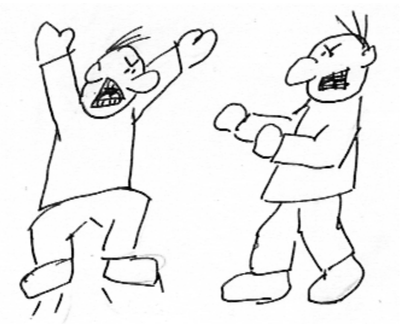
1. Dial-A-Curse: Generating random insults from lists of words

> "He or she is a slimy-sided, frog-mouthed, silt-eating slug with the brains of a turtle." — Dial-A-Curse



Random events are at the heart of interesting games and puzzles. Humans quickly grow bored of things that are always the same, so let's learn how to make our programs more interesting by having them behave differently each time they are run. This exercise will introduce how to randomly select one or more elements from lists of options. To explore randomness, we'll create a program called `abuse.py` that will insult the user by randomly selecting adjectives and nouns to create slanderous epithets.

In order to test randomness, though, we need to control it. It turns out that "random" events on computers are rarely actually random but only "pseudo-random," which means we can control them using a "seed."[1] Each time you use the same seed, you get the same "random" choices!

Shakespeare had some of the best insults, so we'll draw from the vocabulary of his works. Here is the list of adjectives you should use:

> bankrupt base caterwauling corrupt cullionly detestable dishonest false filthsome filthy foolish foul gross heedless indistinguishable infected insatiate irksome lascivious lecherous loathsome lubbery old peevish rascaly rotten ruinous scurilous scurvy slanderous sodden-witted thin-faced toad-spotted unmannered vile wall-eyed

And these are the nouns:

> Judas Satan ape ass barbermonger beggar block boy braggart butt carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool gull harpy jack jolthead knave liar lunatic maw milksop minion ratcatcher recreant rogue scold slave swine traitor varlet villain worm

For instance, it might produce the following:

```
$ ./abuse.py
You slanderous, rotten block!
You lubbery, scurilous ratcatcher!
You rotten, foul liar!
```

In this exercise, you will learn to:

- Use `parser.error()` from `argparse` to throw errors
- Learn about random seeds to control randomness
- Take random choices and samples from Python lists
- Iterate an algorithm a specified number of times with a `for` loop
- Format output strings

1.1. Writing `abuse.py`

The arguments to this program are options that have default values, meaning it can run with no arguments at all. The `-n` or `--number` option will default to `3` and will determine how many insults are created:

```
$ ./abuse.py --number 2
You filthsome, cullionly fiend!
You false, thin-faced minion!
```

And the `-a` or `--adjectives` option should default to `2` and will determine how many adjectives are used in each insult:

```
$ ./abuse.py --adjectives 3
You caterwauling, heedless, gross coxcomb!
You sodden-witted, rascaly, lascivious varlet!
You dishonest, lecherous, foolish varlet!
```

Lastly, your program should accept a `-s` or `--seed` argument (default `None`) that will control the randomness of the program. The following should be exactly reproducible, no matter who runs the program on any machine at any time:

```
$ ./abuse.py --seed 1
You filthsome, cullionly fiend!
You false, thin-faced minion!
You sodden-witted, rascaly cur!
```

When run with no arguments, the program should generate insults using the defaults:

```
$ ./abuse.py
You foul, false varlet!
You filthy, insatiate fool!
You lascivious, corrupt recreant!
```

