inxs Documentation

Release 0.2b1

Frank Sachsenheim

Jun 23, 2019

Contents

CHAPTER 1

Prerequisites

At least Python 3.6 is required, delb is installed as dependency.

CHAPTER 2

From the cheeseshop

To install inxs, run this command in your terminal:

```
$ pip install inxs
```

This is the preferred method to install inxs, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

CHAPTER 3

From the sources

The sources for inxs can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/funkyfuture/inxs
```

Or download the tarball:

```
$ curl  -OL https://github.com/funkyfuture/inxs/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Or install an editable instance:

```
$ python setup.py develop
```

CHAPTER 4

Usage

`inxs` is designed to allow Pythonistas and other cultivated folks to write sparse and readable transformations that take delb objects as input. Most likely they will return the same, but there's no limitation into what the data can be mangled. It does so by providing a framework that traverses an XML tree, tests tag nodes, pulls and manipulates data in a series of steps. It supports the combination of reusable and generalized logical units of expressions and actions. Therefore there's also a library with functions to deploy and a module with contributed transformations.

Though `inxs` should be usable for any problem that XSLT could solve, it is not modeled to address XSLT users to get a quick grip on it. Anyone who enjoys XSLT should continue to do so. So far the framework performs with acceptable speed with uses on text documents from the humanities.

Let's break its usage down with the second example from the `README`:

```
1 transformation = Transformation(
2     generate_skeleton,
3     Rule('person', extract_person),
4     lib.sort('persons', itemgetter(1)),
5     list_persons,
6     result_object='context.html', context={'persons': []})
```

A transformation is set up by instantiating a *`inxs.Transformation`* (line 1) with a series of *transformation steps* (lines 2-5) passed as positional argument s and two *configuration* values (line 6) provided as keyword arguments.

The first step (line 2) is a function that creates a skeleton for the resulting HTML markup and stores it in the `context` namespace:

```
def generate_skeleton(context):
    context.html = new_tag_node(
        "html", namespace='http://www.w3.org/1999/xhtml',
        children=(
            tag("head",
                tag("title", "Testing XML Example")),
            tag("body", (
                tag("h1", "Persons"),
                tag("ul")
            )),
```

(continues on next page)

(continued from previous page)

```
        )
    )
```

When a transformation calls a handler function it does so by applying dependency injection as may be known from pytest's fixtures. The passed arguments are resolved from *inxs.Transformation._available_symbols* where any object that has previously been added to the `context` namespace is available as well as the `context` itself.

Line 3 defines something that is used more often in real world uses than here. A *inxs.Rule* that tests the *transformation root* and its descendants for defined properties. In the example all nodes with a `person` tag will be passed to the associated *handler function*:

```
def extract_person(node: TagNode, persons):
    persons.append(
        (first(node.css_select("name")).full_text,
         first(node.css_select("family-name")).full_text)
    )
```

delb's API is used to fetch child nodes of the matching nodes, extract their text and appends them in a tuple to a list that was defined in the `context` argument of the configuration values (line 7).

Rules can also test anything outside the scope of a node, the utilized functions however aren't 'dependency injected' to avoid overhead. They are called with `node` and `transformation` as arguments and take it from there. See *inxs.If()* for an example.

The last two steps (line 4 and 5) eventually sort (*inxs.lib.sort()* with `operator.itemgetter()`) and append the data to the HTML tree that was prepared by the step in line 2:

```
def list_persons(previous_result, html: TagNode):
    first(html.css_select("html|body html|ul")).append_child(
        *(html.new_tag_node("li", children=[f'{x[1]}, {x[0]}'])
          for x in previous_result)
    )
```

The argument `previous_result` is resolved to the object that the previous function returned, again the `delb` API and Python's f-string s are used to generate the result.

As the transformation was configured with `context.html` as result object, the transformation returns the object referenced as `html` (see handler function in line 2) from the context. If the transformation hasn't explicitly configured a result object, (per default a copy of) the *transformation root* is returned. Any other data is discarded.

The initialized transformation can now be called with a `delb.Document` or `delb.TagNode` instance as *transformation root*:

```
>>> result = transformation(document)  # doctest: +SKIP
```

A *transformation root* can be any node within a document, leaving siblings and ancestors untouched. A transformation works on a copy of the document's tree unless the configuration contains a key `copy` set to `False` or the transformation is called with such keyword argument.

