condition, it will be displayed on the screen. About a hundred such lines are found in the `top_comments_2019.txt` file.

Here's a visual representation of the `pat` regexp:



 The above railroad diagram for the `r'\s(?:[--]|by)\s'` pattern was created using the [debuggex](#) site. You can also visit this [regex101](#) link, which is another popular way to experiment and understand regexp patterns. See my [Python re\(gex\)?](#) ebook if you want to learn more about regular expressions.

And here's a sample of the mismatching lines:

```
**Wayfarers** \- Becky Chambers
**The Broken Earth** *by N.K. Jemisin*
5. Uprooted- Naomi Novik
Empire of Sand, Tasha Suri
```

So, some entries used a slightly different markdown style and some used `,` as the separator. The first two cases can be allowed by optionally matching the `\` or `*` characters. The last two cases will require breaking the whitespace matching rule. For now, this will be allowed so as to proceed further. But in the next section you will see how to apply regexp on a priority basis so that the different rules are applied only for the mismatching lines.

The modified program is shown below. The `re.X` flag allows you to use literal whitespaces for readability purposes. You can also add comments after the `#` character if you wish.

```
# analyze.py
import re

file = 'top_comments_2019.txt'
pat = re.compile(r'''
\s(?:[--]|by)\s
|\s\\[--]\s
|\s*by\s
|[, -]\s
''', flags=re.I|re.X)

with open(file) as f:
    for line in f:
        if re.fullmatch(r'\s+', line):
            continue
        elif not pat.search(line):
            print(line, end='')
```


After applying this rule, there are less than 50 mismatching lines. Some of them are comments irrelevant to the voting, but some of the entries can still be salvaged by manual modification (for example, entries that have the book title and author names in reversed order). These will be completely ignored for this project, but you can try to correct them if you wish.

Changing the input file to `top_comments_2021.txt` gives new kind of mismatches. Some of the mismatches are shown below:

```
The Blue Sword-Robin McKinley
**The Left Hand of Darkness**by Ursula K. Le Guin
Spinning Silver (Naomi Novik)
```

These can be accommodated by modifying the matching criteria, but since the total count of mismatches is less than 40, they will also be ignored. You can try to improve the code as an exercise. In case you are wondering, total entries are more than 1500 and 3400 for the 2019

and 2021 polls respectively. So, ignoring less than 50 mismatches isn't a substantial loss.

 Note that the results you get might be different than what is shown here due to modification of the Reddit comments under analysis. Or, users might have deleted their comments and so on.

Extracting author names

It is time to extract only the author names and save them for further analysis. The regexp patterns seen in the previous section needs to be modified to capture author names at the end of the lines. Also, `.*` is added at the start so that only the furthest match in the line is extracted. To give priority for the best case matches, the patterns are first stored separately as different elements in a `tuple`. By looping over these patterns, you can then quit once the earliest match is found.

```
# extract_author_names.py
import re

ip_files = ('top_comments_2019.txt', 'top_comments_2021.txt')
op_files = ('authors_2019.txt', 'authors_2021.txt')

patterns = (r'.*\s(?:[--]|by)\s+(.+)',
            r'.*\s\[--]\s+(.+)',
            r'.*\s\*by\s+(.+)',
            r'.*\[, -]\s+(.+)')

for ip_file, op_file in zip(ip_files, op_files):
    with open(ip_file) as ipf, open(op_file, 'w') as opf:
        for line in ipf:
            if re.fullmatch(r'\s+', line):
                continue
            for pat in patterns:
                if m := re.search(pat, line, flags=re.I):
                    opf.write(m[1].strip('*\t ') + '\n')
                    break
```

If you check the two output files you get, you'll see some entries like shown below. Again, managing these entries is left as an exercise.

```
Janny Wurts & Raymond E. Feist
Patricia C. Wrede, Caroline Stevermer
Melaine Rawn, Jennifer Roberson, and Kate Elliott
and get to add some stuff I really enjoyed! In no particular order:
but:
Marie Brennan (Memoirs of Lady Trent)
Alice B. Sheldon (as James Tiptree Jr.)
Linda Nagata from The Red trilogy
Novik, Naomi
```

`strip('*\t ')` is applied on the captured portion to remove whitespaces at the end of the line, markdown formatting, etc. Without that, you'll get author names like shown below:

N.K. Jemisin*
ML Wang**
*Mary Robinette Kowal

Data similarity

Now that you have all the author names, the next task is to take care of typos. You'll see how to use the `rapidfuzz` module for calculating the similarity between two strings. This helps to remove majority of the typos — for example *Courtney Schaefer* and *Courtney Shafer*. But, this would also introduce new errors if similar looking names are actually different authors and not typos — for example *R.J. Barker* and *R.J. Parker*.

From [pypi: rapidfuzz](#):

RapidFuzz is a fast string matching library for Python and C++, which is using the string similarity calculations from [FuzzyWuzzy](#).

