regret

Release 2024.2.3.dev11+g321fb15

Julian Berman

Apr 28, 2024

CONTENTS

`regret` is a library for deprecating functionality in Python libraries and applications.

Its documentation lives on Read the Docs.

CHAPTER

ONE

DEPRECATIONS

`regret` can deprecate:

- **[x] callables**
  - [x] functions
  - **[x] classes**
    - [ ] subclassable classes
- **[ ] attributes**
  - [ ] of modules
  - [ ] of classes (& methods)
  - [ ] of instances
- **[ ] descriptors**
  - [ ] classmethod
- **[ ] modules**
  - [ ] current module
  - [ ] other module
- **[ ] parameters to callables**
  - [x] previously required parameters that will be removed
  - [x] optional parameters that are now required
  - [ ] deprecated values for parameters
  - [ ] type changes for parameters
  - [ ] mutual exclusion
- **[ ] interfaces**
  - [ ] PEP 544 protocols
  - [ ] `zope.interface`s
- [x] inheritability of a class

CHAPTER TWO

DESIGN GOALS

`regret` is meant to cover all of the deprecations an author may encounter.

It is intended to:

- be versioning system agnostic (i.e. SemVer, CalVer, HipsTer, etc.), because deprecations originate from a version, a point in time, or both.
- be documentation system aware (i.e. Sphinx, epydoc, Plaintext, etc.), because deprecations need communication.
- be itself fully tested, because deprecations must not break the code they deprecate
- support removal date indication, and likely "policies" which automate choosing default removal dates, because deprecations ultimately intend some ultimate change
- make "clean code" trivially easy to deprecate, and make complex code *possible* to deprecate, because the deprecation process is fraught with edge cases and unforeseen necessity.
- minimize the amount of deprecation-related code required for authors, since deprecations are boring, and we all want to focus on developing our libraries instead.

In particular, as a lofty first milestone, it is intended to cover all of the specific deprecations required for these jsonschema issues, and with luck, to subsume all the functionality present in twisted.python.deprecate.

CHAPTER

THREE

CONTENTS

3.1 Before You Deprecate

Regret is painful.

Making a design or implementation mistake and having to pay for it later can be a significant burden[1] for a library, package or codebase. This burden weighs heavily both on maintainer(s) and on users of the code.

Most of the functionality provided by *regret* is aimed towards these inevitable moments *after* a mistake has been made and is to be corrected.

However, there are a number of things that can be done *before* any mistake has even been made, which may minimize the need for some deprecations, or minimize their damage, or increase the overall trust between you the maintainer and your package's users.

This document discusses a number of these concerns – things you should do as a maintainer before you even do any deprecations at all.

3.1.1 Have a Policy

The most important thing a maintainer can do ahead of time is to have a clear and understandable deprecation *policy*.

The policy can be as formal or informal as appropriate, or as may be known ahead of time.

It should clearly communicate what users of a package can expect out of you the maintainer when a deprecation is needed. It may cover any or all of:

- once a deprecation has been done, how long will it be before the deprecation is finished and the object is removed or changed incompatibly?
- where or how will deprecations be communicated?
- how do you the maintainer, or team of maintainers, view backwards compatibility in the context of your package? Is it important? Is it unimportant (sometimes it won't be!)
- in the event a backwards incompatible change is made *unintentionally*, perhaps due to the introduction of a bug, is fixing the backwards incompatibility itself allowed as a backwards incompatibility in a successive release? Does the answer change if the bug is unnoticed for a long while?
- how will changes to the deprecation policy itself be done?

Your policy will likely fail to predict all of the intricate deprecations that you may need over your package's long and healthy lifetime. Judgment calls may still have to be made (*empathetically*), but a policy sets an initial set of guidelines which can be built upon later.

[1] a burden this library tries to ease, at least a small bit

See also:

PEP 387

The PEP covering the backwards compatibility policy for Python itself as a language

regret's Deprecation Policy

regret's own deprecation policy

Twisted Compatibility Policy

The Twisted project's deprecation (compatibility) policy

3.1.2 Document Your Public API

A *deprecation policy* can clarify to users of a package what to expect as the package evolves and changes APIs or objects over time.

Simply specifying how deprecations will be handled however is not enough. Even with a clear policy on how deprecations will be handled, a key additional piece of guidance is needed around *what objects or APIs are themselves considered "public" and thereby fall under the deprecation policy*[2].