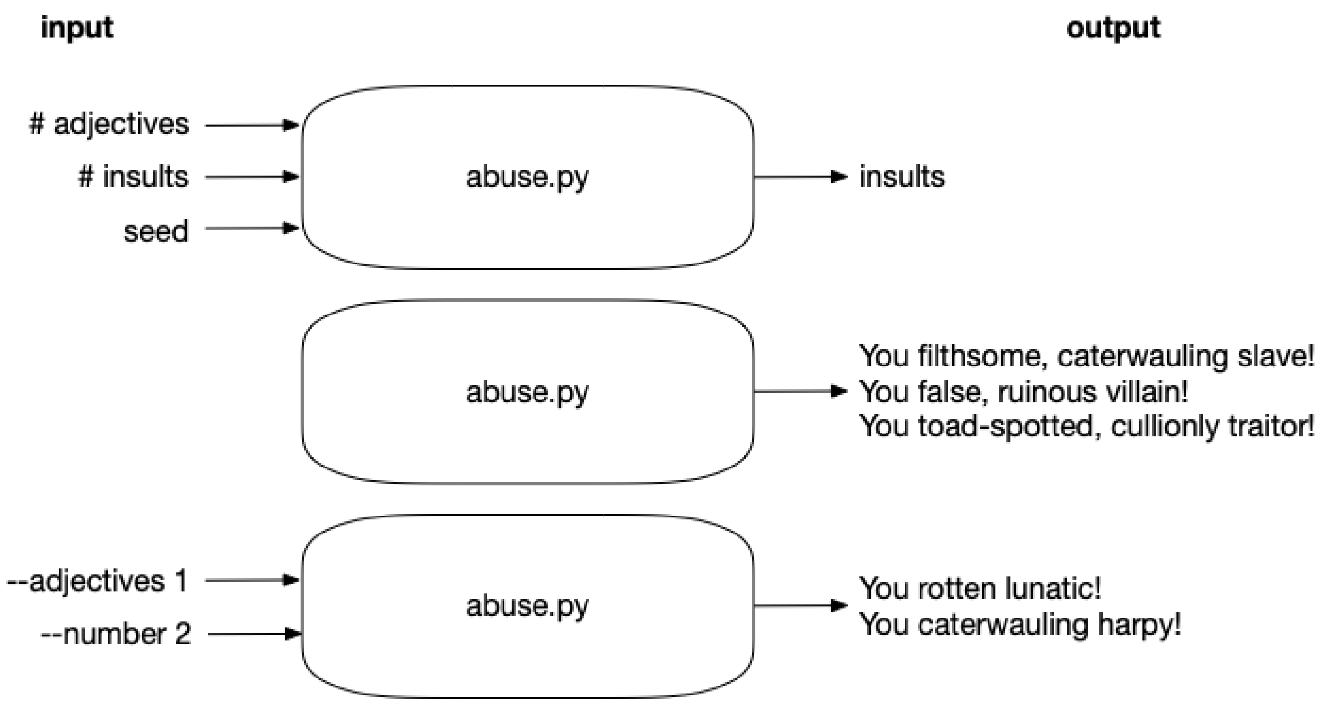
I recommend you start by copying the `template/template.py` to `abuse/abuse.py` or by using `new.py` to create the `abuse.py` program in the `abuse` directory of your repository. The next step is to make your program produce the following usage for `-h` or `--help`:

```
$ ./abuse.py -h
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]

Heap abuse

optional arguments:
  -h, --help            show this help message and exit
  -a adjectives, --adjectives adjectives
                        Number of adjectives (default: 2)
  -n insults, --number insults
                        Number of insults (default: 3)
  -s seed, --seed seed  Random seed (default: None)
```

Here is a string diagram to help you see the program:



1.1.1. Validating arguments

All of the options for number of insults and adjectives as well as the random seed should all be `int` values. If you define each using `type=int` (remember there are no quotes around the `int`), then `argparse` will handle the validation and conversion of the arguments to an `int` value for you. That is, just by defining `type=int`, the following error will be generated for you:

```
$ ./abuse.py -n foo
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: argument -n/--number: invalid int value: 'foo'
```

Additionally, if either `--number` or `--adjectives` less than 1, your program should exit with an error code and message:

```
$ ./abuse.py -a -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-4" must be > 0
$ ./abuse.py -n -4
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --number "-4" must be > 0
```

As you start to write your own programs and tests, I recommend you steal from the tests I've written. [2] Let's take a look at one of the tests in `test.py` to see how the program is tested:

```
def test_bad_adjective_num():                                ①
    """bad_adjectives"""

    n = random.choice(range(-10, 0))                          ②
    rv, out = getstatusoutput(f'{prg} -a {n}')                ③
    assert rv != 0                                            ④
    assert re.search(f'--adjectives "{n}" must be > 0', out) ⑤
```

① The name of the function must start with `test_` in order for `pytest` to find and run it.