From [pypi: fuzzywuzzy](#):

It uses [Levenshtein Distance](#) to calculate the differences between sequences in a simple-to-use package.

```
# virtual environment
$ pip install rapidfuzz

# normal environment
# use py instead of python3.13 for Windows
$ python3.13 -m pip install --user rapidfuzz
```

Examples

Here are some examples of using the `fuzz.ratio()` method to calculate the similarity between two strings. Output of `100.0` means exact match.

```
>>> from rapidfuzz import fuzz

>>> fuzz.ratio('Courtney Schaefer', 'Courtney Schafer')
96.96969696969697
>>> fuzz.ratio('Courtney Schaefer', 'Courtney Shafer')
93.75
```

If you decide `90` as the cut-off limit, here are some cases that will be missed.

```
>>> fuzz.ratio('Ursella LeGuin', 'Ursula K. LeGuin')
80.0
>>> fuzz.ratio('robin hobb', 'Robin Hobb')
80.0
>>> fuzz.ratio('R. F. Kuang', 'RF Kuang')
84.21052631578947
```

Ignoring string case and removing the `.` characters before comparison helps in some cases.

```
>>> fuzz.ratio('robin hobb'.lower(), 'Robin Hobb'.lower())
100.0
>>> fuzz.ratio('R. F. Kuang'.replace('.', ''), 'RF Kuang'.replace('.', ''))
94.11764705882352
```

Here's an example where two different authors have only a single character difference. This would result in a false positive, which can be improved if the book names are also compared.

```
>>> fuzz.ratio('R.J. Barker', 'R.J. Parker')
90.9090909090909
```

Top authors

The below program processes the author lists created earlier.

```
# top_authors.py
from rapidfuzz import fuzz

ip_files = ('authors_2019.txt', 'authors_2021.txt')
op_files = ('top_authors_2019.csv', 'top_authors_2021.csv')

for ip_file, op_file in zip(ip_files, op_files):
    authors = {}
    with open(ip_file) as ipf, open(op_file, 'w') as opf:
        for line in ipf:
            name = line.rstrip('\n')
            authors[name] = authors.get(name, 0) + 1

    fuzzed = {}
    for k1 in sorted(authors, key=lambda k: -authors[k]):
        s1 = k1.lower().replace('.', '')
        for k2 in fuzzed:
            s2 = k2.lower().replace('.', '')
            if round(fuzz.ratio(s1, s2)) >= 90:
                fuzzed[k2] += authors[k1]
                break
        else:
            fuzzed[k1] = authors[k1]

    opf.write(f'Author,votes\n')
    for name in sorted(fuzzed, key=lambda k: -fuzzed[k]):
        votes = fuzzed[name]
        if votes >= 5:
            opf.write(f'{name},{votes}\n')
```


First, a naive histogram is created with the author name as the *key* and the total number of exact matches as the *value*.

Then, `rapidfuzz` is used to merge similar author names. The `sorted()` function is used to allow the most popular spelling to win.

Finally, the fuzzed dictionary is sorted again by highest votes and written to output files. The result is written in `csv` format with a header and a cut-off limit of minimum `5` votes.

Here's a table of the top-10 authors:

2021	Votes	2019	Votes
Ursula K. Le Guin	139	N.K. Jemisin	58
Robin Hobb	127	Ursula K. Le Guin	57
N.K. Jemisin	127	Lois McMaster Bujold	52
Martha Wells	113	Robin Hobb	47
Lois McMaster Bujold	112	J.K. Rowling	47
Naomi Novik	110	Naomi Novik	45
Susanna Clarke	81	Becky Chambers	36
Becky Chambers	76	Katherine Addison	33
Katherine Addison	74	Martha Wells	30
Madeline Miller	72	Jacqueline Carey	29

 Note that the results you get might be different than what is shown here due to modification of the Reddit comments under analysis. Or, users might have deleted their comments and so on.

If you wish to compare with the actual results, visit the threads linked below (see the comment section for author name based counts). The top-10 list shown above happens to match the actual results for both the polls, but with a slightly different order and vote counts.

- [2021 poll results](#)
- [2019 poll results](#)

Exercises

- Combine `extract_author_names.py` and `top_authors.py` into a single script so that the intermediate files aren't needed.
- Give your best shot at salvaging some of the vote entries that were discarded in the above scripts.
- Display a list of author names who got at least **10** votes in 2021 but less than **5** votes in 2019.
 - You'll have to fuzzy match the author names since the spelling that won could be different between the two lists.
- Find out the top-5 authors who had at least **5** votes in both the lists and had the biggest gain in 2021 compared to the 2019 data. You can decide how to calculate the gain — vote count or percentage increase.
- Explore the [word_cloud](#) package for visually displaying the results.

Further Reading

- `praw`
 - praw.readthedocs.io
 - [Authenticating via OAuth](#)
 - [Comment Extraction and Parsing](#)
 - [/r/redditdev/](#) — subreddit for discussion of reddit API clients
 - [stackoverflow: top praw Q&A](#)
- [Python re\(gex\)?](#) — my ebook on Regular Expressions

- [My list of resources for Data Science and Data Analysis](#)
- [rich](#) — library for *rich* text and beautiful formatting in the terminal