As an uncontroversial non-example to illustrate – the specific layout of lines of code within a package's source code is essentially never part of a package's public API. Adding a blank line, or a comment, or even reordering the order in which two functions are defined in a file is essentially universally treated as an acceptable change to make without any indication. This is the case even though in theory[3] a downstream user of the package may have written code that relies on exact line numbers of an object, or on the ordering of definitions of objects within the package.

Within the (Python) community, there is a generally agreed upon set of fundamental norms that most maintainers follow or understand, which covers a portion of this public API definition. But there are a number of areas or subtleties where there is no explicitly discussed standard practice – and in these cases there are maintainers and libraries which conduct themselves in different ways (i.e. which do or do not consider these parts of their API surface to be public or private).

And so – having clear guidelines on what part of a package's APIs are considered public can be hugely helpful for users who wish to understand when and where they are using a package as intended, and can therefore rely on its policy.

Here is a (non-exhaustive) list of potential API surface that you may need or want to clarify:

- for "normal" Python function parameters which may be passed either positionally or by name, is the order of parameters considered public and will not change? How about their names? Will they change? How about the kind of parameter itself? Will positional parameters always be passable positionally? Or may a parameter "convert" into a keyword-only one, or vice versa?
- does the package use a convention to indicate entire modules within it are considered private? (say, `mypackage._foo`, following that of other Python objects)
- is the text content of exceptions defined or raised by the package part of its public API, or may they change?
- which methods on objects are considered part of their public API? Is their `__repr__` considered private, even though it otherwise follows public API conventions? Are any other methods considered similarly exceptional to the "rule"?
- are modules and objects found within the package's test suite considered public API?

[2] Python does not have a particularly formal enforced definition of "public" and "private", but we use the terms here in their commonly understood meaning within the Python ecosystem: a public object or API is one which is expected to be relied upon by end-users of the package and whose compatibility is "guaranteed", and a private object is one whose use is conversely *not* encouraged and not guaranteed for end users, regardless of its accessibility at runtime.

[3] though hopefully not in practice, even if Hyrum's Law might apply.

- are imported objects part of a module's public API? Can a user of the package assume that if `mypackage.foo` imports `bar`, even though `bar` really lives in some other module, that `bar` will not be removed from `mypackage.foo`? Is the answer different if `bar` is an object defined somewhere in `mypackage` vs. in an external package?
- is the layout of your documentation considered public API? More specifically for say, a package documenting itself via `Sphinx`, will the `refs` defined for headings be kept over time? Will the overall document structure change? How about links to specific concrete pages as URLs?
- is being able to *raise* exceptions defined by your library part of its public API? Or is only catching them considered public?
- is the number of stack frames your library uses internally to implement a public API itself public API? After all, a downstream library calling into it may be using `warnings.warn` and be providing a `stacklevel` parameter relying on it!
- a class in your library is defined using `attrs`. Is calling `attrs.evolve` directly on instances of your class considered public API, and your class is thereby permanently coupled to `attrs.evolve`'s public API? Or is your use of `attrs` as a library maintainer simply an implementation detail?
- a function in your library calls `requests.get`, a function which has mutable global state – it allows someone to import the library and e.g. define `Transport Adapters` which, even if done outside your library, affect how it will retrieve the HTTP response. Is your library free to change its HTTP client to another HTTP client even though this will potentially disrupt users who expect their external change to have an effect on your library's behavior?
- your package depends on `foo>1.2.3`. Are these pins part of your public API, and bumping the lower-pinned version of `foo` a breaking change? Doing so may of course affect users who are using `foo==1.2.3` alongside another of their own dependencies.
- today, your package has no binary (non-Python) dependencies. Is that a permanent promise of its installation "API"?
- ...

There are many many more. Think of things that you, a maintainer, rely on from libraries *you* use, and how many subtleties you wish were clearer.

To be clear, some of the above *do* have commonly understood answers in the ecosystem – but even beyond resolving the final bits of doubt, there may still be a lot to gain from explicitly confirming each has been considered in the course of changes made to the package.

See also:

public API

regret's own public API definition

The SemVer specification, step 1

which echoes the requirement of defining a clear public API.

`jsonschema public API`

another example of a public API definition

3.1.3 Empathize

Having a *policy* for how you'll deprecate things, and having a *definition* of what it is that is subject to deprecation are key steps in setting clear expectations.

