pydatajson Documentation

Versión 0.4.10

Datos Argentina

25 de abril de 2018

Índice general

Documentación de pydatajson: librería con funcionalidades para gestionar los metadatos de catálogos de datos abiertos que cumplan con el Perfil Nacional de Metadatos. Pydatajson es parte del Paquete de Apertura de Datos.

Podés colaborar cargando un nuevo issue, o respondiendo a un issue ya existente. Lo mismo te invitamos a hacer en el Paquete de Apertura de Datos.

CAPÍTULO 1

Indice

pydatajson

Paquete en python con herramientas para manipular y validar metadatos de catálogos de datos.

- Versión python: 2 y 3
- Licencia: MIT license
- Documentación: https://pydatajson.readthedocs.io/es/stable

Este README cubre los casos de uso más comunes para la librería, junto con ejemplos de código, pero sin mayores explicaciones. Para una versión más detallada de los comportamientos, revise la documentación oficial o el Manual de Uso de la librería.

Instalación

- **Producción:** Desde cualquier parte

```
$ pip install pydatajson
```

- **Desarrollo:** Clonar este repositorio, y desde su raíz, ejecutar:

```
$ pip install -e .
```

A partir de la versión 0.2.x (Febrero 2017), la funcionalidad del paquete se mantendrá fundamentalmente estable hasta futuro aviso. De todas maneras, si piensa utilizar esta librería en producción, le sugerimos fijar la versión que emplea en un archivo `requirements.txt`.

Usos

La librería cuenta con funciones para cuatro objetivos principales:

- **validación de metadatos de catálogos** y los *datasets*,
- **generación de reportes** sobre el contenido y la validez de los metadatos de catálogos y *datasets*,
- **transformación de archivos de metadatos** al formato estándar (JSON), y
- **generación de indicadores de monitoreo de catálogos** y sus *datasets*.

A continuación se proveen ejemplos de cada uno de estas acciones. Si desea analizar un flujo de trabajo más completo, refiérase a los Jupyter Notebook de `samples/`

Setup

`DataJson` utiliza un esquema default que cumple con el perfil de metadatos recomendado en la Guía para el uso y la publicación de metadatos (v0.1) del Paquete de Apertura de Datos.

```
from pydatajson import DataJson

dj = DataJson()
```

Si se desea utilizar un esquema alternativo, por favor, consulte la sección "Uso > Setup" del manual oficial, o la documentación oficial.

Validación de metadatos de catálogos

- Si se desea un **resultado sencillo (V o F)** sobre la validez de la estructura del catálogo, se utilizará **`is_valid_catalog(catalog)`**.
- Si se desea un **mensaje de error detallado**, se utilizará **`validate_catalog(catalog)`**.

Por conveniencia, la carpeta `tests/samples/` contiene varios ejemplos de `data.json` bien y mal formados con distintos tipos de errores.

Archivo data.json local

```
from pydatajson import DataJson

dj = DataJson()
catalog = "tests/samples/full_data.json"
validation_result = dj.is_valid_catalog(catalog)
validation_report = dj.validate_catalog(catalog)

print validation_result
True

print validation_report
{
    "status": "OK",
    "error": {
        "catalog": {
            "status": "OK",
            "errors": [],
            "title": "Datos Argentina"
        },
        "dataset": [
            {
                "status": "OK",
                "errors": [],
                "title": "Sistema de contrataciones electrónicas"
            }
        ]
    }
}
```

Otros formatos

`pydatajson` puede interpretar catálogos tanto en formato JSON como en formato XLSX (siempre y cuando se hayan creado utilizando la plantilla, estén estos almacenados localmente o remotamente a través de URLs de descarga directa. También es capaz de interpretar diccionarios de Python con metadatos de catálogos.

```
from pydatajson import DataJson

dj = DataJson()
catalogs = [
    "tests/samples/full_data.json", # archivo JSON local
    "http://181.209.63.71/data.json", # archivo JSON remoto
    "tests/samples/catalogo_justicia.xlsx", # archivo XLSX local
    "https://raw.githubusercontent.com/datosgobar/pydatajson/master/tests/samples/
↪catalogo_justicia.xlsx", # archivo XLSX remoto
    {
        "title": "Catálogo del Portal Nacional",
    "description" "Datasets abiertos para el ciudadano."
        "dataset": [...],
    (...)
    } # diccionario de Python
]

for catalog in catalogs:
    validation_result = dj.is_valid_catalog(catalog)
```

```
    validation_report = dj.validate_catalog(catalog)
```

Generación de reportes y configuraciones del Harvester

Si ya se sabe que se desean cosechar todos los *datasets* [válidos] de uno o varios catálogos, se pueden utilizar directamente el método `generate_harvester_config()`, proveyendo `harvest='all'` o `harvest='valid'` respectivamente. Si se desea revisar manualmente la lista de *datasets* contenidos, se puede invocar primero `generate_datasets_report()`, editar el reporte generado y luego proveérselo a `generate_harvester_config()`, junto con la opción `harvest='report'`.

Crear un archivo de configuración eligiendo manualmente los datasets a federar

```
catalogs = ["tests/samples/full_data.json", "http://181.209.63.71/data.json"]
report_path = "path/to/report.xlsx"
dj.generate_datasets_report(
    catalogs=catalogs,
    harvest='none', # El reporte tendrá `harvest==0` para todos los datasets
    export_path=report_path
)

# A continuación, se debe editar el archivo de Excel 'path/to/report.xlsx',
# cambiando a '1' el campo 'harvest' en los datasets que se quieran cosechar.

config_path = 'path/to/config.csv'
dj.generate_harvester_config(
    harvest='report',
    report=report_path,
    export_path=config_path
)
```

El archivo `config_path` puede ser provisto a Harvester para federar los *datasets* elegidos al editar el reporte intermedio `report_path`.

Por omisión, en la salida de `generate_harvester_config` la frecuencia de actualización deseada para cada *dataset* será "R/P1D", para intentar cosecharlos diariamente. De preferir otra frecuencia (siempre y cuando sea válida según ISO 8601), se la puede especificar a través del parámetro opcional `frequency`. Si especifica explícitamente `frequency=None`, se conservarán las frecuencias de actualización indicadas en el campo `accrualPeriodicity` de cada *dataset*.