② Use the `random.choice()` function to randomly select a value from the `range` of numbers from -10 to 0. We will use this same function in our program, so note here how it is called!

③ Run the program using the `getstatusoutput()` from the `subprocess` module using a bad `-a` value. This function returns the exit value (which I put into `rv` for "return value") and standard out (`out`).

④ Assert that the return value (`rv`) is not 0 where "0" would indicate success (or "zero errors").

⑤ Assert that the output somewhere contains the statement that the `--adjectives` argument must be greater than 0.

I would recommend you look at the section on "Manually checking arguments" in the `argparse` appendix. There I introduce the `parser.error()` function that you can call inside the `get_args()` function to do the following:

1. Print the short usage statement.
2. Print an error message to the user.
3. Stop execution of the program.
4. Exit with a non-zero exit value to indicate an error.

That is, your `get_args()` normally finishes with:

```
return args.parse_args()
```

Instead, put the `args` into a variable and check the `args.adjectives` value to see if it’s less than 1. If it is, call `parser.error()` with an error message to report to the user:

```
args = parser.parse_args()

if args.adjectives < 1:
    parser.error(f'--adjectives "{args.adjectives}" must be > 0')
```

Also do this for the `args.number`. If they are both fine, then you can `return` the arguments to the calling function:

```
return args
```

1.1.2. Importing and seeding the `random` module

Once you have defined and validated all the program’s arguments, you are ready to heap scorn upon the user. We need to add `import random` to our program so we can use functions from that module to select adjectives and nouns. It’s best practice to list all your `import` statements, one module at a time, at the top of your program.

As usual first thing we need to do in the `main()` is to call `get_args()` to get our arguments, and the very next step is to pass the `args.seed` value to the `random.seed()` function:

```
def main()
    args = get_args()
    random.seed(args.seed)
```

We can read about the `random.seed()` function in the REPL:

```
>>> import random
>>> help(random.seed)
```

There we learn that the function will "initialize internal state from hashable object." That is, we set an initial value from some *hashable* Python type. Both `int` and `str` types are hashable, but the tests are written with the expectation that you will define the `seed` argument as an `int`. (Remember that the character `'1'` is different from the *integer value* `1`!) The default value for `args.seed` is `None`. If the user has not indicated any seed, this is the same as not setting it at all.

If you look at the `test.py` program, you will notice that all the tests that expect a particular output will pass a `-s` or `--seed` argument. Here is the first test for output:

```
def test_01():
    out = getoutput(f'{prg} -s 1 -n 1') ①
    assert out.strip() == 'You filthsome, cullionly fiend!' ②
```

① Run the program using the `getoutput` from the `subprocess` module. Use a seed value of `1`, anad request 1 insult. This function returns only the output from the program.

② Verify that the entire output is the one expected insult.

1.1.3. Defining the adjectives and nouns

Above I’ve given you a long list of adjectives and nouns that you should use in your program. In order to pass the tests, they must be in the same order as I have provided. (You may notice that they are alphabetically sorted.) You could create a `list` by individually quoting each word:

```
>>> adjectives = ['bankrupt', 'base', 'caterwauling']
```

Or you could save yourself a good bit of typing if you use the `str.split` function to create a new list from a str by splitting on spaces:

```
>>> adjectives = 'bankrupt base caterwauling'.split()
>>> adjectives
['bankrupt', 'base', 'caterwauling']
```