The reality is – they'll never be enough.

End-users of your package will forget or not notice something isn't part of your public API. Or they'll knowingly rely on things that aren't public given "no other" good option for a particular piece of functionality.

Be empathetic! We are all just trying to get our jobs done, whatever they may be.

Empathy in this case means – if you've clearly defined something as private, but you nonetheless see thousands of uses of the private API in downstream code, simply take pause. At the very least, this often may indicate either a UX issue in finding the appropriate public APIs (which can be used to improve your package's overall experience) or the lack of an API entirely.

An API marked "experimental" and not-to-be-relied-upon will *still* be relied upon if it remains unchanged for a number of years in the wild, and breaking it, while justified, will still break downstream users. Do so knowingly, if you do do so.

You may choose not to remove a *private* API if it would cause significant breakage due to its evident use. Doing so indicates empathy! (Though, in contrast, *not* doing so, and removing the API, should not be weaponized into a *lack* of empathy!)

Take situations like these as ways to improve the clarity of your policies and guidance of your documentation overall, and as ways to build healthy relationships, *if* that is your decision.

3.1.4 Analytics

As a final area of consideration, though a challenging one – nothing beats data.

If you as a package author have access to concrete usage data of any kind, use it to make better decisions about your package's evolution.

In simple cases this may be as simple as answering "can I deprecate support for a particular Python version?" by investigating how many installations of your package are done on the version in question, for which `the PyPA provides a dataset that can help`. But the same questions can be asked of any API – "how often is this function used? What data would help quantify its use, and can I access it?".

Any additional data you may have or can easily (and ethically) collect will help drive intelligent and informed decisions.

3.2 What You Can Deprecate and How

Most things should be easy to deprecate and everything should be possible.

This page attempts to demonstrate a variety of practical deprecations that library authors face, alongside how to perform the deprecation using *regret*.

The *API Reference* also contains a full list for completeness.

3.2.1 Functions & Callables

Deprecating a single function or callable in its entirety is one of the simplest and most common deprecations to perform.

Consider as an example a greeting function which we wish to deprecate:

```python
def greeting(first_name, last_name):
    return f"Hello {first_name} {last_name}!"
```

Doing so can be done via the *regret.callable* decorator by simply specifying the version in which the function has been deprecated:

```python
@regret.callable(version="v2020-07-08")
def greeting(first_name, last_name):
    return f"Hello {first_name} {last_name}!"
```

at which point any code which uses the function will receive a suitable deprecation warning:

```python
print(greeting("Joe", "Smith"))
```

```
...: DeprecationWarning: greeting is deprecated.
  print(greeting("Joe", "Smith"))
Hello Joe Smith!
```

Note: Class objects are themselves simply callables, and as such, deprecating an entire class can be done in the same manner.

However, if you have an API that primarily encouraged subclassing of the class to be deprecated, the object returned by *regret.Deprecator.callable* is *not* guaranteed to be a type (i.e. a class), and therefore may not support being subclassed as the original object did.

Replacements

It is often the case when deprecating an object that a newer replacement API subsumes its functionality, and is meant to be used instead.

regret.callable accommodates this common use case by allowing you to specify which object is the replacement while deprecating:

```python
def better_greeting(first_name, last_name):
    return f"Hello {first_name} {last_name}! You are amazing!"

@regret.callable(version="1.0.0", replacement=better_greeting)
def greeting(first_name, last_name):
    return f"Hello {first_name} {last_name}!"
```

which will then show the replacement object in warnings emitted:

```python
print(greeting("Joe", "Smith"))
```

```
...: DeprecationWarning: greeting is deprecated. Please use better_greeting instead.
  print(greeting("Joe", "Smith"))
Hello Joe Smith!
```

3.2.2 Parameters

There are various scenarios in which a callable's signature may require deprecation.

Removing a Required Parameter

regret can help deprecate a parameter (argument) which previously was required and which now is to be removed.

