D-Wave Cloud Client

Release 0.5.4

May 06, 2019

Contents

D-Wave Cloud Client is a minimal implementation of the REST interface used to communicate with D-Wave Sampler API (SAPI) servers.

SAPI is an application layer built to provide resource discovery, permissions, and scheduling for quantum annealing resources at D-Wave Systems. This package provides a minimal Python interface to that layer without compromising the quality of interactions and workflow.

CHAPTER 1

Example

This example instantiates a D-Wave Cloud Client and solver based on the local system's auto-detected default configuration file and samples a random Ising problem tailored to fit the solver's graph.

```
import random
from dwave.cloud import Client

# Connect using the default or environment connection information
with Client.from_config() as client:

    # Load the default solver
    solver = client.get_solver()

    # Build a random Ising model to exactly fit the graph the solver supports
    linear = {index: random.choice([-1, 1]) for index in solver.nodes}
    quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}

    # Send the problem for sampling, include solver-specific parameter 'num_reads'
    computation = solver.sample_ising(linear, quad, num_reads=100)

    # Print the first sample out of a hundred
    print(computation.samples[0])
```

CHAPTER 2

Documentation

2.1 Introduction

D-Wave Cloud Client is a minimal implementation of the REST interface used to communicate with D-Wave Sampler API (SAPI) servers.

SAPI is an application layer built to provide resource discovery, permissions, and scheduling for quantum annealing resources at D-Wave Systems. This package provides a minimal Python interface to that layer without compromising the quality of interactions and workflow.

The D-Wave Cloud Client `Solver` class enables low-level control of problem submission. It is used, for example, by the dwave-system `DWaveSampler`, which enables quick incorporation of the D-Wave system as a sampler in your code.

2.1.1 Configuration

It's recommended you set up your D-Wave Cloud Client configuration through the *interactive CLI utility*.

As described in the Using a D-Wave System section of Ocean Documentation, for your code to access remote D-Wave compute resources, you must configure communication through SAPI; for example, your code needs your API token for authentication. D-Wave Cloud Client provides multiple options for configuring the required information:

- One or more locally saved *configuration files*
- *Environment variables*
- Direct setting of key values in functions

These options can be flexibly used together.

Configuration Files

If a D-Wave Cloud Client configuration file is not explicitly specified when instantiating a client or solver, auto-detection searches for candidate files in a number of standard directories, depending on your local system's operating

system. You can see the standard locations with the *get_configfile_paths()* method.

For example, on a Unix system, depending on its flavor, these might include (in order):

```
/usr/share/dwave/dwave.conf
/usr/local/share/dwave/dwave.conf
~/.config/dwave/dwave.conf
./dwave.conf
```

On Windows 7+, configuration files are expected to be located under:

```
C:\\Users\\<username>\\AppData\\Local\\dwavesystem\\dwave\\dwave.conf
```

On Mac OS X, configuration files are expected to be located under:

```
~/Library/Application Support/dwave/dwave.conf
```

(For details on the D-Wave API for determining platform-independent paths to user data and configuration folders see the homebase tool.)

You can check the directories searched by *get_configfile_paths()* from a console using the *interactive CLI utility*; for example:

```
$ dwave config ls -m
/etc/xdg/xdg-ubuntu/dwave/dwave.conf
/usr/share/upstart/xdg/dwave/dwave.conf
/etc/xdg/dwave/dwave.conf
/home/jane/.config/dwave/dwave.conf
./dwave.conf
```

A single D-Wave Cloud Client configuration file can contain multiple profiles, each defining a separate combination of communication parameters such as the URL to the remote resource, authentication token, solver, etc. Configuration files conform to a standard Windows INI-style format: profiles are defined by sections such as, *[profile-a]* and *[profile-b]*. Default values for undefined profile keys are taken from the *[defaults]* section.

For example, if the configuration file, *~/.config/dwave/dwave.conf*, selected through auto-detection as the default configuration, contains the following profiles:

```
[defaults]
token = ABC-123456789123456789123456789

[first-available-qpu]
solver = {"qpu": true}

[software]
client = sw
solver = c4-sw_sample
token = DEF-98765432198765432198765432l
proxy = http://user:pass@myproxy.com:8080/

[backup-dwave2000q]
endpoint = https://url.of.my.backup.dwavesystem.com/sapi
solver = {"num_qubits__gt": 2000}
```

You can instantiate clients for a D-Wave system and a CPU with:

```
>>> from dwave.cloud import Client
>>> client_qpu = Client.from_config()
>>> client_cpu = Client.from_config(profile='software')
```

Environment Variables

In addition to files, you can set configuration information through environment variables; for example:

- `DWAVE_CONFIG_FILE` may select the configuration file path.
- `DWAVE_PROFILE` may select the name of a profile (section).
- `DWAVE_API_TOKEN` may select the API token.

For details on supported environment variables and prioritizing between these and values set explicitly or through a configuration file, see *dwave.cloud.config*.

Interactive CLI Configuration

As part of the installation of the D-Wave Cloud Client package, a *dwave* executable is installed; for example, in a virtual environment it might be installed as *<virtual_environment>\Scripts\dwave.exe*. Running this file from your system's console opens an interactive command line interface (CLI) that guides you through setting up a D-Wave Cloud Client configuration file. It also provides additional helpful functionality; for example:

- List and update existing configuration files on your system
- Establish a connection to (ping) a solver and return timing information
- Show information on configured solvers

Run *dwave –help* for information on all the CLI options.

Note: If you work in a Bash shell and want command completion for *dwave*, add

```
eval "$(_DWAVE_COMPLETE=source <path>/dwave)"
```

to your shell's *.bashrc* configuration file, where *<path>* is the absolute path to the installed *dwave* executable, for example */home/Mary/my-quantum-app/env/bin*.

2.1.2 Work Flow

A typical workflow may include the following steps:

1. Instantiate a *Client* to manage communication with remote *solver* resources, selecting and authenticating access to available solvers; for example, you can list all solvers available to a client with its *get_solvers()* method and select and return one with its *get_solver()* method.