Crear un archivo de configuración que incluya únicamente los datasets con metadata válida

Conservando las variables anteriores:

```
dj.generate_harvester_config(
    catalogs=catalogs,
    harvest='valid'
    export_path='path/to/config.csv'
)
```

Transformación de un archivo de metados XLSX al estándar JSON

```
from pydatajson.readers import read_catalog
from pydatajson.writers import write_json
from pydatajson import DataJson

dj = DataJson()
catalogo_xlsx = "tests/samples/catalogo_justicia.xlsx"

catalogo = read_catalog(catalogo_xlsx)
write_json(obj=catalogo, path="tests/temp/catalogo_justicia.json")
```

Generación de indicadores de monitoreo de catálogos

`pydatajson` puede calcular indicadores sobre uno o más catálogos. Estos indicadores recopilan información de interés sobre los *datasets* de cada uno, tales como:

- el estado de validez de los catálogos;
- el número de días desde su última actualización;
- el formato de sus distribuciones;
- frecuencia de actualización de los *datasets*;
- estado de federación de los *datasets*, comparándolo con el catálogo central

La función usada es `generate_catalogs_indicators`, que acepta los catálogos como parámetros. Devuelve dos valores:

- una lista con tantos valores como catálogos, con cada elemento siendo un diccionario con los indicadores del catálogo respectivo;
- un diccionario con los indicadores de la red entera (una suma de los individuales)

```
catalogs = ["tests/samples/full_data.json", "http://181.209.63.71/data.json"]
indicators, network_indicators = dj.generate_catalogs_indicators(catalogs)

# Opcionalmente podemos pasar como segundo argumento un catálogo central,
# para poder calcular indicadores sobre la federación de los datasets en 'catalogs'

central_catalog = "http://datos.gob.ar/data.json"
indicators, network_indicators = dj.generate_catalogs_indicators(catalogs, central_
↪catalog)
```

Tests

Los tests se corren con `nose`. Desde la raíz del repositorio:

Configuración inicial:

```
$ pip install -r requirements_dev.txt
$ mkdir tests/temp
```

Correr la suite de tests:

```
$ nosetests
```

Recursos de interés

- Estándar ISO 8601 - Wikipedia
- JSON SChema - Sitio oficial del estándar
- Documentación completa de `pydatajson` - Read the Docs
- Guía para el uso y la publicación de metafatos

Créditos

El validador de archivos `data.json` desarrollado es mayormente un envoltorio (*wrapper*) alrededor de la librería `jsonschema`, que implementa el vocabulario definido por JSONSchema.org para anotar y validar archivos JSON.

Manual de uso

Contexto

La política de Datos Abiertos de la República Argentina que nace con el Decreto 117/2016 (*"Plan de Apertura de Datos"*) se basa en un esquema descentralizado donde se conforma una red de nodos publicadores de datos y un nodo central o indexador.

El pilar fundamental de este esquema es el cumplimiento de un Perfil Nacional de Metadatos común a todos los nodos, en el que cada organismo de la APN que publique un archivo `data.json` o formato alternativo compatible.

Esto posibilita que todos los conjuntos de datos (*datasets*) publicados por organismos de la Administración Pública Nacional se puedan encontrar en el Portal Nacional de Datos: http://datos.gob.ar/.

Glosario

Un *catálogo* de datos abiertos está compuesto por *datasets*, que a su vez son cada uno un conjunto de *distribuciones* (archivos descargables). Ver la Guía para el uso y la publicación de metadatos para más información.

- **Catálogo de datos**: Directorio de conjuntos de datos que recopila y organiza metadatos descriptivos de los datos que produce una organización. Un portal de datos es una implementación posible de un catálogo. También lo es un archivo Excel, un JSON u otras.
- **Dataset**: También llamado conjunto de datos. Pieza principal en todo catálogo. Se trata de un activo de datos que agrupa recursos referidos a un mismo tema, que respetan una estructura de la información. Los recursos que lo componen pueden diferir en el formato en que se los presenta (por ejemplo: .csv, .json, .xls, etc.), la fecha a la que se refieren, el área geográfica cubierta o estar separados bajo algún otro criterio.
- **Distribución o recurso**: Es la unidad mínima de un catálogo de datos. Se trata de los activos de datos que se publican allí y que pueden ser descargados y re-utilizados por un usuario como archivos. Los recursos pueden tener diversos formatos (.csv, .shp, etc.). Están acompañados de información contextual asociada ("metadata") que describe el tipo de información que se publica, el proceso por el cual se obtiene, la descripción de los campos del recurso y cualquier información extra que facilite su interpretación, procesamiento y lectura.
- **data.json y catalog.xlsx**: Son las dos *representaciones externas* de los metadatos de un catálogo que `pydatajson` comprende. Para poder ser analizados programáticamente, los metadatos de un catálogo deben estar representados en un formato estandarizado: el PAD establece el archivo `data.json` para tal fin, y `pydatajson` permite leer una versión en XLSX equivalente.
- **diccionario de metadatos**: Es la *representación interna* que la librería tiene de los metadatos de un catálogo. Todas las rutinas de la librería `pydatajson` que manipulan catálogos, toman como entrada una *representación externa* (`data.json` o `catalog.xlsx`) del catálogo, y lo primero que hacen es "leerla" y generar una *representación interna* de la información que la rutina sea capaz de manipular.

Uso

Setup

`DataJson` valida catálogos contra un esquema default que cumple con el perfil de metadatos recomendado en la Guía para el uso y la publicación de metadatos del Paquete de Apertura de Datos.

```
from pydatajson import DataJson

catalog = DataJson("http://datos.gob.ar/data.json")
```

Si se desea utilizar un esquema alternativo, se debe especificar un **directorio absoluto** donde se almacenan los esquemas (`schema_dir`) y un nombre de esquema de validación (`schema_filename`), relativo al di-

rectorio de los esquemas. Por ejemplo, si nuestro esquema alternativo se encuentra en `/home/datosgobar/metadatos-portal/esquema_de_validacion.json`, especificaremos:

```
from pydatajson import DataJson

catalog = DataJson("http://datos.gob.ar/data.json",
                   schema_filename="esquema_de_validacion.json",
                   schema_dir="/home/datosgobar/metadatos-portal")
```

Lectura

`pydatajson` puede leer un catálogo en JSON, XLSX, CKAN o `dict` de python:

```
from pydatajson.ckan_reader import read_ckan_catalog
import requests

# data.json
catalog = DataJson("http://datos.gob.ar/data.json")
catalog = DataJson("local/path/data.json")

# catalog.xlsx
catalog = DataJson("http://datos.gob.ar/catalog.xlsx")
catalog = DataJson("local/path/catalog.xlsx")

# CKAN
catalog = DataJson(read_ckan_catalog("http://datos.gob.ar"))

# diccionario de python
catalog_dict = requests.get("http://datos.gob.ar/data.json").json()
catalog = DataJson(catalog_dict)
```

