Resolwe SDK for Python

Release 14.1.1.dev11+gcdaaaa0

Genialis, Inc.

May 13, 2022

CONTENTS

Resolwe SDK for Python supports interaction with Genialis Server. Genialis Server is based on Resolwe workflow engine and its plugin Resolwe Bioinformatics. You can use it to upload and inspect biomedical data sets, contribute annotations and run analysis.

CHAPTER

ONE

INSTALL

Install from PyPI:

```
pip install resdk
```

If you would like to contribute to the SDK code base, follow the *installation steps for developers*.

CHAPTER

TWO

USAGE EXAMPLE

We will download a sample containing raw sequencing reads that were aligned to a genome:

```
import resdk

# Create a Resolwe object to interact with the server
res = resdk.Resolwe(url='https://app.genialis.com')

# Enable verbose logging to standard output
resdk.start_logging()

# Get sample meta-data from the server
sample = res.sample.get('resdk-example')

# Download files associated with the sample
sample.download()
```

Multiple files (fastq, fastQC report, bam, bai…) have downloaded to the working directory. Check them out. To learn more about the Resolwe SDK continue with *Tutorials*.

If you have problems connecting to our server, please contact us at info@genialis.com.

CHAPTER

THREE

DOCUMENTATION

3.1 Getting started

This tutorial is for bioinformaticians. It will help you install the ReSDK and explain some basic commands. We will connect to an instance of Genialis server, do some basic queries, and align raw reads to a genome.

3.1.1 Installation

Installing is easy, just make sure you have Python and pip installed on your computer. Run this command in the terminal (CMD on Windows):

```
pip install resdk
```

3.1.2 Registration

The examples presented here require access to a public Genialis Server that is configured for the examples in this tutorial. Some parts of the documentation will work for registered users only. Please request a Demo on Genialis Server before you continue, and remember your username and password.

3.1.3 Connect to Genialis Server

Start the Python interpreter by typing `python` into the command line. You'll recognize the interpreter by '>>>'. Now we can connect to the Genialis Server:

```
import resdk

# Create a Resolwe object to interact with the server and login
res = resdk.Resolwe(url='https://app.genialis.com')
res.login()

# Enable verbose logging to standard output
resdk.start_logging()
```

Note: If you omit the `login()` line you will be logged as anonymous user. Note that anonymous users do not have access to the ful set of features.

Note: When connecting to the server through an interactive session, we suggest you use the `resdk.start_logging()` command. This allows you to see important messages (*e.g.* warnings and errors) when executing commands.

Note: To avoid copy-pasting of the commands, you can `download all the code` used in this section.

3.1.4 Query data

Before we start querying data on the server we should become familiar with what a data object is. Everything that is uploaded or created (via processes) on a server is a data object. The data object contains a complete record of the processing that has occurred. It stores the inputs (files, arguments, parameters…), the process (the algorithm) and the outputs (files, images, numbers…). Let's count all data objects on the server that we can access:

```
res.data.count()
```

This is all of the data on the server you have permissions for. As a new user you can only see a small subset of all data objects. We can see the data objects are referenced by *id*, *slug*, and *name*.

Note: `id` is the auto-generated **unique identifier** of an object. IDs are integers.

`slug` is the **unique name** of an object. The slug is automatically created from the `name` but can also be edited by the user. Only lowercase letters, numbers and dashes are allowed (will not accept white space or uppercase letters).

`name` is an arbitrary, **non unique name** of an object.

Let's say we now want to find some genome indices. We don't always know the *id*, *slug*, or *name* by heart, but we can use filters to find them. We will first count all genome index data objects:

```
res.data.filter(type='data:index').count()
```

This is quite a lot of objects! We can filter even further:

```
res.data.filter(type="data:index:star", name__contains="Homo sapiens")
```

Note: For a complete list of filtering options use a "wrong" filtering argument and you will receive an informative message with all options listed. For example:

```
res.data.filter(foo="bar")
```

For future work we want to get the genome with a specific slug. We will get it and store a reference to it for later:

```
# Get data object by slug
genome_index = res.data.get('resdk-example-genome-index')
```

We have now seen how to use filters to find and get what we want. Let's query and get a paired-end FASTQ data object:

```
# All paired-end fastq objects
res.data.filter(type='data:reads:fastq:paired')
```

(continues on next page)

(continued from previous page)

```
# Get specific object by slug
reads = res.data.get('resdk-example-reads')
```

We now have `genome` and `reads` data objects. We can learn about an object by calling certain object attributes. We can find out who created the object:

```
reads.contributor
```

and inspect the list of files it contains:

```
reads.files()
```

These and many other data object attributes/methods are described here.

3.1.5 Run alignment

A common analysis in bioinformatics is to align sequencing reads to a reference genome. This is done by running a certain *process*. A process uses an algorithm or a sequence of algorithms to turn given inputs into outputs. Here we will only test the STAR alignment process, but many more processes are available (see the Process catalog). This process automatically creates a BAM alignment file and BAI index, along with some other files.

Let's run STAR on our reads, using our genome:

```
bam = res.run(
    slug='alignment-star',
    input={
        'reads': reads.id,
        'genome': genome_index.id,
    },
)
```

This might seem like a complicated statement, but note that we only run a process with specific slug and required inputs. The processing may take some time. Note that we have stored the reference to the alignment object in a `bam` variable. We can check the status of the process to determine if the processing has finished:

```
bam.status
```

Status `OK` indicates that processing has finished successfully. If the status is not `OK` yet, run the `bam.update()` and `bam.status` commands again in few minutes. We can inspect our newly created data object:

```
# Get the latest info about the object from the server
bam.update()
bam.status
```

As with any other data object, it has its own *id*, *slug*, and *name*. We can explore the process inputs and outputs:

```
# Process inputs
bam.input

# Process outputs
bam.output
```

Download the outputs to your local disk:

```
bam.download()
```

We have come to the end of Getting started. You now know some basic ReSDK concepts and commands. Yet, we have only scratched the surface. By continuing with the Tutorials, you will become familiar with more advanced features, and will soon be able to perform powerful analyses on your data.

3.2 Tutorials

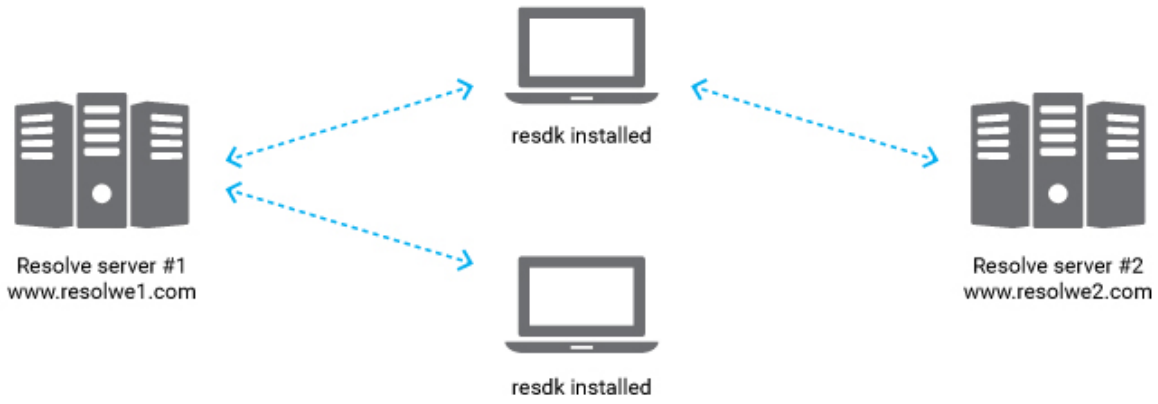
3.2.1 Genialis Server basics

This chapter provides a general overview and explains the basic concepts. We highly recommend reading it even though it is a bit theoretic.

Genialis Server and ReSDK

Genialis Server is a web application that can handle large quantities of biological data, perform complex data analysis, organize results, and automatically document your work in a reproducible fashion. It is based on Resolwe and Resolwe Bioinformatics. Resolwe is an open source dataflow package for the Django framework while Resolwe Bioinformatics is an extension of Resolwe that provides bioinformatics pipelines.

Resolwe SDK for Python allows you to access Genialis Server through Python. It supports accessing, modifying, uploading, downloading and organizing the data.



