Kodexa

Release 2.0.17

May 29, 2020

Contents:

This the documentation for 2.0.17

CHAPTER 1

Kodexa

Kodexa is a core python package for the Kodexa Cloud, and a surrounding set of extensions, that are built to allow developers to work with documents and unstructured data in a way that supports flexibility, extensibility and deployable in a range of platforms.

CHAPTER 2

Mix-Ins

In this section we cover some of the included mix-ins and detail the functions that they add.

2.1 Core

It is the core set of helpers that can be added to content nodes to allow for searching and tagging.

tag(*self*, *tag_name*, *type_re=None*, *content_re=None*, *use_all_content=False*, *node_only=False*, *include_children=False*, *fixed_position=None*, *data=None*)
This will tag (see Feature Tagging) the expression groups identified by the regular expression.

```
>>> document.content_node.find(content_re='.*Cheese.*').tag('is_cheese')
```

Parameters

- **tag_name** – the name of the tag to apply
- **type_re** – regular expression to make the type (default .*)
- **content_re** – the regular expression that you wish to use to tag, note that we will create a tag for each matching group
- **use_all_content** – apply the regular expression to the all_content (include content from child nodes)
- **node_only** – Ignore the matching groups and tag the whole node
- **include_children** – Include recurse into children and tag where matching
- **fixed_position** – use a fixed position, supplied as a tuple i.e. - (4,10) tag from position 4 to 10 (default None)
- **data** – Attach the a dictionary of data for the given tag

tag_range(*self*, *start_content_re*, *end_content_re*, *tag_name*, *type_re='.*'*, *use_all_content=False*)
This will tag all the child nodes between the start and end content regular expressions

```
>>> document.content_node.tag_range(start_content_re='.*Cheese.*', end_content_re=
↪'.*Fish.*', tag_name='foo')
```

Parameters

- **start_content_re** – The regular expression to match the starting child
- **end_content_re** – The regular expression to match the ending child
- **tag_name** – The tag to be applied to the nodes in range
- **type_re** – The type to match (default is all)
- **use_all_content** – Use full content (including child nodes, default is False)

get_all_content(*self*, *separator=' '*)

This will build the complete content, including the content of children.

```
>>> document.content_node.get_all_content()
"This string is made up of multiple nodes"
```

Parameters **separator** – the separate to use to join the content together (default is " ")

get_all_tags(*self*)

Returns a list of the names of the tags on the given node and all its children

```
>>> document.content_node.find(content_re='.*Cheese.*').get_all_tags()
['is_cheese']
```

Returns A list of the tag names

get_tag(*self*, *tag_name*)

Returns the value of a tag, this can be either a single list [start,end,value] or if multiple parts of the content of this node match you can end up with a list of lists i.e. [[start1,end1,value1],[start2,end2,value2]]

```
>>> document.content_node.find(content_re='.*Cheese.*').get_tag('is_cheese')
[0,10,'The Cheese Moved']
```

Parameters **tag_name** – The name of the tag

Returns The tagged location and value (or a list if more than one)

find(*self*, *content_re='.*'*, *type_re='.*'*, *direction=<FindDirection.CHILDREN: 1>*, *tag_name=None*, *instance=0*, *tag_name_re=None*, *use_all_content=False*)

Search for a node that matches on the value and or type using regular expressions

```
>>> document.content_node.find(content_re='.*Cheese.*',instance=2)
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
```

Parameters

- **content_re** – the regular expression to match the nodes content (default '.*')
- **type_re** – the regular expression to match the nodes type (default '.*')
- **direction** – the direction to search, either FindDirection.CHILDREN or FindDirection.PARENT (default CHILDREN)

- **tag_name** – the tag name that must exist
- **instance** – the instance to return (0=first)
- **tag_name_re** – the regular expression to match for the tag_name
- **use_all_content** – match the content for all child nodes to (default False)

Returns either an instance of ContentNode (if found), or None

findall(*self*, *content_re='.*'*, *type_re='.*'*, *direction=<FindDirection.CHILDREN: 1>*, *tag_name=None*, *tag_name_re=None*, *use_all_content=False*)

Search for nodes that matches on the value and or type using regular expressions

```
>>> document.content_node.findall(content_re='.*Cheese.*')
[<kodexa.model.model.ContentNode object at 0x7f80605e53c8>,
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>]
```

