innoConv Documentation

Release 0.1.1

Innocampus

Jun 24, 2020

CONTENTS

CHAPTER

ONE

TABLE OF CONTENTS

1.1 What is innoConv?

innoConv is a converter for educational content.

The software transforms source content into an intermediate JSON representation that can be displayed with the help of an *innodoc-compatible viewer*.

It takes plain-text files as a source. These are written in the Markdown language and stored in a particular *directory structure* reflecting the sections and subsections of the work.

See also:

Check out the *addtional documentation* to see how a real course looks like.

1.1.1 Features

Common features as basic text formatting, links, tables, lists, etc. are already provided by Markdown out-of-the-box.

While staying as close to traditional Markdown as possible innoConv supports a variety of additional constructs. Many of them are targeted specifically at the creation of educational content.

These include

- Localization
- Math formulas
- Images and videos
- Interactive exercises
- Vector graphics
- Table of contents
- Inter-section references
- Index

1.1.2 innoDoc

innoConv is a part in the software package innoDoc. It handles the translation of source content to the an intermediate JSON represenation.

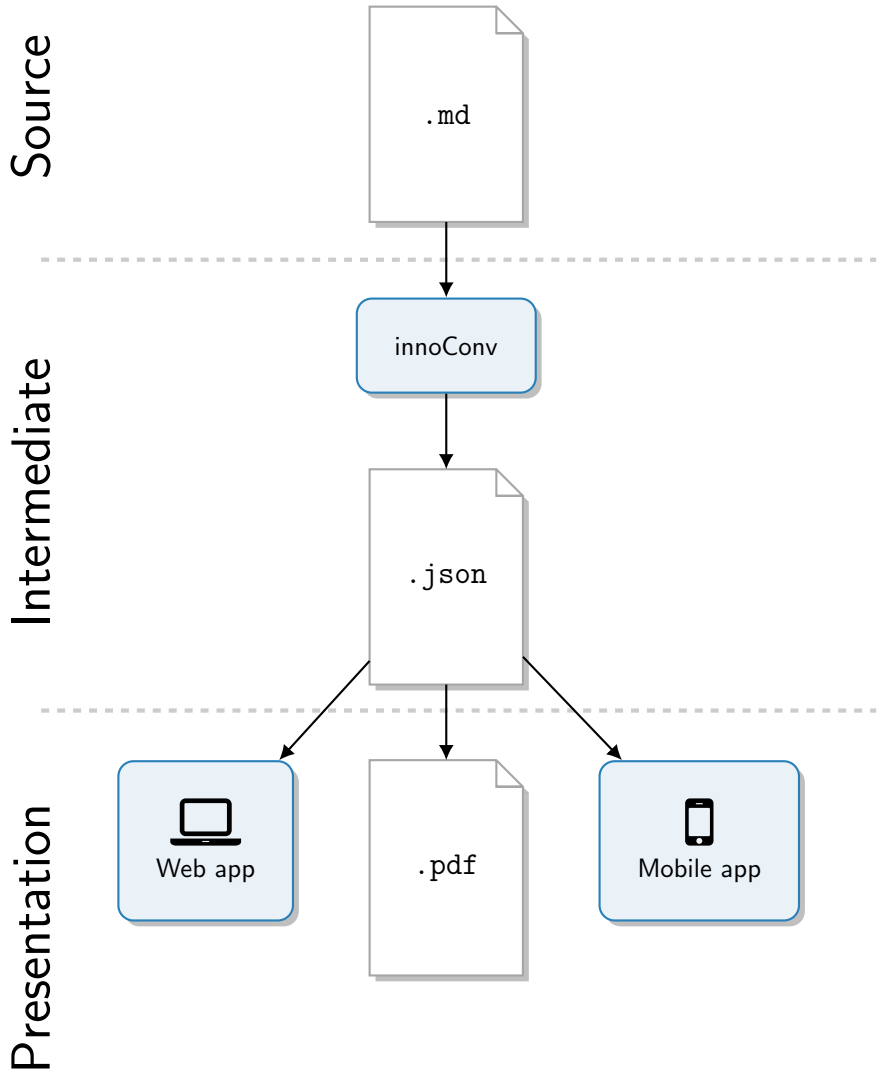

Fig. 1: Overview of the innoDoc software architecture.

innoConv does neither have any busisness with how content is displayed nor helps in its creation. Instead it leaves these tasks completely to others in the processing chain.