Transformations can also be used as simple steps - then invoked with the *transformation root* - or as rule handlers - then invoked with each matching node. Per default these do not operate on copies, to do so *inxs.lib.f()* can be employed:

```
# as a simple step
f(sub_transformation, 'root', copy=True)
# as a rule handler
f(sub_transformation, 'node', copy=True)
```

Any transformation step, condition or handler can be grouped into sequence s to encourage code recycling - But don't take that as a permission to barbarously patching fragments of existing solutions together that you might feel are similar to your problem. It's taken care that the items are retained as when a transformation was initialized if groups were mutable types.

Now that the authoritarian part is reached, be advised that using expressive and unambiguous names is essential when designing transformations and their components. As a rule of thumb, a simple transformation step should fit into one line, rules into two, maybe up to four. If it gets confusing to read, use variables, grouping (more reusability) or dedicated functions (more performance) - again, mind the names! Reciting the Zen of Python on a daily basis makes you a beautiful person. Yes, even more.

To get a grip on implementing own condition test functions and *handler function* s, it's advised to study the `inxs.lib` module.

And now, space for some spots-on-.. sections.

4.1 Traversal strategies

When a rule is evaluated, the document (sub-)tree is traversed in a specified order. There are three aspects that must be combined to define that order and are available as constants that are to be or'ed bitwise:

- `inxs.TRAVERSE_DEPTH_FIRST` / `inxs.TRAVERSE_WIDTH_FIRST`
- `inxs.TRAVERSE_LEFT_TO_RIGHT` / `inxs.TRAVERSE_RIGHT_TO_LEFT`
- `inxs.TRAVERSE_TOP_TO_BOTTOM` / `inxs.TRAVERSE_BOTTOM_TO_TOP`

Rules can be initiated with such value as `traversal_order` argument and override the transformation's one (that one defaults to `..._DEPTH_FIRST | ..._LEFT_TO_RIGHT | ..._TOP_TO_BOTTOM`). Not all strategies are are implemented yet.

`inxs.TRAVERSE_ROOT_ONLY` sets a strategy that only considers the *transformation root*. It is also set implicitly for rules that contain a `'/'` as condition (see *Rule condition shortcuts*).

4.2 Rule condition shortcuts

Strings can be used to specify certain rule conditions:

- `/` selects only the *transformation root*
- `*` selects all nodes - should only be used if there are no other conditions
- any string that contains `://` selects nodes with a namespace that matches the string
- strings that contain only letters select nodes whose *local* name matches the string
- if a string can be translated to an XPath expression with cssselect and thus can be considered a valid css selector, the result is used like the following; mind that you can use namespace prefixes if you know the prefixes, otherwise this is not an option to match a node from a namespace that's not the *transformation root*'s default
- all other strings will select all nodes that an XPath evaluation of that string on the *transformation root* returns

Another shortcut is to pass a dictionary to test an node's attributes, see `inxs.MatchesAttributes()` for details.

Speaking of conditions, see `inxs.Any()`, `inxs.OneOf()` and `inxs.Not()` to overcome the logical `and` evaluation of all tests.

4.3 Global configuration

`inxs` caches and reuses evaluator and handler functions with identical arguments where possible. By default these caches are not limited in size and they might eventually grow larger than the memory that was saved in big, long-running applications that create a lot of short-living transformations. To limit the size of each of these last-recently-used-caches, the environment variable `HANDLER_CACHES_SIZE` can be set. The value should be a power of two.

4.4 Caveats

4.4.1 Modifications during iteration

Similar to iteration over mutable types in Python, adding, moving or deleting nodes to the tree breaks the iteration of a rule over nodes. Thus such modifications must be applied in a simple transformation step; e.g. to remove all `<br>` nodes from a document:

```
def collect_trash(node, trashbin):
    trashbin.append(node)

transformation = Transformation(
    Rule('br', collect_trash),
    lib.remove_nodes('trashbin'),
    context={'trashbin': []})
```

4.5 Debugging / Logging

There are functions in the *inxs.lib* module to log information about a transformation's state at info level. There's a `logger` object in that module too that needs to be set up with a handler and a log level in order to get the output (see `logging`). `inxs` itself produces very noisy messages at debug level.

inxs.lib.debug_dump_document(), *inxs.lib.debug_message()* and *inxs.lib.debug_symbols()* can be used as *handler function*. `inxs.lib.dbg()` and `inxs.lib.nfo()` can be used within test and handler functions.