Consider again our `greeting` function, but where we have decided to replace the separate specification of first and last names with a single `name` parameter, and therefore wish to deprecate providing the name in separate parameters:

```
@regret.parameter(version="v1.2.3", name="first_name")
@regret.parameter(version="v1.2.3", name="last_name")
def greeting(first_name=None, last_name=None, *, name=None):
    if first_name is not None:
        name = first_name
        if last_name is not None:
            name += f" {last_name}"
    return f"Hello {name}!"
```

After the above, using the function with the previous parameters will show a deprecation warning:

```
print(greeting("Joe", "Smith"))
```

```
...: DeprecationWarning: The 'first_name' parameter is deprecated.
  print(greeting("Joe", "Smith"))
...: DeprecationWarning: The 'last_name' parameter is deprecated.
  print(greeting("Joe", "Smith"))
Hello Joe Smith!
```

but via the new parameter, will not:

```
print(greeting(name="Joe Smith"))
```

```
Hello Joe Smith!
```

Making a New or Previously-Optional Parameter Required

regret can help make a parameter which previously was *not* required slowly become required.

Again in our `greeting` function, perhaps we have decided to allow specifying how excited to make the greeting, by specifying whether to end it with a period or exclamation point. We wish to ultimately force users of the function to specify one or the other, but until then, a default is being chosen.

```
@regret.optional_parameter(version="v1.2.3", name="end", default="!")
def greeting(first_name, last_name, end):
    return f"Hello {first_name} {last_name}{end}"
```

Note: *regret* can and should handle ensuring that the default is used when not provided by the caller.

Your wrapped function can assume a value will always be provided.

None can be used as a default if appropriate (and will not be interpreted with any meaning), as can any other Python object.

After the above, using the function without explicitly passing the end parameter will show a deprecation warning:

```
print(greeting("Joe", "Smith"))
```

```
...: DeprecationWarning: Calling greeting without providing the 'end' parameter is␣
↪deprecated. Using '!' as a default.
  print(greeting("Joe", "Smith"))
Hello Joe Smith!
```

but when properly specifying the new parameter, will not:

```
print(greeting("Joe", "Smith", end="."))
```

```
Hello Joe Smith.
```

3.2.3 Subclassability

A deprecation library isn't necessarily the place to opine on the pros and cons of inheritance.

For library authors however who have released public APIs that heavily depend on or require users to inherit from provided superclasses, *regret* provides a mechanism for deprecating the inheritability of classes.

Consider for example:

```
class Contact:
    name: str
    phone: str
    address: str
```

which downstream users of the class extend via e.g.:

```
class EMailContact(Contact):
    email: str
```

We can deprecate the downstream use of the contact class in inheritance hierarchies via:

```
@regret.inheritance(version="v1.2.3")
class Contact:
    name: str
    phone: str
    address: str
```

at which point, the act itself of subclassing will produce:

```
class EMailContact(Contact):
    email: str
```

```
...: DeprecationWarning: Subclassing from Contact is deprecated.
  class EMailContact(Contact):
```

3.3 regret's Deprecation Policy

This document is *regret*'s own deprecation policy.

It attempts to define, with as much forethought as is known, how backwards-incompatible changes to the library will be performed.

3.3.1 Public API

regret's public API broadly follows what is often intuitively or culturally understood in the Python ecosystem. Here though is a more precise (though potentially still incomplete) description of things *not* considered part of its public API:

- any object whose name follows Python's "private" convention (starting with an underscore)
- the entire contents of any *module* named in the above way (with a leading underscore)
- the contents of the `regret.tests` package, i.e. the test suite, even if objects "appear" to be named publicly
- the exact wording or contents of any exception message emitted
- the exact structure or contents of any object `repr`s
- the subclassability (i.e. ability of a class to be a superclass of another) of *any* object not explicitly marked as subclassable
- any "transitive" imported objects – meaning, if a module `regret.foo` imports an object `regret.bar.Baz`, the presence of `Baz` is not public within `foo`, only within `bar`

3.3.2 Versioning

With the above public API in mind, *regret* attempts to follow the Semantic Versioning specification as an (imperfect) communication mechanism. Specifically, major version numbers will be bumped for each backwards incompatible change, including changes which have been through the deprecation period discussed *below*.

3.3.3 Backwards Incompatible Changes

In accordance with the semantic *Versioning* scheme mentioned in this document, *regret*'s public API may change more drastically until it reaches version 1.0.0 (it's "official" public release).

In the event that an API requires a backwards incompatible change, a deprecation period of *six months* or *two releases* will preserve the original behavior unchanged, whilst emitting a `DeprecationWarning`, and introducing any replacement APIs.