See also:

See section *Content creation* for a in-depth discussion on how to write course content.

1.1.2.1 Viewers

At the moment there are two viewers in development.

innodoc-webapp React-based HTML5 web application

innodoc-app React Native-based Smartphone App

Note: Configuration and deployment of viewers is not the scope of this document. Please refer to the respective documentation.

1.2 Getting started

1.2.1 Prerequisites

innoConv is mainly used on Linux machines. It might work on Mac OS and Windows/Cygwin/WSL. You are invited to share your experiences.

1.2.2 Dependencies

The only dependencies you have to provide yourself is Pandoc and the Python interpreter.

1.2.2.1 Python 3

innoConv is being tested and developed with **Python 3.5-3.8**.

Python should be available on the majority of Linux machines nowadays. Usually it's being installed using a package manager.

1.2.2.2 Pandoc

You need to make sure to have a recent version of the **`pandoc`** binary available in `$PATH` (version 2.9.2.1 at the time of writing). There are several ways how to install Pandoc.

1.2.3 Installation

1.2.3.1 Using pip

The easiest way to install innoConv is to use **`pip`**.

Given you have a regular Python setup with **`pip`** available the following installs innoConv in your user directory (usually `~/.local` under Linux).

```
$ pip install --user innoconv
```

For the **`innoconv`** command to work, make sure you have `~/.local/bin` in your `$PATH`.

For a system-wide installation you can omit the `--user` argument.

```
$ pip install innoconv
```

1.2.3.2 In a virtual environment

It's possible to install innoConv into a virtual environment. Setup and activate a virtual environment in a location of your choice.

```
$ python3 -m venv /path/to/venv
$ source /path/to/venv/bin/activate
```

Install innoConv in your virtual environment using **pip**.

```
$ pip install innoconv
```

If everything went fine you should now have access to the **innoconv** command.

The next time you login to your shell make sure to activate your virtual environment before using **innoconv**.

1.3 How to use innoConv

The principle way of using innoConv is the CLI (Command-line interface) **innoconv**. Another option is to use innoConv in a *programmatic way* as Python library.

Run the converter on your content directory.

```
$ innoconv /path/to/my/content
```

This will trigger a conversion and store the result in a folder `innoconv_output`. A return code other than `0` indicates an unsuccessful run.

Note: According to Unix convention you will not see any messages if the conversion was successful. Though you might pass the `--verbose` flag to change this behavior.

1.3.1 Command line arguments

1.3.1.1 innoconv

Converter for interactive educational content.

```
innoconv [OPTIONS] SOURCE_DIR
```

Options

-o, --output-dir <output_dir>
Set output directory. [default: ./innoconv_output]

-e, --extensions <extensions>
Enable extensions (comma-separated). [default: copy_static,generate_toc,index_terms,join_strings,number_boxes,tikz2svg,write_

-f, --force
Force overwriting of output.

-v, --verbose
Print verbose messages.

--version
Show the version and exit.

Arguments

SOURCE_DIR
Required argument

1.3.2 Using innoConv as a library

Using innoConv within a Python program involves creating a *Manifest* object and using it with a *InnoconvRunner*.

```
>>> manifest = Manifest(manifest_data)
>>> runner = InnoconvRunner(source_dir, output_dir, manifest, extensions)
>>> runner.run()
```

Have a look at the source of *innoconv.cli* for a more detailed example.

1.4 Content creation

This section deals with the creation of content files that are fed into innoConv for processing.

As already mentioned, the main format for writing content is Markdown. The available references for the Markdown language generally do apply for innoConv but we will direct you specifically to Pandoc's Markdown. The reason for that is Markdown exists in different flavours and innoConv is using Pandoc under the hood.

See also:

This section will only give an overview on how to structure your content. It will not provide an exhaustive list of formatting options.

Please refer to the *addtional documentation*. It offers more detailed information on how to create content with innoDoc.

1.4.1 Best practices

1.4.1.1 Text editor

Content is written in plain-text. Therefore you will need a text editor to author innoDoc content. The choices are endless. If in doubt your operating system comes with a text editor pre-installed.

