Epydoc

API Documentation Extraction in Python

Edward Loper

Epydoc: Overview

- **Extracts and organizes API documentation for a Python library.**
- **Extracts documentation from…**
  - **Docstrings**
  - **Inspection**
  - **Parsed source code**
- **And organizes it into a coherent reference...**
  - **Webpage (HTML)**
  - **Document (PDF)**

API Documentation

- **What it does:**
  - **Defines the "interface" provided a library.**
  - **Describes each object defined by the library.**
- **Why it's useful:**
  - **Explains how to use a library**
  - **Documents how a library works**

Writing API Documentation

- **API documentation is tightly coupled with source code.**
  - **So it can be difficult to keep it in sync with the implementation.**
- **Solution:**
  - **Keep API documentation in docstrings.**
  - **A *docstring* is a string constant at the top of an object's definition, that is available via inspection.**

Documentation Extraction

- It’s convenient to *write* API docs in the source code…
- But it’s not convenient to *read* them there.
- Solution: use a tool that…
  - Extracts API docs from the source code
  - Converts them into a readable format

Avoiding Duplication

- Multiple objects can share the same documentation:
  - Overridden methods
  - Interface implementations
  - Wrapper functions
- But duplicating their documentation is problematic:
  - It clutters the source code
  - It’s easy for different copies to get out of sync

Avoiding Duplication

- Epydoc provides 2 mechanisms to avoid duplication:
  - *Documentation inheritance:* A method without a docstring inherits documentation from the method it overrides.
  - *The “@include” field:* Special markup that can be used to include documentation from any other object.

Epydoc’s Output

- Epydoc currently supports 2 output formats:
  - Webpage (HTML)
  - Document (LaTeX/DVI/PS/PDF)
- And one more is in the works:
  - Manpage

Webpage Output: A Quick Tour

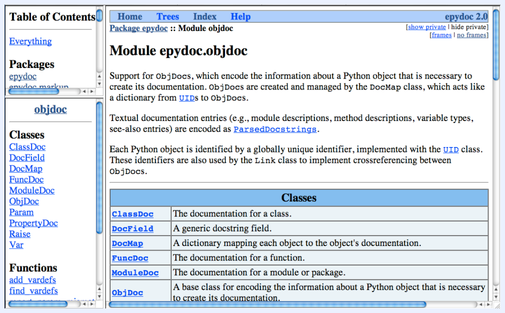


Table of Contents

Everything

Packages

epydoc

objdoc

Classes

ClassDoc
DocField
DocMap
FuncDoc
ModuleDoc
ObjDoc
Param
PropertyDoc
Raise
Var

Functions

add_vardefs
find_vardefs

Home Trees Index Help epydoc 2.0

Package epydoc :: Module objdoc [show private | hide private] [frames | no frames]

Module epydoc.objdoc

Support for `ObjDocs`, which encode the information about a Python object that is necessary to create its documentation. `ObjDocs` are created and managed by the `DocMap` class, which acts like a dictionary from `UIDs` to `ObjDocs`.

Textual documentation entries (e.g., module descriptions, method descriptions, variable types, see-also entries) are encoded as `ParsedDocstrings`.

Each Python object is identified by a globally unique identifier, implemented with the `UID` class. These identifiers are also used by the `Link` class to implement crossreferencing between `ObjDocs`.

Classes	
`ClassDoc`	The documentation for a class.
`DocField`	A generic docstring field.
`DocMap`	A dictionary mapping each object to the object's documentation.
`FuncDoc`	The documentation for a function.
`ModuleDoc`	The documentation for a module or package.
`ObjDoc`	A base class for encoding the information about a Python object that is necessary to create its documentation.