Due to its rather sparse and dynamic design, the exception tracebacks that are produced aren't very helpful as they contain no information about the context of an exception. To tackle one of those, a minimal non-working example is preferred to debug.

4.6 Glossary

configuration The configuration of a transformation is a `types.SimpleNamespace` object that is bound as its `config` property and is populated by passing keywords arguments to its initialization. It is intended to be an immutable container for key-value-pairs that persist through transformation's executions. Mind that it's immutability isn't completely enforced, manipulating it or its members might result in unexpected behaviour. It can be referred to in *handler function*'s signatures as `config`, the same is true for its member unless overridden in *inxs.Transformation._available_symbols*. See *inxs.Transformation* for details on reserved names in the configuration namespace.

context The context of a transformation is a `types.SimpleNamespace` instance and intended to hold any mutable values during a transformation. It is initialized from the values stored in the *configuration*'s `context` value and the overriding keywords provided when calling a *inxs.Transformation* instance.

handler function Handler functions can be employed as simple *transformation steps* or as conditionally executed `handlers` of a *`inxs.Rule`*. Any of their signature's argument s must be available in *`inxs.Transformation._available_symbols`* upon the time the function gets called.

transformation root This is the node that a transformation instance is called with. Any traverser will return neither its ancestors nor its siblings.

transformation steps Transformation steps are *handler functions* or *`inxs.Rule`* s that define the actions taken when a transformation is processed. The steps are stored as a linear graph, rudimentary branching can be achieved by using rules that call other transformations.

CHAPTER 5

API documentation

See also:

5.1 inxs.contrib module

This module contains transformations that are supposedly of common interest.

inxs.contrib.**reduce_whitespaces = <inxs.Transformation object>**
Normalizes any whitespace character in text nodes to a simple space and reduces consecutive ones to one. Leading or tailing whitespaces are not stripped away.

inxs.contrib.**remove_empty_nodes = <inxs.Transformation object>**
Removes nodes without attributes, text and children.

5.2 inxs.lib module

This module contains common functions that can be used for either *Rule* s' tests, as handler functions or simple transformation steps.

Community contributions are highly appreciated, but it's hard to layout hard criteria for what belongs here and what not. In doubt open a pull request with your proposal as far as it proved functional to you, it doesn't need to be polished at that point.

inxs.lib.**add_html_classes**
Adds the string tokens passed as positional arguments to the `classes` attribute of a node specified by `target`. An argument can also be a sequence of strings or a *Ref()* that yields on of the two. Per default that is a *Ref()* to the matching node of a rule.

inxs.lib.**append**
Appends the object referenced by `symbol` (default: the result of the previous *handler function*) to the object available as `name` in the `Transformation._available_symbols`. If the object is a `delb.TagNode` instance and `copy_node` is `True`, a copy that includes all descendant nodes is appended to the target.

inxs.lib.**cleanup_namespaces**(*root: delb.nodes.TagNode*, *previous_result: Any*) → Any
Cleanup the namespaces of the tree. This should always be used at the end of a transformation when nodes' namespaces have been changed.

inxs.lib.**clear_attributes**(*node: delb.nodes.TagNode*, *previous_result: Any*) → Any
Deletes all attributes of an node.

inxs.lib.**concatenate**
Concatenate the given parts which may be lists or strings as well as callables returning such.

inxs.lib.**debug_dump_document**
Dumps all contents of the node referenced by `name` from the *inxs.Transformation._available_symbols* to the log at info level.

inxs.lib.**debug_message**
Logs the provided message at info level.

inxs.lib.**debug_symbols**
Logs the representation strings of the objects referenced by `names` in *inxs.Transformation._available_symbols* at info level.

inxs.lib.**f**(*func*, **args*, ***kwargs*)
Wraps the callable `func` which will be called as `func(*args, **kwargs)`, the function and any argument can be given as *inxs.Ref()*.