   Preferred use is with a context manager—a `with Client.from_config(...) as` construct—to ensure proper closure of all resources. The following example snippet creates a client based on an auto-detected configuration file and instantiates a solver.

   ```
   >>> with Client.from_config() as client:   # doctest: +SKIP
   ...     solver = client.get_solver(qpu=True)
   ```

   Alternatively, the following example snippet creates a client for software resources that it later explicitly closes.

   ```
   >>> client = Client.from_config(client='sw')   # doctest: +SKIP
   >>> # code that uses client
   >>> client.close()    # doctest: +SKIP
   ```

2. Instantiate a selected *Solver*, a resource for solving problems. Solvers are responsible for:

- Encoding submitted problems
- Checking submitted parameters
- Adding problems to a client's submission queue

Solvers that provide sampling for solving *Ising* and *QUBO* problems, such as a D-Wave 2000Q *sampler* `DWaveSampler` or software sampler `SimulatedAnnealingSampler`, might be remote resources.

3. Submit your problem, using your solver, and then process the returned `Future`, instantiated by your solver to handle remotely executed problem solving.

2.1.3 Terminology

Ising Traditionally used in statistical mechanics. Variables are "spin up" (↑) and "spin down" (↓), states that correspond to $+1$ and -1 values. Relationships between the spins, represented by couplings, are correlations or anti-correlations. The objective function expressed as an Ising model is as follows:

$$\mathrm{E}_{ising}(\boldsymbol{s}) = \sum_{i=1}^{N} h_i s_i + \sum_{i=1}^{N} \sum_{j=i+1}^{N} J_{i,j} s_i s_j \quad (2.1)$$

where the linear coefficients corresponding to qubit biases are h_i, and the quadratic coefficients corresponding to coupling strengths are $J_{i,j}$.

A collection of variables with associated linear and quadratic biases.

Quadratic unconstrained binary optimization. QUBO problems are traditionally used in computer science. Variables are TRUE and FALSE, states that correspond to 1 and 0 values. A QUBO problem is defined using an upper-diagonal matrix Q, which is an N x N upper-triangular matrix of real weights, and x, a vector of binary variables, as minimizing the function

$$f(x) = \sum_{i} Q_{i,i} x_i + \sum_{i<j} Q_{i,j} x_i x_j \quad (2.2)$$

where the diagonal terms $Q_{i,i}$ are the linear coefficients and the nonzero off-diagonal terms are the quadratic coefficients $Q_{i,j}$. This can be expressed more concisely as

$$\min_{x \in \{0,1\}^n} x^T Q x. \quad (2.3)$$

In scalar notation, the objective function expressed as a QUBO is as follows:

$$\mathrm{E}_{qubo}(a_i, b_{i,j}; q_i) = \sum_{i} a_i q_i + \sum_{i<j} b_{i,j} q_i q_j. \quad (2.4)$$

A process that samples from low energy states of a problem's objective function. A binary quadratic model (BQM) sampler samples from low energy states in models such as those defined by an Ising equation or a Quadratic Unconstrained Binary Optimization (QUBO) problem and returns an iterable of samples, in order of increasing energy. A dimod sampler provides 'sample_qubo' and 'sample_ising' methods as well as the generic BQM sampler method.

A resource that runs a problem. Some solvers interface to the QPU; others leverage CPU and GPU resources.

2.2 Reference Documentation

Release 0.5.4

Date May 06, 2019

noindex

2.2.1 Configuration

Configuration for communicating with a solver.

Communicating with a solver—submitting a problem, monitoring its progress, receiving samples—requires configuration of several parameters such as the selected solver, its URL, an API token, etc. D-Wave Cloud Client provides multiple options for configuring those parameters:

- One or more locally saved configuration files.
- Environment variables
- Direct setting of key values in functions

These options can be flexibly used together. The standard use is through the *from_config()* classmethod.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments
2. Values specified as environment variables
3. Values specified in the configuration file

Configuration files comply with standard Windows INI-like format, parsable with Python's `configparser`. An optional *defaults* section provides default key-value pairs for all other sections. User-defined key-value pairs (unrecognized keys) are passed through to the client.

Typically configuration files are created, inspected, and changed using interactive CLI commands from your system's console, such as `dwave config create` and `dwave config inspect` (run `dwave --help` for information on CLI options).

Environment variables:

`DWAVE_CONFIG_FILE`: Configuration file path.

`DWAVE_PROFILE`: Name of profile (section).

`DWAVE_API_CLIENT`: API client class. Supported values are `qpu` or `sw`.

`DWAVE_API_ENDPOINT`: API endpoint URL.

`DWAVE_API_TOKEN`: API authorization token.

`DWAVE_API_SOLVER`: Default solver.

`DWAVE_API_PROXY`: URL for proxy connections to D-Wave API.

Examples

The following are typical examples of using *`from_config()`* to create a configured client.

This first example initializes *`Client`* from an explicitly specified configuration file, "~/jane/my_path_to_config/my_cloud_conf.conf":

```
[defaults]
token = ABC-123456789123456789123456789

[first-qpu]
solver = {"qpu": true}

[feature]
endpoint = https://url.of.some.dwavesystem.com/sapi
token = DEF-98765432198765432198765432l
solver = {"num_qubits__gte": 2000, "max_anneal_schedule_points__gte": 4}
```

The example code below creates a client object that connects to a D-Wave QPU, using *`dwave.cloud.qpu.Client`* and the first available online D-Wave system at the default API endpoint URL (https://cloud.dwavesys.com/sapi). The `feature` profile specifies a solver selected based on available features, namely we're requesting the first solver that has at least 2000 qubits and the anneal schedule can be described with at least 4 points.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config(config_file='~/jane/my_path_to_config/my_cloud_conf.
↪conf')  # doctest: +SKIP
>>> # code that uses client
>>> client.close()
```

This second example auto-detects a configuration file on the local system following the user/system configuration paths of *`get_configfile_paths()`*. It passes through to the instantiated client an unrecognized key-value pair my_param=‘my_value‘.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config(my_param=`my_value`)
>>> # code that uses client
>>> client.close()
```