If you try to make one giant string of all the adjectives, it will wrap around and look ugly. I'd recommend you use triple quotes (either single or double quotes) that allow you to include newlines:

```
>>> """
... bankrupt base
... caterwauling
... """.split()
['bankrupt', 'base', 'caterwauling']
```

Once you have variables for adjectives and nouns, you might check that you have the right number:

```
>>> assert len(adjectives) == 36
>>> assert len(nouns) == 39
```

1.1.4. Taking random samples and choices

In addition to the random module's seed() function, we will also use the choice() and sample() functions. In the test_bad_adjective_num function above, we saw one example of using random.choice(). We can use it similarly to select a noun from the list of nouns. Notice that this function returns a single item, so, given a list of str values, it will return a single str:

```
>>> random.choice(nouns)
'braggart'
>>> random.choice(nouns)
'milksop'
```

For the adjectives, you should use random.sample(). If you read the help(random.sample), you will see this function takes the list of adjectives and a k parameter for how many items to sample:

```
sample(population, k) method of random.Random instance
    Chooses k unique random elements from a population sequence or set.
```

Note that this function returns a new list:

```
>>> random.sample(adjectives, 2)
['detestable', 'peevish']
>>> random.sample(adjectives, 3)
['slanderous', 'detestable', 'base']
```

There is also a `random.choices` that works similarly to `sample` but which might select the same items twice because it samples "with replacement," so we will not use that.

1.1.5. Formatting the output

The output of the program is some `--number` of insults, which you could generate using a `for` loop and the `range` function. It doesn't matter here that `range` starts at zero. What's important is that it generates three values:

```
>>> for n in range(3):
...     print(n)
...
0
1
2
```

You can loop the `--number` of times needed, select your sample of adjectives and your noun, and then format the output. Each insult should start with the string "You ", then have the adjectives joined on a comma and a space, then the noun, and finish with an exclamation point. You could use either an f-string or the `str.format` function to `print` the output to `STDOUT`.

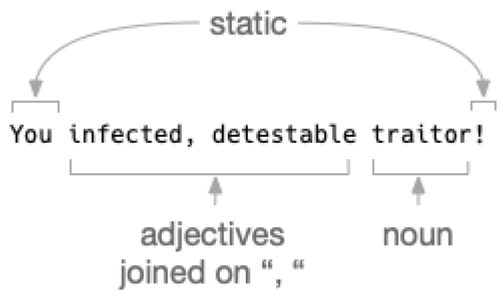


Figure 9. 1. Each insult will combine the chosen adjectives joined on commas with the selected noun and some static bits of text.

Hints:

- Perform the check for positives values for `--adjectives` `--number` *inside* the `get_args()` function and use `parser.error()` to throw the error while printing a message and the usage.
- If you set the default for `args.seed` to `None` while using a `type=int`, you should be able to directly pass the argument's value to `random.seed()` to control testing.
- Use a `for` loop with the `range()` function to create a loop that will execute `--number` of times to generate each insult.
- Look at the `random.sample()` and `random.choice()` functions for help in selecting some adjectives

and a noun.

- You can use three single (`'''`) or double quotes (`"""`) to create a multi-line string and then `str.split()` that to get a list of strings. This is easier than individually quoting a long list of shorter strings (e.g., the list of adjectives and nouns).
- To construct an insult string to print, you can use the `+` operator to concatenate strings, use the `str.join()` method, or use format strings.

Now give this your best shot before reading ahead to the solution!