Warning: Please use *UTF-8 character encoding* exclusively when writing documents for innoConv. Make sure your editor uses the right encoding.

1.4.1.2 Version control

Writing large amounts of text is often a joint effort with a lot of edits and many contributors. Using a VCS (Version control system) (e.g. git) for personal use is highly recommended. On a collaborative project it's indispensable.

1.4.2 Writing content

Your content typically resides in a dedicated directory referred to as *root directory* from now on. There are some conventions that you need to follow. These are explained in this section.

1.4.2.1 The manifest file

The root directory is home to the `manifest.yml` file. It is used to store meta information about your content, like the title, the languages and so on. It is written in YAML format.

Listing 1: A minimal example for a content manifest.

```
title:
  en: Example course
  de: Beispielkurs
languages: en,de
```

Note: If your content uses just one language, you will still need to list the language here.

1.4.2.2 Directory and file structure

Sections and subsections

In the previous section we saw how to specify the available languages for the content. For every language one sub-directory needs to exist in the root directory.

Under each language directory there is a structure of folders reflecting the part/chapter/section structure of the text.

The names of the directories determine the section order. They are sorted alphanumerically. Therefore it's advisable to use a numerical prefix (e.g. `01-section`).

The directory name itself is the ID for the section and can be used to create cross-references from one part in the the text to another. Also, they are used to form URLs (Uniform Resource Locator).

Note: While technically not strictly required, for convenience it's advisable to limit directory names to characters, numbers, hyphen and underscore (`a-z`, `0-9`, `-` and `_`).

A file `content.md` needs to exist in every section folder. It has a small section at the top of the file called YAML metadata block that contains the section title.

Listing 2: Example YAML metadata block.

```
---
title: Example title for this section
---
```

After the metablock you can write your actual content.

Note: A `content.md` needs to exist for every language version, e.g. `en/section01/content.md` and `de/section01/content.md`.

Custom pages

A course can also include custom pages that are not part of the section structure. You can define pages by adding them to the `pages` key of the *manifest file*. You need to define an ID, optionally an icon and can choose if the page should show up in the navigation and footer part of the viewer.

```
pages:
  - id: about
    icon: info-circle
    link_in_nav: true
    link_in_footer: true
```

For every page you need to provide a content file in each language. It uses the page ID as the name (e.g. `about.md`). The content file is placed in the `_pages` directory inside the language folder.

Pages also need a YAML header like described in *Sections and subsections*.

Example directory structure

Listing 3: Example directory structure for two languages.

```
root
├── manifest.yml
├── en
|   ├── _pages
|   |   ├── about.md
|   |   └── ...
|   ├── content.md
|   ├── 01-part
|   |   ├── content.md
|   |   ├── 01-section
|   |   |   └── content.md
|   |   └── ...
|   └── ...
```

(continues on next page)

(continued from previous page)

```
└── de
    ├── _pages
    |   ├── about.md
    |   └── ...
    ├── content.md
    ├── 01-part
    |   ├── content.md
    |   ├── 01-section
    |   |   └── content.md
    |   └── ...
    └── ...
```

Important: The directory structure in each of the language folders need to match!

Static files

The directory `_static` is used for placing static files such as images and videos.

The directory exists under the root and can also be placed inside a language folder for content that needs to be localized. The converter will prefer files from the localized folder.

Listing 4: Locations for static files.

```
root
├── _static
|   ├── chart.svg
|   └── image.png
├── en
|   └── _static
|       └── video.mp4
└── de
    └── _static
        └── video.mp4
```

For the sake of clarity other needed files and directories are omitted in this listing.

1.4.3 Additional documentation

For more detailed instructions including examples on how to author content refer to the innoDoc example course. It features in-depth descriptions on all content elements and the general course structure.

Note: If you want to start compiling content, check out the source code and start using innoConv right away.

1.4.3.1 Links

- innoDoc example course
- Source repository

1.5 Module overview

1.5.1 innoconv.cli

Please see section *Command line arguments* on how to use the CLI.

