Reporting

In chapter 2, we saw scripts that helped librarians and archivists who were limited by issues of data quality and data portability. In this chapter, we'll take one step closer to front-end users by looking at scripts that aid reporting. These scripts are all about organizing and querying the available data and presenting it to library staff—some of them nontechnical—in ways that aid decision making. They analyze logs to highlight actionable information, simplify workflows, and expand the capabilities of the ILS.

Examples

Two survey respondents wrote scripts to analyze logs. Stuart Yeates found that his web server was under attack from bots that ignored his `robots.txt` file; his bash script (figure 3.1) picked out the relevant lines from his logs and sorted them. This let him find the major traffic sources (i.e., bots) and defend against them, improving server performance. Robin Camille Davis wrote a Python script to analyze EZproxy logs. EZproxy logs an enormous amount of data, which can make it hard to find what you're looking for; her script let her zero in on usage patterns by different patron categories (e.g., students vs. faculty) and graph how usage changed throughout the year.

As Davis noted, "We can get pretty good usage stats from the individual database vendors, but with monthly logs like these, why not analyze them yourself? You could do this in Excel, but Python is *much* more flexible, and much faster, and also, I've already written the script for you."[1] In a blog post, she discusses how and why she wrote the script, changes you may need to make to run it on your own logs, and other questions you could answer by modifying the script.[2] (The need to make changes to get others' scripts working in your own environment is recurrent, and we'll tackle it in more depth later this chapter.)

```
tail -100000 /var/log/httpd/access-
nzetc.log |grep facet |awk '{print
$1}'|sort | uniq -c | sort -n
```

Figure 3.1
The entirety of Yeates's bash script

Robin Camille Davis's script
https://github.com/robincamille/ezproxy-analysis/blob/master/ezp-analysis.py

Three survey respondents wrote scripts that generate reports as part of improving various library workflows. We discussed Annie Glerum's quality control script in the last chapter. Matthew S. Collins wrote a Python script to "check a list of ISBNs from a publisher catalog against our holdings to see what we already have or need to order," saving time in the acquisitions workflow. Joe Montibello helped his preservation librarians, who were dissatisfied that they didn't have good metrics on which parts of their collection had been crawled by LOCKSS. He came up with an algorithm to estimate when crawling would be complete. Like many survey respondents, he had his program write its output to a spreadsheet so that it would be easy for his nonprogramming colleagues to integrate into their existing workflows. By automatically generating this spreadsheet as a shared Google doc, he also saved his own time in reporting out, since his coworkers could check the spreadsheet whenever they were curious rather than needing to ask him.

Joe Montibello's script
https://github.com/joemontibello/update-lockss

Finally, three respondents wrote scripts to fill gaps in ILS reporting capacity. Sam Kome simplified weeding by writing a script that "identified print volumes that met a set of rules for de-accession including a rule that [they] be held at more than one of 50+ libraries in our lending network." Cindy Harper found her serials module "cumbersome," so she wrote her own script that lists what's been sent to the bindery and what needs to be claimed. The third reporting script, below, is the subject of this chapter's deep dive.

Deep Dive: Automated ILS Reporting

Esther Verreau wrote an array of scripts (available on GitHub) that "pull stats from the Sierra ILS and report it to the appropriate destinations, Shoutbomb, Civic Technologies Community Connect and Novelist Select." Some of these had originally been written in Millennium's scripting language, but they were "error prone and difficult to maintain" and became obsolete after an upgrade. The new Python scripts run essentially without human intervention.

Esther Verreau's scripts
https://github.com/everreau/sierra-scripts

These scripts cover an array of uses. One generates RSS feeds of new items (compare with LibALERTS, chapter 2). Another generates an internal report of how many patrons had maxed out their holds; this allowed the library to use real data to decide whether it needed to change its holds policy. Indeed, a striking fact about Verreau's scripts is how many of them are *experimental*—written not to provide a new service, but to let the library gather data on whether a new service or policy would be helpful. Once people are moderately fluent at writing code, one-time-use scripts become reasonable to write. This lets people ask questions it wasn't previously feasible to ask and answer them with hard data.

Verreau's `novelist.py` script exports metadata from the library's entire collection, writes it to a file, FTPs that file to Novelist Select, and notifies someone that it happened. Let's walk through the script.

The latest version of the novelist.py#script
https://thatandromeda.github.io/ltr/Chapter3.html

The comments in **Lines 1–14** describe the script's function and clearly indicate what future users will need to change in order to run the script in their own environments. These sorts of explanatory comments are best practice because future users don't necessarily know what you were thinking and don't want to spend time puzzling through the code to find out. (This includes you, six months from now, when you are guaranteed to have forgotten what you were thinking today.)