Parameters

- **content_re** – the regular expression to match the nodes content (default '.*')
- **type_re** – the regular expression to match the nodes type (default '.*')
- **direction** – the direction to search, either FindDirection.CHILDREN or FindDirection.PARENT (default CHILDREN)
- **tag_name** – the tag name that must exist
- **tag_name_re** – the regular expression to match for the tag_name
- **use_all_content** – match the content for all child nodes to (default False)

Returns list of matching content nodes

find_with_feature_value(*self*, *feature_type*, *feature*, *value*, *direction=<FindDirection.CHILDREN: 1>*, *instance=1*)

Search for a node with a specific feature type, name and value

```
>>> document.content_node.find_with_feature_value(feature_type='tag',feature='is_
↪cheese',value=[1,10,'The Cheese has moved'])
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
```

Parameters

- **feature_type** – the feature type
- **feature** – the feature name
- **value** – the feature value
- **direction** – the direction to search, either FindDirection.CHILDREN or FindDirection.PARENT (default CHILDREN)
- **instance** – the instance to get (defaul 1)

Returns either an instance of ContentNode (if found), or None

findall_with_feature_value(*self*, *feature_type*, *feature_name*, *value*, *direction=<FindDirection.CHILDREN: 1>*)

Search for all nodes with a specific feature type, name and value

```
>>> document.content_node.findall_with_feature_value(feature_type='tag',feature=
↪'is_cheese', value=[1,10,'The Cheese has moved'])
[<kodexa.model.model.ContentNode object at 0x7f80605e53c8>]
```

Parameters

- **feature_type** – the feature type
- **feature_name** – the feature name
- **value** – the feature value
- **direction** – the direction to search, either FindDirection.CHILDREN or FindDirection.PARENT (default CHILDREN)

Returns list of the matching content nodes

get_tags (*self*)

Returns a list of the names of the tags on the given node

```
>>> document.content_node.find(content_re='.*Cheese.*').get_tags()
['is_cheese']
```

Returns A list of the tag name

move_child_to_parent (*self*, *target_child*, *target_parent*)

This will move the target_child, which must be a child of the node, to a new parent.

It will be added to the end of the parent

```
>>> document.content_root.move_child_to_parent(document.content_root.find(type_
↪ref='line'), document.content_root)
```

Parameters

- **target_child** – the child node that needs to be moved
- **target_parent** – the parent to attach this node to

adopt_children (*self*, *children*, *replace=False*)

This will take a list of content nodes and adopt them under this node, ensuring they are re-parented.

It will be added to the end of the parent

```
>>> document.content_root.adopt_children(document.content_root.find(type_ref='line
↪'), replace=True)
```

Parameters

- **children** – a list of the children to adopt
- **replace** – if True the node will remove all current children and replace them with the new list

remove_tag (*self*, *tag_name*)

This will remove a tag from the given node

```
>>> document.content_node.remove_tag(tag_name='foo')
```

Parameters **tag_name** – The tag to be applied to the nodes in range

collect_nodes_to(*self*, *end_node*)

Return a list of the sibling nodes between the current node and the end node

```
>>> document.content_node.children[0].collect_nodes_to(end_node=document.content_
↪node.children[5])
```

Parameters **end_node** – The node to end at

tag_nodes_to(*self*, *end_node*, *tag_name*)

Tag all the nodes from this node to the end node with the given tag name

```
>>> document.content_node.children[0].tag_nodes_to(document.content_node.
↪children[5], tag_name='foo')
```

param end_node The node to end with

param tag_name The tag name

get_node_at_index(*self*, *index*)

Returns the node at a specific index, if the index it outside the first (0), or last index it will return null, if not it will return the node at index, if there isn't a node at that index it will return a 'virtual' node that will represent the node to the side of this node and have an index and no features or content

Parameters **index** – The index (zero-based) for the child node

Returns Node at index, or None is the index is outside the boundaries of child nodes

has_next_node(*self*, *type_re='.*'*, *skip_virtual=False*)

Returns True if the node has a next node

Parameters

- **type_re** – Type name (regular expression)
- **skip_virtual** – True to skip any virtual nodes and only return the next real node (default False)

Returns True if there is a next sibling node

has_previous_node(*self*, *type_re='.*'*, *skip_virtual=False*)

Returns True if the node has a previous node

Parameters

- **type_re** – Type name (regular expression)
- **skip_virtual** – True to skip any virtual nodes and only return the next real node (default False)

return True if there is a previous sibling node

next_node(*self*, *type_re='.*'*, *skip_virtual=False*, *has_no_content=False*, *traverse=<Traverse.SIBLING: 1>*)