Escritura

Validación

Los métodos de validación de catálogos procesan un catálogo por llamada. En el siguiente ejemplo, `catalogs` contiene las cinco representaciones de un catálogo que DataJson entiende:

```
from pydatajson import DataJson

catalog = DataJson()
catalogs = [
    "tests/samples/full_data.json", # archivo JSON local
    "http://181.209.63.71/data.json", # archivo JSON remoto
    "tests/samples/catalogo_justicia.xlsx", # archivo XLSX local
    "https://raw.githubusercontent.com/datosgobar/pydatajson/master/tests/samples/
↪catalogo_justicia.xlsx", # archivo XLSX remoto
    {
        "title": "Catálogo del Portal Nacional",
  "description" "Datasets abiertos para el ciudadano."
        "dataset": [...],
  (...)
    } # diccionario de Python
]
```

```
for catalog in catalogs:
    validation_result = catalog.is_valid_catalog(catalog)
    validation_report = catalog.validate_catalog(catalog)
```

Un ejemplo del resultado completo de `validate_catalog()` se puede consultar en el **Anexo I: Estructura de respuestas**.

Federación y restauración

`pydatajson` permite federar o restaurar fácilmente un dataset de un catálogo hacia un Portal Andino (usa todo el perfil de metadatos) o CKAN (sólo usa campos de metadatos de CKAN), utilizando la API de CKAN.

Federar un dataset

Incluye la transformación de algunos metadatos, para adaptar un dataset de un nodo original a cómo debe documentarse en un nodo indexador.

```
catalog_origin = DataJson("https://datos.agroindustria.gob.ar/data.json")

catalog_origin.harvest_dataset_to_ckan(
    owner_org="ministerio-de-agroindustria",
    dataset_origin_identifier="8109e9e8-f8e9-41d1-978a-d20fcd2fe5f5",
    portal_url="http://datos.gob.ar",
    apikey="apikey",
    catalog_id="agroindustria"
)
```

La organización del nodo de destino debe estar previamente creada.

Restaurar un dataset

Los metadatos no sufren transformaciones: se escribe el dataset en el nodo de destino tal cual está en el nodo original.

```
catalog_origin = DataJson("datosgobar/backup/2018-01-01/data.json")

catalog_origin.restore_dataset_to_ckan(
    owner_org="ministerio-de-agroindustria",
    dataset_origin_identifier="8109e9e8-f8e9-41d1-978a-d20fcd2fe5f5",
    portal_url="http://datos.gob.ar",
    apikey="apikey"
)
```

La organización del nodo de destino debe estar previamente creada. En este caso no hace falta `catalog_id` porque el `dataset_identifier` no sufre ninguna transformación.

Transformación de `catalog.xlsx` a `data.json`

La lectura de un archivo de metadatos por parte de `pydatajson.readers.read_catalog` **no realiza ningún tipo de verificación sobre la validez de los metadatos leídos**. Por ende, si se quiere generar un archivo en formato JSON estándar únicamente en caso de que los metadatos de archivo XLSX sean válidos, se deberá realizar la validación por separado.

El siguiente código, por ejemplo, escribe a disco un catálogos de metadatos en formato JSONO sí y sólo sí los metadatos del XLSX leído son válidos:

```
from pydatajson.readers import read_catalog
from pydatajson.writers import write_json
from pydatajson import DataJson

catalog = DataJson()
catalogo_xlsx = "tests/samples/catalogo_justicia.xlsx"

catalogo = read_catalog(catalogo_xlsx)
if catalog.is_valid_catalog(catalogo):
    write_json(obj=catalogo, path="tests/temp/catalogo_justicia.json")
else:
    print "Se encontraron metadatos inválidos. Operación de escritura cancelada."
```

Para más información y una versión más detallada de esta rutina en Jupyter Notebook, dirigirse aquí (metadatos válidos) y aquí (metadatos inválidos).

Generación de reportes

El objetivo final de los métodos `generate_datasets_report`, `generate_harvester_config` y `generate_harvestable_catalogs`, es proveer la configuración que Harvester necesita para cosechar datasets. Todos ellos devuelven una "tabla", que consiste en una lista de diccionarios que comparten las mismas claves (consultar ejemplos en el **Anexo I: Estructura de respuestas**). A continuación, se proveen algunos ejemplos de uso comunes:

Crear un archivo de configuración eligiendo manualmente los datasets a federar

```
catalogs = ["tests/samples/full_data.json", "http://181.209.63.71/data.json"]
report_path = "path/to/report.xlsx"
catalog.generate_datasets_report(
    catalogs=catalogs,
    harvest='none', # El reporte generado tendrá `harvest==0` para todos los datasets
    export_path=report_path
)
# A continuación, se debe editar el archivo de Excel 'path/to/report.xlsx', cambiando␣
↪a '1' el campo 'harvest' para aquellos datasets que se quieran cosechar.

config_path = 'path/to/config.csv'
catalog.generate_harvester_config(
    harvest='report',
    report=report_path,
    export_path=config_path
)
```

El archivo `config_path` puede ser provisto a Harvester para federar los datasets elegidos al editar el reporte intermedio `report_path`.

Alternativamente, el output de `generate_datasets_report()` se puede editar en un intérprete de python:

```
# Asigno el resultado a una variable en lugar de exportarlo
datasets_report = catalog.generate_datasets_report(
    catalogs=catalogs,
    harvest='none', # El reporte generado tendrá `harvest==0` para todos los datasets
```

```
)
# Imaginemos que sólo se desea federar el primer dataset del reporte:
datasets_report[0]["harvest"] = 1

config_path = 'path/to/config.csv'
catalog.generate_harvester_config(
    harvest='report',
    report=datasets_report,
    export_path=config_path
)
```

Crear un archivo de configuración que incluya únicamente los datasets con metadata válida

Conservando las variables anteriores:

```
catalog.generate_harvester_config(
    catalogs=catalogs,
    harvest='valid'
    export_path='path/to/config.csv'
)
```

Para fines ilustrativos, se incluye el siguiente bloque de código que produce los mismos resultados, pero genera el reporte intermedio sobre datasets:

```
datasets_report = catalog.generate_datasets_report(
    catalogs=catalogs,
    harvest='valid'
)

# Como el reporte ya contiene la información necesaria sobre los datasets que se␣
↪pretende cosechar, el argumento `catalogs` es innecesario.
catalog.generate_harvester_config(
    harvest='report'
    report=datasets_report
    export_path='path/to/config.csv'
)
```

Modificar catálogos para conservar únicamente los datasets válidos

```
# Creamos un directorio donde guardar los catálogos
output_dir = "catalogos_limpios"
import os; os.mkdir(output_dir)

catalog.generate_harvestable_catalogs(
    catalogs,
    harvest='valid',
    export_path=output_dir
)
```

Funcionalidades

La librería cuenta con funciones para tres objetivos principales:

- **validación de metadatos de catálogos** y los datasets,
- **generación de reportes** sobre el contenido y la validez de los metadatos de catálogos y datasets,
- **transformación de archivos de metadatos** al formato estándar (JSON),
- **federación de datasets** a portales de destino.

