Top Ten Best Practices (TA1)

Michael P. Gerlek

mgerlek@inferlink.com

v1.1 / 10 Feb 2024

This document provides several suggestions in ten important areas: things you can do today to make your python applications more robust, reliable, maintainable, scalable, debuggable, and usable.

# 1. Command Line Tools

Design your command line tools to be easy for users to use:

* 1. Use clear names for switches: --config-file and --output-dir are more descriptive than just --config and --output.
  2. If you have a command line switch that references a file or directory, add an early, explicit check to make sure that the file exists, the directory is writable, etc. Don’t wait until the app has been running for 10 minutes to throw a cryptic exception because the user typed the output file name incorrectly.
  3. Do not assume a specific directory layout or file locations. Let the user adapt the tool for his own environment.
  4. Always calls sys.exit() with a value of zero (if the tool was successful) or nonzero (if an error occurred). This allows the tool to interoperate with other tools, especially in automated scenarios.

# 2. Error Handling

* 1. If an error is likely in some area, print a meaningful message and exit the tool: don't rely on potentially cryptic exception tracebacks if you can handle the error yourself.
  2. Catch exceptions narrowly, using the exception type you expect: avoid catch Exception: and never write just catch:.
  3. Don't swallow exceptions. In your catch block, handle the error appropriately and don’t let execution continue if the app can’t recover.
  4. Never call exit() from within a library, as this makes it very difficult to use your library in new contexts, e.g. in a server application. Instead, let the error propagate out to the main() routine using status code returns or exception handling, and then have main call exit().

# 3. Logging

* 1. Use Python’s logging facility to report the state of your application.
  2. Don’t be afraid of logging too much information. Allow the user to control the verbosity of your app using the standard logging severity flags.
  3. Output from print() may not be captured or visible in a batch or server context. Using the logger is more robust.
  4. If your application can run for a long time (one minute or more), consider providing a heartbeat log message. This will indicate to users that the app is still making progress and is not hung. (Extra credit if your heartbeat can provide a percent-complete indication.)

# 4. Deployment

* 1. Use a requirements.txt file and keep it up to date. “Pin” your versions whenever possible.
  2. Don’t use old third-party packages. Take the time to upgrade the packages you rely on to their latest stable releases.
  3. Use a virtual environment like venv or poetry (preferred).
  4. If your system has OS-level dependencies, use a build script to install and configure them. Better yet, consider using a tool like conda with a corresponding conda environment file.
  5. If your tool has specific hardware requirements, e.g. GPUs or lots of RAM, document this clearly in your README.
  6. If your system requires a database, provide an init.sql file.

# Docker

* 1. If you are using docker, use a Dockerfile to carefully define your environment. Do not just “commit” new image versions, as that will not provide a replicable history. (The same advice applies to VMs and AMIs.)
  2. Learn how to mount volumes. In some situations, this is more effective (and efficient) than bulk-copying data into your image. Document what mounts your container expects.
  3. Remember than your container will run as root by default. In some circumstances, this can present a security issue.

# 6. Layered, Object-Oriented Design

* 1. Break long functions up into smaller functions. Generally, a function shouldn’t be longer than about a page. Smaller functions make the code easier to understand and debug.
  2. Encapsulate related functions with shared state into classes. And keep only one (nontrivial) class per file.
  3. Consider how your classes can be organized into layers of functionality, with functions in higher (more general/abstract) layers calling down into lower (more granular, specific) levels to get work done; lower layers never talk “up” to higher layers. A typical layering might look like:
     + your main() (or web server endpoints)
     + your system’s main algorithms and workflows
     + your system’s utility classes and helper functions
     + third-party packages
     + python system libraries
  4. Consider making your application have a main() function which only parses command line options and then calls a high-level “engine” class which does all the “real” work. This will make it easier to use your system in a new context, e.g. for unit testing or to running inside a server.

# 7. Testing

* 1. Learn how to use pytest.
  2. Write unit tests. Provide at least a test stub for each (nontrivial) class in your system. Then, when new code is added, or when errors crop up, you can incrementally expand on the stubs. (If you don’t have the stubs already written and passing, the overhead needed to add them while you are debugging something is too much of a distraction.)
  3. Keep the tests’ run time short enough that there is not friction to running the tests all the time. If a test is too long to do “interactively”, move that test out of your “default” test set.
  4. Design for test: as you write your code, think about how you can test it. Can you set up your (major) functions such that given some known inputs, you get back known (deterministic) results that can be compared – in an automated way – to the expected results?
  5. Where unit tests generally exercise individual classes, system tests exercise higher-level functionality of the system, e.g. requiring the composition of multiple classes. (These are also known as component tests, workflow tests, or end-to-end (“e2e”) tests.) Write these tests as if you were starting from scratch, e.g. do no assume your database tables have already been populated.
  6. Do not try to test everything: don’t let the fear of having to write a combinatorial number of tests scare you away. Focus instead on testing where the common cases and likely buggy cases are.

# 8. Global State

* 1. Global variables should be used for read-only constants.
  2. Sometimes “singleton” classes are useful data structures. They can represent global, mutable state, but if constrained in their use and accessibility through functions, they are good to use.
  3. For data from configuration files or from parsing command-line switches, store the data in a class and pass the class to your underlying classes.

# 9. Working with Files

* 1. If you use structured text files (JSON, YAML, CSV, etc.), provide a simple schema explaining what the expected data structure is. This schema can be written in text in your README. For extra credit, use a library like pydantic to map the file directly into a python class – this is preferable to relying on the use of dict objects which take string keys the user might get wrong, with no error until runtime.
  2. If your application needs to reference files, e.g. input pytorch models or output JSON data: do not hard code file paths, do not assume the files live in the current working directory (“.”), and do not assume that “.” Is writable. Instead, allow your users to set the input and output locations using function parameters or command-line switches.

# 10. Your Users

Finally, when you are writing your application, think of your users: the people who don’t have your familiarity your specific algorithms but who nonetheless someday may need to run, debug, and modify your code. Documenting your code is always good, but think about what else you can do make things easier for others. A little of your time spent now – using more descriptive names, say, or documenting expectations about the execution environment – can save a lot of time for someone else later on.