Returns the next sibling content node, note that this logic is based on the index, therefore the next node might actually be a virtual node that is created to fill a gap in the document, since the index allows for sparse documents

Parameters

- **type_re** – the regular expression for the type of node (default '.*')
- **skip_virtual** – True to skip any virtual nodes and only return the next real node (default False)
- **has_no_content** – True if you only want to return a node that has no content
- **traverse** – By default we traverse siblings, however you can include CHILDREN, PARENT or ALL

Returns the next node or None if no node exists

previous_node(*self*, *type_re='.*'*, *skip_virtual=False*, *has_no_content=False*, *traverse=<Traverse.SIBLING: 1>*)

Returns the previous sibling content node, note that this logic is based on the index, therefore the next node might actually be a virtual node that is created to fill a gap in the document, since the index allows for sparse documents

Parameters

- **type_re** – the regular expression for the type of node (default '.*')
- **skip_virtual** – True to skip any virtual nodes and only return the next real node (default False)
- **has_no_content** – True if you only want to return a node that no content
- **traverse** – By default we traverse siblings, however you can include CHILDREN, PARENT or ALL

Returns the previous node or None if no node exists

get_last_child_index(*self*)

Returns the max index value for the children of this node, if the node has no children it returns None

Returns Thhe max index of the children of this node, or None is no children

is_first_child(*self*)

Returns True if this is the first child, also True if it has no parent

Returns True if this is the first child

is_last_child(*self*)

Returns True if this is the last child, also True if it has no parent

Returns True if this is the first child

2.2 Spatial

One of the core mix-ins is Spatial. It is based on the concept of holding spatial information about the content nodes.

This spatial information can then be used by the mix-in's methods to allow you to both pull spatial information, but also to query it.

set_statistics(*self*, *statistics*)

Set the spatial statistics for this node

```
>>> document.content_node.find(type_re='page').set_statistics(NodeStatistics())
```

Parameters **statistics** – the statistics object

get_statistics(*self*)
Get the spatial statistics for this node

```
>>> document.content_node.find(type_re='page').get_statistics()
<kodexa.spatial.NodeStatistics object at 0x7f80605e53c8>
```

Returns the statistics object (or None if not set)

set_bbox(*self*, *bbox*)
Set the bounding box for the node, this is structured as:

[x1,y1,x2,y2]

```
>>> document.content_node.find(type_re='page').set_bbox([10,20,50,100])
```

Parameters **bbox** – the bounding box array

get_bbox(*self*)
Get the bounding box for the node, this is structured as:

[x1,y1,x2,y2]

```
>>> document.content_node.find(type_re='page').get_bbox()
[10,20,50,100]
```

Returns the bounding box array

set_rotate(*self*, *rotate*)
Set the rotate of the node

```
>>> document.content_node.find(type_re='page').set_rotate(90)
```

:param rotate the rotation of the node

get_rotate(*self*)
Get the rotate of the node

```
>>> document.content_node.find(type_re='page').get_rotate()
90
```

Returns the rotation of the node

get_x(*self*)
Get the X position of the node

```
>>> document.content_node.find(type_re='page').get_x()
10
```

Returns the X position of the node

get_y(*self*)
Get the Y position of the node

```
>>> document.content_node.find(type_re='page').get_y()
90
```

Returns the Y position of the node

get_width(*self*)
Get the width of the node

```
>>> document.content_node.find(type_re='page').get_width()
70
```

Returns the width of the node

get_height(*self*)
Get the height of the node

```
>>> document.content_node.find(type_re='page').get_height()
40
```

Returns the height of the node

set_bbox_from_children(*self*)
Set the bounding box for this node based on its children

collapse(*self*, *type_re*)
Will collapse the given type, this will remove this type from the hierarchy.

Parameters **type_re** – the type that you will collapse

Returns

CHAPTER 3

API Documentation

Kodexa is a Python framework to enable flexible data engineering with semi-structured and unstructured documents and data.