Command line interface for the innoConv document converter.

class innoconv.cli.**CustomEpilogCommand**(*name*, *context_settings=None*, *callback=None*, *params=None*, *help=None*, *epilog=None*, *short_help=None*, *options_metavar='[OPTIONS]'*, *add_help_option=True*, *no_args_is_help=False*, *hidden=False*, *deprecated=False*)

Format epilog in a custom way.

format_epilog(*_*, *formatter*)

Format epilog while preserving newlines.

1.5.2 innoconv.constants

Project constants.

innoconv.constants.**DEFAULT_OUTPUT_DIR_BASE**

Default innoconv output directory

innoconv.constants.**DEFAULT_EXTENSIONS**

Default enabled extensions

innoconv.constants.**ENCODING**

Encoding used in this project

innoconv.constants.**CONTENT_BASENAME**

Basename for the content file in a section

innoconv.constants.**MANIFEST_BASENAME**

Manifest filename

innoconv.constants.**STATIC_FOLDER**

Static folder name

1.5.3 innoconv.ext

innoConv extensions.

Extensions are a way of separating concerns of the conversion process into independent modules. They can be enabled on a one-by-one basis as not all features are needed in all cases.

Extensions interface with *InnoconvRunner* through a set of methods defined in *AbstractExtension*.

innoconv.ext.**EXTENSIONS = {'copy_static': <class 'innoconv.ext.copy_static.CopyStatic'>,**
List of available extensions

1.5.4 innoconv.ext.abstract

Base class for all other extensions.

The AbstractExtension is not instantiated directly but serves as super-class to all extensions.

class innoconv.ext.abstract.**AbstractExtension**(*manifest*)
Abstract class for extensions.

The class all extensions inherit from. The to-be-implemented methods document the available events that are triggered during the conversion process.

Extension classes should have a `_helptext` attribute. It's used to display a brief summary.

Parameters **manifest** (innoconv.manifest.Manifest) – Content manifest.

classmethod helptext()
Return a brief summary of what the extension is doing.

extension_list(*extensions*)
Receive list of active extension instances.

Parameters **extensions** (*list*) – List of all active extension instances

start(*output_dir*, *source_dir*)
Conversion is about to start.

Parameters

- **output_dir** (*str*) – Base output directory
- **source_dir** (*str*) – Content source directory

pre_conversion(*language*)
Conversion of a single language folder is about to start.

Parameters **language** (*str*) – Language that is currently being converted.

pre_process_file(*path*)
Conversion of a single file is about to start.

Parameters **path** (*str*) – Output path

post_process_file(*ast*, *title*, *content_type*)
Conversion of a single file finished. The AST can be modified.

Parameters

- **ast** (*List of content nodes*) – File content as parsed by pandoc.
- **title** (*str*) – Section title (localized)
- **content_type** (*str*) – Content type ('section' or 'custom')

post_conversion(*language*)
Conversion of a single language folder finished.

Parameters **language** (*str*) – Language that is currently being converted.

finish()
Conversion finished.

1.5.5 innoconv.ext.copy_static

Extension that copies static files.

Content can include figures, images and videos. Static files can be included in a special folder named `_static`. The files will be copied to the output directory automatically.

1.5.5.1 Translation

It's possible to have language-specific versions of a static file.

For that to work you need to have a `_static` folder beneath the language folder. Files in this folder will take precedence over the common `_static` folder for that language.

Example: `en/_static/example.png` takes precedence over `_static/example.png` in the English version.

1.5.5.2 Relative and absolute reference

Files can be referenced using relative or absolute paths.

Absolute paths are resolved to the root folder, either the common (`_static`) or language-specific (`en/_static`) folder.

Relative paths are resolved to the root folder but have the chapters path fragment appended.

Example

This example shows how a reference to an image is resolved. The references happen inside the section `chapter01` in the English language version.

Type	Resolution
Relative	`subdir/my_picture.png` → `en/_static/chapter01/subdir/my_picture.png`
Absolute	`/subdir/my_picture.png` → `en/_static/subdir/my_picture.png`

class innoconv.ext.copy_static.**CopyStatic**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Copy static files to the output folder.

This extension copies checks the AST for references to static files and copies them from the content source directory to the output directory.

process_element(*elem*, *_*)
Respond to AST element.

start(*output_dir*, *source_dir*)
Remember directories.

pre_conversion(*language*)
Remember current conversion language.

pre_process_file(*path*)
Remember file path.

post_process_file(*ast*, _, __)
Find all static files in AST.

finish()
Copy static files to the output folder.

manifest_fields()
Add *logo* field to manifest.