Genialis Server runs on computers with strong computational capabilities. On the contrary, `resdk` is a Python package on a local computer that interacts with Genialis Server through a RESTful API. The power of `resdk` is its lightweight character. It is installed with one simple command, but supports manipulation of large data sets and heavy computation on a remote server.

Data and Process

The two most fundamental resources in Genialis Server are *Data* and *Process*.

Process stores an algorithm that transforms inputs into outputs. It is a blueprint for one step in the analysis.

Data is an instance of a Process. It is a complete record of the performed processing. It remembers the inputs (files, arguments, parameters...), the algorithm used and the outputs (files, images, numbers...). In addition, Data objects store some useful meta data, making it easy to reproduce the dataflow and access information.

Example use case: you have a file `reads.fastq` with NGS read sequences and want to map them to the genome `genome.fasta` with aligner `STAR`. Reads are one Data object and genome is another one. Alignment is done by creating a third Data. At the creation, one always needs to define the Process (`STAR`) and inputs (first and second Data). When the Data object is created, the server automatically runs the given process with provided inputs and computes all inputs, outputs, and meta data.

Samples and Collections

Eventually, you will have many Data objects and want to organize them. Genialis server includes different structures to help you group Data objects: *Sample* and *Collection*.

Sample represents a biological entity. It includes user annotations and Data objects associated with this biological entity. In practice, all Data objects in the Sample are derived from an initial single Data object. Typically, a Sample would contain the following Data: raw reads, preprocessed reads, alignment (bam file), and expressions. A Data object can belong to only one Sample. Two distinct Samples cannot contain the same Data object.

Collection is a group of Samples. In addition to Samples and their Data, Collections may contain Data objects that store other analysis results. Example of this are differential expressions - they are done as combination of many Samples and cannot belong to only one Sample. Each Sample and Data object can only be in one Collection.

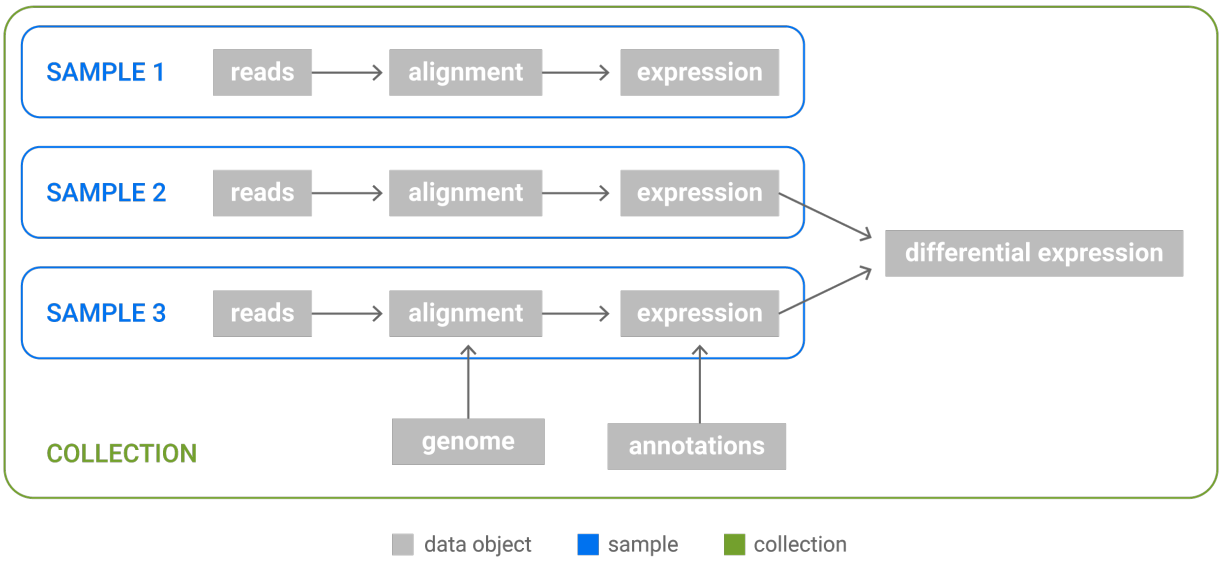


Fig. 1: Relations between Data, Samples and Collection. Samples are groups of Data objects originating from the same biological sample: all Data objects in a Sample are derived from a single NGS reads file. Collections are arbitrary groups of Samples and Data objects that store analysis results.

When a new Data object that represents a biological sample (*i.e.* fastq files, bam files) is uploaded, the unannotated Sample is automatically created. It is the duty of the researcher to properly annotate the Sample. When a Data object

that belongs to an existing Sample is used as an input to trigger a new analysis, the output of this process is automatically attached to an existing Sample.

3.2.2 Query, inspect and download data

Login

By now, you should have an account on the Genialis Server. If not, you can request a Demo. Let's connect to the server by creating a *Resolwe* object:

```
import resdk

# Create a Resolwe object to interact with the server and login
res = resdk.Resolwe(url='https://app.genialis.com')
res.login()

# Enable verbose logging to standard output
resdk.start_logging()
```

If you omit the `login()` you will be logged as anonymous user. Note that this will strongly limit the things you can do.

Note: To avoid copy-pasting of the commands, you can `download all the code` used in this section.

Query resources

As you have read in the *Genialis Server basics* section, there are various resources: *Data*, *Sample*, *Collection*, *Process*... each of which has a corresponding entry-point on `Resolwe` object (in our case, this is the `res` variable). For example, to count all `Data` or `Sample` objects:

```
res.data.count()
res.sample.count()
```

Note:

`id` is the autogenerated unique identifier of an object. IDs are integers.

`slug` is the **unique name** of an object. The slug is automatically created from the `name` but can also be edited by the user, although we do not recommend that. Only lowercase letters, numbers and dashes are allowed (will not accept white space or uppercase letters).

`name` is an arbitrary, **non unique name** of an object.

In practice one typically wants to narrow down the amount of results. This can be done with the *filter(**fields)* method. It returns a list of objects under the conditions defined by `**fields`. For example:

```
# Get all Collection objects with "RNA-Seq" in their name
res.collection.filter(name__contains='RNA-Seq')

# Get all Processes with category "Align"
res.process.filter(category='Align')
```

Note: For a complete list of processes, their categories and definitions, please visit resolwe-bio docs

But the real power of the `filter()` method is in combining multiple parameters:

```
# Filter by using several fields:
from datetime import datetime

res.data.filter(
  status='OK',
  created__gt=datetime(2018, 10, 1),
  created__lt=datetime(2025, 11, 1),
  ordering='-modified',
  limit=3,
)
```

This will return data objects with OK status, created in October 2018, order them by descending modified date and return first 3 objects. Quite powerful isn't it?

Note: For a complete list of filtering options use a "wrong" filtering argument and you will receive an informative message with all options listed. For example:

```
res.data.filter(foo="bar")
```

The *get(**fields)* method searches by the same parameters as `filter` and returns a single object (`filter` returns a list). If only one parameter is given, it will be interpreted as a unique identifier `id` or `slug`, depending on if it is a number or string:

```
# Get object by slug
res.sample.get('resdk-example')
```

Inspect resources

We have learned how to query the resources with `get` and `filter`. Now we will look at how to access the information in these resources. All of the resources share some common attributes like `name`, `id`, `slug`, `created`, `modified`, `contributor` and `permissions`. You can access them like any other Python class attributes:

```
# Get a data object:
data = res.data.get('resdk-example-reads')

# Object creator:
data.contributor
# Date and time of object creation:
data.created
# Name
data.name
# List of permissions
data.permissions
```

Aside from these attributes, each resource class has some specific attributes and methods. For example, some of the most used ones for `Data` resource:

```
data = res.data.get('resdk-example-reads')
data.status
data.process
data.started
data.finished
data.size
```

You can check list of methods defined for each of the resources in the *reference section*. Note that some attributes and methods are defined in the *BaseResource* and *BaseCollection* classes. *BaseResource* is the parent of all resource classes in `resdk`. *BaseCollection* is the parent of all collection-like classes: *Sample* and *Collection*

Quite commonly, one wants to inspect list of `Data` objects in `Collection` or to know the `Sample` of a given `Data`... For such purposes, there are some handy shortcuts:

- *data.sample*
- *data.collection*
- *sample.data*
- *sample.collection*
- *collection.data*
- *collection.samples*

Download data

Resource classes `Data`, `Sample` and `Collection` have the methods `files()` and `download()`.