3.1 Kodexa Cloud

Provides out of the box integration with the Kodexa cloud, enabling the universe of content services that are available

class KodexaCloudPipeline(*slug*, *version=None*, *attach_source=True*, *options=None*, *auth=None*, *cloud_url='https://cloud.kodexa.com'*, *access_token=None*)

Allow you to interact with a pipeline that has been deployed in the Kodexa Cloud

class KodexaCloudService(*slug*, *version=None*, *attach_source=False*, *options={}*, *auth=[]*, *cloud_url='https://cloud.kodexa.com'*, *access_token=None*)

Allows you to interact with a content service that has been deployed in the Kodexa cloud

3.2 Connectors

Connectors provide a way to access document (files or otherwise) from a source, and they form the starting point for Pipelines

class FolderConnector(*path*, *file_filter='*'*)

class UrlConnector(*url*, *headers=None*)

3.3 Core Model

The core model provides the object structure for Documents, ContentNodes and Features which is used as the foundation for working with unstructured data in the framework.

Create a new instance of a Document, you will be required to provide a DocumentMetadata object

```
>>> document = Document(DocumentMetadata())
```

class Document(*metadata=None*, *content_node: kodexa.model.model.ContentNode = None*, *source=<kodexa.model.model.SourceMetadata object>*)

A Document is a collection of metadata and a set of content nodes.

add_mixin(*mixin*)

Add the given mixin to this document, this will apply the mixin to all the content nodes, and also register it with the document so that future invocations of create_node will ensure the node has the mixin appled.

```
>>> document.add_mixin('spatial')
```

create_node(*type: str = <class 'type'>*, *content: str = None*, *virtual: bool = False*, *parent: kodexa.model.model.ContentNode = None*, *index: int = 0*)

Creates a new node for the document, this doesn't add the node to the document however it does ensure that any mixins that have been applied to the document will also be available on the new node

```
>>> document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
```

static from_dict(*doc_dict*)

Build a new document from a dictionary

```
>>> Document.from_dict(doc_dict)
```

static from_json(*json_string*)

From a JSON string create an instance of a Document

```
>>> document.from_json()
```

static from_kdxa(*file_path*)

Read an .kdxa file from the given file_path and

```
>>> document = Document.from_kdxa('my-document.kdxa')
```

Parameters **file_path** – the path to the mdoc file

static from_msgpack(*bytes*)

From a message pack byte array create an instance of a Document

```
>>> document.from_msgpack()
```

get_mixins()

Get the list of mixins that have been enabled on this document.

```
>>> document.get_mixins()
['spatial','finders']
```

get_root()

Get the root content node for the document (same as content_node)

```
>>> node = document.get_node()
```

to_dict()

Convert this document object structure into a simple set of dictionaries

```
>>> document.to_dict()
```

to_json()
Convert this document object structure into a JSON object

```
>>> document.to_json()
```

to_kdxa(*file_path*)
Write the document to the kdxa format (msgpack) which can be used with the Kodexa platform

```
>>> document.to_mdoc('my-document.kdxa')
```

Parameters **file_path** – the path to the mdoc you wish to create

to_msgpack()
Convert this document object structure into a message pack

```
>>> document.to_msgpack()
```

to_text()
Convert this document object structure into a text representation, which can be useful when trying to review the structure.

```
>>> document.to_text()
```

class DocumentMetadata(**args*, ***kwargs*)
A flexible dict based approach to capturing metadata for the document

class ContentNode(*document*, *type*, *content=''*, *content_parts=[]*)
A Content Node identifies a section of the document containing logical grouping of information

The node will have content and can include n number of features.

You should always create a node using the Document create_node method to ensure that the correct mixins are applied.

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> current_content_node.add_child(new_page)
```

or

```
>>> new_page = document.create_node(type='page', content='This is page 1')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> current_content_node.add_child(new_page)
```

add_child(*child*, *index=None*)
Add a ContentNode as a child of this ContentNode

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> current_content_node.add_child(new_page)
```

add_feature(*feature_type*, *name*, *value*, *single=True*, *serialized=False*)
Add a new feature to this ContentNode.

You will need to provide the feature type, the name of the feature and then the value.

Note this will add a value to an existing feature, therefore the feature value might switch to being a list

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> new_page.add_feature('pagination','pageNum',1)
```

get_children()
Returns a list of the children of this node

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> new_page.get_children()
[]
```

get_content()
Returns the content of the node

```
>>> new_page.get_content()
"This is page one"
```

get_feature(*feature_type*, *name*)
Gets the value for the given feature.

