Chapter 26. v3.7 to v3.n Migration

This book spans several versions of Python and covers some substantial (and still evolving!) new features, including:

- Order-preserving `dicts`
- Type annotations
- `:=` assignment expressions (informally called "the walrus operator")
- Structural pattern matching

Individual developers may be able to install each new Python version as it is released, and solve compatibility issues as they go. But for Python developers working in a corporate environment or maintaining a shared library, migrating from one version to the next involves deliberation and planning.

This chapter deals with the changing shape of the Python language, as seen from a Python programmer's viewpoint. (There have been many changes in Python internals as well, including to the Python C API, but those are beyond the scope of this chapter: for details, see the "What's New in Python 3.*n*" sections of each release's online documentation.)

Significant Changes in Python Through 3.11

Most releases have a handful of significant new features and improvements that characterize that release, and it is useful to have these in mind as high-level reasons for targeting a particular release. **Table 26-1** details only major new features and breaking changes in versions 3.6–3.11[1] that are likely to affect many Python programs; see the **Appendix** for a more complete list.

Table 26-1. Significant changes in recent Python releases

Version	New features	Breaking changes
3.6	• `dicts` preserve order (as an implementation detail of CPython) • F-strings added • _ in numeric literals supported • Annotations can be used for types, which can be checked with external tools such as `mypy` • `asyncio` is no longer a provisional module *Initial release: December 2016* *End of support: December 2021*	• Unknown escapes of `\` a ASCII letter no longer su ported in pattern argum to most `re` functions (sti mitted in `re.sub()` only

Version	New features	Breaking changes
3.7	• `dicts` preserve order (as a formal language guarantee) • `dataclasses` module added • `breakpoint()` function added *Initial release: June 2018* *Planned end of support: June 2023*	• Unknown escapes of `\` a ASCII letter no longer su ported in pattern argum to `re.sub()` • Named arguments no lo supported in `bool()`, `fl` `list()`, and `tuple()` • Leading named argume `int()` no longer suppor
3.8	• Assignment expressions (`:=` , aka the walrus operator) added • `/` and `*` in function argument lists to indicate positional-only and named-only arguments • Trailing = for debugging in f-strings (`f'{x=}'` short form for `f'x={x!r}'`) • Typing classes added (`Literal`, `TypedDict`, `Final`, `Protocol`)	• `time.clock()` removed `time.perf_counter()` • `pyvenv` script removed; **`python -m venv`** • **`yield`** and **`yield from`** r longer allowed in comp hensions or genexps • `SyntaxWarnings` on **`is`** a **`not`** tests against `str` an literals added

Version	New features	Breaking changes
	Initial release: October 2019 *Planned end of support: October 2024*	
3.9	• Union operators | and |= on `dict`s supported • `str.removeprefix()` and `str.removesuffix()` methods added • `zoneinfo` module added for IANA time zone support (to replace third-party `pytz` module) • Type hints can now use built-in types in generics (`list[int]` instead of `List[int]`) *Initial release: October 2020* *Planned end of support: October 2025*	• `array.array.tostring(` `fromstring()` removed • `threading.Thread.isAl` removed (use `is_alive` instead) • `ElementTree` and `Eleme` `getchildren()` and `getiterator()` remove • `base64.encodestring()` `decodestring()` remov (use `encodebytes()` and `decodebytes()` instead) • `fractions.gcd()` remov (use `math.gcd()` instead • `typing.NamedTuple._fi` removed (use `__annotations__` instea
3.10	• **`match`**/**`case`** structural pattern matching supported	• Importing ABCs from `collections` removed (

Version	New features	Breaking changes
	• Writing union types as `X | Y` (in type annotations and as second argument to `isinstance()`) allowed • Optional `strict` argument added to `zip()` built-in to detect sequences of differing lengths • Parenthesized context managers now officially supported; e.g., **`with`**`(`*`ctxmgr`*`,` *`ctxmgr`*`, ...):` *Initial release: October 2021* *Planned end of support: October 2026*	now import from `collections.abc`) • `loop` parameter remove from most of `asyncio`'s level API
3.11	• Improved error messages • General performance boost • Exception groups and `except*` added • Typing classes added (`Never`, `Self`) • `tomllib` TOML parser added to stdlib	• `binhex` module remove • `int` to `str` conversion r stricted to 4,300 digits

Version	New features	Breaking changes
	Initial release: October 2022 Planned end of support: October 2027 (est.)	

Planning a Python Version Upgrade

Why upgrade in the first place? If you have a stable, running application, and a stable deployment environment, a reasonable decision might be to leave it alone. But version upgrades do come with benefits:

- New versions usually introduce new features, which may allow you to simplify code.
- Updated versions include bug fixes and refactorings, which can improve system stability and performance.
- Security vulnerabilities identified in an older version may be fixed in a new version.[2]

Eventually, old Python versions fall out of support, and projects running on older versions become difficult to staff and more costly to maintain. Upgrading might then become a necessity.

Choosing a Target Version

Before deciding which version to migrate to, sometimes you have to figure out first, “What version am I running now?” You may be unpleasantly surprised to find old software running unsupported Python versions lurking in your company’s systems. Often this happens when those systems depend on some third-party package that is itself behind in version upgrades, or does not have an upgrade available. The situation is even more dire when such a system is critical in some way for company operations. You may be able to isolate the lagging package behind a remote-access API, allowing that package to run on the old version while permitting your own code to safely upgrade. The presence of systems with these upgrade constraints must be made visible to senior management, so they can be advised of the risks and trade-offs of retaining, upgrading, isolating, or replacing.