inxs.lib.**get_attribute**
Gets the value of the node's attribute named `name`.

inxs.lib.**get_localname**(*node*)
Gets the node's local tag name.

inxs.lib.**get_text**(*node: delb.nodes.TagNode*)
Returns the content of the matched node's descendants of `delb.TextNode` type.

inxs.lib.**get_variable**
Gets the object referenced as `name` from the *context*. It is then available as symbol `previous_result`.

inxs.lib.**has_attributes**(*node: delb.nodes.TagNode*, _)
Returns `True` if the node has attributes.

inxs.lib.**has_children**(*node: delb.nodes.TagNode*, _)
Returns `True` if the node has descendants.

inxs.lib.**has_matching_text**
Returns `True` if the text contained by the node and its descendants has a matches the provided `pattern`.

inxs.lib.**has_text**(*node: delb.nodes.TagNode*, _)
Returns `True` if the node has any `delb.TextNode`.

inxs.lib.**insert_fontawesome_icon**
Inserts the html markup for an icon from the fontawesome set with the given `name` at `position` of which only `after` is implemented atm.

It employs semantics for Font Awesome 5.

inxs.lib.**join_to_string**
Joins the object referenced by `symbol` around the given `separator` and returns it.

inxs.lib.**lowercase**(*previous_result*)
Processes `previous_result` to be all lower case.

inxs.lib.**make_node**(***node_args*)
Creates a new tag node in the root node's context, takes the arguments of delb.TagNode.new_tag_node() that must be provided as keyword arguments. The node is then available as symbol previous_result.

inxs.lib.**pop_attribute**
Pops the node's attribute named name.

inxs.lib.**pop_attributes**
Pops all attributes with name from names and returns a mapping with names and values. When ignore_missing is True KeyError exceptions pass silently.

inxs.lib.**prefix_attributes**(*prefix: str*, **attributes*)
Prefixes the attributes with prefix.

inxs.lib.**put_variable**
Puts value``as ``name to the *context* namespace, by default the value is determined by a *inxs.Ref()* to previous_result.

inxs.lib.**remove_attributes**
Removes all attributes with the keys provided as names from the node.

inxs.lib.**remove_namespace**(*node: delb.nodes.TagNode*, *previous_result*)
Removes the namespace from the node. When used, *cleanup_namespaces()* should be applied at the end of the transformation.

inxs.lib.**remove_node**(*node: delb.nodes.TagNode*)
A very simple handler that just removes a node and its descendants from a tree.

inxs.lib.**remove_nodes**
Removes all nodes from their tree that are referenced in a list that is available as references. The nodes' children are retained when keep_children is passed as True, or only the contained text when preserve_text is passed as True. The reference list is cleared afterwards if clear_ref is True.

inxs.lib.**rename_attributes**(*translation_map: Mapping[str, str]*) → Callable
Renames the attributes of a node according to the provided translation_map that consists of old name keys and new name values.

inxs.lib.**resolve_xpath_to_node**
Resolves the objects from the context namespace (which are supposed to be XPath expressions) referenced by names with the *one* node that the expression matches or None. This is useful when a copied tree is processed and 'XPath pointers' are passed to the *context* when a *inxs.Transformation* is called.

inxs.lib.**set_attribute**
Sets an attribute name with value.

inxs.lib.**set_localname**
Sets the node's localname to name.

inxs.lib.**set_text**
Sets the nodes's first child node that is of delb.TextNode type to the one provided as text, it can also be a *inxs.Ref()*. If the first node isn't a text node, one will be inserted.

inxs.lib.**sort**
Sorts the object referenced by name in the *context* using key as key function.

inxs.lib.**text_equals**
Tests whether the evaluated node's text contained by its descendants is equal to text.

5.3 inxs.utils module

inxs.utils.**is_Ref**(*obj*)
Tests whether the given object is a reference to a symbol.

inxs.utils.**reduce_whitespaces**(*text: str*, *translate_to_space: str = '\t\n\r\x0b\x0c'*, *strip: str = 'lr'*) → str
Reduces the whitespaces of the provided string by replacing any of the defined whitespaces with a simple space (U+20) and stripping consecutive ones to a single one.

