Romanesco Documentation

Release 0.1.0

Kitware, Inc.

January 27, 2016

Contents

CHAPTER 1

What is Romanesco?

Romanesco is a python application for generic task execution. It can be run within a celery worker to provide a distributed batch job execution platform.

The application can run tasks in a variety of languages and environments, including python, R, spark, and docker, all via a single python or celery broker interface. Tasks can be chained together into workflows, and these workflows can actually span multiple languages and environments seamlessly. Data flowing between tasks can be automatically converted into a format understandable in the target environment. For example, a python object from a python task can be automatically converted into an R object for an R task at the next stage of a pipeline.

Romanesco defines a specification that prescribes a loose coupling between a task and its runtime inputs and outputs. That specification is described in the API documentation section. This specification is language-independent and instances of the spec are best represented by a hierarchical data format such as JSON or YAML, or an equivalent serializable type such as a `dict` in python.

Romanesco is designed to be easily extended to new languages and environments, or to support new data types and formats, or modes of data transfer. This is accomplished via its plugin system, which is described in Plugins.

1.1 Installation

To install the romanesco worker on your system, we recommend using `pip` to install the package. (If you wish to install from source, see the *Installing from source* section of the developer documentation.) The following command will install the core dependencies:

```
pip install romanesco
```

That will install the core romanesco library, but not the third-party dependencies for any of its plugins. If you want to enable a set of plugins, their IDs should be included as extras to the pip install command. For instance, if you are planning to use the R plugin and the spark plugin, you would run:

```
pip install romanesco[r,spark]
```

You can run this command at any time to install dependencies of other plugins, even if romanesco is already installed.

1.2 Configuration

Several aspects of Romanesco's behavior are controlled via its configuration file. This file is found within the installed package directory as `worker.local.cfg`. If this file does not exist, simply run:

```
cd romanesco ; cp worker.dist.cfg worker.local.cfg
```

The core configuration parameters are outlined below.

- `celery.app_main`: The name of the celery application. Clients will need to use this same name to identify what app to send tasks to. It is recommended to call this "romanesco" unless you have a reason not to.
- `celery.broker`: This is the broker that celery will connect to in order to listen for new tasks. Celery recommends using RabbitMQ as your message broker.
- `romanesco.tmp_root`: Each romanesco task is given a temporary directory that it can use if it needs filesystem storage. This config setting points to the root directory under which these temporary directories will be created.
- `romanesco.plugins_enabled`: This is a comma-separated list of plugin IDs that will be enabled at runtime, e.g. `spark,vtk`.
- `romanesco.plugin_load_path`: If you have any external plugins that are not inside the **romanesco/plugins** package directory, set this value to a colon-separated list of directories to search for external plugins that need to be loaded.

Note: After making changes to values in the config file, you will need to restart the worker before the changes will be reflected.

1.3 Types and formats

In Romanesco, every analysis input and output is typed. A *type* in Romanesco is a high-level description of a data structure useful for intuitive workflows. It is not tied to a particular representation. For example, the *table* type may be defined as a list of rows with ordered, named column fields. This description does not specify any central representation since the information may be stored in a variety of ways. A type is specified by a string unique to your Romanesco environment, such as `"table"` for the table type.

An explicit representation of data is called a *format* in Romanesco. A format is a low-level description of data layout. For example, the table type may have formats for CSV, database table, R data frame, or JSON. The format may be text, serialized binary, or even in-memory data layouts. Just like types, a format is specified by a string unique to your Romanesco environment, such as `"csv"` for the CSV format. Formats under the same type should be convertable between each other.

Notice that the above uses the phrases such as "may be defined" and "may have formats". This is because at its core Romanesco does not contain types or formats. The *`romanesco.run()`* function will attempt to match given input bindings to analysis inputs, validating data and performing conversions as needed. To make Romanesco aware of certain types and formats, you must define validation and conversion routines. These routines are themselves Romanesco algorithms of a particular form, loaded with *`romanesco.format.import_converters()`*. See that function's documentation for how to define validators and converters.