1.2. Solution

```
#!/usr/bin/env python3
"""Heap abuse"""

import argparse
import random                                         ①


# --------------------------------------------------
def get_args():
    """Get command-line arguments"""

    parser = argparse.ArgumentParser(
        description='Heap abuse',
        formatter_class=argparse.ArgumentDefaultsHelpFormatter)

    parser.add_argument('-a',                         ②
                        '--adjectives',
                        help='Number of adjectives',
                        metavar='adjectives',
                        type=int,
                        default=2)

    parser.add_argument('-n',                         ③
                        '--number',
                        help='Number of adjectives',
                        metavar='adjectives',
                        type=int,
                        default=3)

    parser.add_argument('-s',                         ④
                        '--seed',
                        help='Random seed',
                        metavar='seed',
                        type=int,
                        default=None)

    args = parser.parse_args()                        ⑤

    if args.adjectives < 1:                           ⑥
        parser.error('--adjectives "{}" must be > 0'.format(args.adjectives))

    if args.number < 1:                               ⑦
        parser.error('--number "{}" must be > 0'.format(args.number))

    return args                                       ⑧


# --------------------------------------------------
```

```
def main():
    """Make a jazz noise here"""

    args = get_args()                                    ⑨
    random.seed(args.seed)                               ⑩

    adjectives = """                                     ⑪
    bankrupt base caterwauling corrupt cullionly detestable dishonest false
    filthsome filthy foolish foul gross heedless indistinguishable infected
    insatiate irksome lascivious lecherous loathsome lubbery old peevish
    rascaly rotten ruinous scurilous scurvy slanderous sodden-witted
    thin-faced toad-spotted unmannered vile wall-eyed
    """.strip().split()

    nouns = """                                          ⑫
    Judas Satan ape ass barbermonger beggar block boy braggart butt
    carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
    gull harpy jack jolthead knave liar lunatic maw milksop minion
    ratcatcher recreant rogue scold slave swine traitor varlet villain worm
    """.strip().split()

    for _ in range(args.number):                         ⑬
        adjs = ', '.join(random.sample(adjectives, k=args.adjectives)) ⑭
        print(f'You {adjs} {random.choice(nouns)}!')     ⑮


# --------------------------------------------------
if __name__ == '__main__':
    main()
```

① Bring in the `random` module so we can use functions.

② Define the parameter for the number of adjectives, setting the `type=int` and the default value.

③ Similarly define the parameter for the number insults.

④ The random seed default should be `None`.

⑤ Get the `args` from parsing the argument. Error such as non-integer values will be handled at this point by `argparse`.

⑥ If we make it to this point, we can perform additional checking such as verifying that the `args.adjectives` is greater than 0. If there is a problem, call `parser.error()` with the error message to print.

⑦ Similarly check for the `args.number`.

⑧ If we make it to this point, then all the user's arguments have been validated.

⑨ This is where the program actually begins as it is the first action inside our `main()`. We always start off by getting the arguments.

⑩ Set the `random.seed()` using whatever value was passed by the user. Any `int` value is valid, and we know that `argparse` has handled the validation and conversion of the argument to an `int`.

⑪ Create a `list` of `adjectives` by splitting the very long string contained in the triple quotes.

⑫ Do the same for the `list` of `nouns`.

⑬ Use a `for` loop over the `range` of the `args.number`. Since we don't actually need the value from the `range`, we can use the `_` to disregard it.

⑭ Use the `random.sample()` function to select the correct number of adjectives and join them on the comma-space `str`.

⑮ Use an f-string to format the output to `print`.

1.3. Discussion

1.3.1. Defining the arguments

More than half of my solution is just in defining the program's arguments to `argparse`. The effort is well worth the result, because `argparse` will ensure that each argument is a valid integer value because I set `type=int`. Notice there are no quotes around the `int` — it's not the string `'int'` but a reference to the class in Python.

```
parser.add_argument('-a',                          ①
                    '--adjectives',                ②
                    help='Number of adjectives',   ③
                    metavar='adjectives',          ④
                    type=int,                      ⑤
                    default=2)                     ⑥
```

① The short flag.

② The long flag.

③ The help message.

④ A description of the parameter.

⑤ The actual Python `type` for converting the input, note that this is the bareword `int` for the integer class.

⑥ The default value for the number of adjectives per insult.

