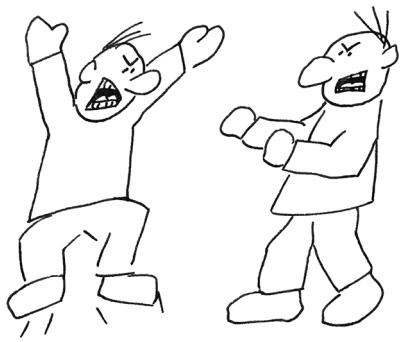
9 Dial-a-Curse: Generating random insults from lists of words

"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. I think one reason people may choose to have pets and children is to inject some randomness into their lives. Let's learn how to make our programs more interesting by having them behave differently each time they are run.

This exercise will show you 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 truly random but only *pseudo-random*, which means we can control them by using a "seed."[1] Each time you use the same seed, you will 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

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

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
- Control randomness with random seeds
- Take random choices and samples from Python lists
- Iterate an algorithm a specified number of times with a `for` loop
- Format output strings

9.1 Writing abuse.py

You should go into the 09_abuse directory to create your new program. Let's start by looking at the usage statement it should produce:

```
$ ./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)
```

All parameters are options that have default values, so our program will be able to run with no arguments at all.

For instance, the `-n` or `--number` option will have a default of `3` and will control the number of insults:

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

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, the `-s` or `--seed` option will control the random choices in the program by setting an initial value. The default should be the special `None` value, which is like an undefined value.

Because the program will use a random seed, the following output should be exactly reproducible by any user 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 file to abuse/abuse.py or by using new.py to create the abuse.py program in the 09_abuse directory of your repository.

Figure 9.1 is a string diagram that illustrates the parameters for the program.

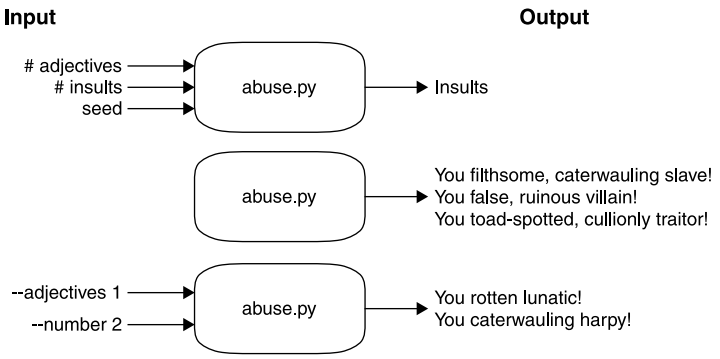


Figure 9.1 The abuse.py program will accept options for the number of insults to create, the number of adjectives per insult, and a random seed value.

9.1.1 Validating arguments

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

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

Not only must the value be a number, but it must be an *integer* which means it must be a whole number, so `argparse` will complain if you give it something that looks like a `float`. Note that you can use `type=float` when you actually want a floating-point value:

```
$ ./abuse.py -a 2.1
usage: abuse.py [-h] [-a adjectives] [-n insults] [-s seed]
abuse.py: error: argument -a/--adjectives: invalid int value: '2.1'
```

Additionally, if either `--number` or `--adjectives` is 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:

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

The name of the function must start with "test_" in order for Pytest to find and run it.

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.

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

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

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

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

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

[3] The `subprocess` module allows you to run a command from inside your program. The `subprocess.getoutput()` function will capture the output from the command, while the `subprocess.getstatusoutput()` will capture both the exit value and the output from the command.

There's no simple way to tell `argparse` that the numbers for adjectives and insults must be greater than zero, so we'll have to check those values ourselves. We'll use the verification ideas from section A.4.7 in the appendix. There I introduce the `parser.error()` function, which 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 nonzero exit value to indicate an error

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

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

Instead, we'll put the `args` into a variable and check the `args.adjectives` value to see if it's less than 1. If it is, we'll 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')
```

We'll also do this for `args.number`. If they are both fine, you can `return` the arguments to the calling function:

```
return args
```

9.1.2 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. First, 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 the `import` statements, one module at a time, at the top of a program.

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

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

We call the random.seed() function to set the initial value of the random module's state. There is no return value from random.seed()—the only change is internal to the random module.

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

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

There you'll learn that the function will "initialize internal state [of the `random` module] from hashable object." That is, we set an initial value from some *hashable* Python type. Both the `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` should be `None`. If the user has not indicated any seed, then setting `random.seed(None)` 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 an `-s` or `--seed` argument. Here is the first test for output:

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

Run the program using getoutput() from the subprocess module using a seed value of 1 and requesting 1 insult. This function returns only the output from the program.

Verify that the entire output is the one expected insult.

This means test.py will run your program and capture the output into the `out` variable:

```
$ ./abuse.py -s 1 -n 1
You filthsome, cullionly fiend!
```

It will then verify that the program did in fact produce the expected number of insults with the expected selection of words.

9.1.3 Defining the adjectives and nouns

Earlier in the chapter, I gave you a long list of adjectives and nouns that you should use in your program. You could create a `list` by individually quoting each word:

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

Or you could save yourself a good bit of typing by using the `str.split()` method 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 be very long and so will wrap around in your code editor and look ugly. I recommend you use triple quotes (either single or double quotes), which will allow you to include newlines:

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

Once you have variables for `adjectives` and `nouns`, you should check that you have the right number of each:

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

NOTE In order to pass the tests, your adjectives and nouns must be in alphabetical order as they were provided.

9.1.4 Taking random samples and choices

In addition to the `random.seed()` function, we will also use the `random.choice()` and `random.sample()` functions. In the `test_bad_adjective_num` function in section 9.1.1, you 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)` output, you will see that this function takes some `list` of items and a `k` parameter for how many items to return:

```
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()` function that works similarly but which might select the same items twice because it samples "with replacement." We will not use that.

9.1.5 Formatting the output

The output of the program is a `--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 (figure 9.2). You could use either an f-string or the `str.format()` function to `print()` the output to `STDOUT`.

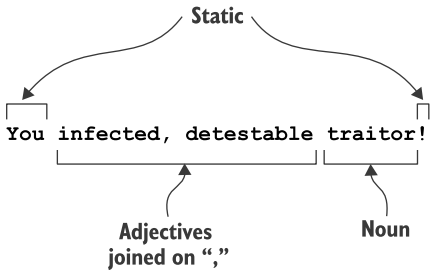


Figure 9.2 Each insult will combine the chosen adjectives joined on commas with the selected noun and some static bits of text.

Here are a few hints:

- Perform the check for positive values for `--adjectives` and `--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 of `args.seed` to `None` and use `type=int`, you can directly pass the value to `random.seed()`. When the value is `None`, it will be like not setting the value at all.
- Use a `for` loop with the `range()` function to create a loop that will execute the `--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 quotes (`'''`) or double quotes (`"""`) to create a multiline string and then use `str.split()` to get a list of strings. This is easier than individually quoting a long list of shorter strings (such as the list of adjectives and nouns).
- To construct an insult 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, you snotty-faced heap of parrot droppings!

9.2 Solution

This is the first solution where I use `parser.error()` to augment the validation of the arguments. I also incorporate triple-quoted strings and introduce the `random` module, which is quite fun unless you're a vacuous, coffee-nosed, malodorous git.

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

import argparse
import random
```

Bring in the random module so we can call functions.

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

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

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

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

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

    args = parser.parse_args()

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

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

    return args

# --------------------------------------------------
def main():
    """Make a jazz noise here"""

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

    adjectives = """
    bankrupt base caterwauling corrupt cullionly detestable dishonest false
    filthsome filthy foolish foul gross heedless indistinguishable infected
```

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

Similarly define the parameter for the number of insults as an integer with a default.

The random seed default should be None.

Get the result of parsing the command-line arguments. The argparse module will handle errors such as non-integer values.

Check that args.adjectives is greater than 0. If there is a problem, call parser.error() with the error message.

Similarly check args.number.

At this point, all the user's arguments have been validated, so return the arguments to the caller.