The following are the types available in Romanesco core. Plugins may add their own types and formats using the `romanesco.format.import_converters` function. See the Plugins section for details on plugin-specific types and formats.

1.3.1 `"boolean"` type

A true or false value. Formats:

`"boolean"` An in-memory Python `bool`.

"json" A JSON string representing a single boolean (`"true"` or `"false"`).

1.3.2 "number" type

A numeric value (integer or real). Formats:

"number" An in-memory Python `int` or `float`.

"json" A JSON string representing a single number.

1.3.3 "string" type

A sequence of characters.

"text" A raw string of characters (`str` in Python).

"json" A JSON string representing a single string. This is a quoted string with certain characters escaped.

1.3.4 "table" type

A list of rows with ordered, named column attributes. Formats:

"rows" A Python dictionary containing keys `"fields"` and `"rows"`. `"fields"` is a list of column names that specifies column order. `"rows"` is a list of dictionaries of the form `field: value` where `field` is the field name and `value` is the value of the field for that row. For example:

```
{
    "fields": ["one", "two"],
    "rows": [{"one": 1, "two": 2}, {"one": 3, "two": 4}]
}
```

"rows.json" The equivalent JSON representation of the `"rows"` format.

"objectlist" A Python list of dictionaries of the form `field: value` where `field` is the field name and `value` is the value of the field for that row. For example:

```
[{"one": 1, "two": 2}, {"one": 3, "two": 4}]
```

This is identical to the `"rows"` field of the `"rows"` format. Note that this format does not preserve column ordering.

"objectlist.json" The equivalent JSON representation of the `"objectlist"` format.

"objectlist.bson" The equivalent BSON representation of the `"objectlist"` format. This is the format of MongoDB collections.

"csv" A string containing the contents of a comma-separated CSV file. Column headers will be reasonably detected if present, otherwise columns will be named `"Column 1"`, `Column 2"`, etc. See **'has_header'_** for details on header detection.

"tsv" A string containing the contents of a tab-separated TSV file. Column headers are detected the same as for the `"csv"` format.

1.3.5 "tree" type

A hierarchy of nodes with node and/or link attributes. Formats:

"nested" A nested Python dictionary representing the tree. All nodes may contain a "children" key containing a list of child nodes. Nodes may also contain "node_data" and "edge_data" which are name: value dictionaries of node and edge attributes. The top-level (root node) dictionary contains "node_fields" and "edge_fields" which are lists of node and edge attribute names to preserve ordering. The root should not contain "edge_data" since it does not have a parent edge. For example:

```
{
    "edge_fields": ["weight"],
    "node_fields": ["node name", "node weight"],
    "node_data": {"node name": "", "node weight": 0.0},
    "children": [
        {
            "node_data": {"node name": "", "node weight": 2.0},
            "edge_data": {"weight": 2.0},
            "children": [
                {
                    "node_data": {"node name": "ahli", "node weight": 2.0},
                    "edge_data": {"weight": 0.0}
                },
                {
                    "node_data": {"node name": "allogus", "node weight": 3.0},
                    "edge_data": {"weight": 1.0}
                }
            ]
        },
        {
            "node_data": {"node name": "rubribarbus", "node weight": 3.0},
            "edge_data": {"weight": 3.0}
        }
    ]
}
```

"nested.json" The equivalent JSON representation of the "nested" format.

"newick" A tree in Newick format.

"nexus" A tree in Nexus format.

"phyloxml" A phylogenetic tree in PhyloXML format.

1.3.6 "graph" type

A collection of nodes and edges with optional attributes. Formats:

"networkx" A representation of a graph using an object of type nx.Graph (or any of its subclasses).

"networkx.json" A JSON representation of a NetworkX graph.

"clique.json" A JSON representation of a Clique graph.

"graphml" An XML String representing a valid GraphML representation.

"adjacencylist" A string representing a very simple adjacency list which does not preserve node or edge attributes.

1.3.7 "image" type

A 2D matrix of uniformly-typed numbers. Formats:

"png" An image in PNG format.