The `files()` method returns a list of all files on the resource but does not download anything.

```
# Get data by slug
data = res.data.get('resdk-example-reads')

# Print a list of files
data.files()

# Filter the list of files by file name
data.files(file_name='reads.fastq.gz')

# Filter the list of files by field name
data.files(field_name='output.fastq')
```

The method `download()` downloads the resource files. The optional parameters `file_name` and `field_name` have the same effect as in the `files` method. There is an additional parameter, `download_dir`, that allows you to specify the download directory:

```
# Get sample by slug
sample = res.sample.get('resdk-example')

# Download the FASTQ reads file into current directory
sample.download(
    file_name='reads.fastq.gz',
    download_dir='./',
)
```

3.2.3 Create, modify and organize data

To begin, we need some sample data to work with. You may use your own reads (.fastq) files, or download an example set we have provided:

```
import resdk

res = resdk.Resolwe(url='https://app.genialis.com')
res.login()

# Get example reads
example = res.data.get('resdk-example-reads')
# Download them to current working directory
example.download(
    field_name='fastq',
    download_dir='./',
)
```

Note: To avoid copy-pasting of the commands, you can `download all the code` used in this section.

Organize resources

Before all else, one needs to prepare space for work. In our case, this means creating a "container" where the produced data will reside. So let's create a collection and than put some data inside!

```
# create a new collection object in your running instance of Resolwe (res)
test_collection = res.collection.create(name='Test collection')
```

Upload files

We will upload fastq single end reads with the upload-fastq-single process.

```
# Upload FASTQ reads
reads = res.run(
    slug='upload-fastq-single',
    input={
        'src': './reads.fastq.gz',
    },
    collection=test_collection,
)
```

What just happened? First, we chose a process to run, using its slug `upload-fastq-single`. Each process requires some inputs—in this case there is only one input with name `src`, which is the location of reads on our computer. Uploading a fastq file creates a new `Data` on the server containing uploaded reads. Additionally, we ensured that the new `Data` is put inside `test_collection`.

The upload process also created a Sample object for the reads data to be associated with. You can access it by:

```
reads.sample
```

Note: You can also upload your files by providing url. Just replace path to your local files with the url. This comes handy when your files are large and/or are stored on a remote server and you don't want to download them to your computer just to upload them to Resolwe server again...

Modify data

Both `Data` with reads and `Sample` are owned by you and you have permissions to modify them. For example:

```
# Change name
reads.name = 'My first data'
reads.save()
```

Note the `save()` part! Without this, the change is only applied locally (on your computer). But calling `save()` also takes care that all changes are applied on the server.

Note: Some fields cannot (and should not) be changed. For example, you cannot modify `created` or `contributor` fields. You will get an error if you try.

Annotate Samples and Data

The obvious next thing to do after uploading some data is to annotate it. Annotations are encoded as bundles of descriptors, where each descriptor references a value in a descriptor schema (*i.e.* a template). Annotations for data objects, samples, and collections each follow a different descriptor format. For example, a reads data object can be annotated with the 'reads' descriptor schema, while a sample can be annotated by the 'sample' annotation schema. Each data object that is associated with the sample is also connected to the sample's annotation, so that the annotation for a sample (or collection) represents all Data objects attached to it. Descriptor schemas are described in detail (with accompanying examples) in the Resolwe Bioinformatics documentation.

Here, we show how to annotate the `reads` data object by defining the descriptor information that populates the annotation fields as defined in the 'reads' descriptor schema:

```
# define the chosen descriptor schema
reads.descriptor_schema = 'reads'

# define the descriptor
reads.descriptor = {
    'description': 'Some free text...',
}

# Very important: save changes!
reads.save()
```

We can annotate the sample object using a similar process with a 'sample' descriptor schema:

```
reads.sample.descriptor_schema = 'sample'

reads.sample.descriptor = {
    'general': {
        'description': 'This is a sample...',
```

(continues on next page)

(continued from previous page)

```
        'species': 'Homo sapiens',
        'strain': 'F1 hybrid FVB/N x 129S6/SvEv',
        'cell_type': 'glioblastoma',
    },
    'experiment': {
        'assay_type': 'rna-seq',
        'molecule': 'total_rna',
    },
}

reads.sample.save()
```

Warning: Many descriptor schemas have required fields with a limited set of choices that may be applied as annotations. For example, the 'species' annotation in a sample descriptor must be selected from the list of options in the sample descriptor schema, represented by its Latin name.

We can also define descriptors and descriptor schema directly when calling `res.run` function. This is described in the section about the `run()` method below.

Run analyses

Various bioinformatic processes are available to properly analyze sequencing data. Many of these pipelines are available via Resolwe SDK, and are listed in the Process catalog of the Resolwe Bioinformatics documentation.

After uploading reads file, the next step is to align reads to a genome. We will use STAR aligner, which is wrapped in a process with slug `alignment-star`. Inputs and outputs of this process are described in STAR process catalog. We will define input files and the process will run its algorithm that transforms inputs into outputs.

```
# Get genome
genome_index = res.data.get('resdk-example-genome-index')

alignment = res.run(
    slug='alignment-star',
    input={
        'genome': genome_index,
        'reads': reads,
    },
)
```

Lets take a closer look to the code above. We defined the alignment process, by its slug `'alignment-star'`. For inputs we defined data objects `reads` and `genome`. `Reads` object was created with 'upload-fastq-single' process, while `genome` data object was already on the server and we just used its slug to identify it. The `alignment-star` processor will automatically take the right files from data objects, specified in inputs and create output files: `bam` alignment file, `bai` index and some more...

You probably noticed that we get the result almost instantly, while the typical assembling process runs for hours. This is because processing runs asynchronously, so the returned data object does not have an OK status or outputs when returned.

```
# Get the latest meta data from the server
alignment.update()
```

(continues on next page)

(continued from previous page)

```
# See the process progress
alignment.process_progress

# Print the status of data
alignment.status
```

Status `OK` indicates that processing has finished successfully, but you will also find other statuses. They are given with two-letter abbreviations. To understand their meanings, check the *status reference*. When processing is done, all outputs are written to disk and you can inspect them:

```
# See process output
alignment.output
```

Until now, we used `run()` method twice: to upload reads (yes, uploading files is just a matter of using an upload process) and to run alignment. You can check the full signature of the *run()* method.

Run workflows

Typical data analysis is often a sequence of processes. Raw data or initial input is analysed by running a process on it that outputs some data. This data is fed as input into another process that produces another set of outputs. This output is then again fed into another process and so on. Sometimes, this sequence is so commonly used that one wants to simplify it's execution. This can be done by using so called "workflow". Workflows are special processes that run a stack of processes. On the outside, they look exactly the same as a normal process and have a process slug, inputs… For example, we can run workflow "General RNA-seq pipeline" on our reads:

```
# Run a workflow
res.run(
    slug='workflow-bbduk-star-featurecounts-qc',
    input={
        'reads': reads,
        'genome': res.data.get('resdk-example-genome-index'),
        'annotation': res.data.get('resdk-example-annotation'),
        'rrna_reference': res.data.get('resdk-example-rrna-index'),
        'globin_reference': res.data.get('resdk-example-globin-index'),
    }
)
```

Solving problems

Sometimes the data object will not have an "OK" status. In such case, it is helpful to be able to check what went wrong (and where). The *stdout()* method on data objects can help—it returns the standard output of the data object (as string). The output is long but exceedingly useful for debugging. Also, you can inspect the info, warning and error logs.

```
# Update the data object to get the most recent info
alignment.update()

# Print process' standard output
print(alignment.stdout())
```

(continues on next page)

(continued from previous page)

```
# Access process' execution information
alignment.process_info

# Access process' execution warnings
alignment.process_warning

# Access process' execution errors
alignment.process_error
```

3.3 Topical documentation

Here you can browse through topical documentation about various parts of ReSDK.