3.3.4 Python Version Support

In general, *regret*'s support for particular versions of Python will end at *latest* on or around the `end of life dates` for each respective version, but more typically once they constitute a meaningful maintenance burden and constitute a smaller proportion of installations. Exceptions may be made in some circumstances where versions see continued use, but should not be relied upon. Support contracts are available for situations which require them.

Unless otherwise noted, only the *latest* patch version of each CPython release is officially supported.

Attempts will always be made to have *regret*'s supported Python versions reflected in its built distributions (i.e. via `python_requires`), such that installation tools do not install versions unless they are compatible with the running interpreter.

3.3.5 Further Notes

regret's continuous integration and test suite should be referenced as a representation of execution "environments" it supports.

This includes both operating system support, as well as broader concerns – as a hypothetical example, if a new way of installing or running *regret* is used which is not already being tested in the automated suite, its support or continuing function is *not* guaranteed.

Pull requests are welcome to add additional supported environments, though some discretion may still be applied if there is likelihood of ongoing maintenance burden.

3.4 API Reference

3.4.1 `regret`

Submodules

`regret.emitted`

Objects emitted whilst a deprecated object is being used.

class `regret.emitted.`**`Deprecation`**(*kind: Deprecatable*, *name_of: name_of = <function _qualname>*, *replacement: Any = None*, *removal_date: date | None = None*, *addendum: str | None = None*)

Bases: `object`

A single emitted deprecation.

`message`() → str

Express this deprecation as a comprehensible message.

class `regret.emitted.`**`Callable`**(*object: Any*)

Bases: `object`

A parameter for a particular callable.

`message`(*name_of:* name_of) → str

Express this deprecation as a comprehensible message.

class `regret.emitted.`**`Inheritance`**(*type: type[Any]*)

Bases: `object`

The subclassing of a given parent type.

`message`(*name_of:* name_of) → str

Express this deprecation as a comprehensible message.

class `regret.emitted.`**`Parameter`**(*callable:* _Callable*[..., Any]*, *parameter: inspect.Parameter*)

Bases: `object`

A parameter for a particular callable which should no longer be used.

`message`(*name_of:* name_of) → str

Express this deprecation as a comprehensible message.

class `regret.emitted.`**OptionalParameter**(*callable: _Callable[..., Any]*, *parameter: inspect.Parameter*, *default: Any*)

Bases: `object`

A parameter for a particular callable which will become mandatory.

message(*name_of: name_of*) → str

Express this deprecation as a comprehensible message.

`regret.testing`

Helpers for testing your regret.

exception `regret.testing.`**ExpectedDifferentDeprecations**

Bases: `AssertionError`

Different deprecation(s) were seen than the ones which were expected.

class `regret.testing.`**Recorder**(*saw: list[Deprecatable] = NOTHING*)

Bases: `object`

Recorders keep track of deprecations as they are emitted.

They provide helper methods for asserting about the deprecations afterwards.

emit(*deprecation: Deprecatable*, *extra_stacklevel: int*) → None

"Emit" a deprecation by simply storing it.

An emitter suitable for passing to *`regret.Deprecator`* instances.

expect(***kwargs: Any*) → contextlib.AbstractContextManager[None]

Expect a given set of deprecations to be emitted.

expect_deprecations(**deprecations: Deprecation*) → Iterator[None]

Expect a given set of deprecations to be emitted.

expect_clean() → AbstractContextManager[None]

Expect no deprecations to be emitted.

`regret.typing`

Typing related helpers for regret.

class `regret.typing.`**Deprecatable**(**args*, ***kwargs*)

Bases: `Protocol`

A single kind of deprecatable behavior.

message(*name_of: Callable[[Any], str]*) → str

Return a message summarizing this deprecation.

class `regret.typing.`**Emitter**(**args*, ***kwargs*)

Bases: `Protocol`

A callable which reports when deprecated things are used.

`regret.typing.`**`name_of`**

Return a string name for any object.

alias of `Callable`[`Any`, `str`]

`class regret.typing.`**`new_docstring`**(**args*, ***kwargs*)

Bases: `Protocol`

A callable which transforms docstrings to include deprecation info.

Contents

You made a thing, but now you wish it'd go away… Deprecations, a love story.

`class regret.`**`Deprecator`**(*emit: Emitter = <function emit>*, *name_of: name_of = <function _qualname>*, *new_docstring: new_docstring = <function doc_with_deprecated_directive>*)

Bases: `object`

Deprecators help manifest regret.