Webpage Output: Table of Contents

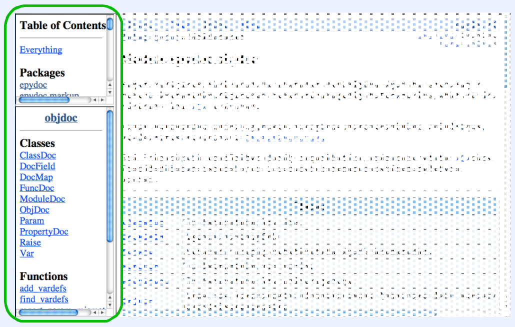


Table of Contents

Everything

Packages

epydoc

objdoc

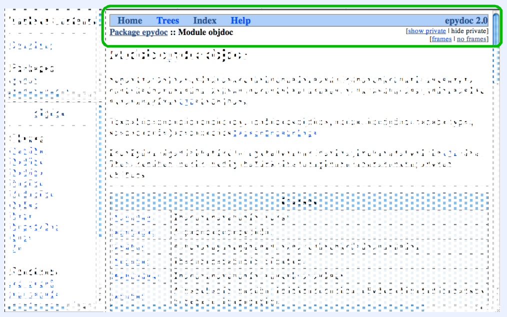
Classes

ClassDoc
DocField
DocMap
FuncDoc
ModuleDoc
ObjDoc
Param
PropertyDoc
Raise
Var

Functions

add_vardefs
find_vardefs

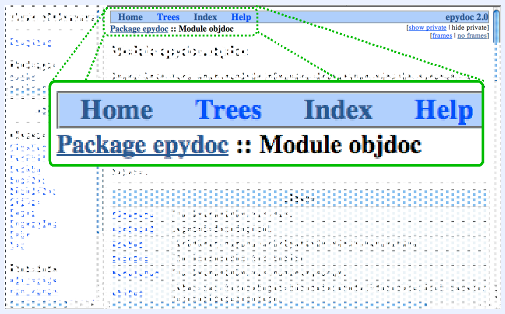
Webpage Output: Navigation Bar



Home Trees Index Help epydoc 2.0

Package epydoc :: Module objdoc [show private | hide private] [frames | no frames]

Webpage Output: Navigation Bar



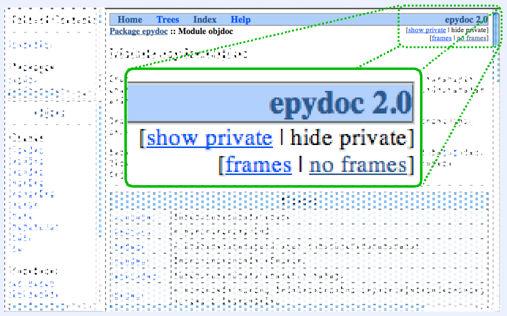
Home Trees Index Help epydoc 2.0

Package epydoc :: Module objdoc [show private | hide private] [frames | no frames]

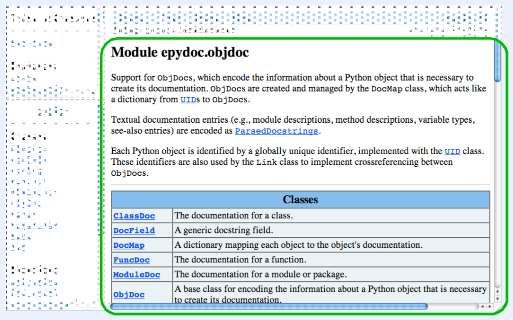
Home Trees Index Help

Package epydoc :: Module objdoc

Webpage Output: Navigation Bar


Home
Trees
Index
Help
epydoc 2.0
Package epydoc :: Module objdoc
[show private | hide private]
[frames | no frames]

Webpage Output: Module Documentation



Module epydoc.objdoc

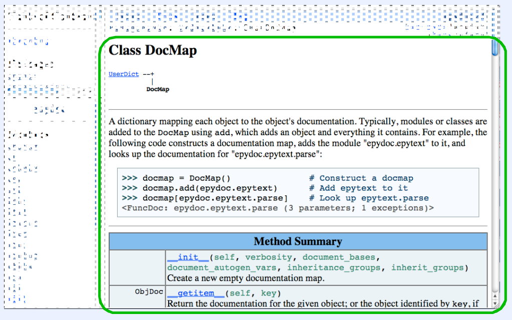
Support for `ObjDocs`, which encode the information about a Python object that is necessary to create its documentation. `ObjDocs` are created and managed by the `DocMap` class, which acts like a dictionary from `UIDs` to `ObjDocs`.