Parameters

- **text** – The input string.
- **translate_to_space** – The characters that should are defined as whitespace. Defaults to all common whitespace characters from the ASCII set.
- **strip** – The 'sides' of the string to strip from any whitespace at all, indicated by 'l' for the beginning and/or 'r' for the end of the string.

Returns The resulting string.

inxs.utils.**resolve_Ref_values_in_mapping**(*mapping*, *transformation*)
Returns a mapping where all references to symbols are replaced with the current value of these symbols.

5.4 inxs module contents

inxs.**logger = <Logger inxs (WARNING)>**
Module logger, configure as you need.

exception inxs.**AbortRule**
Bases: inxs.FlowControl

Can be raised to abort the evaluation of all the currently processed *inxs.Rule* 's remaining tests and handlers. No further nodes will be considered for that rule. This is similar to Python's builtin `break` in iterations.

exception inxs.**AbortTransformation**
Bases: inxs.FlowControl

Can be raised to cancel the remaining *transformation steps*.

exception inxs.**SkipToNextNode**
Bases: inxs.FlowControl

Can be raised to abort handling of the current node. This is similar to Python's builtin `continue` in iterations.

exception inxs.**InxsException**
Bases: Exception

Base class for inxs exceptions.

inxs.**Any**(**conditions*) → Callable
Returns a callable that evaluates the provided test functions and returns `True` if any of them returned that.

inxs.**Not**(**conditions*) → Callable
Returns a callable that evaluates the provided test functions and returns `True` if any of them returned `False`.

inxs.**OneOf**(**conditions*) → Callable
Returns a callable that evaluates the provided test functions and returns `True` if exactly one of them returned that.

inxs.**HasNamespace**
Returns a callable that tests an node for the given tag namespace.

inxs.**HasLocalname**
Returns a callable that tests an node for the given local tag name.

inxs.**MatchesAttributes**(*constraints: Union[Dict[Union[str, Pattern[~AnyStr]], Union[str, Pattern[~AnyStr], None]], Callable]*) → Callable
Returns a callable that tests an node's attributes for constrains defined in a mapping. All constraints must be matched to resolve as true. Expected keys and values can be provided as string or compiled regular expression object from the `re` module. A `None` as value constraint evaluates as true if the key is in the attributes regardless its value. It also implies that at least one attribute must match the key's constraint if this one is a regular expression object. Alternatively a callable can be passed that returns such mappings during the transformation.

inxs.**MatchesXPath**
Returns a callable that tests an node for the given XPath expression (whether the evaluation result on the *transformation root* contains it). If the `xpath` argument is a callable, it will be called with the current transformation as argument to obtain the expression.

inxs.**If**(*x: Any*, *operator: Callable*, *y: Any*) → Callable
Returns a callable that can be used as condition test in a *Rule*. The arguments `x` and `y` can be given as callables that will be used to get the `operator`'s input values during execution. Before you implement your own operators, mind that there are a lot available within Python's `__builtins__` and the standard library, in particular the `operator` module.

Examples:

```
>>> If(Ref('previous_result'), operator.is_not, None)  # doctest: +SKIP
```

inxs.**Ref**
Returns a callable that can be used for value resolution in a condition test or *handler function* that supports such. The value will be looked up during the processing of a transformation in *Transformation._available_symbols* by the given `name`. This allows to reference dynamic values in *transformation steps* and *Rule* s.

class inxs.**Rule**(*conditions: Union[Callable, AnyStr, Dict[Union[str, Pattern[~AnyStr]], Union[str, Pattern[~AnyStr], None]], Sequence[Union[Callable, AnyStr, Dict[Union[str, Pattern[~AnyStr]], Union[str, Pattern[~AnyStr], None]]]]], handlers: Union[Callable, Sequence[Callable]], name: str = None, traversal_order: int = None*)
Bases: `object`

Instances of this class can be used as conditional *transformation steps* that are evaluated against all traversed nodes.