The choice of target version often defaults to "whatever version is the most current." This is a reasonable choice, as it is usually the most cost-effective option with respect to the investment involved in doing the upgrade: the most recent release will have the longest support period moving forward. A more conservative position might be "whatever version is the most current, minus 1." You can be reasonably sure that version *N*–1 has undergone some period of in-production testing at other companies, and someone else has shaken out most of the bugs.

Scoping the Work

After you have selected your target version of Python, identify all the breaking changes in the versions after the version your software is currently using, up to and including the target version (see the **Appendix** for a detailed table of features and breaking changes by version; additional details can be found in the "What's New in Python 3.*n*" sections of the **online docs**). Breaking changes are usually documented with a compatible form that will work with both your current version and the target version. Document and communicate the source changes that development teams will need to make before upgrading. (There may be significantly more work than expected involved in moving directly to the selected target version, if a lot of your code is affected by breaking changes or compatibility issues with related software. You may even end up revisiting the choice of target version or considering smaller steps. Perhaps you'll decide on upgrading to *target*–1 as a first step and deferring the task of the upgrade to *target* or *target*+1 for a subsequent upgrade project.)

Identify any third-party or open source libraries that your codebase uses, and ensure that they are compatible with (or have plans to be compatible with) the target Python version. Even if your own codebase is ready for upgrading to the target, an external library that lags behind may hold up your upgrade project. If necessary, you may be able to isolate such a library in a separate runtime environment (using virtual machines or container technologies), if that library offers a remote access programming interface.

Make the target Python version available in development environments, and optionally in deployment environments, so that developers can confirm that their upgrade changes are complete and correct.

Applying the Code Changes

Once you have decided on your target version and identified all the breaking changes, you'll need to make changes in your codebase to make it compatible with the target version. Your goal, ideally, is to have the code in a form that is compatible with both the current *and* target Python versions.

IMPORTS FROM __FUTURE__

`__future__` is a standard library module containing a variety of features, documented in the online docs, to ease migration between versions. It is unlike any other module, because importing features can affect the syntax, not just the semantics, of your program. Such imports *must* be the initial executable statements of your code.

Each "future feature" is activated using the statement:

```
from __future__ import feature
```

where *`feature`* is the name of the feature you want to use.

In the span of versions this book covers, the only future feature you might consider using is:

```
from __future__ import annotations
```

which permits references to as-yet-undefined types without enclosing them in quotes (as covered in Chapter 5). If your current version is Python 3.7 or later, then adding this `__future__` import will permit use of the unquoted types in type annotations, so you don't have to redo them later.

Begin by reviewing libraries that are shared across multiple projects. Removing the blocking changes from these libraries will be a crucial first step, since you will be unable to deploy any dependent applications on the target version until this is done. Once a library is compatible with both versions, it can be deployed for use in the migration project. Moving forward, the library code must maintain compatibility with both the current Python version and the target version: shared libraries will likely be the *last* projects that will be able to utilize any new features of the target version.

Standalone applications will have earlier opportunities to use the new features in the target version. Once the application has removed all code affected by breaking changes, commit it to your source control system as a cross-version-compatible snapshot. Afterwards, you may add new features to the application code and deploy it into environments that support the target version.

If version compatibility changes affect type annotations, you can use *.pyi* stub files to isolate version-dependent typing from your source code.

Upgrade Automation Using pyupgrade

You may be able to automate much of the toil in upgrading your code using automation tools such as the **pyupgrade package**. `pyupgrade` analyzes the abstract syntax tree (AST) returned by Python's `ast.parse` function to locate issues and make corrections to your source code. You can select a specific target Python version using command-line switches.

Whenever you use automatic code conversion, review the output of the conversion process. A dynamic language like Python makes it impossible to perform a perfect translation; while testing helps, it can't pick up all imperfections.

Multiversion Testing

Make sure that your tests cover as much of your project as possible, so that inter-version errors are likely to be picked up during testing. Aim for at least 80% testing coverage; much more than 90% can be difficult to achieve, so don't spend too much effort trying to reach a too-ambitious standard. (*Mocks*, mentioned in **"Unit Testing and System Testing"**, can help you increase the breadth of your unit testing coverage, if not the depth.)

The **tox package** is useful to help you manage and test multiversion code. It lets you test your code under a number of different virtual environments, and it supports multiple CPython versions, as well as PyPy.

Use a Controlled Deployment Process

Make the target Python version available in deployment environments, with an application environment setting to indicate whether an application should run using the current or target Python version. Continuously track, and periodically report, the completion percentage to your management team.

How Often Should You Upgrade?

The PSF releases Python on a minor-release-per-year cadence, with each version enjoying five years of support after release. If you apply a latest-release-minus-1 strategy, it provides you with a stable, proven version to migrate to, with a four-year support horizon (in case a future upgrade needs to be deferred). Given the four-year time window, doing upgrades to the latest release minus 1 every year or two should provide a reasonable balance of periodic upgrade cost and platform stability.

Summary

Maintaining the version currency of the software that your organization's systems depend on is an ongoing habit of proper "software hygiene," in Python just like in any other development stack. By performing regular upgrades of just one or two versions at a time, you can keep this work at a steady and manageable level, and it will become a recognized and valued activity in your organization.

[1] While Python 3.6 is outside the range of versions covered in this book, it introduced some significant new features, and we include it here for historical context.

[2] When this happens, it is usually an "all hands on deck" emergency situation to do the upgrade in a hurry. These events are the very ones you are trying to avoid, or at least minimize, by implementing a steady and ongoing Python version upgrade program.