"png.base64" A Base-64 encoded PNG image.

"pil" An image as a `PIL.Image` from the Python Imaging Library.

1.4 API documentation

1.4.1 Overview

The main purpose of Romanesco is to execute a broad range of tasks. These tasks, along with a set of input bindings and output bindings are passed to the *romanesco.run()* function, which is responsible for fetching the inputs as necessary and executing the task, and finally populating any output variables and sending them to their destination.

The task, its inputs, and its outputs are each passed into the function as python dictionaries. In this section, we describe the structure of each of those dictionaries.

The task specification

The first argument to *romanesco.run()* describes the task to execute, independently of the actual data that it will be executed upon. The most important field of the task is the `mode`, which describes what type of task it is. The structure for the task dictionary is described below. Uppercase names within angle braces represent symbols defined in the specification. Optional parts of the specification are surrounded by parenthesis to avoid ambiguity with the square braces, which represent lists in python or Arrays in JSON. The Python task also accepts a `write_script` paramater that when set to 1 will write task scripts to disk before executing them. This aids in readability for interactive debuggers such as `pdb`.

```
<TASK> ::= <PYTHON_TASK> | <R_TASK> | <DOCKER_TASK> | <WORKFLOW_TASK>

<PYTHON_TASK> ::= {
    "mode": "python",
    "script": <python code to run as a string>
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
    (, "write_script": 1)
}

<R_TASK> ::= {
    "mode": "r",
    "script": <r code to run (as a string)>
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}

<DOCKER_TASK> ::= {
    "mode": "docker",
    "docker_image": <docker image name to run>
    (, "container_args": [<container arguments>])
    (, "entrypoint": <custom override for container entry point>)
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
```

```
}

<WORKFLOW_TASK> ::= {
    "mode": "workflow",
    "steps": [<WORKFLOW_STEP> (, <WORKFLOW_STEP>, ...)],
    "connections": [<WORKFLOW_CONNECTION> (, <WORKFLOW_CONNECTION>, ...)]
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}

<WORKFLOW_STEP> ::= {
    "name": <step name>,
    "task": <TASK>
}

<WORKFLOW_CONNECTION> ::= {
    ("name": <name of top-level input to bind to>)
    (, "input": <input id to bind to for a step>)
    (, "input_step": <input step name to connect>)
    (, "output_step": <output step name to connect>)
}
```

The workflow mode simply allows for a directed acyclic graph of tasks to be specified to *romanesco.run()*.

```
<TASK_INPUT> ::= {
    "id": <string, the variable name>,
    "type": <data type>,
    "format": <data format>
    (, "default": <default value if none is bound at runtime>)
    (, "target": <INPUT_TARGET_TYPE>)   ; default is "memory"
    (, "filename": <name of file if target="filepath">)
}

<INPUT_TARGET_TYPE> ::= "memory" | "filepath"

<TASK_OUTPUT> ::= {
    "id": <string, the variable name>,
    "type": <data type>,
    "format": <data format>
    (, "target": <INPUT_TARGET_TYPE>)   ; default is "memory"
}
```

The input specification

The `inputs` argument to *romanesco.run()* specifies the inputs to the task described by the `task` argument. Specifically, it tells what data should be placed into the task input ports.

```
<INPUTS> ::= {
    <id> : <INPUT_BINDING>
    (, <id> : <INPUT_BINDING>)
    (, ...)
}
```

The input spec is a dictionary mapping each `id` (corresponding to the `id` key of each task input) to its data binding for this execution.

```
<INPUT_BINDING> ::= <INPUT_BINDING_HTTP> | <INPUT_BINDING_LOCAL> |
                    <INPUT_BINDING_MONGODB> | <INPUT_BINDING_INLINE>

<INPUT_BINDING_HTTP> ::= {
    "mode": "http",
    "format": <data format>,
    "url": <url of data to download>
    (, "headers": <dict of HTTP headers to send when fetching>)
    (, "method": <http method to use, default is "GET">)
    (, "maxSize": <integer, max size of download in bytes>)
}
```

