# Python Security Best Practices Cheat Sheet - 2021 Edition

snyk

In 2019, Snyk released its first Python cheat sheet. Since then, many aspects of Python have changed, so we've had to make updates. Here is the 2021 version.

# 1. Always sanitize external data

A vector of attack for any application is external data, which can be used for injection, XSS, or denial of service (DOS) attacks. So the general rule needs to be to sanitize data as soon as possible and to use secure functions throughout the application.

Starting with sanitization, it always makes more sense to check for what the input should be than to try to handle the exceptions. It's also recommended to use well maintained libraries for sanitization. Here are two:

- schema is "a library for validating Python data structures, such as those obtained from config-files, forms, external services or command-line parsing, converted from JSON/YAML (or something else) to Python data-types.
- beach is "an allowed-list-based HTML sanitizing library that escapes or strips markup and attributes."

The major frameworks come with their own sanitation functions like Flask's `flask.escape()` or Django's `django.utils.html.escape()`. The goal of any of these functions is to secure possibly malicious HTML input such as this:

```
>>> import bleach
>>> bleach.clean('an XSS <script>navigate(...)</script> example')
'an XSS &lt;script&gt;navigate(...)&lt;/script&gt; example'
```

The limitation of this approach is that these libraries are not good for everything. They are specialized in their domain. As a note, be sure to read to the end of this post to get more information about other data formats, such as XML.

Another often used option is to leave the rendering of HTML to templating engines such as Jinja. It provides lots of capabilities, and amongst them is auto-escaping to prevent XSS using MarkupSafe.

Another aspect of sanitization is preventing data from being used as a command. A typical example is an SQL injection. Instead of stitching strings and variables together to generate an SQL query, it is advisable to use named-parameters to tell the database what to treat as a command and what as data.

```
# Instead of this …
cursor.execute(f"SELECT admin FROM users WHERE username =
'{username}'");
# ...do this...
cursor.execute("SELECT admin FROM users WHERE username = %(username)
s", {'username': username});
```

Or even better, use Object-Relational Mapping (ORM), such as sqlalchemy, which would make the example query look like this:

```
query = session.query(User).filter(User.name.like('%{username}'))
```

Here you also declare clearly what you want to treat as command and what as data or parameter plus you give the ORM layer the opportunity to apply optimizations like caching. So you benefit beyond being more secure.

If you want to learn more, check out our cheat sheet on SQL injection.

# Scan your code

Python developers have a wide array of static code analysis tools at their disposal. Let's take a look at three different levels of tools.

First, the linter level. PEP8 has been serving for decades now as a style guide for Python. Various tools are available (and built into IDEs) to check against this style guide, like pep8, pylint, flake8, and more.

Next, tools like bandit transform code into an abstract syntax tree and perform queries on it to find typical security issues. This is a level above what typical linters do which work on a syntactical level. Still, bandit is limited by its intermediate representation and its performance. For example, bandit cannot detect data flow related issues (known as taint-analysis) and these result in the most devastating flaws (injections like SQL injection or XSS as an example).