Textual documentation entries (e.g., module descriptions, method descriptions, variable types, see-also entries) are encoded as `ParsedDocstrings`.

Each Python object is identified by a globally unique identifier, implemented with the `UID` class. These identifiers are also used by the `Link` class to implement crossreferencing between `ObjDocs`.

Classes	
`ClassDoc`	The documentation for a class.
`DocField`	A generic docstring field.
`DocMap`	A dictionary mapping each object to the object's documentation.
`FuncDoc`	The documentation for a function.
`ModuleDoc`	The documentation for a module or package.
`ObjDoc`	A base class for encoding the information about a Python object that is necessary to create its documentation.

Webpage Output: Class Documentation



Class DocMap

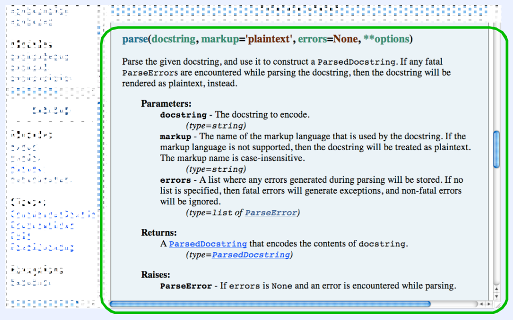
```
UserDict --+
           |
          DocMap
```

A dictionary mapping each object to the object's documentation. Typically, modules or classes are added to the `DocMap` using `add`, which adds an object and everything it contains. For example, the following code constructs a documentation map, adds the module "epydoc.epytext" to it, and looks up the documentation for "epydoc.epytext.parse":

```
>>> docmap = DocMap()                # Construct a docmap
>>> docmap.add(epydoc.epytext)       # Add epytext to it
>>> docmap[epydoc.epytext.parse]     # Look up epytext.parse
<FuncDoc: epydoc.epytext.parse (3 parameters; 1 exceptions)>
```

Method Summary	
	`__init__(self, verbosity, document_bases, document_autogen_vars, inheritance_groups, inherit_groups)` Create a new empty documentation map.
`ObjDoc`	`__getitem__(self, key)` Return the documentation for the given object; or the object identified by `key`, if

Webpage Output: Function Documentation



parse(docstring, markup='plaintext', errors=None, **options)

Parse the given docstring, and use it to construct a `ParsedDocstring`. If any fatal `ParseErrors` are encountered while parsing the docstring, then the docstring will be rendered as plaintext, instead.

Parameters:

- **`docstring`** - The docstring to encode.
  (type=`string`)
- **`markup`** - The name of the markup language that is used by the docstring. If the markup language is not supported, then the docstring will be treated as plaintext. The markup name is case-insensitive.
  (type=`string`)
- **`errors`** - A list where any errors generated during parsing will be stored. If no list is specified, then fatal errors will generate exceptions, and non-fatal errors will be ignored.
  (type=`list` of `ParseError`)

Returns:

A `ParsedDocstring` that encodes the contents of `docstring`.
(type=`ParsedDocstring`)

Raises:

`ParseError` - If `errors` is `None` and an error is encountered while parsing.