The http input mode specifies that the data should be fetched over HTTP. Depending on the `target` field of the corresponding task input specifier, the data will either be passed in memory, or streamed to a file on the local filesystem, and the variable will be set to the path of that file.

```
<INPUT_BINDING_LOCAL> ::= {
    "mode": "local",
    "format": <data format>,
    "path": <path on local filesystem to the file>
}
```

The local input mode denotes that the data exists on the local filesystem. Its contents will be read into memory and the variable will point to those contents.

```
<INPUT_BINDING_MONGODB> ::= {
    "mode": "mongodb",
    "format": <data format>,
    "db": <the database to use>,
    "collection": <the collection to fetch from>
    (, "host": <mongodb host, default is "localhost">)
}
```

The mongodb input mode specifies that the data should be fetched from a mongo collection. This simply binds the entire BSON-encoded collection to the input variable.

```
<INPUT_BINDING_INLINE> ::= {
    "mode": "inline",
    "format": <data format>,
    "data": <data to bind to the variable>
}
```

The inline input mode simply passes the data directly in the input binding dictionary as the value of the "data" key. Do not use this for any data that could be large.

The output specification

The optional `outputs` argument to *romanesco.run()* specifies output variables of the task that should be handled in some way.

```
<OUTPUTS> ::= {
    <id> : <OUTPUT_BINDING>
    (, <id> : <OUTPUT_BINDING>)
    (, ...)
}
```

The output spec is a dictionary mapping each `id` (corresponding to the `id` key of each task output) to some behavior that should be performed with it. Task outputs that do not have bindings in the ouput spec simply get their results set

in the return value of *romanesco.run()*.

```
<OUTPUT_BINDING> ::= <OUTPUT_BINDING_HTTP> | <OUTPUT_BINDING_LOCAL> |
                     <OUTPUT_BINDING_MONGODB>

<OUTPUT_BINDING_HTTP> ::= {
    "mode": "http",
    "url": <url to upload data to>,
    "format": <data format>
    (, "headers": <dict of HTTP headers to send with the request>)
    (, "method": <http method to use, default is "POST">)
}

<OUTPUT_BINDING_LOCAL> ::= {
    "mode": "local",
    "format": <data format>,
    "path": <path to write data on the local filesystem>
}
```

The local output mode writes the data to the specified path on the local filesystem.

```
<OUTPUT_BINDING_MONGODB> ::= {
    "mode": "mongodb",
    "db": <mongo database to write to>,
    "format": <data format>,
    "collection": <mongo collection to write to>
    (, "host": <mongo host to connect to>)
}
```

The mongodb output mode attempts to BSON-decode the bound data, and then overwrites any data in the specified collection with the output data.

1.4.2 Script execution

`romanesco.`**`convert`**(*type*, *input*, *output*, ***kwargs*)

Convert data from one format to another.

Parameters

- **`type`** – The type specifier string of the input data.
- **`input`** – A binding dict of the form `{"format": format, "data", data}`, where `format` is the format specifier string, and `data` is the raw data to convert. The dict may also be of the form `{"format": format, "uri", uri}`, where `uri` is the location of the data (see `romanesco.uri` for URI formats).
- **`output`** – A binding of the form `{"format": format}`, where `format` is the format specifier string to convert the data to. The binding may also be in the form `{"format": format, "uri", uri}`, where `uri` specifies where to place the converted data.

Returns The output binding dict with an additional field `"data"` containing the converted data. If `"uri"` is present in the output binding, instead saves the data to the specified URI and returns the output binding unchanged.

`romanesco.`**`isvalid`**(*type*, *binding*, ***kwargs*)

Determine whether a data binding is of the appropriate type and format.

Parameters

- **`type`** – The expected type specifier string of the binding.

- **binding** – A binding dict of the form `{"format": format, "data", data}`, where `format` is the format specifier string, and `data` is the raw data to test. The dict may also be of the form `{"format": format, "uri", uri}`, where `uri` is the location of the data (see `romanesco.uri` for URI formats).

Returns `True` if the binding matches the type and format, `False` otherwise.