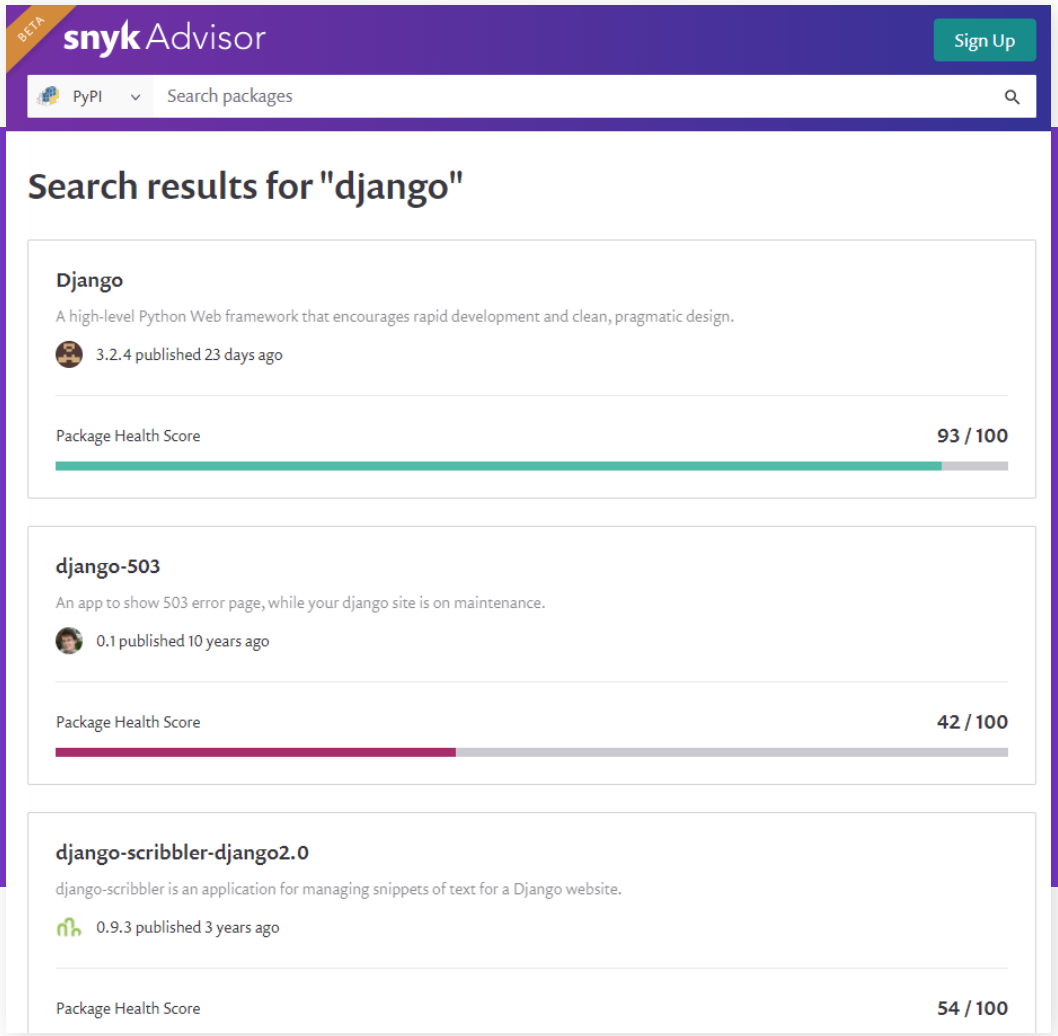
Finally, Static Application Security Testing (SAST) tools like Snyk Code run a semantic analysis taking even complex interfile issues into account. Unlike other tools on this level, Snyk Code is developer-friendly by scanning fast and integrating into the IDE. Snyk Code explains its highly accurate findings and provides help including examples how to fix it. And to top that, it's easy to get started with and free to use on open source (plus a limited amount of non-OSS tests).

# Be careful when downloading packages

It is easy to install Python packages. Typically developers use the standard package installer for Python (pip) which uses the PyPI marketplace. This makes it important to understand how packages are added to PyPI.

PyPI has a procedure for reporting security concerns. If someone reports a malicious package or a problem within PyPI it is addressed, but packages added to PyPI do not undergo review — this would be an unrealistic expectation of the volunteers who maintain PyPI.

Therefore, it is wise to assume that there are malicious packages within PyPI and you should act accordingly. Reasonable steps include doing a bit of research on the package you want to install and ensuring that you carefully spell out the package name (a package named for a common misspelling of a popular package could execute malicious code). Before downloading a package, make sure to check it on Snyk Advisor.




Doing a quick search for a package on Snyk Advisor gives you a lot of information on the package, its support in the community, its history of bugs and fixes, and a lot more. A best practice before downloading a reference is to type the name in an editor and copy-paste using that spelling to prevent typosquatting. Snyk Advisor can tell you whether or not you should trust a package. You can see the history of security issues and the time it took to get them fixed.

Another best practice is to use virtual environments to isolate projects from each other. Also, use `pip freeze` or a comparable command to record changes in the environment in the requirement list.

Maintaining references in an up to date manner, Snyk Open Source is based on an industry leading vulnerability database recording security issues and possible fixes. Snyk Open Source runs scans using the requirements and provides actionable information about discovered vulnerabilities of direct and transitive references and helps you to fix them right away.

# Review your dependency licenses

When considering using an open source project, it is important to understand how these projects are licensed. Open source projects are free and available to use, but there may still be terms and conditions applied. These terms usually involve how the software is used, whether you need to make any changes you make to the software publicly available, and other similar requirements. You should become familiar with the licenses necessary for the projects you use, so you are sure that you are not compromising yourself legally.

If the project adopts a more restrictive license than you anticipated (GPL, SSPL, etc.), you can end up cornering yourself, leaving you to either comply with the terms of the license or cease using the project. Additionally, if you need to make changes to the project that does not have a license, you might run afoul with copyright law.