For `--adjectives` and `--number`, I can set reasonable defaults so that no input is required from the user. This makes your program dynamic, interesting, and testable. How do you know if your values are being used correctly unless you change them and test that the proper change was made in your program? Maybe you started off hardcoding the number of insults and forgot to change the `range` to use a variable. Without changing the input value and testing that the number of insults changed accordingly, it might be a user who discovers your bug, and that's somewhat embarrassing.

1.3.2. Using `parser.error()`

I really love the `argparse` module for all the work it saves me. For instance, the `type=int` saves me all the trouble of verifying that the input is an integer, creating an error message, and then converting the user's input to an actual `int` value.

Something else I enjoy about `argparse` is that, if I find there is a problem with an argument, I can use `parser.error()` to do four things:

1. Print the short usage of the program to the user
2. Print a specific message about the problem
3. Halt execution of the program
4. Return an error code to the operating system

I can’t very easily tell `argparse` that the `--number` should be a positive integer, only that it must be of type `int`. I can, however, inspect the value myself and call `parser.error('message')` if there is a problem. I do all this inside `get_args()` so that, by the time I get the `args` in my `main()` function, I know they have been validated.

I highly recommend you tuck this tip into your back pocket. It can prove quite handy, saving you loads of time validating user input and generating useful error messages. (And it’s really quite likely that the future user of your program will be *you*, and you will really appreciate your efforts!)

1.3.3. Program exit values and `STDERR`

I would like to highlight the exit value of your program. Under normal circumstances, your program should exit with a value of `0`. In computer science, we often think of `0` as a `False` value, but here it’s quite positive. In this instance we should think of it like "zero errors." If you use `sys.exit()` in your code to exit a program prematurely, the default exit value is `0`. If you want to indicate to the operating system or some calling program that your program exited with an error, you should return *any value other than* `0`. The `parser.error()` function does this for you automatically.

Additionally, it’s common for all error messages to be printed not to `STDOUT` (standard out) but to `STDERR` (standard error). Many command shells (like `bash`) can segregate these two output channels using `1` for `STDOUT` and `2` for `STDERR`. Notice how I can use `2>` to redirect `STDERR` to the file called `err` so that nothing appears on `STDOUT`:

```
$ ./abuse.py -a -1 2>err
```

And now we can verify that the expected error messages are in the `err` file:

```
$ cat err
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: --adjectives "-1" must be > 0
```

If you were to handle all of this yourself, you would need to write something like:

```
if args.adjectives < 1:
    parser.print_usage() ①
    print(f'--adjectives "{args.adjectives}" must be > 0', file=sys.stderr) ②
    sys.exit(1) ③
```

① Print the short "usage". You can also `parser.print_help()` to print the more verbose output for `-h`.

② Print the error message to the `sys.stderr` file handle. This is similar to the `sys.stdout` file handle we used in the "Howler" exercise.

③ Exit the program with a value that is not `0` to indicate an error.

Writing Pipelines

As you write more programs in your career, you may eventually start chaining them together. We often call these "pipelines" as the output of one program is "piped" to become the input for the next program. If there is an error in any part of the pipeline, we want the entire operation to stop so that the problems can be fixed. A non-zero return value from any program is a warning flag to halt operations.

1.3.4. Controlling randomness with `random.seed()`

Once I'm in `main()` and have my arguments, I can control the randomness of the program by calling `random.seed(args.seed)` because the default value of the `seed` is `None`, and setting `random.seed()` to `None` is the same as not setting it at all. The `type` of `args.seed` is `int` which is the expected type for our tests. I do not have to validate the argument further. Note that negative integer values are valid!

