`!importlib.resources` -- Package resource reading, opening and access

- Functional API

importlib.resources

Source code: `Lib/importlib/resources/__init__.py`

3.7

This module leverages Python's import system to provide access to *resources* within *packages*.

"Resources" are file-like resources associated with a module or package in Python. The resources may be contained directly in a package, within a subdirectory contained in that package, or adjacent to modules outside a package. Resources may be text or binary. As a result, a package's Python module sources (.py), compilation artifacts (pycache), and installation artifacts (like `reserved filenames <os.path.isreserved>` in directories) are technically de-facto resources of that package. In practice, however, resources are primarily those non-Python artifacts exposed specifically by the package author.

Resources can be opened or read in either binary or text mode.

Resources are roughly akin to files inside directories, though it's important to keep in mind that this is just a metaphor. Resources and packages **do not** have to exist as physical files and directories on the file system: for example, a package and its resources can be imported from a zip file using `zipimport`.

> `importlib.resources` follows the same security model as the built-in `open` function. Passing untrusted inputs to the functions in this module is unsafe.

> The standalone backport of this module provides more information on using importlib.resources and migrating from pkg_resources to importlib.resources.

`Loaders <importlib.abc.Loader>` that wish to support resource reading should implement a `get_resource_reader(fullname)` method as specified by `importlib.resources.abc.ResourceReader`.

Represents an anchor for resources, either a `module object <types.ModuleType>` or a module name as a string. Defined as `Union[str, ModuleType]`.

files(anchor: Optional[Anchor] = None)

Returns a `~importlib.resources.abc.Traversable` object representing the resource container (think directory) and its resources (think files). A Traversable may contain other containers (think subdirectories).

anchor is an optional `Anchor`. If the anchor is a package, resources are resolved from that package. If a module, resources are resolved adjacent to that module (in the same package or the package root). If the anchor is omitted, the caller's module is used.

3.9

3.12 *package* parameter was renamed to *anchor*. *package* was still accepted, but deprecated.

3.15 *package* parameter was fully removed. *anchor* can now be a non-package module and if omitted will default to the caller's module. *package* is no longer accepted since Python 3.15. Consider passing the anchor positionally or using `importlib_resources >= 5.10` for a compatible interface on older Pythons.

as_file(traversable)

Given a `~importlib.resources.abc.Traversable` object representing a file or directory, typically from `importlib.resources.files`, return a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary file or directory created when the resource was extracted from e.g. a zip file.

Use `as_file` when the Traversable methods (`read_text`, etc) are insufficient and an actual file or directory on the file system is required.

3.9

3.12 Added support for *traversable* representing a directory.

Functional API

A set of simplified, backwards-compatible helpers is available. These allow common operations in a single function call.

For all the following functions:

- *anchor* is an `~importlib.resources.Anchor`, as in `~importlib.resources.files`. Unlike in `files`, it may not be omitted.
- *path_names* are components of a resource's path name, relative to the anchor. For example, to get the text of resource named `info.txt`, use:

  ```
  importlib.resources.read_text(my_module, "info.txt")
  ```

  Like `Traversable.joinpath <importlib.resources.abc.Traversable>`, The individual components should use forward slashes (`/`) as path separators. For example, the following are equivalent:

  ```
  importlib.resources.read_binary(my_module, "pics/painting.png")
  importlib.resources.read_binary(my_module, "pics", "painting.png")
  ```

  For backward compatibility reasons, functions that read text require an explicit *encoding* argument if multiple *path_names* are given. For example, to get the text of `info/chapter1.txt`, use:

  ```
  importlib.resources.read_text(my_module, "info", "chapter1.txt",
                                encoding='utf-8')
  ```

open_binary(anchor, *path_names)

Open the named resource for binary reading.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*.

This function returns a `~typing.BinaryIO` object, that is, a binary stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('rb')
```

3.13 Multiple *path_names* are accepted.

open_text(anchor, *path_names, encoding='utf-8', errors='strict')

Open the named resource for text reading. By default, the contents are read as strict UTF-8.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function returns a `~typing.TextIO` object, that is, a text stream open for reading.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).open('r', encoding=encoding)
```

3.13 Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

read_binary(anchor, *path_names)

Read and return the contents of the named resource as `bytes`.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_bytes()
```

3.13 Multiple *path_names* are accepted.

read_text(anchor, *path_names, encoding='utf-8', errors='strict')

Read and return the contents of the named resource as `str`. By default, the contents are read as strict UTF-8.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*. *encoding* and *errors* have the same meaning as in built-in `open`.

For backward compatibility reasons, the *encoding* argument must be given explicitly if there are multiple *path_names*. This limitation is scheduled to be removed in Python 3.15.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).read_text(encoding=encoding)
```

3.13 Multiple *path_names* are accepted. *encoding* and *errors* must be given as keyword arguments.

path(anchor, *path_names)

Provides the path to the *resource* as an actual file system path. This function returns a context manager for use in a `with` statement. The context manager provides a `pathlib.Path` object.

Exiting the context manager cleans up any temporary files created, e.g. when the resource needs to be extracted from a zip file.

For example, the `~pathlib.Path.stat` method requires an actual file system path; it can be used like this:

```
with importlib.resources.path(anchor, "resource.txt") as fspath:
    result = fspath.stat()
```

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
as_file(files(anchor).joinpath(*path_names))
```

3.13 Multiple *path_names* are accepted.

is_resource(anchor, *path_names)

Return `True` if the named resource exists, otherwise `False`. This function does not consider directories to be resources.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
files(anchor).joinpath(*path_names).is_file()
```

3.13 Multiple *path_names* are accepted.

contents(anchor, *path_names)

Return an iterable over the named items within the package or path. The iterable returns names of resources (e.g. files) and non-resources (e.g. directories) as `str`. The iterable does not recurse into subdirectories.

See `the introduction <importlib_resources_functional>` for details on *anchor* and *path_names*.

This function is roughly equivalent to:

```
for resource in files(anchor).joinpath(*path_names).iterdir():
    yield resource.name
```

3.11 Prefer `iterdir()` as above, which offers more control over the results and richer functionality.

□□□ #2
□ □□□ □□□ 2026-10-31 17:12:47 UTC
□ □□□ □□□ 2026-10-31 17:12:50 UTC