Parameters

- **conditions** (A single callable, string or mapping, or a sequence of such.) – All given conditions must evaluate as `True` in order for this rule to be applied. Strings and mappings can be provided as shortcuts, see *Rule condition shortcuts* for details. The condition test functions are always called with the currently evaluated `node` and the *Transformation* instance as arguments. There are helper functions for grouping conditions logically: *Any()*, *Not()* and *OneOf()*.
- **handlers** (A single callable or a sequence of such.) – These handlers will be called if the conditions matched. They can take any argument whose name is available in *Transformation._available_symbols*.
- **name** (*String.*) – The optional rule's name.

- **traversal_order** (*Integer.*) – An optional traversal order that overrides the transformation's default `Transformation.config.traversal_order`, see *Traversal strategies* for details.

class inxs.**Once**(**args*, ***kwargs*)
Bases: *inxs.Rule*

This is a variant of *Rule* that is only applied on the first match.

class inxs.**Transformation**(**steps*, ***config*)
Bases: object

A transformation instance is defined by its *transformation steps* and *configuration*. It is to be called with a `delb.Document` or `delb.TagNode` instance as *transformation root*, only this node (or the root node of a `Document`) and its children will be considered during traversal.

Parameters

- **steps** – The designated transformation steps of the instance are given as a sequence of positional arguments.
- **config** – The configuration values for the instance are passed as keyword arguments. Beside the following keywords, it can be populated with any key-value-pairs that will be available in *inxs.Transformation._available_symbols* during a transformation. The defaults are defined in `config_defaults`.
  - `context` can be provided as mapping with items that are added to the *context* before a (sub-)document is processed.
  - `common_rule_conditions` can be used to define one or more conditions that must match in all rule evaluations. E.g. a transformation could be restricted to nodes with a certain namespace without redundantly defining that per rule. Can be given as a single object (e.g. a string) or as sequence.
  - `copy` is a boolean that defaults to `True` and indicates whether to process on a copy of the document's tree object.
  - `name` can be used to identify a transformation.
  - `result_object` sets the transformation's attribute that is returned as result. Dot-notation lookup (e.g. `context.target`) is implemented. Per default the *transformation root* is returned.
  - `traversal_order` sets the default traversal order for rule evaluations and itself defaults to depth first, left to right, to to bottom. See *Traversal strategies* for possible values.

_available_symbols
This mapping contains items that are used for the dependency injection of handler functions. These names are included:

- All attributes of the transformation's *configuration*, overridden by the following.
- All attributes of the transformation's *context*, overridden by the following.
- `config` - The *configuration* namespace object.
- `context` - The *context* namespace object.
- `node` - The node that matched a *Rule*'s conditions or `None` in case of simple *transformation steps*.
- `previous_result` - The result that was returned by the previously evaluated handler function.
- `root` - The root node of the processed (sub-)document a.k.a. *transformation root*.
- `transformation` - The calling *Transformation* instance.

config_defaults = {'common_rule_conditions': None, 'context': {}, 'copy': True, 'na
The default *configuration* values. Changing members on an instance actually affects the class unless a copy of this mapping as copied and bound as instance attribute.

context
This property can be used to access the *context* while the transformation is processing.

name
The `name` member of the transformation's *configuration*.

root
This property can be used to access the root node of the currently processed (sub-)document.

CHAPTER 6

General Index

CHAPTER 7

Frequently Asked Questions

7.1 Why, oh, why?

TODO

7.2 Am I cognitively fit to use `inxs`?

If you're comfortable with Python and delb, you propably are. Just give it a try to solve a smaller problem. If you don't get a grip on something that may be due to the immature documentation.

If you aren't, you should be willing to get acquainted with both. In this case it is recommended to test your understanding and assumptions without `inxs` as well. bpython and Jupyter are great playgrounds.

7.3 Can I get help?

In case you carefully studied the documentation, just open an issue on the issue tracker. Mind that you can't get supported to solve your actual problem, but rather to understand and use `inxs` as a tool to do so.

7.4 Can I produce HTML output with `inxs`?

One thing you may do is to rather produce XHTML, but that is lacking modern HTML features you may want to use. Here's a trick to produce actual HTML:

- produce an XML tree without namespace declarations using the HTML tag set
- serialize the result into a string
- mangle that through pytidylib

CHAPTER 8

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

8.1 Types of Contributions

8.1.1 Report Bugs

Report bugs at https://github.com/funkyfuture/inxs/issues.

If you are reporting a bug, please include:

- Your Python interpreter and *inxs*' version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to fix it.