Docstring Markup

- Why use markup in docstrings?
  - More expressive power
  - Display medium independence
- Epydoc supports 4 markup languages:
  - Epytext
  - reStructuredText
  - Javadoc
  - Plaintext
- Markup language declaration:

  __docformat__ = “restructuredtext”

Docstring Markup: Epytext

- A lightweight markup language
  - Easy to write
  - Easy to read as plaintext
  - Easy to understand
- A conservative markup language
  - Uses common conventions for basic formatting.
  - If encounters unknown formatting, it falls back to verbatim plaintext.
  - Works well with docstrings that were written in plaintext.
- The default markup language

Docstring Markup: reStructuredText

- An “easy-to-read, what-you-see-is-what-you-get” markup language
- Supports a large (and growing) number of constructions
- Quickly becoming a standard markup language for Python documentation
  - Currently used for PEPs
  - Might be used for the standard library reference documentation in the future.

Fields

- A “tagged” portion of a docstring that describes a specific property of an object.
  - Descriptions of parameters & return values
  - Information about how objects are organized
  - Metadata about objects
- Why use fields?
  - Specialized presentation
  - Specialized processing

Fields: Signature Specification

- Describe individual function/method parameters.
- Specify a function/method's type signature.

@param *p*: …	Describes parameter *p*
@return: …	Describes of the return value
@kwparam *p*: …	Describes keyword param *p*
@type *p*: …	Parameter *p*'s type
@returntype: …	The return value's type
@raise *e*: …	Conditions that cause an exception

Fields: Variable Documentation

- Describe variables & specify their types
  - Variables can't define docstrings.

@var *v*: …	Describes module variable *v*
@ivar *v*: …	Describes instance variable *v*
@cvar *v*: …	Describes class variable *v*
@type *v*: …	Variable *v*'s type

- In the works:
  - Read pseudo-docstrings for variables (from string literals or specially marked constants).

Fields: Content Organization

- Specify how objects are organized.

@group g: c_1, …, c_n	Defines a named collection of related objects.
@sort: c_1, …, c_n	Specifies the order in which objects should be listed
@undocumented: *c*	Indicates that an object should not be listed in the documentation

Fields: Metadata & Tagged Information

- Describe specific aspects of an object.
  - Consistent presentation of information
  - Automatic processing (e.g. creating a bug index)

@seealso: …	@author: …
@bug: …	@version: …
@todo: …	@depreciated: …
@warning: …	@copyright: …
@license: …	@precondition: …

etc.

Fields: Create Your Own!

- Epydoc provides two mechanisms for defining new fields:
  - A special field:
    @newfield *tag*: *label* [, *plural-label*]
  - A module-level variable:
    __extra_epydoc_fields__ = [
            (*tag* [, *label* [, *plural-label*]])
            ]

Extracting Documentation

- Two prevalent methods for extracting API documentation from Python:
  - *Inspection:* Import the library, and examine each object's attributes directly.

```
>>> import zipfile
>>> docstring = zipfile.__doc__
>>> …
```

  - *Source code parsing:* Parse the library's source code, and extract relevant information.

```
>>> sourcecode = open('zipfile.py').read()
>>> ast = parser.suite(sourcecode)
>>> …
```

Extracting Documentation: Limitations of Parsing

- Can't capture the effects of dynamic transformations
  - Metaclasses
  - Namespace manipulation
- Can't document non-python modules
  - Extension modules
  - Javadoc modules
  - Non-python base classes for python modules

Extracting Documentation: Limitations of Inspection

- Some information is unavailable via inspection:
  - What module defines a given function?
  - Which objects are imported vs defined locally?
    - E.g., integer constants
  - Pseudo-docstrings for variables.
- Can't document "insecure" code
- Can't document modules that perform complex or interactive tasks when imported
  - E.g., opening a Tkinter window

Extracting Documentation

- Epydoc's answer: use a hybrid approach!
  - Inspection forms the basis for documentation extraction
    - Inspection gives a more accurate representation of the user-visible behavior.
  - Source code parsing is used to overcome the limitations of inspection, where necessary.
- Using this hybrid approach, Epydoc can generate comprehensive API documentation for almost any libraries.

Thank you!

ed@loper.org
http://epydoc.sourceforge.net/