3.3.1 Knowledge base

Genialis Knowledge base (KB) is a collection of "features" (genes, transcripts, ...) and "mappings" between these features. It comes very handy when performing various tasks with genomic features e.g.:

- find all aliases of gene `BRCA2`
- finding all genes of type `protein_coding`
- find all transcripts of gene `FHIT`
- converting `gene_id` to `gene_symbol`
- ...

Feature

`Feature` object represents a genomic feature: a gene, a transcript, etc. You can query `Feature` objects by `feature` endpoint, similarly like `Data`, `Sample` or any other ReSDK resource:

```
feature = res.feature.get(feature_id="BRCA2")
```

To examine all attributes of a `Feature`, see the *SDK Reference*. Here we will list a few most commonly used ones:

```
# Get the feature:
feature = res.feature.get(feature_id="BRCA2")

# Database where this feature is defined, e.g. ENSEMBL, UCSC, NCBI, ...
feature.source

# Unique identifier of a feature
feature.feature_id

# Feature species
feature.species

# Feature type, e.g. gene, transcript, exon, ...
feature.type
```

(continues on next page)

(continued from previous page)

```
# Feature name
feature.name

# List of feature aliases
feature.aliases
```

The real power is in the filter capabilities. Here are some examples:

```
# Count number of Human "protein-conding" transcripts in ENSEMBL database
res.feature.filter(
    species="Homo sapiens",
    type="transcript",
    subtype="protein-coding",
    source="ENSEMBL",
).count()

# Convert all gene IDs in a list ``gene_ids`` to gene symbols::
gene_ids = ["ENSG00000139618", "ENSG00000189283"]
genes = res.feature.filter(
    feature_id__in=gene_ids,
    type="gene",
    species="Homo sapiens",
)
mapping = {g.feature_id: g.name for g in genes}
gene_symbols = [mapping[gene_id] for gene_id in gene_ids]
```

Warning: It might look tempting to simplify the last example with:

```
gene_symbols = [g.name for g in genes]
```

Don't do this. The order of entries in the `genes` can be arbitrary and therefore cause that the resulting list `gene_symbols` is not ordered in the same way as `gene_ids`.

Mapping

Mapping is a *connection* between two features. Features can be related in various ways. The type of mapping is indicated by `relation_type` attribute. It is one of the following options:

- `crossdb`: Two features from different sources (databases) that describe same feature. Example: connection for gene BRCA2 between database "UniProtKB" and "UCSC".
- `ortholog`: Two features from different species that describe orthologous gene.
- `transcript`: Connection between gene and it's transcripts.
- `exon`: Connection between gene / transcript and it's exons.

Again, we will only list an example and then let your imagination fly:

```
# Find UniProtKB ID for gene with given ENSEMBL ID:
mapping = res.mapping.filter(
    source_id="ENSG00000189283",
```

(continues on next page)

(continued from previous page)

```
    source_db="ENSEMBL",
    target_db="UniProtKB",
    source_species="Homo sapiens",
    target_species="Homo sapiens",
)
uniprot_id = mapping[0].target_id
```

3.3.2 ReSDK Tables

ReSDK tables are helper classes for aggregating collection data in tabular format. Currently, we have four flavours:

- *RNATables*
- *MethylationTables*
- *MATables*
- *VariantTables*

RNATables

Imagine you are modelling gene expression data from a given collection. Ideally, you would want all expression values organized in a table where rows represents samples and columns represent genes. Class `RNATables` gives you just that (and more).

We will present the functionality of `RNATables` through an example. We will:

- Create an instance of `RNATables` and examine it's attributes
- Fetch raw expressions and select TIS signature genes with sufficient coverage
- Normalize expression values (log-transform) and visualize samples in a simple PCA plot

First, connect to a Resolwe server, pick a collection and create and instance of `RNATables`:

```
import resdk
from resdk.tables import RNATables
res = resdk.Resolwe(url='https://app.genialis.com/')
res.login()
collection = res.collection.get("sum149-fresh-for-rename")
sum149 = RNATables(collection)
```

Object `sum149` is an instance of `RNATables` and has many attributes. For a complete list see the *SDK Reference*, here we list the most commonly used ones:

```
# Expressions raw counts
sum149.rc

# Expressions normalized counts
sum149.exp
# See normalization method
sum149.exp.attrs["exp_type"]

# Sample metadata
sum149.meta
```

(continues on next page)

(continued from previous page)

```
# Sample QC metrics
sum149.qc

# Dictionary that maps gene ID's into gene symbols
sum149.id_to_symbol
# This is handy to rename column names (gene ID's) to gene symbols
sum149.rc.rename(columns=sum149.id_to_symbol)
```

Note: Expressions and metadata are cached in memory as well as on disk. At each time they are re-requested a check is made that local and server side of data is synced. If so, cached data is used. Otherwise, new data will be pulled from server.

In our example we will only work with a set of TIS signature genes:

```
TIS_GENES = ["CD3D", "IDO1", "CIITA", "CD3E", "CCL5", "GZMK", "CD2", "HLA-DRA", "CXCL13",
↪ "IL2RG", "NKG7", "HLA-E", "CXCR6", "LAG3", "TAGAP", "CXCL10", "STAT1", "GZMB"]
```

We will identify low expressed genes and only keep the ones with average raw expression above 20:

```
tis_rc = sum149.rc.rename(columns=sum149.id_to_symbol)[TIS_GENES]
mean = tis_rc.mean(axis=0)
high_expressed_genes = mean.loc[mean > 20].index
```

Now, lets select TPM normalized expressions and keep only highly expressed tis genes. We also transform to `log2(TPM + 1)`:

```
import numpy as np
tis_tpm = sum149.exp.rename(columns=sum149.id_to_symbol)[high_expressed_genes]
tis_tpm_log = np.log(tis_tpm + 1)
```

Finally, we perform PCA and visualize the results:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2, whiten=True)
Y = pca.fit_transform(tis_tpm_log)

import matplotlib.pyplot as plt
for ((x, y), sample_name) in zip(Y, tis_tpm.index):
    plt.plot(x, y, 'bo')
    plt.text(x, y, sample_name)
plt.xlabel(f"PC1 ({pca.explained_variance_ratio_[0]})")
plt.ylabel(f"PC2 ({pca.explained_variance_ratio_[1]})")
plt.show()
```

MethylationTables

Similar as `RNATables` provide access to raw counts and normalized expression values of RNA data, `MethylationTables` allow for fast access of beta and m-values of methylation data:

```
meth = resdk.tables.MethylationTables(<collection-with-methylation-data>)

# Methylation beta-values
meth.beta

# Methylation m-values
meth.mval
```

MATables

Similar as `RNATables` provide access to raw counts and normalized expression values of RNA data, `MATables` allow for fast access of expression values per probe of microarray:

```
ma = resdk.tables.MATables(<collection-with-microarray-data>)

# Microarray expressions values (columns are probe ID's)
ma.exp
```

VariantTables

Similar as `RNATables` provide access to raw counts and normalized expression values of RNA data, `VariantTables` allow for fast access of variant data present in Data of type `data:mutationstable`:

```
vt = resdk.tables.VariantTables(<collection-with-variant-data>)
vt.variants
```

The output of the above would look something like this:

sample_id	chr1_123_C>T_Gly11Asp	chr1_126_T>C_Asp12Gly
101	2	0
102	0	1

In rows, there are sample ID's. In columns there are variants where each variant is given as: `<chromosome>_<position>_<nucleotide-change>_<amino-acid-change>`. Values in table can be 0 (no mutation), 1 (heterozygous mutation) or 2 (homozygous mutation).

The above example gives an ideal situation where the mutation status for each position is known. However, this is not always the case.

Missing values and `discard_fakes` argument

Very often, there is no info about a certain variant / sample, so values can also be `NaN` (unknown). Other common case is just the info that there is no mutation on a given position. This is a valid information also. Given the above, a more realistic example of output is:

sample_id	chr1_123_C>T_Gly11Asp	chr1_126_T>C_Asp12Gly	chr1_127
101	2	NaN	0
102	0	1	NaN

One can se that for some combination of variants / samples there is no information: a value in table is `NaN`. It is up to a user if this is interpreted as no variant or something else. In the first case, one can quickly convert `NaN` to 0 with `vt.variants.fillna(0)`. One can also see that there is a column (chr1_127) that is not actually representing a variant. One may call this a "fake" variant. It is a way of signalling the absence of variant on a given position. Usually this is not useful, but is some cases it is. If you would like your output to contain such fake variants please specify `discard_fakes=False` in `VariantTables` constructor.

Inspecting depth

The reason for NaN values may be that the read depth on certain position is too low for GATK to reliably call a variant. In such case, it is worth inspecting the depth or depth per base:

```
# Similar as above but one gets depth on particular variant / sample
vt.depth
# One can also get depth for specific base
vt.depth_a
vt.depth_c
vt.depth_t
vt.depth_g
```

Filtering mutations

Process `mutations-table` accepts an input `mutations` which specifies the gene (and optionally amino acid change) of interest. It restricts the scope of mutation to just a given gene or amino acid.

However, it can happen that not all the samples have the same `mutations` input. In such cases, it makes little sense to merge the information about mutations from multiple samples. By default, `VariantTables` checks that all Data is computed with same `mutations` input. If this is not true, it will raise an error.

But if you provide additional argument `mutations` it will limit the mutations to only those in the given gene. An example:

```
# Sample 101 has mutations input "FHIT, BRCA2"
# Sample 102 has mutations input "BRCA2"

# This would cause error, since the mutations inputs are not the same
vt = resdk.tables.VariantTables(<collection>)
vt.variants

# This would limit the variants to just the ones in BRCA2 gene.
vt = resdk.tables.VariantTables(<collection> mutations=["BRCA2"])
vt.variants
```

3.3.3 Genesets

Geneset is a special kind of `Data` resource. In addition to all of the functionality of `Data`, it also has `genes` attribute and support for set-like operations (intersection, union, etc...).

In the most common case, genesets exist somewhere on Resolwe server and user just fetches them:

```
# Get one geneset by slug
gs = res.geneset.get("my-slug")

# Get all human genesets in a given collection:
genesets = res.geneset.filter(collection=<my-collection>, species="Homo sapiens"):
```

What one gets is an object (or list of them) of type `Geneset`. This object has all the attributes of `Data` plus some additional ones:

```
# Set of genes in the geneset:
gs.genes
# Source of genes, e.g. ENSEMBL, UCSC, NCBI...
gs.source
# Species of the genes in the geneset
gs.species
```

A common thing to do with `Geneset` objects is to perform set-like operations on them to create new `Geneset`. This is easily done with exactly the same syntax as for Python `set` objects:

```
gs1 = res.geneset.get("slug-1")
gs2 = res.geneset.get("slug-2")

# Union
gs3 = gs1 | gs2
# Intersection
gs3 = gs1 & gs2
# Difference
gs3 = gs1 - gs2
```

Note: Performing these operations is only possible on genesets that have equal values of `species` and `source` attribute. Otherwise newly created sets would not make sense and would be inconsistent.

So far, geneset `gs3` only exists locally. One can easily save it to Resolwe server:

```
gs3.save()
# As with Data, it is a good practice to include it in a collection:
gs3.collection = <my_collection>
gs.save()
```

Alternative way of creating genesets is to use `Resolwe.geneset.create` method. In such case, you need to enter the `genes`, `species` and `source` information manually:

```
res.geneset.create(genes=["MYC", "FHT"], source="UCSC", species="Homo sapiens")
```

3.3.4 Metadata

Samples are normally annotated with the use of `descriptor` and `descriptor_schema`. However in some cases the fields defined in `DescriptorSchema` do not suffice and it comes handy to upload sample annotations in a table where each row holds information about some sample in collection. In general, there can be multiple rows referring to the same sample in the collection (for example one sample received two or more distinct treatments). In such cases one can upload this tables with the process Metadata table. However, quite often there is exactly one-on-one mapping between rows in such table and samples in collection. In such case, please use the "unique" flavour of the above process, Metadata table (one-to-one).

Metadata in ReSDK is just a special kind of `Data` resource that simplifies retrieval of the above mentioned tables. In addition to all of the functionality of `Data`, it also has two additional attributes: `df` and `unique`:

```
# The "df" attribute is pandas.DataFrame of the output named "table"
# The index of df are sample ID's
m.df
# Attribute "unique" is signalling if this is metadata is unique or not
m.unique
```

Note: Behind the scenes, `df` is not an attribute but rather a property. So it has getter and setter methods (`get_df` and `set_df`). This comes handy if the default parsing logic does not suffice. In such cases you can provide your own parser and keyword arguments for it. Example:

```
import pandas
m.get_df(parser=pandas.read_csv, sep="\t", skiprows=[1, 2, 3])
```

In the most common case, Metadata objects exist somewhere on Resolwe server and user just fetches them:

```
# Get one metadata by slug
m = res.metadata.get("my-slug")

# Filter metadata by some conditions, e.g. get all metadata
# from a given collection:
ms = res.metadata.filter(collection=<my-collection>):
```

Sometimes, these objects need to be updated, and one can easily do that. However, `df` and `unique` are upload protected - they can be set during object creation but cannot be set afterwards:

```
m.unique = False  # Will fail on already existing object
m.df = <new-df>  # Will fail on already existing object
```

Sometimes one wishes to create a new Metadata. This can be achieved in the same manner as for other ReSDK resources:

```
m = res.metadata.create(df=<my-df>, collection=<my-collection>)

# Creating metadata without specifying  df / collection will fail
m = res.metdata.create()  # Fail
m = res.metdata.create(collection=<my-collection>)  # Fail
m = res.metdata.create(df=<my-df>)  # Fail
```

Alternatively, one can also build this object gradually from scratch and than call `save()`:

```
m = Metadata(resolwe=<resolwe>)
m.collection = <my-collection>
my_df = m.set_index(<my-df>)
m.df = my_df
m.save()
```

where `m.set_index(<my-df>)` is a helper function that finds `Sample name/slug/ID` column or index name, maps it to `Sample ID` and sets it as index. This function is recommended to use because the validation step is trying to match `m.df` index with `m.collection` sample ID's.

Deleting Metadata works the same as for any other resource. Be careful, this cannot be undone and you need to have sufficient permissions:

```
m.delete()
```

3.4 SDK Reference

3.4.1 Resolwe

class resdk.**Resolwe**(*username=None*, *password=None*, *url=None*)

Connect to a Resolwe server.

Parameters

- **username** (*str*) – user's username
- **password** (*str*) – user's password
- **url** (*str*) – Resolwe server instance

data_usage(***query_params*)

Get per-user data usage information.

Display number of samples, data objects and sum of data object sizes for currently logged-in user. For admin users, display data for **all** users.

get_or_run(*slug=None*, *input={}*)

Return existing object if found, otherwise create new one.

Parameters

- **slug** (*str*) – Process slug (human readable unique identifier)
- **input** (*dict*) – Input values

get_query_by_resource(*resource*)

Get ResolweQuery for a given resource.

login(*username=None*, *password=None*)

Interactive login.

Ask the user to enter credentials in command prompt. If username / email and password are given, login without prompt.

run(*slug=None*, *input={}*, *descriptor=None*, *descriptor_schema=None*, *collection=None*, *data_name=''*, *process_resources=None*)

Run process and return the corresponding Data object.

1. Upload files referenced in inputs
2. Create Data object with given inputs
3. Command is run that processes inputs into outputs
4. Return Data object

The processing runs asynchronously, so the returned Data object does not have an OK status or outputs when returned. Use data.update() to refresh the Data resource object.

Parameters

- **slug** (*str*) – Process slug (human readable unique identifier)
- **input** (*dict*) – Input values
- **descriptor** (*dict*) – Descriptor values
- **descriptor_schema** (*str*) – A valid descriptor schema slug
- **collection** (*int/resource*) – Collection resource or it's id into which data object should be included
- **data_name** (*str*) – Default name of data object
- **process_resources** (*dict*) – Process resources

Returns data object that was just created

Return type Data object

3.4.2 Resolwe Query

class resdk.**ResolweQuery**(*resolwe*, *resource*, *slug_field='slug'*)

Query resource endpoints.

A Resolwe instance (for example "res") has several endpoints:

- res.data
- res.collection
- res.sample
- res.process
- ...

Each such endpoint is an instance of the ResolweQuery class. ResolweQuery supports queries on corresponding objects, for example:

```
res.data.get(42)  # return Data object with ID 42.
res.sample.filter(contributor=1)  # return all samples made by contributor 1
```

This object is lazy loaded which means that actual request is made only when needed. This enables composing multiple filters, for example:

```
res.data.filter(contributor=1).filter(name='My object')
```

is the same as:

```
res.data.filter(contributor=1, name='My object')
```

This is especially useful, because all endpoints at Resolwe instance are such queries and can be filtered further before transferring any data.

To get a list of all supported query parameters, use one that does not exist and you will et a helpful error message with a list of allowed ones.