Como se menciona en el Glosario estos métodos no tienen acceso *directo* a ningún catálogo, dataset ni distribución, sino únicamente a sus *representaciones externas*: archivos o partes de archivos en formato JSON que describen ciertas propiedades. Por conveniencia, en este documento se usan frases como "validar el dataset X", cuando una versión más precisa sería "validar la fracción del archivo `data.json` que consiste en una representación del dataset X en forma de diccionario". La diferencia es sutil, pero conviene mantenerla presente.

Todos los métodos públicos de la librería toman como primer parámetro `catalog`:

- o bien un diccionario de metadatos (una *representación interna*),
- o la ruta (local o remota) a un archivo de metadatos en formato legible (idealmente JSON, alternativamente XLSX).

Cuando el parámetro esperado es `catalogs`, en plural, se le puede pasar o un único catálogo, o una lista de ellos.

Todos los métodos comienzan por convertir `catalog(s)` en una **representación interna** unívoca: un diccionario cuyas claves son las definidas en el Perfil de Metadatos. La conversión se realiza a través de `pydatajson.readers.read_catalog(catalog)`: éste es la función que todos ellos invocan para obtener un diccionario de metadatos estándar.

Métodos de validación de metadatos

- **pydatajson.DataJson.is_valid_catalog(catalog) -> bool**: Responde `True` únicamente si el catálogo no contiene ningún error.
- **pydatajson.DataJson.validate_catalog(catalog) -> dict**: Responde un diccionario con información detallada sobre la validez "global" de los metadatos, junto con detalles sobre la validez de los metadatos a nivel catálogo y cada uno de sus datasets. De haberlos, incluye una lista con información sobre los errores encontrados.

Métodos de transformación de formatos de metadatos

Transformar un archivo de metadatos de un formato a otro implica un primer paso de lectura de un formato, y un segundo paso de escritura a un formato distinto. Para respetar las disposiciones del PAD, sólo se pueden escribir catálogos en formato JSON.

- **pydatajson.readers.read_catalog()**: Método que todas las funciones de DataJson llaman en primer lugar para interpretar cualquier tipo de representación externa de un catálogo.
- **pydatajson.writers.write_json_catalog()**: Fina capa de abstracción sobre `pydatajson.writers.write_json`, que simplemente vuelca un objeto de Python a un archivo en formato JSON.

Métodos de generación de reportes

Para federación de datasets

Los siguientes métodos toman una o varias representaciones externas de catálogos, y las procesan para generar reportes específicos sobre su contenido:

- **pydatajson.DataJson.generate_datasets_report()**: Devuelve un reporte con información clave sobre cada dataset incluido en un catálogo, junto con variables indicando la validez de sus metadatos.

- **pydatajson.DataJson.generate_harvester_config()**: Devuelve un reporte con los campos mínimos que requiere el Harvester para federar un conjunto de datasets.
- **pydatajson.DataJson.generate_harvestable_catalogs()**: Devuelve la lista de catálogos ingresada, filtrada de forma que cada uno incluya únicamente los datasets que se pretende que el Harvester federe.

Los tres métodos toman los mismos cuatro parámetros, que se interpretan de manera muy similar:

- **catalogs**: Representación externa de un catálogo, o una lista compuesta por varias de ellas.
- **harvest**: Criterio de decisión utilizado para marcar los datasets a ser federados/cosechados. Acepta los siguientes valores:
  - `'all'`: Cosechar todos los datasets presentes en **catalogs**.
  - `'none'`: No cosechar ninguno de los datasets presentes en **catalogs**.
  - `'valid'`: Cosechar únicamente los datasets que no contengan errores, ni en su propia metadata ni en la metadata global del catálogo.
  - `'report'`: Cosechar únicamente los datasets indicados por el reporte provisto en `report`.
- **report**: En caso de que se pretenda cosechar un conjunto específico de catálogos, esta variable debe recibir la representación externa (path a un archivo) o interna (lista de diccionarios) de un reporte que identifique los datasets a cosechar.
- **export_path**: Esta variable controla el valor de retorno de los métodos de generación. Si es `None`, el método devolverá la representación interna del reporte generado. Si especifica el path a un archivo, el método devolverá `None`, pero escribirá a `export_path` la representación externa del reporte generado, en formato CSV o XLSX.

generate_harvester_config() puede tomar un parámetro extra, `frequency`, que permitirá indicarle a la rutina de cosecha de con qué frecuencia debe intentar actualizar su versión de cierto dataset. Por omisión, lo hará diariamente.

Para presentación de catálogos y datasets

Existen dos métodos, cuyos reportes se incluyen diariamente entre los archivos que disponibiliza el repositorio `libreria-catalogos`:

- **pydatajson.DataJson.generate_datasets_summary()**: Devuelve un informe tabular (en formato CSV o XLSX) sobre los datasets de un catálogo, detallando cuántas distribuciones tiene y el estado de sus propios metadatos.
- **pydatajson.DataJson.generate_catalog_readme()**: Genera un archivo de texto plano en formato Markdown para ser utilizado como "README", es decir, como texto introductorio al contenido del catálogo.

Métodos para federación de datasets

- **pydatajson.DataJson.push_dataset_to_ckan()**: Copia la metadata de un dataset y la escribe en un portal de CKAN. Toma los siguientes parámetros:
  - **owner_org**: La organización a la que pertence el dataset. Debe encontrarse en el portal de destino.
  - **dataset_origin_identifier**: Identificador del dataset en el catálogo de origen.
  - **portal_url**: URL del portal de CKAN de destino.
  - **apikey**: La apikey de un usuario del portal de destino con los permisos para crear el dataset bajo la organización pasada como parámetro.
  - **catalog_id** (opcional, default: None): El prefijo que va a preceder el id y name del dataset en el portal destino, separado por un guión.