Parameters

- **`emit`** – a callable which will be called with one argument, a *`regret.emitted.Deprecation`* instance, whenever a deprecated object has been used. If unprovided, by default, a warning will be shown using the standard library `warnings` module.
- **`name_of`** – a callable which given any Python object should return a suitable name for the object. If unprovided, the `__qualname__` will be used, and therefore an object's (non-fully-)qualified name will appear in messages.
- **`new_docstring`** – a callable which should produce a docstring for newly deprecated objects. It will be called with three *keyword* :param * `object`: :param the object that is being deprecated: :param * `name_of`: calculating object names :param the callable described above for use in: calculating object names :param * `version`: :param the version that deprecates the provided object:

  and it should return a single string which will become the new docstring for a deprecated object. If unprovided, deprecation docstrings will be constructed using syntax suitable for `Sphinx`, via the `deprecated` directive.

`callable`(*version: str*, *replacement: Any = None*, *removal_date: datetime.date | None = None*, *addendum: str | None = None*)

Deprecate a callable as of the given version.

Parameters

- **`version`** – the first version in which the deprecated object was considered deprecated
- **`replacement`** – optionally, an object that is the (direct or indirect) replacement for the functionality previously performed by the deprecated callable
- **`removal_date`** (*`datetime.date`*) – optionally, a date when the object is expected to be removed entirely
- **`addendum`** (*`str`*) – an optional additional message to include at the end of warnings emitted for this deprecation

`parameter`(*version: str*, *name: str*)

Deprecate a parameter that was previously required and will be removed.

Parameters

- **version** – the first version in which the deprecated parameter was considered deprecated
- **name** – the name of the parameter as specified in the callable's signature.

  Deprecating a parameter that was previously accepted only via arbitrary keyword arguments ("kwargs") is also supported and should be specified using the name of the parameter as retrieved from the keyword arguments.

optional_parameter(*version: str*, *name: str*, *default: Any*)

Deprecate a parameter that was optional and will become required.

Parameters

- **version** – the first version in which the parameter was to warn when unprovided
- **name** – the name of the parameter as specified in the callable's signature.

  Requiring an optional parameter that was previously accepted only via arbitrary keyword arguments ("kwargs") is also supported and should be specified using the name of the parameter as retrieved from the keyword arguments.
- **default** – whilst the parameter remains optional, the value that should be used when it is unprovided by a caller.

  It will be passed through to the wrapped callable, which can therefore assume the argument will always be present.

inheritance(*version: str*)

Deprecate allowing a class to be subclassed.

Parameters

version – the first version in which the deprecated object was considered deprecated

regret.**callable**(*version: str*, *replacement: Any = None*, *removal_date: datetime.date | None = None*, *addendum: str | None = None*)

Deprecate a callable as of the given version.

Parameters

- **version** – the first version in which the deprecated object was considered deprecated
- **replacement** – optionally, an object that is the (direct or indirect) replacement for the functionality previously performed by the deprecated callable
- **removal_date** (`datetime.date`) – optionally, a date when the object is expected to be removed entirely
- **addendum** (`str`) – an optional additional message to include at the end of warnings emitted for this deprecation

regret.**inheritance**(*version: str*)

Deprecate allowing a class to be subclassed.

Parameters

version – the first version in which the deprecated object was considered deprecated

regret.**optional_parameter**(*version: str*, *name: str*, *default: Any*)

Deprecate a parameter that was optional and will become required.

Parameters

- **version** – the first version in which the parameter was to warn when unprovided

- **`name`** – the name of the parameter as specified in the callable's signature.

  Requiring an optional parameter that was previously accepted only via arbitrary keyword arguments ("`kwargs`") is also supported and should be specified using the name of the parameter as retrieved from the keyword arguments.

- **`default`** – whilst the parameter remains optional, the value that should be used when it is unprovided by a caller.

  It will be passed through to the wrapped callable, which can therefore assume the argument will always be present.

`regret.`**`parameter`**`(`*version: str*, *name: str*`)`

Deprecate a parameter that was previously required and will be removed.

Parameters

- **`version`** – the first version in which the deprecated parameter was considered deprecated
- **`name`** – the name of the parameter as specified in the callable's signature.

  Deprecating a parameter that was previously accepted only via arbitrary keyword arguments ("`kwargs`") is also supported and should be specified using the name of the parameter as retrieved from the keyword arguments.

PYTHON MODULE INDEX

INDEX