This third example instantiates two clients, for managing both QPU and software solvers. Common key-value pairs are taken from the defaults section of a shared configuration file:

```
[defaults]
token = ABC-123456789123456789123456789

[primary-qpu]
solver = {"qpu": true}

[sw-solver]
client = sw
solver = c4-sw_sample
endpoint = https://url.of.some.software.resource.com/my_if
```

(continues on next page)

(continued from previous page)

```
token = DEF-98765432198765432198765432l

[backup-qpu]
solver = {"qpu": true, "num_qubits__gte": 2000}
endpoint = https://url.of.some.dwavesystem.com/sapi
proxy = http://user:pass@myproxy.com:8080/
token = XYZ-01010101001123412341234123412341234
```

The example code below creates client objects for two QPU solvers (at the same URL but each with its own solver ID and token) and one software solver.

```
>>> from dwave.cloud import Client
>>> client_qpu1 = Client.from_config(profile='primary-qpu')     # doctest: +SKIP
>>> client_qpu1 = Client.from_config(profile='backup-qpu')     # doctest: +SKIP
>>> client_sw1 = Client.from_config(profile='sw-solver')    # doctest: +SKIP
>>> client_qpu1.default_solver    # doctest: +SKIP
u'EXAMPLE_2000Q_SYSTEM_A'
>>> client_qpu2.endpoint    # doctest: +SKIP
u'https://url.of.some.dwavesystem.com/sapi'
>>> # code that uses client
>>> client_qpu1.close() # doctest: +SKIP
>>> client_qpu2.close() # doctest: +SKIP
>>> client_sw1.close() # doctest: +SKIP
```

This fourth example loads configurations auto-detected in more than one configuration file, with the higher priority file (in the current working directory) supplementing and overriding values from the lower priority user-local file. After instantiation, an endpoint from the default section and client from the profile section is provided from the user-local `/usr/local/share/dwave/dwave.conf` file:

```
[defaults]
solver = {"qpu": true}

[dw2000]
endpoint = https://int.se.dwavesystems.com/sapi
token = ABC-12345678912345678912345678l
```

A solver is supplemented from the file in the current working directory, which also overrides the token value. `./dwave.conf` is the file in the current directory:

```
[dw2000]
token = DEF-98765432198765432198765432l
```

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> client.default_solver    # doctest: +SKIP
u'EXAMPLE_2000Q_SYSTEM_A'
>>> client.endpoint  # doctest: +SKIP
u'https://int.se.dwavesystems.com/sapi'
>>> client.token  # doctest: +SKIP
u'DEF-98765432198765432198765432l'
>>> # code that uses client
>>> client.close() # doctest: +SKIP
```

The next example uses *load_config()* to load profile values. **Most users do not need to use this method.** It loads from the following configuration file, dwave_c.conf, located in the current working directory, and specified explicitly:

```
[defaults]
endpoint = https://url.of.some.dwavesystem.com/sapi
solver = {"qpu": true}

[dw2000a]
solver = {"software": true, "name": "EXAMPLE_2000Q"}
token = ABC-123456789123456789123456789

[dw2000b]
solver = {"qpu": true}
token = DEF-98765432198765432198765432l
```

This configuration file contains two profiles in addition to the defaults section. In the following example code, first no profile is specified, and the first profile after the defaults section is loaded with the solver overridden by the environment variable. Next, the second profile is selected with the explicitly named solver overriding the environment variable setting.

```
>>> import dwave.cloud as dc
>>> import os
>>> os.environ['DWAVE_API_SOLVER'] = 'EXAMPLE_2000Q_SYSTEM'    # doctest: +SKIP
>>> dc.config.load_config("./dwave_c.conf")    # doctest: +SKIP
{'client': u'sw',
 'endpoint': u'https://url.of.some.dwavesystem.com/sapi',
 'proxy': None,
 'solver': 'EXAMPLE_2000Q_SYSTEM',
 'token': u'ABC-123456789123456789123456789'}
>>> dc.config.load_config("./dwave_c.conf", profile='dw2000b', solver='Solver3')    #␣
↪doctest: +SKIP
{'client': u'qpu',
 'endpoint': u'https://url.of.some.dwavesystem.com/sapi',
 'proxy': None,
 'solver': 'Solver3',
 'token': u'DEF-98765432198765432198765432l'}
```

Methods

Most users do not need to use these methods.

Loading Configuration

These functions deploy D-Wave Cloud Client settings from a configuration file.

load_config([config_file, profile, client, ...])	Load D-Wave Cloud Client configuration based on a configuration file.

dwave.cloud.config.load_config

dwave.cloud.config.**load_config**(*config_file=None*, *profile=None*, *client=None*, *endpoint=None*, *token=None*, *solver=None*, *proxy=None*)

Load D-Wave Cloud Client configuration based on a configuration file.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments in *load_config()*. These values replace values read from a configuration file, and therefore must be **strings**, including float values for timeouts, boolean flags (tested for "truthiness"), and solver feature constraints (a dictionary encoded as JSON).
2. Values specified as environment variables.
3. Values specified in the configuration file.

Configuration-file format is described in *dwave.cloud.config*.

If the location of the configuration file is not specified, auto-detection searches for existing configuration files in the standard directories of *get_configfile_paths()*.

If a configuration file explicitly specified, via an argument or environment variable, does not exist or is unreadable, loading fails with *ConfigFileReadError*. Loading fails with *ConfigFileParseError* if the file is readable but invalid as a configuration file.

Similarly, if a profile explicitly specified, via an argument or environment variable, is not present in the loaded configuration, loading fails with ValueError. Explicit profile selection also fails if the configuration file is not explicitly specified, detected on the system, or defined via an environment variable.

Environment variables: DWAVE_CONFIG_FILE, DWAVE_PROFILE, DWAVE_API_CLIENT, DWAVE_API_ENDPOINT, DWAVE_API_TOKEN, DWAVE_API_SOLVER, DWAVE_API_PROXY.

Environment variables are described in *dwave.cloud.config*.

Parameters

- **config_file** (*str/[str]/None/False/True, default=None*) – Path to configuration file(s).

  If *None*, the value is taken from *DWAVE_CONFIG_FILE* environment variable if defined. If the environment variable is undefined or empty, auto-detection searches for existing configuration files in the standard directories of *get_configfile_paths()*.

  If *False*, loading from file(s) is skipped; if *True*, forces auto-detection (regardless of the *DWAVE_CONFIG_FILE* environment variable).

- **profile** (*str, default=None*) – Profile name (name of the profile section in the configuration file).

  If undefined, inferred from *DWAVE_PROFILE* environment variable if defined. If the environment variable is undefined or empty, a profile is selected in the following order:

  1. From the default section if it includes a profile key.
  2. The first section (after the default section).
  3. If no other section is defined besides *[defaults]*, the defaults section is promoted and selected.

- **client** (*str, default=None*) – Client type used for accessing the API. Supported values are *qpu* for *dwave.cloud.qpu.Client* and *sw* for *dwave.cloud.sw.Client*.

- **endpoint** (*str, default=None*) – API endpoint URL.

- **token** (*str, default=None*) – API authorization token.

- **solver** (*str, default=None*) – *solver* features, as a JSON-encoded dictionary of feature constraints, the client should use. See *get_solvers()* for semantics of supported feature constraints.

  If undefined, the client uses a solver definition from environment variables, a configuration file, or falls back to the first available online solver.

For backward compatibility, solver name in string format is accepted and converted to `{"name": <solver name>}`.

- **proxy** (*str, default=None*) – URL for proxy to use in connections to D-Wave API. Can include username/password, port, scheme, etc. If undefined, client uses the system-level proxy, if defined, or connects directly to the API.

Returns Mapping of configuration keys to values for the profile (section), as read from the configuration file and optionally overridden by environment values and specified keyword arguments. Always contains the *client*, *endpoint*, *token*, *solver*, and *proxy* keys.

Return type dict

Raises

- `ValueError` – Invalid (non-existing) profile name.
- *ConfigFileReadError* – Config file specified or detected could not be opened or read.
- *ConfigFileParseError* – Config file parse failed.

Examples This example loads the configuration from an auto-detected configuration file in the home directory of a Windows system user.