- **demote_superThemes** (opcional, default: True):Si está en true, los ids de los themes del dataset, se escriben como groups de CKAN.
- **demote_themes** (opcional, default: True): Si está en true, los labels de los themes del dataset, se escriben como tags de CKAN; sino,se pasan como grupo.

Retorna el id en el nodo de destino del dataset federado.

Advertencia: La función `push_dataset_to_ckan()` sólo garantiza consistencia con los estándares de CKAN. Para mantener una consistencia más estricta dentro del catálogo a federar, es necesario validar los datos antes de pasarlos a la función.

- **pydatajson.federation.remove_dataset_from_ckan()**: Hace un borrado físico de un dataset en un portal de CKAN. Toma los siguientes parámetros:
  - **portal_url**: La URL del portal CKAN. Debe implementar la funcionalidad de `/data.json`.
  - **apikey**: La apikey de un usuario con los permisos que le permitan borrar el dataset.
  - **filter_in**: Define el diccionario de filtro en el campo `dataset`. El filtro acepta los datasets cuyos campos coincidan con todos los del diccionario `filter_in['dataset']`.
  - **filter_out**: Define el diccionario de filtro en el campo `dataset`. El filtro acepta los datasets cuyos campos coincidan con alguno de los del diccionario `filter_out['dataset']`.
  - **only_time_series**: Borrar los datasets que tengan recursos con series de tiempo.
  - **organization**: Borrar los datasets que pertenezcan a cierta organizacion.

  En caso de pasar más de un parámetro opcional, la función `remove_dataset_from_ckan()` borra aquellos datasets que cumplan con todas las condiciones.
- **pydatajson.DataJson.push_theme_to_ckan()**: Crea un tema en el portal de destino Toma los siguientes parámetros:
  - **portal_url**: La URL del portal CKAN.
  - **apikey**: La apikey de un usuario con los permisos que le permitan crear un grupo.
  - **identifier** (opcional, default: None): Id del `theme` que se quiere federar, en el catálogo de origen.
  - **label** (opcional, default: None): label del `theme` que se quiere federar, en el catálogo de origen.

  Debe pasarse por lo menos uno de los 2 parámetros opcionales. En caso de que se provean los 2, se prioriza el identifier sobre el label.
- **pydatajson.DataJson.push_new_themes()**: Toma los temas de la taxonomía de un DataJson y los crea en el catálogo de destino si no existen. Toma los siguientes parámetros:
  - **portal_url**: La URL del portal CKAN adonde se escribiran los temas.
  - **apikey**: La apikey de un usuario con los permisos que le permitan crear los grupos.

Hay también funciones que facilitan el uso de `push_dataset_to_ckan()`:

- **pydatajson.DataJson.harvest_dataset_to_ckan()**: Federa la metadata de un dataset en un portal de CKAN. Toma los siguientes parámetros:
  - **owner_org**: La organización a la que pertence el dataset. Debe encontrarse en el portal de destino.
  - **dataset_origin_identifier**: Identificador del dataset en el catálogo de origen.
  - **portal_url**: URL del portal de CKAN de destino.
  - **apikey**: La apikey de un usuario del portal de destino con los permisos para crear el dataset bajo la organización pasada como parámetro.

- **catalog_id**: El prefijo que va a preceder el id y name del dataset en el portal destino, separado por un guión.

Retorna el id en el nodo de destino del dataset federado.

- **pydatajson.DataJson.restore_dataset_to_ckan()**: Restaura la metadata de un dataset en un portal de CKAN. Toma los siguientes parámetros:
  - **owner_org**: La organización a la que pertence el dataset. Debe encontrarse en el portal de destino.
  - **dataset_origin_identifier**: Identificador del dataset en el catálogo de origen.
  - **portal_url**: URL del portal de CKAN de destino.
  - **apikey**: La apikey de un usuario del portal de destino con los permisos para crear el dataset bajo la organización pasada como parámetro.

Retorna el id del dataset restaurado.

- **pydatajson.DataJson.harvest_catalog_to_ckan()**: Federa los datasets de un catálogo al portal pasado por parámetro. Toma los siguientes parámetros:
  - **dataset_origin_identifier**: Identificador del dataset en el catálogo de origen.
  - **portal_url**: URL del portal de CKAN de destino.
  - **apikey**: La apikey de un usuario del portal de destino con los permisos para crear el dataset.
  - **catalog_id**: El prefijo que va a preceder el id y name del dataset en el portal destino, separado por un guión.
  - **dataset_list** (opcional, default: None): Lista de ids de los datasets a federar. Si no se pasa, se federan todos los datasets del catálogo.
  - **owner_org** (opcional, default: None): La organización a la que pertence el dataset. Debe encontrarse en el portal de destino. Si no se pasa, se toma como organización el catalog_id

  Retorna el id en el nodo de destino de los datasets federados.

Anexo I: Estructura de respuestas

validate_catalog()

El resultado de la validación completa de un catálogo, es un diccionario con la siguiente estructura:

```
{
    "status": "OK",  # resultado de la validación global
    "error": {
    "catalog": {
            # validez de la metadata propia del catálogo, ignorando los
            # datasets particulares
        "status": "OK",
        "errors": []
        "title": "Título Catalog"},
    "dataset": [
        {
        # Validez de la metadata propia de cada dataset
                "status": "OK",
        "errors": [],
        "title": "Titulo Dataset 1"
        },
        {
```

```
        "status": "ERROR",
        "errors": [
                    {
                        "error_code": 2,
                        "instance": "",
                        "message": "'' is not a 'email'",
                        "path": ["publisher", "mbox"],
                        "validator": "format",
                        "validator_value": "email"
                    },
                    {
                        "error_code": 2,
                        "instance": "",
                        "message": """ is too short",
                        "path": ["publisher", "name"],
                        "validator": "minLength",
                        "validator_value": 1
                    }
                ],
                "title": "Titulo Dataset 2"
        }
    ]
    }
}
```

Si `validate_catalog()` encuentra algún error, éste se reportará en la lista `errors` del nivel correspondiente, a través de un diccionario con las siguientes claves:

- **path**: Posición en el diccionario de metadata del catálogo donde se encontró el error.
- **instance**: Valor concreto que no pasó la validación. Es el valor de la clave `path` en la metadata del catálogo.
- **message**: Descripción humanamente legible explicando el error.
- **validator**: Nombre del validador violado, ("type" para errores de tipo, "minLength" para errores de cadenas vacías, et cétera).
- **validator_value**: Valor esperado por el validador `validator`, que no fue respetado.
- **error_code**: Código describiendo genéricamente el error. Puede ser:
  - **1**: Valor obligatorio faltante: Un campo obligatorio no se encuentra presente.
  - **2**: Error de tipo y formato: se esperaba un `array` y se encontró un `dict`, se esperaba un `string` en formato `email` y se encontró una `string` que no cumple con el formato, et cétera.

generate_datasets_report()

El reporte resultante tendrá tantas filas como datasets contenga el conjunto de catálogos ingresado, y contará con los siguientes campos, casi todos autodescriptivos:

- **catalog_metadata_url**: En caso de que se haya provisto una representación externa de un catálogo, la string de su ubicación; sino `None`.
- **catalog_title**
- **catalog_description**
- **valid_catalog_metadata**: Validez de la metadata "global" del catálogo, es decir, ignorando la metadata de datasets particulares.

- **dataset_title**
- **dataset_description**
- **dataset_index**: Posición (comenzando desde cero) en la que aparece el dataset en cuestión en lista del campo `catalog["dataset"]`.
- **valid_dataset_metadata**: Validez de la metadata *específica a este dataset* que figura en el catálogo (`catalog["dataset"][dataset_index]`).
- **harvest**: '0' o '1', según se desee excluir o incluir, respectivamente, un dataset de cierto proceso de cosecha. El default es '0', pero se puede controlar a través del parámetro 'harvest'.
- **dataset_accrualPeriodicity**
- **dataset_publisher_name**
- **dataset_superTheme**: Lista los valores que aparecen en el campo dataset["superTheme"], separados por comas.
- **dataset_theme**: Lista los valores que aparecen en el campo dataset["theme"], separados por comas.
- **dataset_landingPage**
- **distributions_list**: Lista los títulos y direcciones de descarga de todas las distribuciones incluidas en un dataset, separadas por "newline".

La *representación interna* de este reporte es una lista compuesta en su totalidad de diccionarios con las claves mencionadas. La *representación externa* de este reporte, es un archivo con información tabular, en formato CSV o XLSX. A continuación, un ejemplo de la *lista de diccionarios* que devuelve `generate_datasets_report()`:

```
[
    {
        "catalog_metadata_url": "http://181.209.63.71/data.json",
        "catalog_title": "Andino",
        "catalog_description": "Portal Andino Demo",
        "valid_catalog_metadata": 0,
        "dataset_title": "Dataset Demo",
        "dataset_description": "Este es un dataset de ejemplo, se incluye como␣
↪material DEMO y no contiene ningun valor estadistico.",
        "dataset_index": 0,
        "valid_dataset_metadata": 1,
        "harvest": 0,
        "dataset_accrualPeriodicity": "eventual",
        "dataset_publisher_name": "Andino",
        "dataset_superThem"": "TECH",
        "dataset_theme": "Tema.demo",
        "dataset_landingPage": "https://github.com/datosgobar/portal-andino",
        "distributions_list": ""Recurso de Ejemplo": http://181.209.63.71/dataset/
↪6897d435-8084-4685-b8ce-304b190755e4/resource/6145bf1c-a2fb-4bb5-b090-bb25f8419198/
↪download/estructura-organica-3.csv"
    },
    {
        "catalog_metadata_url": "http://datos.gob.ar/data.json",
        "catalog_title": "Portal Nacional de Datos Abiertos",
        ( ... )
    }
]
```

generate_harvester_config()

Este reporte se puede generar a partir de un conjunto de catálogos, o a partir del resultado de `generate_datasets_report()`, pues no es más que un subconjunto del mismo. Incluye únicamente las claves necesarias para que el Harvester pueda federar un dataset, si `'harvest'==1`:

- **catalog_metadata_url**
- **dataset_title**
- **dataset_accrualPeriodicity**

La *representación interna* de este reporte es una lista compuesta en su totalidad de diccionarios con las claves mencionadas. La *representación externa* de este reporte, es un archivo con información tabular, en formato CSV o XLSX. A continuación, un ejemplo con la *lista de diccionarios* que devuelve `generate_harvester_config()`:

```
[
    {
        "catalog_metadata_url": "tests/samples/full_data.json",
        "dataset_title": "Sistema de contrataciones electrónicas",
        "dataset_accrualPeriodicity": "R/P1Y"
    },
    {
        "catalog_metadata_url": "tests/samples/several_datasets_for_harvest.json",
        "dataset_title": "Sistema de Alumbrado Público CABA",
        "dataset_accrualPeriodicity": "R/P1Y"
    },
    {
        "catalog_metadata_url": "tests/samples/several_datasets_for_harvest.json",
        "dataset_title": "Listado de Presidentes Argentinos",
        "dataset_accrualPeriodicity": "R/P1Y"
    }
]
```

generate_datasets_summary()

Se genera a partir de un único catálogo, y contiene, para cada uno de dus datasets:

- **Índice**: El índice, identificador posicional del dataset dentro de la lista `catalog["dataset"]`.
- **Título**: dataset[“title”], si lo tiene (es un campo obligatorio).
- **Identificador**: dataset[“identifier”], si lo tiene (es un campo recomendado).
- **Cantidad de Errores**: Cuántos errores de validación contiene el dataset, según figure en el detalle de `validate_catalog`
- **Cantidad de Distribuiones**: El largo de la lista `dataset["distribution"]`