1.5.6 innoconv.ext.generate_toc

Extension that generates a table of contents.

A table of contents is generated from the course sections and added to the *Manifest*.

class innoconv.ext.generate_toc.**GenerateToc**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Generate a TOC from content sections.

start(*output_dir*, _)
Remember output directory.

pre_conversion(*language*)
Remember current conversion language.

pre_process_file(*path*)
Remember current path.

post_process_file(_, *title*, *content_type*)
Add this section file to the TOC.

manifest_fields()
Add *toc* field to manifest.

1.5.7 innoconv.ext.index_terms

Scan documents for index terms.

Viewers may generate a list of words with links to locations in the documents. In order to achieve this a list of index terms is provided in the *Manifest* so viewers don't have to scan the whole documents themselves. Also for every occurence of an index term in the text an ID is attached.

This extension modifies the AST.

Note: Index terms are not supported in custom pages.

class innoconv.ext.index_terms.**IndexTerms**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Scan the documents for index terms.

process_element(*elem*, *_*)
Respond to AST element.

pre_conversion(*language*)
Remember current conversion language.

pre_process_file(*path*)
Remember current path.

post_process_file(*ast*, *_*, *content_type*)
Scan the AST.

manifest_fields()
Add *index_terms* field to manifest.

start(*output_dir*, *source_dir*)
Conversion is about to start.

Parameters

- **output_dir** (*str*) – Base output directory
- **source_dir** (*str*) – Content source directory

1.5.8 innoconv.ext.join_strings

Merge consecutive sequences of strings and spaces into a single string element.

The motivation behind this extension is to make the AST more readable and also to save space by compressing the representation. The actual appearance in a viewer remains identical.

This extension modifies the AST.

1.5.8.1 Example

```
{"t":"Str","c":"Foo"},{"t":"Space"},{"t":"Str","c":"bar"}]  →  {"t":"Str",
"c":"Foo bar"}]
```

class innoconv.ext.join_strings.**JoinStrings**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Merge consecutive strings and spaces in the AST.

post_process_file(*ast*, *_*, *__*)
Process AST in-place.

1.5.9 innoconv.ext.number_boxes

Scan documents for numbered boxes (info, example, exercise).

Viewers need to be able to quickly number and reference boxes without scanning the whole document structure themselves. This extension adds a field to the *Manifest* that lists all boxes per section.

It will assign an auto-generated `ID` based on the box type and number to the box div element (in case it doesn't already have one). Also a `data-number` attribute is attached.

Furthermore this extension faciliates the course-wide tracking of exercise progress. For each exercise, the achievable points are stored. A viewer application can easily display total points per section without having to scan all documents for exercises.

class innoconv.ext.number_boxes.**NumberBoxes**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Scan the documents for boxes.

process_element(*elem*, *_*)
Respond to AST element.

pre_conversion(*language*)
Remember current conversion language.

post_conversion(*_*)
Set scan to done.

pre_process_file(*path*)
Remember current path.

post_process_file(*ast*, *_*, *content_type*)
Scan the AST.

manifest_fields()
Add `boxes` field to manifest.

1.5.10 innoconv.ext.tikz2svg

Convert and insert Ti*k*Z figures.

SVG files are rendered from Ti*k*Z code and saved in the folder `_tikz` in the static folder of the output directory.

Ti*k*Z code blocks are replaced by image elements.

Note: In order to use this extension you need to have the following software installed on your system:

- LaTeX distribution with PGF/Ti*k*Z
- pdf2svg

1.5.10.1 Example

A Ti*k*Z image is directly embedded into Markdown using a code block.

```
```tikz
\\begin{tikzpicture}
\\shade[left color=blue,right color=red,rounded corners=8pt] (-0.5,-0.5)
  rectangle (2.5,3.45);
\\draw[white,thick,dashed,rounded corners=8pt] (0,0) -- (0,2) -- (1,3.25)
  -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
\\node[white] at (1,-0.25) {\\footnotesize House of Santa Claus};
\\end{tikzpicture}
```
```

During conversion the code block will be turned into an image tag, similar to the following.

```
![](/_tikz/tikz_abcdef0123456789.svg "Alt text")
```

class innoconv.ext.tikz2svg.**Tikz2Svg**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Convert and insert Ti*k*Z images.

process_element(*elem*, *parent*)
Respond to AST element.

start(*output_dir*, *source_dir*)
Initialize the list of images to be converted.

post_process_file(*ast*, _, __)
Find TikZ images in AST and replace with image tags.

finish()
Render images and copy SVG files to the static folder.

