Lesson 4: Configuring Programs

Reference:
Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson
https://merely-useful.tech/py-rse/config.html

Why configure?

- Programs are only useful if they can be controlled, and work is only **reproducible** if those controls are **explicit** and **shareable**.

- The problem of configuring a program illustrates the difference between “works for me on my machine” and **“works for everyone, everywhere.”**

Four layers of configuration:

1. A system-wide configuration file for general settings.
2. A user-specific configuration file for personal preferences.
3. A job-specific file with settings for a particular run.
4. Command-line options to change things that commonly change.

Overlay configuration:

- A technique for configuring programs in which several layers of configuration are used, each overriding settings in the ones before.

System-wide configuration file

Purpose: To provide default settings **applicable to all users** of a system.

Impact: Changes **affect every user** and application on the system.

Modification and Risks: Editing system-wide files requires caution as incorrect settings can **impact system stability**.

Example: Modifying a system-wide configuration file in an operating system to change the default network timeout settings. This change will affect all network-related operations for all users.

User-Specific Configuration

Scope: Settings that apply only to **a single user's environment**.

Flexibility: Allows **personalization** without affecting other users.

Storage: Typically stored in user's home directory or specified user profile sections.

Example: A user creates a configuration file in their document editor to set a default font size and page layout, different from the system-wide defaults.

Job-Specific Configuration

Application: Used for settings that apply to **a specific task** or project.

Priority: Overrides system and user-specific settings **for the job's duration**.

Creation and Use: Crafted for individual projects or tasks, often located within the project directory.

Example: Setting up a configuration file in a data analysis project to specify data sources and output formats unique to that project.

Command-line options

Use: For temporary adjustments or **one-off changes**.

Flexibility: Allows quick, on-the-fly changes **without altering permanent configurations**.

Best Practices: Use for frequent or minor changes; **avoid for complex configurations**.

Example: Running a file compression tool with command-line options to set a high compression level for a specific large file, overriding the default compression setting.

Configuration File Formats

Popular Formats: Python modules for Python-centric tools; INI for simple, structured data; YAML for readability and complex configurations.

Pros and Cons: Python modules can't be used by other languages, INI is becoming outdated, YAML is versatile but requires careful syntax.

Due to its flexibility and readability, especially for complex configurations, YAML is recommended.

Example: Configuring plot parameters

Creating a configuration file called `plotparams.yml`:

```
# Plot characteristics
axes.labelsize   : x-large  ## fontsize of the x and y labels
xtick.labelsize  : large    ## fontsize of the tick labels
ytick.labelsize  : large    ## fontsize of the tick labels
```

Inspect it:

```
with open('plotparams.yml', 'r') as reader:
    plot_params = yaml.load(reader, Loader=yaml.BaseLoader)
print(plot_params)
```

```
{'axes.labelsize': 'x-large',
 'xtick.labelsize': 'large',
 'ytick.labelsize': 'large'}
```

Example: Configuring plot parameters

Write function to set plot parameters drawing from YAML:

```
def set_plot_params(param_file):
    """Set the matplotlib parameters."""
    if param_file:
        with open(param_file, 'r') as reader:
            param_dict = yaml.load(reader,
                                   Loader=yaml.BaseLoader)
    else:
        param_dict = {}
    for param, value in param_dict.items():
        mpl.rcParams[param] = value
```