You will need to provide the type and name of the feature. If no feature is find you will get None

```
>>> new_page.get_feature('pagination','pageNum')
1
```

get_feature_value(*feature_type*, *name*)
Returns the assigned value for a given feature

You will need to provide the type and name of the feature. If the feature is present it will return the value otherwise it will return None.

```
>>> new_page.get_feature_value('pagination','pageNum')
1
```

get_features()
Returns a list of the features on this content node

Returns a list of the features present

get_features_of_type(*feature_type*)
Return a list of all the features of a specific type

```
>>> new_page.get_features_of_type('tag')
[]
```

Returns list of the tags

get_type()
Returns the type of the node

```
>>> new_page.get_content()
"page"
```

has_feature(*feature_type*, *name*)
Determines if the feature with the given name and type exists on this content node

You will need to provide the type and name of the feature. If the feature is present it will return True, else it will return False

```
>>> new_page.has_feature('pagination','pageNum')
True
```

Returns True if the feature is present

remove_feature (*feature_type*, *name*)
Determines if the feature with the given name and type exists on this content node

You will need to provide the type and name of the feature. If the feature is present it will return True, else it will return False

```
>>> new_page.remove_feature('pagination','pageNum')
```

set_feature (*feature_type*, *name*, *value*)
Sets a feature to this ContentNode, replacing the value

You will need to provide the feature type, the name of the feature and then the value.

Note this will replace any matching feature (i.e. with the same type and name)

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> new_page.add_feature('pagination','pageNum',1)
```

to_dict ()
Convert the ContentNode, and all its children into a simple dictionary

```
>>> new_page = document.create_node(type='page')
<kodexa.model.model.ContentNode object at 0x7f80605e53c8>
>>> current_content_node.to_dict()
```

to_json ()
Convert this node structure into a JSON object

```
>>> node.to_json()
```

to_text ()
Convert this node structure into a text representation, which can be useful when trying to review the structure.

```
>>> node.to_text()
```

3.4 Pipeline

A Pipeline is a way to bring together a Connector, set of steps and then a sink to perform data cleansing, normalization, analysis and more.

class Pipeline (*connector*, *name='Default'*, *stop_on_exception=True*, *logging_level=20*)
A pipeline represents a way to bring together parts of the kodexa framework to solve a specific problem.

When you create a Pipeline you must provide the connector that will be used to source the documents.

```
>>> pipeline = Pipeline(FolderConnector(path='/tmp/', file_filter='example.pdf'))
```

Parameters

- **connector** – the connector that will be the starting point for the pipeline
- **name** – the name of the pipeline (default 'Default')
- **stop_on_exception** – Should the pipeline raise exceptions and stop (default True)
- **logging_level** – The logging level of the pipeline (default INFO)

add_step(*step*)

Add the given step to the current pipeline

```
>>> pipeline = Pipeline(FolderConnector(path='/tmp/', file_filter='example.pdf
↪'))
>>> pipeline.add_step(ExampleStep())
```

Note that it is also possible to add a function as a step, for example

```
>>> def my_function(doc):
>>>       doc.metadata.fishstick = 'foo'
>>>       return doc
>>> pipeline.add_step(my_function)
```

Parameters **step** – the step to add

add_store(*name*, *store*)

Add the store to the pipeline so that it is available to the pipeline

```
>>> pipeline = Pipeline(FolderConnector(path='/tmp/', file_filter='example.pdf
↪'))
>>> pipeline.add_store("test-store", InMemoryObjectStore())
```

Parameters

- **name** – the name of the store (to refer to it)
- **store** – the store that should be added

run()

Run the current pipeline, note that you must have a sink in place to allow the pipeline to run

```
>>> pipeline = Pipeline(FolderConnector(path='/tmp/', file_filter='example.pdf
↪'))
>>> pipeline.set_sink(ExampleSink())
>>> pipeline.run()
```

Returns The context from the run

set_sink(*sink*)

Set the sink you wish to use, note that it will replace any currently assigned sink