`romanesco.`**`load`**(*task_file*)

Load a task JSON into memory, resolving any `"script_uri"` fields by replacing it with a `"script"` field containing the contents pointed to by `"script_uri"` (see `romanesco.uri` for URI formats). A `script_fetch_mode` field may also be set

Parameters **analysis_file** – The path to the JSON file to load.

Returns The analysis as a dictionary.

`romanesco.`**`register_executor`**(*name*, *fn*)

Register a new executor in the romanesco runtime. This is used to map the "mode" field of a task to a function that will execute the task.

Parameters

- **name** (*str*) – The value of the mode field that maps to the given function.
- **fn** (*function*) – The implementing function.

`romanesco.`**`run`**(**args*, ***kwargs*)

Run a Romanesco task with the specified I/O bindings.

Parameters

- **task** (*dict*) – Specification of the task to run.
- **inputs** – Specification of how input objects should be fetched into the runtime environment of this task.
- **outputs** (*dict*) – Speficiation of what should be done with outputs of this task.
- **auto_convert** – If `True` (the default), perform format conversions on inputs and outputs with *convert()* if they do not match the formats specified in the input and output bindings. If `False`, an expection is raised for input or output bindings do not match the formats specified in the analysis.
- **validate** – If `True` (the default), perform input and output validation with *isvalid()* to ensure input bindings are in the appropriate format and outputs generated by the script are formatted correctly. This guards against dirty input as well as buggy scripts that do not produce the correct type of output. An invalid input or output will raise an exception. If `False`, perform no validation.
- **write_script** – If `True` task scripts will be written to file before being passed to `exec`. This improves interactive debugging with tools such as `pdb` at the cost of additional file I/O. Note that when passed to run *all* tasks will be written to file including validation and conversion tasks.

Returns A dictionary of the form `name: binding` where `name` is the name of the output and `binding` is an output binding of the form `{"format": format, "data": data}`. If the *outputs* param is specified, the formats of these bindings will match those given in *outputs*. Additionally, `"data"` may be absent if an output URI was provided. Instead, those outputs will be saved to that URI and the output binding will contain the location in the `"uri"` field.

`romanesco.`**`unregister_executor`**(*name*)

Unregister an executor from the map.

Parameters **name** (*str*) – The name of the executor to unregister.

1.4.3 Formats

class romanesco.format.**Validator**

type
The validator type, like string.

format
The validator format, like text.

romanesco.format.**converter_path**(*source*, *target*)
Gives the shortest path that should be taken to go from a source type/format to a target type/format.

Throws a NetworkXNoPath exception if it can not find a path.

Parameters

- **source** – Validator tuple indicating the type/format being converted *from*.
- **target** – Validator tuple indicating the type/format being converted *to*.

Returns An ordered list of the analyses that need to be run to convert from

source to target.

romanesco.format.**get_validator**(*validator*)
Gets a validator node from the conversion graph by its type and format.

```
>>> validator = get_validator(Validator('string', 'text'))
```

Returns a tuple containing 2 elements >>> len(validator) 2

First is the Validator namedtuple >>> validator[0] Validator(type=u'string', format=u'text')

and second is the validator itself >>> validator[1].keys() ['validator', 'type', 'format']

If the validator doesn't exist, an exception will be raised >>> get_validator(Validator('foo', 'bar')) Traceback (most recent call last):

...

Exception: No such validator foo/bar

Parameters **validator** – A Validator namedtuple

Returns A tuple including the passed Validator namedtuple, with a second

element being the analysis data.

romanesco.format.**has_converter**(*source*, *target=Validator(type=None*, *format=None)*)
Determines if any converters exist from a given type, and optionally format.

Underneath, this just traverses the edges until it finds one which matches the arguments.

Parameters

- **source** – Validator tuple indicating the type/format being converted *from*.
- **target** – Validator tuple indicating the type/format being converted *to*.

Returns True if it can converter from source to target, False

otherwise.

romanesco.format.**import_converters**(*search_paths*)
Import converters and validators from the specified search paths. These functions are loaded into romanesco.format.conv_graph with nodes representing validators, and directed edges representing converters.