```
>>> import dwave.cloud as dc
>>> dc.config.load_config()
{'client': u'qpu',
 'endpoint': u'https://url.of.some.dwavesystem.com/sapi',
 'proxy': None,
 'solver': u'EXAMPLE_2000Q_SYSTEM_A',
 'token': u'DEF-98765432198765432198765432l'}
>>> See which configuration file was loaded
>>> dc.config.get_configfile_paths()
[u'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf']
```

Additional examples are given in *dwave.cloud.config*.

Managing Files

These functions manage your D-Wave Cloud Client configuration files. It's recommended you set up your configuration through the *interactive CLI utility* instead.

get_configfile_paths([system, user, local, ...])	Return a list of local configuration file paths.
get_configfile_path()	Return the highest-priority local configuration file.
get_default_configfile_path()	Return the default configuration-file path.

dwave.cloud.config.get_configfile_paths

`dwave.cloud.config.`**`get_configfile_paths`**(*system=True, user=True, local=True, only_existing=True*)

Return a list of local configuration file paths.

Search paths for configuration files on the local system are based on homebase and depend on operating system; for example, for Linux systems these might include `dwave.conf` in the current working directory (CWD), user-local `.config/dwave/`, and system-wide `/etc/dwave/`.

Parameters

- **system** (*boolean, default=True*) – Search for system-wide configuration files.
- **user** (*boolean, default=True*) – Search for user-local configuration files.
- **local** (*boolean, default=True*) – Search for local configuration files (in CWD).
- **only_existing** (*boolean, default=True*) – Return only paths for files that exist on the local system.

Returns List of configuration file paths.

Return type list[str]

Examples

This example displays all paths to configuration files on a Windows system running Python 2.7 and then finds the single existing configuration file.

```
>>> import dwave.cloud as dc
>>> # Display paths
>>> dc.config.get_configfile_paths(only_existing=False)    # doctest: +SKIP
[u'C:\ProgramData\dwavesystem\dwave\dwave.conf',
 u'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf',
 '.\dwave.conf']
>>> # Find existing files
>>> dc.config.get_configfile_paths()    # doctest: +SKIP
[u'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf']
```

dwave.cloud.config.get_configfile_path

dwave.cloud.config.**get_configfile_path**()

Return the highest-priority local configuration file.

Selects the top-ranked configuration file path from a list of candidates returned by *get_configfile_paths()*, or None if no candidate path exists.

Returns Configuration file path.

Return type str

Examples

This example displays the highest-priority configuration file on a Windows system running Python 2.7.

```
>>> import dwave.cloud as dc
>>> # Display paths
>>> dc.config.get_configfile_paths(only_existing=False)    # doctest: +SKIP
[u'C:\ProgramData\dwavesystem\dwave\dwave.conf',
 u'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf',
 '.\dwave.conf']
>>> # Find highest-priority local configuration file
>>> dc.config.get_configfile_path()    # doctest: +SKIP
u'C:\Users\jane\AppData\Local\dwavesystem\dwave\dwave.conf'
```

dwave.cloud.config.get_default_configfile_path

dwave.cloud.config.**get_default_configfile_path**()

Return the default configuration-file path.

Typically returns a user-local configuration file; e.g: `~/.config/dwave/dwave.conf`.

Returns Configuration file path.

Return type str

Examples

This example displays the default configuration file on an Ubuntu Unix system running IPython 2.7.

```
>>> import dwave.cloud as dc
>>> # Display paths
>>> dc.config.get_configfile_paths(only_existing=False)    # doctest: +SKIP
['/etc/xdg/xdg-ubuntu/dwave/dwave.conf',
 '/usr/share/upstart/xdg/dwave/dwave.conf',
 '/etc/xdg/dwave/dwave.conf',
 '/home/mary/.config/dwave/dwave.conf',
 './dwave.conf']
>>> # Find default configuration path
>>> dc.config.get_default_configfile_path()    # doctest: +SKIP
'/home/mary/.config/dwave/dwave.conf'
```

2.2.2 Clients

The *solver*s that provide sampling for solving *Ising* and *QUBO* problems, such as a D-Wave 2000Q QPU or a software *sampler* such as the dimod simulated annealing sampler, are typically remote resources. The D-Wave Cloud Client *Client* class manages such remote solver resources.

Preferred use is with a context manager—a `with Client.from_config(...) as` construct—to ensure proper closure of all resources. The following example snippet creates a client based on an auto-detected configuration file and instantiates a solver.