```
res.data.filter(foo="bar")
```

all()

Return copy of the current queryset.

This is handy function to get newly created query without any filters.

clear_cache()

Clear cache.

count()

Return number of objects in current query.

create(***model_data*)

Return new instance of current resource.

delete(*force=False*)

Delete objects in current query.

Parameters **force** (*bool*) – Do not trigger confirmation prompt. WARNING: Be sure that you really know what you are doing as deleted objects are not recoverable.

filter(***filters*)

Return clone of current query with added given filters.

get(**args*, ***kwargs*)

Get object that matches given parameters.

If only one non-keyworded argument is given, it is considered as id if it is number and as slug otherwise.

Parameters **uid** (*int for ID or string for slug*) – unique identifier - ID or slug

Return type object of type self.resource

Raises

- **ValueError** – if non-keyworded and keyworded arguments are combined or if more than one non-keyworded argument is given
- **LookupError** – if none or more than one objects are returned

iterate(*chunk_size=100*)

Iterate through query.

This can come handy when one wishes to iterate through hundreds or thousands of objects and would otherwise get "504 Gateway-timeout".

The method cannot be used together with the following filters: limit, offset and ordering, and will raise a `ValueError`.

search(*text*)

Full text search.

3.4.3 Resources

Resource classes

class resdk.resources.base.**BaseResource**(*resolwe*, ***model_data*)

Abstract resource.

One and only one of the identifiers (slug, id or model_data) should be given.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

delete(*force=False*)

Delete the resource object from the server.

Parameters **force** (*bool*) – Do not trigger confirmation prompt. WARNING: Be sure that you really know what you are doing as deleted objects are not recoverable.

classmethod fetch_object(*resolwe*, *id=None*, *slug=None*)

Return resource instance that is uniquely defined by identifier.

fields()

Resource fields.

id

unique identifier of an object

save()

Save resource to the server.

update()

Update resource fields from the server.

class resdk.resources.base.**BaseResolweResource**(*resolwe*, ***model_data*)

Base class for Resolwe resources.

One and only one of the identifiers (slug, id or model_data) should be given.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property contributor

Contributor.

property created

Creation time.

current_user_permissions

current user permissions

property modified

Modification time.

name

name of resource

property permissions

Permissions.

slug

human-readable unique identifier

update()

Clear permissions cache and update the object.

version

resource version

class resdk.resources.**Data**(*resolwe*, ***model_data*)

Resolwe Data resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

checksum

checksum field calculated on inputs

property children

Get children of this Data object.

property collection

Get collection.

descriptor

annotation data, with the form defined in descriptor_schema

descriptor_dirty

indicate whether *descriptor* doesn't match *descriptor_schema* (is dirty)

property descriptor_schema

Get descriptor schema.

download(*file_name=None*, *field_name=None*, *download_dir=None*)

Download Data object's files and directories.

Download files and directoriesfrom the Resolwe server to the download directory (defaults to the current working directory).

Parameters

- **file_name** (*string*) – name of file or directory
- **field_name** (*string*) – file or directory field name
- **download_dir** (*string*) – download path

Return type None

Data objects can contain multiple files and directories. All are downloaded by default, but may be filtered by name or output field:

- re.data.get(42).download(file_name='alignment7.bam')
- re.data.get(42).download(field_name='bam')

duplicate(*inherit_collection=False*)

Duplicate (make copy of) `data` object.

Parameters **inherit_collection** – If `True` then duplicated data will be added to collection of the original data.

Returns Duplicated data object

duplicated

duplicated

files(*file_name=None*, *field_name=None*)

Get list of downloadable file fields.

Filter files by file name or output field.

Parameters

- **file_name** (*string*) – name of file
- **field_name** (*string*) – output field name

Return type List of tuples (data_id, file_name, field_name, process_type)

property finished

Get finish time.

input

actual input values

output

actual output values

property parents

Get parents of this Data object.

property process

Get process.

process_cores

process cores

process_error

error log message (list of strings)

process_info

info log message (list of strings)

process_memory

process memory

process_progress

process progress in percentage

process_rc

Process algorithm return code

process_resources

process_resources

process_warning

warning log message (list of strings)

property sample

Get sample.

scheduled

scheduled

size

size

property started

Get start time.

status

process status - Possible values: UP (Uploading - for upload processes), RE (Resolving - computing input data objects) WT (Waiting - waiting for process since the queue is full) PP (Preparing - preparing the environment for processing) PR (Processing) OK (Done) ER (Error) DR (Dirty - Data is dirty)

stdout()

Return process standard output (stdout.txt file content).

Fetch stdout.txt file from the corresponding Data object and return the file content as string. The string can be long and ugly.

Return type string

tags

data object's tags

update()

Clear cache and update resource fields from the server.

class resdk.resources.collection.**BaseCollection**(*resolwe*, ***model_data*)

Abstract collection resource.

One and only one of the identifiers (slug, id or model_data) should be given.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property data

Return list of attached Data objects.

data_types()

Return a list of data types (process_type).

Return type List

description

description

descriptor

descriptor

descriptor_dirty

descriptor_dirty

property descriptor_schema

Descriptor schema.

download(*file_name=None*, *field_name=None*, *download_dir=None*)

Download output files of associated Data objects.

Download files from the Resolwe server to the download directory (defaults to the current working directory).

Parameters

- **file_name** (*string*) – name of file
- **field_name** (*string*) – field name
- **download_dir** (*string*) – download path

Return type None

Collections can contain multiple Data objects and Data objects can contain multiple files. All files are downloaded by default, but may be filtered by file name or Data object type:

- re.collection.get(42).download(file_name='alignment7.bam')
- re.collection.get(42).download(data_type='bam')

duplicated

duplicatied

files(*file_name=None*, *field_name=None*)

Return list of files in resource.

settings

settings

tags

tags

update()

Clear cache and update resource fields from the server.

class resdk.resources.**Collection**(*resolwe*, ***model_data*)

Resolwe Collection resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property data

Return list of data objects on collection.

duplicate()

Duplicate (make copy of) `collection` object.

Returns Duplicated collection

property relations

Return list of data objects on collection.

property samples

Return list of samples on collection.

update()

Clear cache and update resource fields from the server.

class resdk.resources.**Sample**(*resolwe*, ***model_data*)

Resolwe Sample resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property background

Get background sample of the current one.

property collection

Get collection.

property data

Get data.

duplicate(*inherit_collection=False*)

Duplicate (make copy of) `sample` object.

Parameters **inherit_collection** – If `True` then duplicated samples (and their data) will be added to collection of the original sample.

Returns Duplicated sample

property is_background

Return `True` if given sample is background to any other and `False` otherwise.

property relations

Get `Relation` objects for this sample.

update()

Clear cache and update resource fields from the server.

update_descriptor(*descriptor*)

Update descriptor and descriptor_schema.

class resdk.resources.**Relation**(*resolwe*, ***model_data*)

Resolwe Relation resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

add_sample(*sample*, *label=None*, *position=None*)

Add `sample` object to relation.

category

category of the relation

property collection

Return collection object to which relation belongs.

partitions

list of `RelationPartition` objects in the `Relation`

remove_samples(**samples*)

Remove `sample` objects from relation.

property samples

Return list of sample objects in the relation.

save()

Check that collection is saved and save instance.

type

type of the relation

unit(*where applicable*, *e.g. for serieses*)

unit (where applicable, e.g. for serieses)

update()

Clear cache and update resource fields from the server.

class resdk.resources.**Process**(*resolwe*, ***model_data*)

Resolwe Process resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

category

used to group processes in a GUI. Examples: `upload:`, `analyses:variants:`, ...

data_name

the default name of data object using this process. When data object is created you can assign a name to it. But if you don't, the name of data object is determined from this field. The field is a expression which can take values of other fields.

description

process description

entity_always_create

entity_always_create

entity_descriptor_schema

entity_descriptor_schema

entity_input

entity_input

entity_type

entity_type

input_schema

specifications of inputs

is_active

Boolean stating wether process is active

output_schema

specification of outputs

persistence

Measure of how important is to keep the process outputs when optimizing disk usage. Options: RAW/CACHED/TEMP. For processes, used on frontend use TEMP - the results of this processes can be quickly re-calculated any time. For upload processes use RAW - this data should never be deleted, since it cannot be re-calculated. For analysis use CACHED - the results can stil be calculated from imported data but it can take time.

print_inputs()

Pretty print input_schema.

priority

process priority - not used yet

requirements

required Docker image, amount of memory / CPU ...

run

the heart of process - here the algorithm is defined.

scheduling_class

Scheduling class

type

the type of process "type:sub_type:sub_sub_type:..."

class resdk.resources.**DescriptorSchema**(*resolwe*, ***model_data*)

Resolwe DescriptorSchema resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

description

description

schema

schema

class resdk.resources.**User**(*resolwe=None*, ***model_data*)

Resolwe User resource.

One and only one of the identifiers (slug, id or model_data) should be given.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

first_name

user's first name

get_name()

Return user's name.

class resdk.resources.**Group**(*resolwe=None*, ***model_data*)

Resolwe Group resource.

One and only one of the identifiers (slug, id or model_data) should be given.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

add_users(**users*)

Add users to group.

name

group's name

remove_users(**users*)

Remove users from group.

update()

Clear cache and update resource fields from the server.

property users

Return list of users in group.

class resdk.resources.**Geneset**(*resolwe*, *genes=None*, *source=None*, *species=None*, ***model_data*)

Resolwe Geneset resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property genes

Get genes.

save()

Save Geneset to the server.

If Geneset is already on the server update with save() from base class. Otherwise, create a new Geneset by running process with slug "create-geneset".

set_operator(*operator*, *other*)

Perform set operations on Geneset object by creating a new Genseset.

Parameters

- **operator** – string -> set operation function name
- **other** – Geneset object

Returns new Geneset object

property source

Get source.

property species

Get species.

class resdk.resources.**Metadata**(*resolwe*, ***model_data*)

Metadata resource.

Parameters

- **resolwe** (*Resolwe object*) – Resolwe instance
- **model_data** – Resource model data

property df

Get table as pd.DataFrame.

property df_bytes

Get file contents of table output in bytes form.

get_df(*parser=None*, ***kwargs*)

Get table as pd.DataFrame.

save()

Save Metadata to the server.

If Metadata is already uploaded: update. Otherwise, create new one.

set_df(*value*)

Set df.

set_index(*df*)

Set index of df to Sample ID.

If there is a column with `Sample ID` just set that as index. If there is `Sample name` or `Sample slug` column, map sample name / slug to sample ID's and set ID's as an index. If no suitable column in there, raise an error. Works also if any of the above options is already an index with appropriate name.

property unique

Get unique attribute.

This attribute tells if Metadata has one-to-one or one-to-many relation to collection samples.

validate_df(*df*)

Validate df property.

Validates that df:

- is an instance of pandas.DataFrame
- index contains sample IDs that match some samples:
  - If not matches, raise warning
  - If there are samples in df but not in collection, raise warning
  - If there are samples in collection but not in df, raise warning

class resdk.resources.kb.**Feature**(*resolwe*, ***model_data*)

Knowledge base Feature resource.

aliases

Aliases

description

Description

feature_id

Feature ID

full_name

Full name

name

Name

source

Source

species

Species

sub_type

Feature subtype (tRNA, protein coding, rRNA, ...)

type

Feature type (gene, transcript, exon, ...)

class resdk.resources.kb.**Mapping**(*resolwe*, ***model_data*)

Knowledge base Mapping resource.

source_db

Source database

source_id

Source feature ID

source_species

Source feature species

target_db

Target database

target_id

Target feature ID

target_species

Target feature species

Permissions

Resources like *resdk.resources.Data*, *resdk.resources.Collection*, *resdk.resources.Sample*, and *resdk.resources.Process* include a *permissions* attribute to manage permissions. The *permissions* attribute is an instance of *resdk.resources.permissions.PermissionsManager*.

class resdk.resources.permissions.**PermissionsManager**(*all_permissions*, *api_root*, *resolwe*)

Helper class to manage permissions of the `BaseResource`.

clear_cache()

Clear cache.

copy_from(*source*)

Copy permissions from some other object to self.

property editors

Get users with `edit` permission.

fetch()

Fetch permissions from server.

property owners

Get users with `owner` permission.

set_group(*group*, *perm*)

Set `perm` permission to `group`.

When assigning permissions, only the highest permission needs to be given. Permission hierarchy is:

- none (no permissions)
- view
- edit
- share
- owner

Some examples:

```
collection = res.collection.get(...)
# Add share, edit and view permission to BioLab:
collection.permissions.set_group('biolab', 'share')
# Remove share and edit permission from BioLab:
collection.permissions.set_group('biolab', 'view')
# Remove all permissions from BioLab:
collection.permissions.set_group('biolab', 'none')
```

set_public(*perm*)

Set `perm` permission for public.

Public can only get two sorts of permissions:

- none (no permissions)
- view

Some examples:

```
collection = res.collection.get(...)
# Add view permission to public:
collection.permissions.set_public('view')
# Remove view permission from public:
collection.permissions.set_public('none')
```

set_user(*user*, *perm*)

Set `perm` permission to `user`.

When assigning permissions, only the highest permission needs to be given. Permission hierarchy is:

- none (no permissions)
- view
- edit
- share
- owner

Some examples:

```
collection = res.collection.get(...)
# Add share, edit and view permission to John:
collection.permissions.set_user('john', 'share')
```

(continues on next page)

(continued from previous page)

```
# Remove share and edit permission from John:
collection.permissions.set_user('john', 'view')
# Remove all permissions from John:
collection.permissions.set_user('john', 'none')
```

property viewers

Get users with `view` permission.

Utility functions

Resource utility functions.

resdk.resources.utils.**fill_spaces**(*word*, *desired_length*)

Fill spaces at the end until word reaches desired length.

resdk.resources.utils.**flatten_field**(*field*, *schema*, *path*)

Reduce dicts of dicts to dot separated keys.

Parameters

- **field** (*dict*) – Field instance (e.g. input)
- **schema** (*dict*) – Schema instance (e.g. input_schema)
- **path** (*string*) – Field path

Returns flattened instance

Return type dictionary

resdk.resources.utils.**get_collection_id**(*collection*)

Return id attribute of the object if it is collection, otherwise return given value.

resdk.resources.utils.**get_data_id**(*data*)

Return id attribute of the object if it is data, otherwise return given value.

resdk.resources.utils.**get_descriptor_schema_id**(*dschema*)

Get descriptor schema id.

Return id attribute of the object if it is descriptor schema, otherwise return given value.

resdk.resources.utils.**get_process_id**(*process*)

Return id attribute of the object if it is process, otherwise return given value.

resdk.resources.utils.**get_relation_id**(*relation*)

Return id attribute of the object if it is relation, otherwise return given value.

resdk.resources.utils.**get_sample_id**(*sample*)

Return id attribute of the object if it is sample, otherwise return given value.

resdk.resources.utils.**get_user_id**(*user*)

Return id attribute of the object if it is relation, otherwise return given value.

resdk.resources.utils.**is_collection**(*collection*)

Return `True` if passed object is Collection and `False` otherwise.

resdk.resources.utils.**is_data**(*data*)

Return `True` if passed object is Data and `False` otherwise.

resdk.resources.utils.**is_descriptor_schema**(*data*)

Return `True` if passed object is DescriptorSchema and `False` otherwise.

resdk.resources.utils.**is_group**(*group*)

Return `True` if passed object is Group and `False` otherwise.

resdk.resources.utils.**is_process**(*process*)

Return `True` if passed object is Process and `False` otherwise.

resdk.resources.utils.**is_relation**(*relation*)

Return `True` if passed object is Relation and `False` otherwise.

resdk.resources.utils.**is_sample**(*sample*)

Return `True` if passed object is Sample and `False` otherwise.

resdk.resources.utils.**is_user**(*user*)

Return `True` if passed object is User and `False` otherwise.

resdk.resources.utils.**iterate_fields**(*fields*, *schema*)

Recursively iterate over all DictField sub-fields.

Parameters

- **fields** (*dict*) – Field instance (e.g. input)
- **schema** (*dict*) – Schema instance (e.g. input_schema)

resdk.resources.utils.**iterate_schema**(*fields*, *schema*, *path=None*)

Recursively iterate over all schema sub-fields.

Parameters

- **fields** (*dict*) – Field instance (e.g. input)
- **schema** (*dict*) – Schema instance (e.g. input_schema)

Path schema Field path

Path schema string

resdk.resources.utils.**parse_resolwe_datetime**(*dtime*)

Convert string representation of time to local datetime.datetime object.

3.4.4 ReSDK Tables

Helper classes for aggregating collection data in tabular format.

Table classes

class resdk.tables.rna.**RNATables**(*collection: resdk.resources.collection.Collection*, *cache_dir: Optional[str] = None*, *progress_callable: Optional[Callable] = None*, *expression_source: Optional[str] = None*, *expression_process_slug: Optional[str] = None*)

A helper class to fetch collection's expression and meta data.

This class enables fetching given collection's data and returning it as tables which have samples in rows and expressions/metadata in columns.

When calling *RNATables.exp*, *RNATables.rc* and RNATables.meta for the first time the corresponding data gets downloaded from the server. This data than gets cached in memory and on disc and is used in consequent calls. If the data on the server changes the updated version gets re-downloaded.

A simple example:

```
# Get Collection object
collection = res.collection.get("collection-slug")

# Fetch collection expressions and metadata
tables = RNATables(collection)
exp = tables.exp
rc = tables.rc
meta = tables.meta
```

check_heterogeneous_collections()

Ensure consistency among expressions.

property exp: pandas.core.frame.DataFrame

Return expressions table as a pandas DataFrame object.

Which type of expressions (TPM, CPM, FPKM, ...) get returned depends on how the data was processed. The expression type can be checked in the returned table attribute *attrs['exp_type']*:

```
exp = tables.exp
print(exp.attrs['exp_type'])
```

Returns table of expressions

property id_to_symbol: Dict[str, str]

Map of source gene ids to symbols.

property rc: pandas.core.frame.DataFrame

Return expression counts table as a pandas DataFrame object.

Returns table of counts

property readable_columns: Dict[str, str]

Map of source gene ids to symbols.

This also gets fetched only once and then cached in memory and on disc. *RNATables.exp* or *RNATables.rc* must be called before this as the mapping is specific to just this data. Its intended use is to rename table column labels from gene ids to symbols.

Example of use:

```
exp = exp.rename(columns=tables.id_to_symbol)
```

Returns dict with gene ids as keys and gene symbols as values

class resdk.tables.methylation.**MethylationTables**(*collection:* resdk.resources.collection.Collection, *cache_dir: Optional[str] = None*, *progress_callable: Optional[Callable] = None*)

A helper class to fetch collection's methylation and meta data.

This class enables fetching given collection's data and returning it as tables which have samples in rows and methylation/metadata in columns.

A simple example:

```
# Get Collection object
collection = res.collection.get("collection-slug")

# Fetch collection methylation and metadata
tables = MethylationTables(collection)
meta = tables.meta
beta = tables.beta
m_values = tables.mval
```

property beta: pandas.core.frame.DataFrame

Return beta values table as a pandas DataFrame object.

property mval: pandas.core.frame.DataFrame

Return m-values as a pandas DataFrame object.

3.4.5 Exceptions

Custom ReSDK exceptions.

class resdk.exceptions.**ValidationError**

An error while validating data.

3.4.6 Logging

Module contents:

1. Parent logger for all modules in resdk library
2. Handler STDOUT_HANDLER is "turned off" by default
3. Handler configuration functions
4. Override sys.excepthook to log all uncaught exceptions

Parent logger

Loggers in resdk are named by their module name. This is achieved by:

```
logger = logging.getLogger(__name__)
```

This makes it easy to locate the source of a log message.

Logging handlers

The handler STDOUT_HANDLER is created but not automatically added to ROOT_LOGGER, which means they do not do anything. The handlers are activated when users call logger configuration functions like `start_logging()`.

Handler configuration functions

As a good logging practice, the library does not register handlers by default. The reason is that if the library is included in some application, developers of that application will probably want to register loggers by themself. Therefore, if a user wishes to register the pre-defined handlers she can run:

```
import resdk
resdk.start_logging()
```

resdk_logger.**start_logging**(*logging_level=logging.INFO*)

Start logging resdk with the default configuration.

Parameters **logging_level** (*int*) – logging threshold level - integer in [0-50]

Return type None

Logging levels:

- logging.DEBUG(10)
- logging.INFO(20)
- logging.WARNING(30)
- logging.ERROR(40)
- logging.CRITICAL(50)

resdk_logger.**log_to_stdout**(*level=None*)

Configure logging to stdout.

Parameters

- **is_on** (*bool*) – If True, log to standard output
- **level** (*int*) – logging threshold level - integer in [0-50]

Return type None

Log uncaught exceptions

All python exceptions are handled by function, stored in `sys.excepthook`. By rewriting the default implementation, we can modify it for our puruses - to log all uncaught exceptions.

Note#1: Modified behaviour (logging of all uncaught exceptions) applies only when runing in non-interactive mode.

Note#2: Any exception can be caught/uncaught and it can happen in interactive/non-interactive mode. This makes 4 different scenarios. The sys.excepthook modification takes care of uncaught exceptions in non-interactive mode. In interactive mode, user is notified directly if exception is raised. If exception is caught and not reraised, it should be logged somehow, since it can provide valuable information for developer when debugging. Therefore, we should use the following convention for logging in resdk: "Exceptions are explicitly logged only when they are caught and not re-raised."

3.5 Contributing

3.5.1 Installing prerequisites

Make sure you have Python 3.7+ installed on your system. If you don't have it yet, follow these instructions.

3.5.2 Preparing environment

Fork the main Resolwe SDK for Python git repository.

If you don't have Git installed on your system, follow these instructions.

Clone your fork (replace <username> with your GitHub account name) and change directory:

```
git clone https://github.com/<username>/resolwe-bio-py.git
cd resolwe-bio-py
```

Prepare Resolwe SDK for Python for development:

```
pip install -e .[docs,package,test]
```

Note: We recommend using venv to create an isolated Python environment.

3.5.3 Running tests

Run unit tests:

```
py.test
```

3.5.4 Coverage report

To see the tests' code coverage, use:

```
py.test --cov=resdk
```

To generate an HTML file showing the tests' code coverage, use:

```
py.test --cov=resdk --cov-report=html
```

3.5.5 Building documentation

```
python setup.py build_sphinx
```

3.5.6 Preparing release

Checkout the latest code and create a release branch:

```
git checkout master
git pull
git checkout -b release-<new-version>
```

Replace the *Unreleased* heading in `docs/CHANGELOG.rst` with the new version, followed by release's date (e.g. *13.2.0 - 2018-10-23*).

Commit changes to git:

```
git commit -a -m "Prepare release <new-version>"
```

Push changes to your fork and open a pull request:

```
git push --set-upstream <resdk-fork-name> release-<new-version>
```

Wait for the tests to pass and the pull request to be approved. Merge the code to master:

```
git checkout master
git merge --ff-only release-<new-version>
git push <resdk-upstream-name> master <new-version>
```

Tag the new release from the latest commit:

```
git checkout master
git tag -sm "Version <new-version>" <new-version>
```

Push the tag to the main ReSDK's git repository:

```
git push <resdk-upstream-name> master <new-version>
```

Now you can release the code on PyPI. Clean `build` directory:

```
python setup.py clean -a
```

Remove previous distributions in `dist` directory:

```
rm dist/*
```

Remove previous `egg-info` directory:

```
rm -r *.egg-info
```

Create source distribution:

```
python setup.py sdist
```

Build wheel:

```
python setup.py bdist_wheel
```

Upload distribution to PyPI:

```
twine upload dist/*
```

PYTHON MODULE INDEX

r

INDEX

E

F

G

I

L

M

N

O

P

R

S

T

U

V