Lines 17–25 import other Python functionality the script will rely on. These include interoperating with databases (`psycopg2`), the operating system (`os`), and e-mail (`smtplib`).

Lines 27–31 define a function, `strify`, that the program will use later when writing individual lines of metadata to the output file. The notable thing here is *error handling*—`if obj == None:` recognizes that there may be some empty lines returned by the database query and ensures that they don't result in anything being written to the output file.

Lines 33–41 also define a utility function, `put_file`, which FTPs a file to a given directory. Like `strify`, it shows error handling: if it encounters an exception while trying to FTP the file, it prints the exception to the console rather than letting the program crash. This is a great early step in writing programs because it lets you explore how they might fail. (Again, some degree of failure is the normal case for programs; understanding and handling failures is often more achievable than avoiding them entirely.) More elegant revisions would identify the specific types of exceptions that the program encounters in practice and write thoughtful handling of each. Depending on the nature of the problem, good options might be logging the error, retrying the file transfer in case the error was temporary, or removing broken lines from the file and attempting to send the remainder while maintaining a record of the broken lines for subsequent human intervention.

Lines 43–59 construct the database query in raw SQL, pulling bar codes, titles, and unique identifiers from the entire collection. While the rest of the program could be used in any context, these lines rely on the specific ILS being used. Indeed, Verreau would write them differently today because III's new API would allow her to dramatically simplify this part of the code.

For a sense of how much easier and more readable an API, using the same programming language as its surrounding code, can be, have a look at the documentation for credit card processor Stripe. The right sidebar shows how to charge a user's credit card. On Stripe's end, this information all lives in a database that can be queried in SQL, but the API lets you use simple and Pythonic statements like `stripe.Charge.create()` instead of constructing the SQL query. Indeed, you need not know how the database is structured at all, and your code does not have to change if Stripe decides to change its database schema.

Stripe API documentation: Creating a new charge
https://stripe.com/docs/api#create_charge

Lines 61–66 connect to the database and provide a cursor we'll be able to use to examine the results. This cursor will let us step through the results line by line.

Lines 68–78 remove outdated files from our Novelist directory and create today's filename.

Lines 80–81 connect to the database and fetch the results.

Lines 83–92 open our new file and write a title line. The loop (indicated by `for r in rows:`) then writes one line per record from the database, each on a separate line (`\n` is the newline character). Once all records have been written, it closes the file.

Lines 94–104 attempt to log in to the FTP server and transfer the file. The `message` variable created here is the body of the e-mail we will send in **lines 106–114**.

This program demonstrates several best practices: comments, error handling, and descriptive variable and function names. The functions `strify` and `put_file` also demonstrate the usefulness of breaking logically coherent units of functionality into actual functions. By keeping them separate from the main body of the program, we make the overall logic more readable; the program reads like an outline, and we can dig into the specifics only as needed. The program is also easier to debug when you can isolate functionality and zero in on the parts that may need to be fixed. In a larger program, these functions could also be reused. For instance, if we needed to FTP our output file to several servers instead of just one, we don't need to rewrite all that code—we can just call `put_file` again. And if we found that `put_file` was so useful that we needed to use it in multiple programs, we could grow it into a library that could be imported by other programs, just like this program imports `psycopg2`, `os`, and `smtplib`.

Want to modify this program for use locally, and practice your Python skills while you're at it? Here are some things you might try:

- Replace the SQL with a query suitable to your ILS (ideally using an API).
- Have the program import all the local parameters (like `DB_NAME`) from a separate file, *and make sure you keep that file out of GitHub* so you don't inadvertently share sensitive data. (For instance, name it `parameters.py`, and add `parameters.py` to your `.gitignore`.) Alternately, have it harvest this information from environment variables (and add some error checking so that the program will exit if it doesn't have all the data it needs).
- Find out what kind of exceptions might actually be thrown by the `try/except` clauses, and handle them specifically.
- Add error handling in case the e-mail-sending fails.
- Think through what might happen if `NOVELIST_DIR` contains files you didn't expect (e.g., if it's the directory where text files for some other project are being stored) and what you can do about those risks.
- Identify some other case where you need to harvest data from your ILS and e-mail or FTP the results. Modify this script to do that instead.

Scripts in This Chapter

Robin Camille Davis's script
https://github.com/robincamille/ezproxy-analysis/blob/master/ezp-analysis.py

Joe Montibello's script
https://github.com/joemontibello/update-lockss

Esther Verreau's scripts
https://github.com/everreau/sierra-scripts

Notes

1. Robin Camille Davis, "Analyzing EZproxy Logs with Python," *Emerging Tech in Libraries* (blog), April 22, 2014, http://emerging.commons.gc.cuny.edu/2014/04/analyzing-ezproxy-logs-python.
2. Ibid.