```
>>> with Client.from_config() as client:    # doctest: +SKIP
...     solver = client.get_solver(num_qubits__gt=2000)
```

Alternatively, the following example snippet creates a client for software resources that it later explicitly closes.

```
>>> client = Client.from_config(software=True)    # doctest: +SKIP
>>> # code that uses client
>>> client.close()    # doctest: +SKIP
```

Typically you use the *Client* class. By default, it instantiates a QPU client. You can also use the specialized QPU and CPU/GPU clients directly.

Client (Base Client)

D-Wave API clients handle communications with *solver* resources: problem submittal, monitoring, samples retrieval, etc.

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a *QUBO* problem (a Boolean NOT gate represented by a penalty model), and samples 5 times.

```
>>> from dwave.cloud import Client
>>> Q = {(0, 0): -1, (0, 4): 0, (4, 0): 2, (4, 4): -1}
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     computation = solver.sample_qubo(Q, num_reads=5)
...
>>> for i in range(5):      # doctest: +SKIP
...     print(computation.samples[i][0], computation.samples[i][4])
...
(1, 0)
(1, 0)
(0, 1)
(0, 1)
(0, 1)
```

Class

class dwave.cloud.client.**Client**(*endpoint=None*, *token=None*, *solver=None*, *proxy=None*, *permissive_ssl=False*, *request_timeout=60*, *polling_timeout=None*, *connection_close=False*, ***kwargs*)

Base client class for all D-Wave API clients. Used by QPU and software *sampler* classes.

Manages workers and handles thread pools for submitting problems, cancelling tasks, polling problem status, and retrieving results.

Parameters

- **endpoint** (*str*) – D-Wave API endpoint URL.
- **token** (*str*) – Authentication token for the D-Wave API.
- **solver** (*dict/str*) – Default solver features (or simply solver name).
- **proxy** (*str*) – Proxy URL to be used for accessing the D-Wave API.
- **permissive_ssl** (*bool, default=False*) – Disables SSL verification.
- **request_timeout** (*float, default=60*) – Connect and read timeout (in seconds) for all requests to the D-Wave API.
- **polling_timeout** (*float, default=None*) – Problem status polling timeout (in seconds), after which polling is aborted.
- **connection_close** (*bool, default=False*) – Force HTTP(S) connection close after each request.

Other Parameters **Unrecognized keys** (*str*) – All unrecognized keys are passed through to the appropriate client class constructor as string keyword arguments.

An explicit key value overrides an identical user-defined key value loaded from a configuration file.

Examples

This example directly initializes a *Client*. Direct initialization uses class constructor arguments, the minimum being a value for *token*.

```
>>> from dwave.cloud import Client
>>> client = Client(token='secret')
>>> # code that uses client
>>> client.close()
```

Methods

client.Client.from_config([config_file, ...])	Client factory method to instantiate a client instance from configuration.
client.Client.solvers([refresh])	Deprecated in favor of *get_solvers()*.
client.Client.get_solver([name, refresh])	Load the configuration for a single solver.
client.Client.get_solvers([refresh, order_by])	Return a filtered list of solvers handled by this client.
client.Client.is_solver_handled(solver)	Determine if the specified solver should be handled by this client.
client.Client.close()	Perform a clean shutdown.

dwave.cloud.client.Client.from_config

classmethod Client.**from_config**(*config_file=None*, *profile=None*, *client=None*, *endpoint=None*, *token=None*, *solver=None*, *proxy=None*, *legacy_config_fallback=False*, ***kwargs*)

Client factory method to instantiate a client instance from configuration.

Configuration values can be specified in multiple ways, ranked in the following order (with 1 the highest ranked):

1. Values specified as keyword arguments in *from_config()*
2. Values specified as environment variables
3. Values specified in the configuration file

Configuration-file format is described in *dwave.cloud.config*.

If the location of the configuration file is not specified, auto-detection searches for existing configuration files in the standard directories of `get_configfile_paths()`.

If a configuration file explicitly specified, via an argument or environment variable, does not exist or is unreadable, loading fails with *ConfigFileReadError*. Loading fails with *ConfigFileParseError* if the file is readable but invalid as a configuration file.

Similarly, if a profile explicitly specified, via an argument or environment variable, is not present in the loaded configuration, loading fails with `ValueError`. Explicit profile selection also fails if the configuration file is not explicitly specified, detected on the system, or defined via an environment variable.

Environment variables: `DWAVE_CONFIG_FILE`, `DWAVE_PROFILE`, `DWAVE_API_CLIENT`, `DWAVE_API_ENDPOINT`, `DWAVE_API_TOKEN`, `DWAVE_API_SOLVER`, `DWAVE_API_PROXY`.

Environment variables are described in *dwave.cloud.config*.

Parameters

- **config_file** (*str/[str]/None/False/True, default=None*) – Path to configuration file.

  If `None`, the value is taken from `DWAVE_CONFIG_FILE` environment variable if defined. If the environment variable is undefined or empty, auto-detection searches for existing configuration files in the standard directories of `get_configfile_paths()`.

  If `False`, loading from file is skipped; if `True`, forces auto-detection (regardless of the `DWAVE_CONFIG_FILE` environment variable).

- **profile** (*str, default=None*) – Profile name (name of the profile section in the configuration file).

  If undefined, inferred from `DWAVE_PROFILE` environment variable if defined. If the environment variable is undefined or empty, a profile is selected in the following order:

  1. From the default section if it includes a profile key.
  2. The first section (after the default section).
  3. If no other section is defined besides `[defaults]`, the defaults section is promoted and selected.

- **client** (*str, default=None*) – Client type used for accessing the API. Supported values are `qpu` for *dwave.cloud.qpu.Client* and `sw` for *dwave.cloud.sw.Client*.

- **endpoint** (*str, default=None*) – API endpoint URL.

- **token** (*str, default=None*) – API authorization token.

- **solver** (*dict/str, default=None*) – Default *solver* features to use in *get_solver()*.

  Defined via dictionary of solver feature constraints (see *get_solvers()*). For backward compatibility, a solver name, as a string, is also accepted and converted to `{"name": <solver name>}`.

  If undefined, *get_solver()* uses a solver definition from environment variables, a configuration file, or falls back to the first available online solver.

- **proxy** (*str, default=None*) – URL for proxy to use in connections to D-Wave API. Can include username/password, port, scheme, etc. If undefined, client uses the system-level proxy, if defined, or connects directly to the API.

- **legacy_config_fallback** (*bool, default=False*) – If True and loading from a standard D-Wave Cloud Client configuration file (`dwave.conf`) fails, tries loading a legacy configuration file (`~/.dwrc`).

Other Parameters **Unrecognized keys** (*str*) – All unrecognized keys are passed through to the appropriate client class constructor as string keyword arguments.

An explicit key value overrides an identical user-defined key value loaded from a configuration file.

Returns Appropriate instance of a QPU or software client.

Return type *Client* (*dwave.cloud.qpu.Client* or *dwave.cloud.sw.Client*, default=:class:*dwave.cloud.qpu.Client*)

Raises

- *ConfigFileReadError* – Config file specified or detected could not be opened or read.
- *ConfigFileParseError* – Config file parse failed.

Examples

A variety of examples are given in *dwave.cloud.config*.

This example initializes *Client* from an explicitly specified configuration file, "~/jane/my_path_to_config/my_cloud_conf.conf":