```
>>> pipeline = Pipeline(FolderConnector(path='/tmp/', file_filter='example.pdf
↪'))
>>> pipeline.set_sink(ExampleSink())
```

Parameters **sink** – the sink for the pipeline

class PipelineContext(*content_provider=<kodexa.pipeline.pipeline.InMemoryContentProvider object>*, *store_provider=<kodexa.pipeline.pipeline.InMemoryStoreProvider object>*, *existing_content_objects=None*, *context=None*)

Pipeline context is created when you create a pipeline and it provides a way to access information about the pipeline that is running. It can be made available to steps/functions so they can interact with it.

It also provides access to the 'stores' that have been added to the pipeline

add_store(*name*, *store*)

Add a store with given name to the context

Parameters

- **name** – the name to refer to the store with
- **store** – the instance of the store

get_store(*name*, *default=None*)

Get a store with given name from the context

Parameters

- **name** – the name to refer to the store with
- **default** – optionally the default to create the store as if it isn't there

Returns the store, or None is not available

get_store_names()

Return the list of store names in context

Returns the list of store names

set_output_document(*output_document*)

Set the output document from the pipeline

Parameters **output_document** – the final output document from the pipeline

Returns the final output document

class PipelineStatistics

A set of statistics for the processed document

documents_processed document_exceptions

processed_document(*document*)

Update statistics based on this document completing processing

Parameters **document** – the document that has been processed

3.5 Sinks

Sinks are the end-point of a Pipeline and allow for the final output of the pipeline to be either stored or written out

class InMemoryDocumentSink

An in-memory document sink can be used for testing where you want to capture a set of the documents as basic list in-memory and then access them

get_document(*index*)

Get document at given index

Parameters **index** – index to get the document at

sink(*document*)
Adds the document to the sink

Parameters **document** – document to add

3.6 Stores

Stores are persistence components for Documents, typically they can act as either a Connector or a Sink

class JsonDocumentStore(*store_path: str*, *force_initialize: bool = False*)
An implementation of a document store that uses JSON files to store the documents and maintains an index.json containing some basics of the documents

add(*document: kodexa.model.model.Document*)
Add a new document and return the index position

Returns The index of the document added

count()
The number of documents in the store

Returns The number of documents

delete(*idx: int*)
Delete the document at the given index

Returns The Document that was removed

get(*idx: int*)
Load the document at the given index

Returns Document at given index

get_document(*index: int*)
Gets the document from the specific index

Parameters **index** – index of document to get

Returns the document

load(*document_id: str*)
Loads the document with the given document ID

:return the document

read_index()
Method to read the document index from the store path

reset_connector()
Reset the index back to the beginning

save_index()
Method to write the JSON store index back to store path

class TableDataStore(*columns=None*, *rows=None*)
Stores data as a list of lists that can represent a table.

This is a good store when you are capturing nested or tabular data.

Parameters

- **columns** – a list of the column names (default to dynamic)

- **rows** – initial set of rows (default to empty)

add(*row*)
Writes a row to the Data Store

Parameters **row** – the row (as a list) to add

count()
Returns the number of rows in the store

Returns number of rows

class DictDataStore(*dicts=None*)
Stores data as a list of dictionaries that can be any structure

This is a good store when you are capturing nested or semi-structured data

add(*dict*)
Writes a dict to the Data Store

Parameters **dict** – the dict to add to the store

count()
Returns the number of dictionaries in the store

Returns number of dictionaries

class NodeTagger(*type_re*, *content_re*, *tag_name*, *use_all_content=True*, *node_only=False*)
A node tagger allows you to provide a type and content regular expression and then tag content in all matching nodes.

It allows for multiple matching groups to be defined, also the ability to use all content and also just tag the node (ignoring the matching groups)

class TextParser(*decode=False*, *encoding='utf-8'*)
The text parser can load a source file as a text document and creates a single content node with the text

class Rollup(*collapse_type_res=[]*, *reindex=True*)
The rollup step allows you to decide how you want to collapse content in a document by removing nodes while maintaining content and features as needed

class ExtractTagsToKeyValuePair(*store_name*, *include=[]*, *exclude=[]*, *include_node_content=True*)
Extract all the tags from a document into a key/value pair table store

class JsonParser
Parse JSON file into kodexa Document

class ExtractTagsToKeyValuePair(*store_name*, *include=[]*, *exclude=[]*, *include_node_content=True*)
Extract all the tags from a document into a key/value pair table store

class JsonParser
Parse JSON file into kodexa Document

class NodeTagger(*type_re*, *content_re*, *tag_name*, *use_all_content=True*, *node_only=False*)
A node tagger allows you to provide a type and content regular expression and then tag content in all matching nodes.

It allows for multiple matching groups to be defined, also the ability to use all content and also just tag the node (ignoring the matching groups)

class Rollup(*collapse_type_res=[]*, *reindex=True*)
The rollup step allows you to decide how you want to collapse content in a document by removing nodes while maintaining content and features as needed

class TextParser(*decode=False*, *encoding='utf-8'*)

The text parser can load a source file as a text document and creates a single content node with the text

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

k

Index