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

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

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

```
    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()

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


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

Do the same for the list of nouns.

Use an f-string to format the output to print().

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

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

9.3 Discussion

I trust you did not peek at the solution before you passed all the tests or else you are a rascaly, filthsome swine.

9.3.1 Defining the arguments

More than half of my solution is defining the program's arguments to `argparse`. The effort is well worth the result. Because I set `type=int`, `argparse` will ensure that each argument is a valid integer value. Notice that 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',
                    '--adjectives',
                    help='Number of adjectives',
                    metavar='adjectives',
                    type=int,
                    default=2)
```

The short flag

The long flag

The help message

A description of the parameter

The actual Python type for converting the input; note that this is the bare word int for the integer class

The default value for the number of adjectives per insult

I set reasonable defaults for all the program's options so that no input is required from the user. The `--seed` option should default to `None` so that the default behavior is to generate pseudo-random insults. This value is only important for testing purposes.

9.3.2 Using parser.error()

I really love the `argparse` module for all the work it saves me. In particular, I often use `parser.error()` when I find there is a problem with an argument. This function will 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'm using `parser.error()` here because, while I can ask `argparse` to verify that a given value is an `int`, I can't as easily say that it must be a *positive* value. I can, however, inspect the value myself and halt the program 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 quite likely that the future user of your program will be *you*, so you will really appreciate your efforts.)

9.3.3 Program exit values and STDERR

I would like to highlight the exit value of the program. Under normal circumstances, programs 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 as "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`. You can also call the function with a string, which will be printed as an error message, and Python will exit with the value `1`. If you run this in the REPL, you will be returned to the command line:

```
>>> import sys
>>> sys.exit('You gross, thin-faced worm!')
You gross, thin-faced worm!
```

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`. When using the Bash shell, note 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
```

I 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 this:

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

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

Print the error message to the sys.stderr file handle. This is similar to the sys.stdout file handle we used in chapter 5.

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

Writing pipelines

As you write more and more programs, 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, you'll generally want the entire operation to stop so that the problems can be fixed. A nonzero return value from any program is a warning flag to halt operations.

9.3.4 Controlling randomness with random.seed()

The pseudo-random events in the `random` module follow from a given starting point. That is, each time you start from a given state, the events will happen in the same way. We can use the `random.seed()` function to set that starting point.

The seed value must be *hashable*. According to the Python documentation (https://docs.python.org/3.1/glossary.html), "all of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are." In this program, we have to use an integer value because the tests were written using integer seeds. When you write you own programs, you may choose to use a string or other hashable type.

The default for our seed is the special `None` value, which is a bit like an undefined state. Calling `random.seed(None)` is essentially the same as not setting the seed at all, so it makes it safe to write this:

```
random.seed(args.seed)
```

9.3.5 Iterating with range() and using throwaway variables

To generate some `--number` of insults, we can use the `range()` function. Because we don't need the numbers returned by `range()`, we use the underscore (`_`) as the variable name to indicate this is throwaway value:

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

The underscore is a valid variable name in Python. You can assign to it and use it:

```
>>> _ = 'You indistinguishable, filthsome carbuncle!'
>>> _
'You indistinguishable, filthsome carbuncle!'
```

The use of the underscore as a variable name is a convention to indicate that we don't intend to use the value. That is, if we had said `for` num in `range(...)`, some tools like Pylint will see that the `num` variable is not used and will report this as a possible error (and well it could be). The _ indicates that you're throwing this value away, which is good information for your future self, some other user, or external tools to know.

Note that 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'
```

9.3.6 Constructing the insults

To create my list of adjectives, I used the `str.split()` method on a long, multiline string enclosed in triple quotes. I think this is probably the easiest way to get all these strings into my program. The triple quotes allow us to enter line breaks, which single quotes would not allow:

```
>>> 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
```

Because we need one or more adjectives, the `random.sample()` function is a good choice. It will return a `list` of items randomly selected from a given `list`:

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

The `random.choice()` function is appropriate for selecting just one item from a list, such as the noun for our invective:

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

Next we need to concatenate the epithets using `', '` (a comma and a space) similar to what we did in chapter 3 for our picnic items. The `str.join()` function is perfect for this:

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

To create the insult, we can combine the adjectives and nouns inside our template using an f-string:

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

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

9.4 Going further

- Read your adjectives 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.

Summary

- Use the `parser.error()` function to print a short usage statement, report the problem, and exit the program with an error value.
- Triple-quoted strings may contain line breaks, unlike regular single- or double-quoted strings.

- 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 pseudo-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.