```
>>> from dwave.cloud import Client
>>> client = Client.from_config(config_file='~/jane/my_path_to_config/my_cloud_
↪conf.conf')
>>> # code that uses client
>>> client.close()
```

dwave.cloud.client.Client.solvers

Client.**solvers**(*refresh=False*, ***filters*)
Deprecated in favor of *get_solvers()*.

dwave.cloud.client.Client.get_solver

Client.**get_solver**(*name=None*, *refresh=False*, ***filters*)
Load the configuration for a single solver.

Makes a blocking web call to *{endpoint}/solvers/remote/{solver_name}/*, where *{endpoint}* is a URL configured for the client, and returns a *Solver* instance that can be used to submit sampling problems to the D-Wave API and retrieve results.

Parameters

- **name** (*str*) – ID of the requested solver. `None` returns the default solver. If default solver is not configured, `None` returns the first available solver in `Client`'s class (QPU/software/base).
- ****filters** (*keyword arguments, optional*) – Dictionary of filters over features this solver has to have. For a list of feature names and values, see: *get_solvers()*.
- **order_by** (*callable/str, default='id'*) – Solver sorting key function (or `Solver` attribute name). By default, solvers are sorted by ID/name.
- **refresh** (*bool*) – Return solver from cache (if cached with `get_solvers()`), unless set to `True`.

Returns *Solver*

Examples

This example creates two solvers for a client instantiated from a local system's auto-detected default configuration file, which configures a connection to a D-Wave resource that provides two solvers. The first uses the default solver, the second explicitly selects another solver.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> client.get_solvers()   # doctest: +SKIP
[Solver(id='2000Q_ONLINE_SOLVER1'), Solver(id='2000Q_ONLINE_SOLVER2')]
>>> solver1 = client.get_solver()    # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
>>> solver2 = client.get_solver(name='2000Q_ONLINE_SOLVER2')     # doctest: +SKIP
>>> solver1.id  # doctest: +SKIP
'2000Q_ONLINE_SOLVER1'
>>> solver2.id   # doctest: +SKIP
'2000Q_ONLINE_SOLVER2'
>>> # code that uses client
>>> client.close() # doctest: +SKIP
```

dwave.cloud.client.Client.get_solvers

`Client.`**`get_solvers`**(*refresh=False*, *order_by='avg_load'*, ***filters*)

Return a filtered list of solvers handled by this client.

Parameters

- **`refresh`** (*`bool`*, *`default=False`*) – Force refresh of cached list of solvers/properties.
- **`order_by`** (*`callable/str/None`*, *`default='avg_load'`*) – Solver sorting key function (or `Solver` attribute/item dot-separated path). By default, solvers are sorted by average load. To explicitly not sort the solvers (and use the API-returned order), set `order_by=None`.

  Signature of the *key callable* is:

  ```
  key :: (Solver s, Ord k) => s -> k
  ```

  Basic structure of the *key* string path is:

  ```
  "-"? (attr|item) ( "." (attr|item) )*
  ```

  For example, to use solver property named `max_anneal_schedule_points`, available in `Solver.properties` dict, you can either specify a callable *key*:

  ```
  key=lambda solver: solver.properties['max_anneal_schedule_points']
  ```

  or, you can use a short string path based key:

  ```
  key='properties.max_anneal_schedule_points'
  ```

  Solver derived properties, available as `Solver` properties can also be used (e.g. `num_active_qubits`, `online`, `avg_load`, etc).

  Ascending sort order is implied, unless the key string path does not start with `-`, in which case descending sort is used.

  Note: the sort used for ordering solvers by *key* is **stable**, meaning that if multiple solvers have the same value for the key, their relative order is preserved, and effectively they are in the same order as returned by the API.

  Note: solvers with `None` for key appear last in the list of solvers. When providing a key callable, ensure all values returned are of the same type (particularly in Python 3). For solvers with undefined key value, return `None`.

- **`**filters`** – See *Filtering forms* and *Operators* below.

Solver filters are defined, similarly to Django QuerySet filters, with keyword arguments of form *<key1>__...__<keyN>[__<operator>]=<value>*. Each *<operator>* is a predicate (boolean) function that acts on two arguments: value of feature *<name>* (described with keys path *<key1.key2...keyN>*) and the required *<value>*.