1.3.5. Iterating `for` loops with `range`

To generate some `--number` of insults, I use the `range` function. Because I don't need the number of the insult, I can use the underscore (`_`) as a throwaway value:

```
>>> num_insults = 2
>>> for _ in range(num_insults):
...     print('An insult!')
...
An insult!
An insult!
```

The underscore is a way to unpack a value and indicate that you do not intend to use it. That is, it's not possible to write this:

```
>>> for in range(num_insults):
  File "<stdin>", line 1
    for in range(num_insults):
```

You have to put *something* after the `for` that looks like a variable. If you put a named variable like `n` and then don't use it in the loop, some tools like `pylint` will detect this as a possible error (and well it could be). The `_` shows that you won't use it, which is good information for your future self, some other user, or external tools to know.

You can use multiple `_` variables in the same statement. For instance, I can unpack a 3-tuple so as to get the middle value:

```
>>> x = 'Jesus', 'Mary', 'Joseph'
>>> _, name, _ = x
>>> name
'Mary'
```

1.3.6. Constructing the insults

To create my list of adjectives, I used the `str.split` method on a long, multi-line string I created using three quotes:

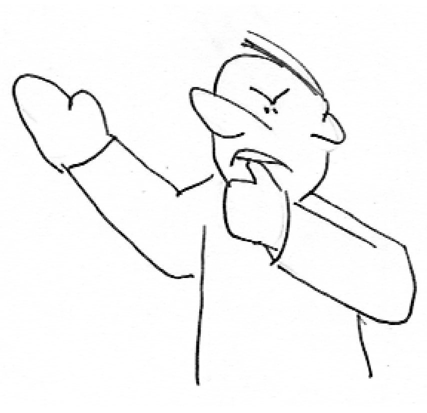
```
>>> adjectives = """
... bankrupt base caterwauling corrupt cullionly detestable dishonest
... false filthsome filthy foolish foul gross heedless indistinguishable
... infected insatiate irksome lascivious lecherous loathsome lubbery old
... peevish rascaly rotten ruinous scurilous scurvy slanderous
... sodden-witted thin-faced toad-spotted unmannered vile wall-eyed
... """.strip().split()
>>> nouns = """
... Judas Satan ape ass barbermonger beggar block boy braggart butt
... carbuncle coward coxcomb cur dandy degenerate fiend fishmonger fool
... gull harpy jack jolthead knave liar lunatic maw milksop minion
... ratcatcher recreant rogue scold slave swine traitor varlet villain worm
... """.strip().split()
>>> len(adjectives)
36
>>> len(nouns)
39
```

To select some number of adjectives, I chose to use `random.sample()` function since I needed more than one:

```
>>> import random
>>> random.sample(adjectives, k=3)
['filthsome', 'cullionly', 'insatiate']
```

For just one randomly selected value, I use `random.choice()`:

```
>>> random.choice(nouns)
'boy'
```



To concatenante them together, I need to put `', '` (a comma and a space) between each of the adjectives, and I can use `str.join` for that:

```
>>> adjs = random.sample(adjectives, k=3)
>>> adjs
['thin-faced', 'scurvy', 'sodden-witted']
>>> ', '.join(adjs)
'thin-faced, scurvy, sodden-witted'
```

And feed all this to a format string:

```
>>> adjs = ', '.join(random.sample(adjectives, k=3))
>>> print(f'You {adjs} {random.choice(nouns)}!')
You heedless, thin-faced, gross recreant!
```

And now you have a handy way to make enemies and influence people.

1.4. Review

- To indicate a problem to `argparse`, use the `parser.error()` function to print a short usage, report the problem, and exit the program with an error value to indicate a problem.
- The `str.split()` method is a useful way to create a `list` of string values from a long string.
- The `random.seed()` function can be used to make reproducible "random" selections each time a program is run.
- The `random.choice()` and `random.sample()` functions are useful for randomly selecting one or several items from a list of choices, respectively.

1.5. Going Further

- Read your adjective and nouns from files that are passed as arguments.
- Add tests to verify that the files are processed correctly and new insults are still stinging.

[1] "The generation of random numbers is too important to be left to chance." — Robert R. Coveyou

[2] "Good composers borrow, Great ones steal." — Igor Stravinsky