1.5.11 innoconv.ext.write_manifest

Extension that writes a `manifest.json` to the output folder.

Every course needs a *Manifest*. Additionally to the fields from the source manifest it can include a table of contents and a glossary.

class innoconv.ext.write_manifest.**WriteManifest**(**args*, ***kwargs*)
Bases: *innoconv.ext.abstract.AbstractExtension*

Write a manifest file when conversion is done.

start(*output_dir*, _)
Remember output directory.

finish()
Output course manifest.

1.5.12 innoconv.manifest

The manifest comprises course metadata.

A `manifest.yml` file needs to exist in every course content and resides at the content root directory.

There is also a representation in JSON format. It is generated automatically by the extension *WriteManifest* and copied to the output folder.

Other extensions may add custom fields to the output manifest by implementing a method `manifest_fields()`. It needs to return a `dict` that is merged into the manifest.

A `manifest.yml` is written by course authors while the `manifest.json` is generated by the converter and used by innoconv-compatible viewers. The included information for both versions differ.

1.5.12.1 Example

```
title:
  en: Example title
  de: Beispiel-Titel
languages: en,de
min_score: 90
```

class innoconv.manifest.**Manifest**(*data*)
Represents course metadata.

Parameters **data** (*dict*) – A dict with the manifest fields as keys

classmethod from_directory(*dirpath*)
Read manifest from content directory.

Parameters **dirpath** (*str*) – Full path to content directory.

Return type *Manifest*

Returns Manifest object

classmethod from_yaml(*yaml_data*)
Create a manifest from YAML data.

Parameters **yaml_data** (*str*) – YAML representation of a manifest

Return type *Manifest*

Returns Manifest object

1.5.13 innoconv.runner

The innoConv runner is the core of the conversion process.

It traverses the source directory recursively and finds all content files. These are converted one-by-one to JSON. Under the hood is uses Pandoc.

It receives a list of extensions that are instantiated and notified upon certain events. The events are documented in *AbstractExtension*.

class innoconv.runner.**InnoconvRunner**(*source_dir*, *output_dir*, *manifest*, *extensions*)
Convert content files in a directory tree.

Parameters

- **source_dir** (*str*) – Content source directory.
- **output_dir** (*str*) – Output directory.
- **manifest** (innoconv.manifest.Manifest) – Content manifest.
- **extensions** (*list[str]*) – List of extension names to use.

run()
Start the conversion by iterating over language folders.

1.5.14 innoconv.traverse_ast

This module helps with traversing an AST.

exception innoconv.traverse_ast.**IgnoreSubtree**
Used to signal an elements sub-tree should not be traversed.

class innoconv.traverse_ast.**TraverseAst**(*func*)
Traverse an AST calling a custom function on each element.

Parameters **func** (*function(dict, dict)*) – Callback for handling an element. Receives element and the parent as parameters.

traverse(*ast*, *parent=None*)
Traverse an AST calling a function on each element.

Parameters

- **ast** (*list*) – Abstract syntax tree to traverse.

- **parent** (*dict*) – Parent of current subtree.

1.5.15 innoconv.utils

Utility module.

innoconv.utils.**to_ast**(*filepath*, *ignore_missing_title=False*)
Convert a file to abstract syntax tree using pandoc.

Parameters

- **filepath** (*str*) – Path of file
- **ignore_missing_title** (*bool*) – Accept missing title in source file

Return type (list of dicts, str)

Returns (Pandoc AST, title)

Raises

- **RuntimeError** – if pandoc exits with an error
- **ValueError** – if no title was found

innoconv.utils.**to_string**(*ast*)
Convert AST to string (handles String and Space only).

Parameters **ast** (*List*) – Content AST.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

i

INDEX

Symbols

A

C

D

E

F

G

H

I

J

M

N

P

R

S

T

W