Any files in a search path matching validate_*.json are loaded as validators. Validators should be fast (ideally O(1)) algorithms for determining if data is of the specified format. These are algorithms that have a single input named "input" and a single output named "output". The input has the type and format to be checked. The output must have type and format "boolean". The script performs the validation and sets the output variable to either true or false.

Any *_to_*.json files are imported as converters. A converter is simply an analysis with one input named "input" and one output named "output". The input and output should have matching type but should be of different formats.

Parameters **search_paths** (*str or list of str*) – A list of search paths relative to the current working directory. Passing a single path as a string also works.

romanesco.format.**import_default_converters**()
Import converters from the default search paths. This is called when the *romanesco.format* module is first loaded.

romanesco.format.**print_conversion_graph**()
Print a graph of supported conversion paths in DOT format to standard output.

romanesco.format.**print_conversion_table**()
Print a table of supported conversion paths in CSV format with "from" and "to" columns to standard output.

1.4.4 URIs

1.5 Developer documentation

1.5.1 Installing from source

Clone from git:

```
git clone https://github.com/Kitware/romanesco.git
cd romanesco
```

Test it:

```
python -m unittest -v tests.table_test
python -m unittest -v tests.tree_test
```

Some things not working? You can install a few things so they do. For example, install MongoDB and R, in addition to their Python bindings:

```
pip install pymongo rpy2  # may need sudo
```

You'll need to get a MongoDB server listening on localhost by running mongod.

In R, you'll need to install some stuff too, currently just the ape package:

```
install.packages("ape")
```

Some things depend on VTK Python bindings. Romanesco uses some features from cutting-edge VTK, so you'll likely need to build it from scratch (takes ~30 minutes). First get CMake, then do the following:

```
git clone git://vtk.org/VTK.git
cd VTK
mkdir build
cd build
cmake .. -DVTK_WRAP_PYTHON:BOOL=ON -DBUILD_TESTING:BOOL=OFF
make
export PYTHONPATH=`pwd`/Wrapping/Python:`pwd`/lib
python -c "import vtk"  # should work without an error
```

Want to run things remotely? On the client and server install celery:

```
pip install celery
```

Then fire up the celery worker:

```
python -m romanesco
```

On the client, run a script akin to the following example:

```
python clients/client.py
```

This section of the documentation is meant for those who wish to contribute to the Romanesco core platform.

1.5.2 Note on building VTK

The VTK binaries used for testing in the .travis.yml is built in an Ubuntu precise 64-bit VM using the following commands:

```
# from host
vagrant up
vagrant ssh

# from Vagrant guest
sudo apt-get update
sudo apt-get install g++ make
wget http://www.cmake.org/files/v2.8/cmake-2.8.12.2.tar.gz
tar xzvf cmake-2.8.12.2.tar.gz
cd cmake-2.8.12.2
./bootstrap --prefix=~/cmake-2.8.12.2-precise64
make install
cd ~
git clone git://vtk.org/VTK.git
cd VTK
mkdir build
cd build
~/cmake-2.8.12.2-precise64/bin/cmake .. -DBUILD_TESTING:BOOL=OFF -DVTK_WRAP_PYTHON:BOOL=ON -DVTK_Grou
make install
tar czvf vtk-precise64.tar.gz ~/vtk-precise64/
exit

# from host
scp -P 2222 vagrant@localhost:~/vtk-precise64-118242.tar.gz .
```

1.5.3 Creating a new release

Romanesco releases are uploaded to PyPI for easy installation via `pip`. The recommended process for generating a new release is described here.