8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" is open to whoever wants to implement it.

8.1.4 Write Documentation

inxs could always use more documentation, whether as part of the official *inxs* docs, in docstrings, or even on the web in blog posts, articles, and such.

8.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/funkyfuture/inxs/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

8.2 Get Started!

Ready to contribute? Here's how to set up *inxs* for local development.

1. Install the needed *Prerequisites*
2. Fork the *inxs* repo on GitHub.
3. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/inxs.git
```

4. Install your local copy into a virtualenv. Assuming you have pew installed, this is how you set up your fork for local development:

```
$ cd inxs/
$ pew new -a $(pwd) inxs
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

5. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

6. When you're done making changes, format the code with black and check that your changes pass all QA tests:

```
$ make black
$ tox
```

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated.
3. The pull request should work for Python 3.6. Check https://travis-ci.org/funkyfuture/inxs/pull_requests and make sure that the tests pass for all supported Python versions.

CHAPTER 9

History

9.1 0.2b1 (2019-06-23)

- refactored to base on delb instead of lxml
- ***removed*** **from the available symbols for handler functions:**
  - `tree`
  - `xpath_evaluator` (use `root.xpath` instead)
- ***renamed*** **available symbols for handler functions:**
  - `element` -> `node`
- ***renamed*** **in core:**
  - `SkipToNextElement` -> `SkipToNextNode`
- ***removed*** **from the lib:**
  - `drop_siblings`
  - `extract_text`
  - `has_tail`
  - `init_elementmaker`
  - `merge`
  - `replace_text`
  - `sub`
- ***renamed*** **in the lib:**
  - `make_element` -> `make_node`
  - `remove_element` -> `remove_node`
  - `remove_elements` -> `remove_nodes`

  - `sorter` -> `sort`
  - `strip_attributes` -> `remove_attributes`
  - `strip_namespace` -> `remove_namespace`

Various arguments to functions and methods have been renamed accordingly.

9.2 0.1b1 (2017-06-25)

- *new*: Allows the definition that any rule must match per transformation as `common_rule_conditions`.
- Minor improvements and fixes.

9.3 0.1b0 (2017-06-19)

- First beta release.

9.4 0.1a0 (2017-05-02)

- First release on PyPI.

CHAPTER 10

inxs – A Python framework for XML transformations without boilerplate.

inxs is inexcessive.

inxs is not XSLT.

inxs is ISC-licensed.

inxs is fully documented here: https://inxs.readthedocs.io/en/latest/

10.1 At a glimpse

Solving the Wikipedia XSLT example #1:

```
def extract_person(node: TagNode):
    return node.attributes['username'], first(node.css_select("name")).full_text

def append_person(previous_result, result: TagNode):
    result.append_child(result.new_tag_node(
        "name", attributes={"username": previous_result[0]},
        children=[previous_result[1]]
    ))

transformation = Transformation(
    Rule('person', (extract_person, append_person)),
    result_object='context.result', context={'result': new_tag_node('root')})
```

(continues on next page)

(continued from previous page)

```
# that's four lines less LOC than the XSLT implementation
```

Solving the Wikipedia XSLT example #2:

```
def generate_skeleton(context):
    context.html = new_tag_node(
        "html", namespace='http://www.w3.org/1999/xhtml',
        children=(
            tag("head",
                tag("title", "Testing XML Example")),
            tag("body", (
                tag("h1", "Persons"),
                tag("ul")
            )),
        )
    )

def extract_person(node: TagNode, persons):
    persons.append(
        (first(node.css_select("name")).full_text,
         first(node.css_select("family-name")).full_text)
    )

def list_persons(previous_result, html: TagNode):
    first(html.css_select("html|body html|ul")).append_child(
        *(html.new_tag_node("li", children=[f'{x[1]}, {x[0]}'])
          for x in previous_result)
    )

transformation = Transformation(
    generate_skeleton,
    Rule('person', extract_person),
    lib.sort('persons', itemgetter(1)),
    list_persons,
    result_object='context.html', context={'persons': []})

# that's four lines more LOC than the XSLT implementation
```

Here you can find the source repository and issue tracker of inxs.

Python Module Index

i

Index

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T