A continuación, un fragmento del resultado de este método al aplicarlo sobre el Catálogo del Ministerio de Justicia:

```
[OrderedDict([(u'indice', 0),
              (u'titulo', u'Base de datos legislativos Infoleg'),
              (u'identificador', u'd9a963ea-8b1d-4ca3-9dd9-07a4773e8c23'),
              (u'estado_metadatos', u'OK'),
              (u'cant_errores', 0),
              (u'cant_distribuciones', 3)]),
 OrderedDict([(u'indice', 1),
              (u'titulo', u'Centros de Acceso a la Justicia -CAJ-'),
              (u'identificador', u'9775fcdf-99b9-47f6-87ae-6d46cfd15b40'),
```

```
              (u'estado_metadatos', u'OK'),
              (u'cant_errores', 0),
              (u'cant_distribuciones', 1)]),
 OrderedDict([(u'indice', 2),
              (u'titulo',
               u'Sistema de Consulta Nacional de Rebeld\xedas y Capturas - Co.Na.R.C.
↪'),
              (u'identificador', u'e042c362-ff39-476f-9328-056a9de753f0'),
              (u'estado_metadatos', u'OK'),
              (u'cant_errores', 0),
              (u'cant_distribuciones', 1)]),

( ... 13 datasets más ...)

 OrderedDict([(u'indice', 15),
              (u'titulo',
               u'Registro, Sistematizaci\xf3n y Seguimiento de Hechos de Violencia␣
↪Institucional'),
              (u'identificador', u'c64b3899-65df-4024-afe8-bdf971f30dd8'),
              (u'estado_metadatos', u'OK'),
              (u'cant_errores', 0),
              (u'cant_distribuciones', 1)])]
```

generate_catalog_readme()

Este reporte en texto plano se pretende como primera introducción somera al contenido de un catálogo, como figurarán en la Librería de Catálogos. Incluye datos clave sobre el editor responsable del catálogo, junto con:

- estado de los metadatos a nivel catálogo,
- estado global de los metadatos, y
- cantidad de datasets y distribuciones incluidas.

A continuación, el resultado de este método al aplicarlo sobre el Catálogo del Ministerio de Justicia:

```
# Catálogo: Datos Justicia Argentina

## Información General

- **Autor**: Ministerio de Justicia y Derechos Humanos
- **Correo Electrónico**: justiciaabierta@jus.gov.ar
- **Nombre del catálogo**: Datos Justicia Argentina
- **Descripción**:

> Portal de Datos de Justicia de la República Argentina. El Portal publica datos del␣
↪sistema de justicia de modo que pueda ser reutilizada para efectuar visualizaciones␣
↪o desarrollo de aplicaciones. Esta herramienta se propone como un punto de␣
↪encuentro entre las organizaciones de justicia y la ciudadanía.

## Estado de los metadatos y cantidad de recursos

Estado metadatos globales | Estado metadatos catálogo | # de Datasets | # de␣
↪Distribuciones
----------------|--------------------------|---------------|--------------------
OK | OK | 16 | 56

## Datasets incluidos
```

```
Por favor, consulte el informe [`datasets.csv`](datasets.csv).
```

CAPÍTULO 2

Documentación automática

- modindex
- genindex

CAPÍTULO 3

Versiones

Versiones

0.4.4 (2018-04-09)

- Agrega wrappers para `push_dataset_to_ckan()`

0.4.3 (2018-03-20)

- Mejora el manejo de themes para recrear un catálogo

0.4.2 (2018-03-13)

- Agrega funciones auxiliares para la administración de un CKAN vía API para facilitar la administración de la federación de datasets
  - `remove_dataset_to_ckan()`
- Incorpora nuevas validaciones (formatos y fileNames)
- Agrega flags opcionales para que `push_dataset_to_ckan()` sea un método que transforma opcionalmente la metadata de un dataset

0.4.1 (2018-02-16)

- `datasets_equal()` permite especificar los campos a tener en cuenta para la comparación, como un parámetro.

0.4.0 (2018-02-08)

- Incorpora métodos para federar un dataset de un catálogo a un CKAN o un Andino: `push_dataset_to_ckan()`.
- Actualiza validaciones y esquema de metadatos al Perfil Nacional de Metadatos versión 1.1.

0.3.21 (2017-12-22)

- Agrega soporte para Python 3.6

0.3.20 (2017-11-16)

- Agrego método `get_theme()` para devolver un tema de la taxonomía específica del catálogo según su `id` o `label`.

0.3.19 (2017-10-31)

- Agrego métodos de búsqueda de series de tiempo en un catálogo (`get_time_series()`) y un parámetro `only_time_series=True or False` para filtrar datasets y distribuciones en sus métodos de búsqueda (`get_datasets(only_time_series=True)` devuelve sólo aquellos datasets que tengan alguna serie de tiempo).

0.3.18 (2017-10-19)

- Agrego posibilidad de pasar un logger desde afuera a la función de lectura de catálogos en Excel.

0.3.15 (2017-10-09)

- Agrega filtro por series de tiempo en `get_datasets()` y `get_distributions()`. Tienen un parámetro `only_time_series` que devuelve sólo aquellos que tengan o sean distribuciones con series de tiempo.

0.3.12 (2017-09-21)

- Agrega función para escribir un catálogo en Excel.
- Agrega funciones para remover datasets o distribuciones de un catálogo.

0.3.11 (2017-09-13)

- Incorpora parámetro para excluir campos de metadatos en la devolución de la búsqueda de datasets y distribuciones.

0.3.10 (2017-09-11)

- Agregar referencia interna a los ids de las entidades padre de otras (distribuciones y fields.)

0.3.9 (2017-09-05)

- Flexibiliza lectura de extras en ckan to datajson.
- Flexibiliza longitud mínima de campos para recomendar su federación o no.
- Agrega método para devolver los metadatos a nivel de catálogo.
- Resuelve la escritura de objetos python como texto en excel.

0.3.8 (2017-08-25)

- Agrega stop words a `helpers.title_to_name()`

0.3.4 (2017-08-21)

- Agrega método para buscar la localización de un `field` en un catálogo.

0.3.3 (2017-08-20)

- Agrega método para convertir el título de un dataset o distribución en un nombre normalizado para la creación de URLs.

0.3.2 (2017-08-16)

- Amplía reporte de federación en markdown.

0.3.0 (2017-08-14)

- Agrega métodos para navegar un catálogo desde el objeto DataJson.

0.2.27 (2017-08-11)

- Agrega validacion de que el campo `superTheme` sólo contenga ids en mayúsculas o minúsculas de alguno de los 13 temas de la taxonomía temática de datos.gob.ar.
- Agrega validación limitando a 60 caracteres los nombres de los campos `field_title`.
- Mejoras al reporte de asistencia a la federación.

0.2.26 (2017-08-04)

- Agrega validación de que no haya ids repetidos en la lista de temas de `themeTaxonomy`.
- Agrega traducción de ckan del campo extra `Cobertura temporal` a `temporal`.

0.2.24 (2017-08-03)

- Mejoras en los reportes de errores y análisis de datasets para federación
- Métodos `DataJson.validate_catalog()` y `DataJson.generate_datasets_report()` tienen nuevas opciones para mejorar los reportes, especialmente en excel.