1. From the target commit, set the desired version number in `plugin.json`. Create a new commit and note the SHA; this will become the release tag.
2. Ensure that all tests pass.
3. Clone the repository in a new directory and checkout the release SHA. (Packaging in an old directory could cause extraneous files to be mistakenly included in the source distribution.)
4. Run `python setup.py sdist --dist-dir .` to generate the distribution tarball in the project directory, which looks like `romanesco-x.y.z.tar.gz`.
5. Create a new virtual environment and install the python package into it. This should not be done in the repository directory because the wrong package will be imported.

```
mkdir test && cd test
virtualenv release
source release/bin/activate
pip install ../romanesco-<version>.tar.gz
```

6. Once that finishes, you should be able to start the worker by simply running `romanesco-worker`.
7. When you are confident everything is working correctly, generate a new release on GitHub. You must be sure to use a tag version of `v<version>`, where `<version>` is the version number as it exists in `plugin.json`. For example, `v0.2.4`. Attach the three tarballs you generated to the release.
8. Add the tagged version to readthedocs and make sure it builds correctly.
9. Finally, upload the release to PyPI with the following command:

```
python setup.py sdist upload
```

Note: The first time you create a release, you will need to register to PyPI before you can run the upload step. To do so, simply run `python setup.py sdist register`.

1.6 Plugins

The Romanesco plugin system is used to extend the core functionality of Romanesco in a number of ways. Plugins can execute any python code when they are loaded at runtime, but the most common augmentations they perform are:

- **Adding new execution modes.** Without any plugins enabled, the core Romanesco application can only perform two types of tasks: `python` and `workflow` modes. It's common for plugins to implement other task execution modes.
- **Adding new data types or formats.** Plugins can make Romanesco aware of new data types and formats, and provide implementations for how to validate and convert to and from those formats.
- **Adding new IO modes.** One of the primary functions of Romanesco is to fetch input data from heterogenous sources and expose it to tasks in a uniform way. Plugins can implement novel modes of fetching and pushing input and output data for a task.

Below is a list of the plugins that are shipped with the romanesco package. They can be enabled via the configuration file (see *Configuration*).

1.6.1 Docker

- **Plugin ID:** `docker`

- **Description:** This plugin exposes a new task execution mode, `docker`. These tasks pull a docker image and run a container using that image, with optional command line arguments. Docker tasks look like:

```
<DOCKER_TASK> ::= {
    "mode": "docker",
    "docker_image": <docker image name to run>
    (, "container_args": [<container arguments>])
    (, "entrypoint": <custom override for container entry point>)
    (, "inputs": [<TASK_INPUT> (, <TASK_INPUT>, ...)])
    (, "outputs": [<TASK_OUTPUT> (, <TASK_OUTPUT>, ...)])
}
```

The optional `container_args` parameter is a list of arguments to pass to the container. If an `entrypoint` argument is passed, it will override the built-in `ENTRYPOINT` directive of the image. Since it's often the case that task inputs will need to passed to the container as arguments, a special syntax can be used to declare that a command line argument should be expanded at runtime to the value of an input:

```
"container_args": ["$input{my_input_id}"]
```

It is not necessary for the entire argument to be a variable expansion; any part of an argument can also be expanded, e.g.:

```
"container_args": ["--some-parameter=$input{some_parameter_value}"]
```

The temporary directory for the romanesco task is mapped into the running container under the directory `/data`, so any files that were fetched into that temp directory will be available inside the running container at that path.

1.6.2 Girder IO

- **Plugin ID:** `girder_io`
- **Description:** This plugin adds new fetch and push modes called `girder`. The fetch mode for inputs supports downloading folders, items, or files from a Girder server. Inputs can be downloaded anonymously (if they are public) or using an authentication token. This data is always written to disk within the task's temporary directory, and is always a `string/text` format since the data itself is simply the path to the downloaded file or directory.

```
<GIRDER_INPUT> ::= {
    "mode": "girder",
    "id": <the _id value of the resource to download>,
    "name": <the name of the resource to download>,
    "host": <the hostname of the girder server>,
    "format": "text",
    "type": "string"
    (, "port": <the port of the girder server, default is 80 for http: and 443 for https:>)
    (, "api_root": <path to the girder REST API, default is "/api/v1")
    (, "scheme": <"http" or "https", default is "http">)
    (, "token": <girder token used for authentication>)
    (, "resource_type": <"file", "item", or "folder", default is "file">)
}
```