Feature *<name>* can be:

1) a derived solver property, available as an identically named `Solver`'s property (*name*, *qpu*, *software*, *online*, *num_active_qubits*, *avg_load*)
2) a solver parameter, available in `Solver.parameters`
3) a solver property, available in `Solver.properties`
4) a path describing a property in nested dictionaries

Filtering forms are:

- <derived_property>__<operator> (object <value>)
- <derived_property> (bool)

  This form ensures the value of solver's property bound to *derived_property*, after applying *operator* equals the *value*. The default operator is *eq*.

  For example:

```
>>> client.get_solvers(avg_load__gt=0.5)
```

  but also:

```
>>> client.get_solvers(online=True)
>>> # identical to:
>>> client.get_solvers(online__eq=True)
```

- <parameter>__<operator> (object <value>)
- <parameter> (bool)

  This form ensures that the solver supports *parameter*. General operator form can be used but usually does not make sense for parameters, since values are human-readable descriptions. The default operator is *available*.

  Example:

```
>>> client.get_solvers(flux_biases=True)
>>> # identical to:
>>> client.get_solvers(flux_biases__available=True)
```

- <property>__<operator> (object <value>)
- <property> (bool)

  This form ensures the value of the solver's *property*, after applying *operator* equals the righthand side *value*. The default operator is *eq*.

Note: if a non-existing parameter/property name/key given, the default operator is *eq*.

Operators are:

- ***available* (<name>: str, <value>: bool):** Test availability of <name> feature.
- ***eq*, *lt*, *lte*, *gt*, *gte* (<name>: str, <value>: any):** Standard relational operators that compare feature <name> value with <value>.

- ***regex* (<name>: str, <value>: str):** Test regular expression matching feature value.
- ***covers* (<name>: str, <value>: single value or range expressed as 2-tuple/list):** Test feature <name> value (which should be a *range*) covers a given value or a subrange.
- ***within* (<name>: str, <value>: range expressed as 2-tuple/list):** Test feature <name> value (which can be a *single value* or a *range*) is within a given range.
- ***in* (<name>: str, <value>: container type):** Test feature <name> value is *in* <value> container.
- ***contains* (<name>: str, <value>: any):** Test feature <name> value (container type) *contains* <value>.
- ***issubset* (<name>: str, <value>: container type):** Test feature <name> value (container type) is a subset of <value>.
- ***issuperset* (<name>: str, <value>: container type):** Test feature <name> value (container type) is a superset of <value>.

Derived properies are:

- *name* (str): Solver name/id.
- *qpu* (bool): Is solver QPU based?
- *software* (bool): Is solver software based?
- *online* (bool, default=True): Is solver online?
- *num_active_qubits* (int): Number of active qubits. Less then or equal to *num_qubits*.
- *avg_load* (float): Solver's average load (similar to Unix load average).

Common solver parameters are:

- *flux_biases*: Should solver accept flux biases?
- *anneal_schedule*: Should solver accept anneal schedule?

Common solver properties are:

- *num_qubits* (int): Number of qubits available.
- *vfyc* (bool): Should solver work on "virtual full-yield chip"?
- *max_anneal_schedule_points* (int): Piecewise linear annealing schedule points.
- *h_range* ([int,int]), j_range ([int,int]): Biases/couplings values range.
- *num_reads_range* ([int,int]): Range of allowed values for *num_reads* parameter.

Returns List of all solvers that satisfy the conditions.

Return type list[*Solver*]

Note: Client subclasses (e.g. `dwave.cloud.qpu.Client` or `dwave.cloud.sw.Client`) already filter solvers by resource type, so for *qpu* and *software* filters to have effect, call `get_solvers()` on base class `Client`.

Examples:

```
client.get_solvers(
    num_qubits__gt=2000,               # we need more than 2000 qubits
    num_qubits__lt=4000,               # ... but fewer than 4000 qubits
    num_qubits__within=(2000, 4000),   # an alternative to the previous two lines
```

(continues on next page)

(continued from previous page)

```
    num_active_qubits=1089,             # we want a particular number of active␣
↪qubits
    vfyc=True,                          # we require a fully yielded Chimera
    vfyc__in=[False, None],             # inverse of the previous filter
    vfyc__available=False,              # we want solvers that do not advertize␣
↪the vfyc property
    anneal_schedule=True,               # we need support for custom anneal␣
↪schedule
    max_anneal_schedule_points__gte=4,  # we need at least 4 points for our␣
↪anneal schedule
    num_reads_range__covers=1000,       # our solver must support returning 1000␣
↪reads
    extended_j_range__covers=[-2, 2],   # we need extended J range to contain␣
↪subrange [-2,2]
    couplings__contains=[0, 128],       # coupling (edge between) qubits (0,128)␣
↪must exist
    couplings__issuperset=[[0,128], [0,4]],
                                        # two couplings required: (0,128) and (0,
↪4)
    qubits__issuperset={0, 4, 215},     # qubits 0, 4 and 215 must exist
    supported_problem_types__issubset={'ising', 'qubo'},
                                        # require Ising, QUBO or both to be␣
↪supported
    name='DW_2000Q_3',                  # full solver name/ID match
    name__regex='.*2000.*',             # partial/regex-based solver name match
    chip_id__regex='DW_.*',             # chip ID prefix must be DW_
    topology__type__eq="chimera"        # topology.type must be chimera
)
```

dwave.cloud.client.Client.is_solver_handled

static Client.**is_solver_handled**(*solver*)

Determine if the specified solver should be handled by this client.

Default implementation accepts all solvers (always returns True). Override this predicate function with a subclass if you want to specialize your client for a particular type of solvers.

Examples

This function accepts only solvers named “My_Solver_*”.

```
@staticmethod
def is_solver_handled(solver):
    return solver and solver.id.startswith('My_Solver_')
```

dwave.cloud.client.Client.close

Client.**close**()

Perform a clean shutdown.

Waits for all the currently scheduled work to finish, kills the workers, and closes the connection pool.

Note: Ensure your code does not submit new work while the connection is closing.

Where possible, it is recommended you use a context manager (a `with Client.from_config(...) as` construct) to ensure your code properly closes all resources.