0.2.23 (2017-08-02)

- Bug fixes

0.2.22 (2017-08-02)

- Agrega estilo y formato al reporte de datasets
- Agrega nuevos campos al reporte de datasets
- Agrega un campo identificador del catálogo en el archivo de configuración de federación

0.2.21 (2017-08-02)

- Tolera el caso de intentar escribir un reporte de datasets sobre un catálogo que no tiene datasets. Loggea un warning en lugar de levantar una excepción.

0.2.20 (2017-08-01)

- Elimina la verificación de SSL en las requests de ckan_reader.

0.2.19 (2017-08-01)

- Elimina la verificación de SSL en las requests.

0.2.18 (2017-07-25)

- Mejora la validación del campo `temporal`
- Agrega formas de reporte de errores para el método `DataJson.validate_catalog()`:
  - Devuelve sólo errores con `only_errors=True`
  - Devuelve una lista de errores lista para ser convertida en tabla con `fmt="list"`

0.2.17 (2017-07-18)

- Agrega un método para convertir un intervalo repetido (Ej.: R/P1Y) en su representación en prosa ("Anualmente").
- Agrego método que estima los datasets federados que fueron borrados de un catálogo específico. Se consideran datasets federados y borrados de un catálogo específico aquellos cuyo publisher.name existe dentro de algún otro dataset todavía presente en el catálogo específico.

0.2.16 (2017-07-13)

- Bug fix: convierte a unicode antes de escribir un objeto a JSON.

0.2.15 (2017-07-11)

- Modifica la definición de dataset actualizado usando el campo "modified" del perfil de metadatos. Si este campo no está presente en la metadata de un dataset, se lo considera desactualizado.

0.2.14 (2017-07-10)

- Modifica la definición de dataset usada para comparar limitándola a la comparación por "title" y "publisher_name".

0.2.13 (2017-06-22)

- Agrega método para verificar si un dataset individual está actualizado

0.2.12 (2017-06-22)

- Se modifica el template de CATALOG README
- Se agrega el indicador "datasets_no_federados" a generate_catalogs_indicators

0.2.11 (2017-05-23)

- Se agrega en `core` el método `DataJson.generate_catalogs_indicators`, que genera indicadores de monitoreo de catálogos, recopilando información sobre, entre otras cosas, su validez, actualidad y formato de sus contenidos.

0.2.10 (2017-05-11)

- Correción ortográfica del listado de frecuencias de actualización admisibles (`pydatajson/schemas/accrualPeriodicity.json`).

0.2.9 (2017-05-04)

- Hotfixes para que `pydatajson` sea deployable en nuevos entornos donde el `setup.py` estaba fallando.

0.2.5 (2017-02-16)

- Se agrega una nueva función a `readers`, `read_ckan_catalog`, que traduce los metadatos que disponibiliza la Action API v3 de CKAN al estándar `data.json`. Esta función *no* está integrada a `read_catalog`.
- Se modifican todos los esquemas de validación, de modo que los campos opcionales de cualquier tipo y nivel acepten strings vacías.

0.2.0 (2017-01-31)

- Se reestructura la librería en 4 módulos: `core`, `readers`, `writers` y `helpers`. Toda la funcionalidad se mantiene intacta, pero algunas funciones muy utilizadas cambian de módulo. En particular, `pydatajson.pydatajson.read_catalog` es ahora `pydatajson.readers.read_catalog`, y `pydatajson.xlsx_to_json.write_json_catalog` es ahora `pydatajson.writers.write_json_catalog` (o `pydatajson.writers.write_json`).
- Se agrega el parámetro `frequency` a `pydatajson.DataJson.generate_harvester_config`, que controla la frecuencia de cosecha que se pretende de los datasets a incluir en el archivo de configuración. Por omisión, se usa `'R/P1D'` (diariamente) para todos los datasets.
- Se agrega la carpeta `samples/`, con dos rutinas de transformación y reporte sobre catálogos de metadatos en formato XLSX.

0.1.7 (2017-01-10)

- Se agrega el módulo `xlsx_to_json`, con dos métodos para lectura de archivos locales o remotos, sean JSON genéricos (`xlsx_to_json.read_json()`) o metadatos de catálogos en formato XLSX (`read_local_xlsx_catalog()`).
- Se agrega el método `pydatajson.read_catalog()` que interpreta todos las representaciones externas o internas de catálogos conocidas, y devuelve un diccionario con sus metadatos.

0.1.6 (2017-01-04)

- Se incorpora el método `DataJson.generate_harvestable_catalogs()`, que filtra los datasets no deseados de un conjunto de catálogos.
- Se agrega el parámetro `harvest` a los métodos `DataJson.generate_harvestable_catalogs()`, `DataJson.generate_datasets_report()` y `DataJson.generate_harvester_config()`, para controlar el criterio de elección de los datasets a cosechar.
- Se agrega el parámetro `export_path` a los métodos `DataJson.generate_harvestable_catalogs()`, `DataJson.generate_datasets_report()` y `DataJson.generate_harvester_config()`, para controlar la exportación de sus resultados.

0.1.4 (2016-12-23)

- Se incorpora el método `DataJson.generate_datasets_report()`, que reporta sobre los datasets y la calidad de calidad de metadatos de un conjunto de catálogos.
- Se incorpora el método `DataJson.generate_harvester_config()`, que crea archivos de configuración para el Harvester a partir de los reportes de `generate_datasets_report()`.

0.1.3 (2016-12-19)

- Al resultado de `DataJson.validate_catalog()` se le incorpora una lista (`."errors"`) con información de los errores encontrados durante la validación en cada nivel de jerarquía ("catalog" y cada elemento de "dataset")

0.1.2 (2016-12-14)

- Se incorpora validación de tipo y formato de campo
- Los métodos `DataJson.is_valid_catalog()` y `DataJson.validate_catalog()` ahora aceptan un `dict` además de un `path/to/data.json` o una url a un data.json.

0.1.0 (2016-12-01)

Primera versión para uso productivo del paquete.

- La instalación via `pip install` debería reconocer correctamente la ubicación de los validadores por default.
- El manejo de data.json's ubicados remotamente se hace en función del resultado de `urlparse.urlparse`
- El formato de respuesta de `validate_catalog` se adecúa a la última especificación (ver `samples/validate_catalog_returns.json`.

0.0.13 (2016-11-25)

- Intentar que la instalación del paquete sepa donde están instalados los schemas por default

0.0.12 (2016-11-25)

- Primera versión propuesta para v0.1.0