The output mode also assumes data of format `string/text` that is a path to a file in the filesystem. That file will then be uploaded under an existing folder (under a new item with the same name as the file), or into an existing item.

```
<GIRDER_OUTPUT> ::= {
    "mode": "girder",
    "parent_id": <the _id value of the folder or item to upload into>,
    "host": <the hostname of the girder server>,
```

```
    "format": "text",
    "type": "string"
    (, "name": <optionally override name of the file to upload>)
    (, "port": <the port of the girder server, default is 80 for http and 443 for https>)
    (, "api_root": <path to the girder REST API, default is "/api/v1")
    (, "scheme": <"http" or "https", default is "http">)
    (, "token": <girder token used for authentication>)
    (, "parent_type": <"folder" or "item", default is "folder">)
}
```

1.6.3 R

- **Plugin ID:** `r`
- **Description:** The R plugin enables the execution of R scripts as tasks via the `r` execution mode. It also exposes a new data type, `r`, and several new data formats and converters for existing data types. Just like `python` mode, the R code to run is passed via the `script` field of the task specification. The `r` data type refers to objects compatible with the R runtime environment.
- **Converters added:**
  - `r/object r/serialized`
  - `table/csv table/r.dataframe`
  - `tree/newick tree/r.apetree`
  - `tree/nexus tree/r.apetree`
  - `r/apetree` → `tree/treestore`
- **Validators added:**
  - `r/object`: An in-memory R object.
  - `r/serialized`: A serialized version of an R object created using R's `serialize` function.
  - `table/r.dataframe`: An R data frame. If the first column contains unique values, these are set as the row names of the data frame.
  - `tree/r.apetree`: A tree in the R package `ape` format.

1.6.4 Spark

- **Plugin ID:** `spark`
- **Description:** Adds a new execution mode `spark.python` that allows tasks to run inside a pyspark environment with a SparkContext variable automatically exposed. That is, each task will have a variable exposed in its python runtime called `sc` that is a valid SparkContext. This plugin exposes a new type, `collection`, referring to something that can be represented by a Spark RDD.
- **Converters added:**
  - `collection/json collection/spark.rdd`: Convert between a JSON list and an RDD created from calling `sc.parallelize` on the list.
- **Validators added:**
  - `collection/json`
  - `collection/spark.rdd`

1.6.5 VTK

- **Plugin ID:** `vtk`
- **Description:** This plugin exposes the `geometry` type and provides converters and validators for several types. This plugin requires that you have the VTK python package exposed in Romanesco's python environment. The `geometry` type represents 3D geometry.
- **Converters added:**
  - `geometry/vtkpolydata geometry/vtkpolydata.serialized`
  - `table/rows table/vtktable`
  - `table/vtktable table/vtktable.serialized`
  - `tree/nested tree/vtktree`
  - `tree/vtktree` → `tree/newick`
  - `tree/vtktree tree/vtktree.serialized`
  - `graph/networkx graph/vtkgraph`
  - `graph/vtkgraph graph/vtkgraph.serialized`
- **Validators added:**
  - `geometry/vtkpolydata`: A vtkPolyData object.
  - `geometry/vtkpolydata.serialized`: A vtkPolyData serialized with vtkPolyDataWriter.
  - `table/vtktable`: A vtkTable.
  - `table/vtktable.serialized`: A vtkTable serialized with vtkTableWriter.
  - `tree/vtktree`: A vtkTree.
  - `tree/vtktree.serialized`: A vtkTree serialized with vtkTreeWriter.
  - `graph/vtkgraph`: A vtkGraph.
  - `graph/vtkgraph.serialized`: A vtkGraph serialized with vtkGraphWriter.

Note: vtkGraphs lose their actual node values as they are represented by their index. In addition, nodes and edges are given all metadata attributes with defaults if they do not specify the metadatum themselves. This is noted further in `romanesco.plugins.vtk.converters.graph.networkx_to_vtkgraph`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

r

C

F

G

H

I

L

P

R

T

U

V