To ensure that your project is sustainable and you do not expose yourself to unnecessary Python security and legal risks, scan and fix license and vulnerability issues in your project's dependencies.

Snyk Open Source can help you here with open source license compliance management. It provides a developer-friendly way to gain end-to-end visibility while providing a flexible governance.

# Do not use the system standard version of Python

Most POSIX systems come preloaded with a version of Python. The problem with this version is that it is not kept current in a way as the specialized deployment package. Sometimes this is even a Python 2 version.

So make sure to use the latest version of Python available for your system and official containers designed to run Python and keep it updated. Snyk is here to help you. Scan your containers for necessary updates using Snyk Container and check your dependencies using Snyk Open Source.

# Use Python's capability for virtual environments

Use Python's capability for virtual environments Python is equipped to separate application development into virtual environments. A virtual environment isolates the Python interpreter, libraries, and scripts installed into it. It is a directory tree which contains executables and other files that make up the virtual environment. Common installation tools such as pip work as expected in a virtual environment. A virtual environment is activated by running a script. Most IDEs or dashboards, such as Anaconda Navigator, have built in functions to switch between virtual environments.

**Pro Tip:** As of Python version 3.5 on the use of venv is recommended and with version 3.6 `pyvenv` was deprecated.

Virtual environments make developing, packaging, and shipping Python applications easier. Using them is highly recommended not only for security reasons. See the Python venv doc for more details.

# Set `DEBUG = False` in production

In a development environment, it makes sense to have verbose error messages. In production though, you want to prevent any leaks of information that might help an attacker to learn more about your environment, libraries, or code.

Per default, most frameworks have debugging switched on. For example, Django has it enabled in settings.py. Make sure to switch debugging to `False` in production.

**Pro Tip**: Having a test case in the integration tests to see if the production configuration is activated.

# Be careful with string formatting

Despite Python's idea of having one and only one way to do things, it actually has four different ways to format strings (three methods for versions prior to Python 3.6).

String formatting has gotten progressively more flexible and powerful (f-strings are particularly interesting), but as flexibility increases, so does the potential for exploits. For this reason, Python users should carefully consider how they format strings with data entered by users.

Python has a built-in module named `string`. This module includes the `Template` class, which is used to create template strings.

Consider the following example.

```
from string import Template
greeting_template = Template("Hello World, my name is $name.")
greeting = greeting_template.substitute(name="Hayley")
```

For the above code, the variable greeting is evaluated as: “Hello World, my name is Hayley.”

This string format is a bit cumbersome because it requires an import statement and it is less flexible with types. It also doesn’t evaluate Python statements the way f-strings do. These constraints make template strings an excellent choice when dealing with user input.

Another quick note on string formatting: Be extra careful with raw SQL as mentioned above.

# Deserialize very cautiously

Python provides a built-in mechanism to serialize and deserialize Python objects called “pickling” using the pickle module. This is known to be insecure and it is advisable to **use it very cautiously and only on trusted sources.**

The new de-facto standard for serialization / deserialization is YAML using PyYAML. The package provides a mechanism to serialize custom data types to YAML and back into Python projects. But PyYAML is riddled with various possible attack vectors. A simple but effective way to secure the usage of PyYAML is using `yaml.SafeLoader()` instead of `yaml.Loader()` as a loader.

```
Data = yaml.load(input_file, Loader=yaml.SafeLoader)
```

This prevents loading of custom classes but supports standard types like hashes and arrays.

Another typical use case is XML. Typically standard libraries are used and they are vulnerable to typical attacks — namely DOS attacks or external entity expansion (an external source is references). A good first line of defense is a package called defusedxml. It has safeguards against these typical attacks.

# Bonus, non-security tip: Use Python type annotations

With version 3.5, type hints were introduced. While the Python runtime does not enforce type annotations, tools such as type checkers, IDEs, linters, SASTs, and others can benefit from the developer being more explicit. Here is an example to highlight the idea:

```
MODE = Literal['r', 'rb', 'w', 'wb']
def open_helper(file: str, mode: MODE) -> str:
    ...

open_helper('/some/path', 'r')  # Passes type check
open_helper('/other/path', 'typo')  # Error in type checker
```

Literal[...] was introduced with version 3.8 and is not enforced by the runtime (you can pass whatever string you want in our example) but type checkers can now discover that the parameter is outside the allowed set and warn you.

**Note:** As it is not enforced by the runtime, the security usage of type hints are limited.

# Secure your Python code with Snyk

Scan your Python code for quality and security issues, and get fix advice right in your IDE. Get started with Snyk for free.

Get started for free