Examples

This example creates a client (based on an auto-detected configuration file), executes some code (represented by a placeholder comment), and then closes the client.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> # code that uses client
>>> client.close()
```

Specialized Clients

Typically you use the *Client* class. By default, it instantiates a QPU client. You can also instantiate a QPU or CPU/GPU client directly.

QPU Client

An implementation of the REST API for D-Wave Solver API (SAPI) servers.

SAPI servers provide authentication, queuing, and scheduling services, and provide a network interface to *solver*s. This API enables you submit a binary quadratic (*Ising* or *QUBO*) model and receive samples from a distribution over the model as defined by a selected solver.

SAPI server workflow is roughly as follows:

1. Submitted problems enter an input queue. Each user has an input queue per solver.
2. Drawing from all input queues for a solver, problems are scheduled.
3. Results are cached for retrieval by the client.

Class

class dwave.cloud.qpu.**Client**(*endpoint=None*, *token=None*, *solver=None*, *proxy=None*, *permissive_ssl=False*, *request_timeout=60*, *polling_timeout=None*, *connection_close=False*, ***kwargs*)

D-Wave API client specialized to work with QPU solvers.

This class is instantiated by default, or explicitly when *client=qpu*, with the typical base client instantiation `with Client.from_config() as client:` of a client. (You should not instantiate this class with *client=sw* or use it with solver feature constraint *software=True*.)

Examples

This example explicitly instantiates a `dwave.cloud.qpu.client` based on the local system's default D-Wave Cloud Client configuration file to sample a random Ising problem tailored to fit the client's default solver's graph.

```
import random
from dwave.cloud.qpu import Client

# Use context manager to ensure resources (thread pools used by Client) are␣
↪released
with Client.from_config() as client:

    solver = client.get_solver()

    # Build problem to exactly fit the solver graph
    linear = {index: random.choice([-1, 1]) for index in solver.nodes}
    quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}

    # Sample 100 times and print out the first sample
    computation = solver.sample_ising(linear, quad, num_reads=100)
    print(computation.samples[0])
```

Methods

qpu.Client.is_solver_handled(solver)	Determine if the specified solver should be handled by this client.

dwave.cloud.qpu.Client.is_solver_handled

static Client.**is_solver_handled**(*solver*)

Determine if the specified solver should be handled by this client.

This predicate function overrides superclass to filter out any non-QPU solvers.

Current implementation filters out D-Wave software clients with solver IDs prefixed with *c4-sw*. If needed, update this method to suit your solver naming scheme.

Examples

This example filters solvers for those prefixed 2000Q.

```
@staticmethod
def is_solver_handled(solver):
    return solver and solver.id.startswith('2000Q')
```

Software-Samplers Client

Class

class dwave.cloud.sw.**Client**(*endpoint=None*, *token=None*, *solver=None*, *proxy=None*, *permissive_ssl=False*, *request_timeout=60*, *polling_timeout=None*, *connection_close=False*, ***kwargs*)

D-Wave API client specialized to work with remote software solvers (samplers).

This class is instantiated by default, or explicitly when *client=sw*, with the typical base client instantiation `with Client.from_config() as client:` of a client. (You should not instantiate this class with *client=qpu* or use it with solver feature constraint *qpu=True*.)

Examples

This example indirectly instantiates a `dwave.cloud.sw.client` based on the local system's default D-Wave Cloud Client configuration file to sample a random Ising problem tailored to fit the client's default solver's graph.

```
import random
from dwave.cloud import Client

# Use context manager to ensure resources (thread pools used by Client) are␣
↪released
with Client.from_config(solver={"software": True}) as client:

    solver = client.get_solver()

    # Build problem to exactly fit the solver graph
    linear = {index: random.choice([-1, 1]) for index in solver.nodes}
    quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}

    # Sample 100 times and print out the first sample
    computation = solver.sample_ising(linear, quad, num_reads=100)
    print(computation.samples[0])
```

Methods

sw.Client.is_solver_handled(solver)	Determine if the specified solver should be handled by this client.

dwave.cloud.sw.Client.is_solver_handled

static Client.**is_solver_handled**(*solver*)

Determine if the specified solver should be handled by this client.

This predicate function overrides superclass to allow only remote software solvers.

Current implementation allows only D-Wave software clients with solver IDs prefixed with *c4-sw*. If needed, update this method to suit your solver naming scheme.

Examples

This example filters solvers for those prefixed My_SW_Solver.

```
@staticmethod
def is_solver_handled(solver):
    return solver and solver.id.startswith('My_SW_Solver')
```

2.2.3 Solver

A *solver* is a resource for solving problems.

Solvers are responsible for:

- Encoding submitted problems

- Checking submitted parameters
- Adding problems to a client's submission queue

You can list all solvers available to a *Client* with its *get_solvers()* method and select and return one with its *get_solver()* method.

Class

class dwave.cloud.solver.**Solver**(*client*, *data*)

Class for D-Wave solvers.

This class provides *Ising* and *QUBO* sampling methods and encapsulates the solver description returned from the D-Wave cloud API.

Parameters

- **client** (Client) – Client that manages access to this solver.
- **data** (*dict*) – Data from the server describing this solver.

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file and checks the identity of its default solver.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> solver = client.get_solver()
>>> solver.data['id']    # doctest: +SKIP
u'EXAMPLE_2000Q_SYSTEM'
```

Methods

Solver.check_problem(linear, quadratic)	Test if an Ising model matches the graph provided by the solver.
Solver.sample_ising(linear, quadratic, **params)	Sample from the specified Ising model.
Solver.sample_qubo(qubo, **params)	Sample from the specified QUBO.

dwave.cloud.solver.Solver.check_problem

Solver.**check_problem**(*linear*, *quadratic*)

Test if an Ising model matches the graph provided by the solver.

Parameters

- **linear** (*list/dict*) – Linear terms of the model (h).
- **quadratic** (*dict of (int, int)* – float): Quadratic terms of the model (J).

Returns boolean

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, and tests a simple *Ising* model for two target embeddings (that is, representations of the model's graph by coupled qubits on the QPU's sparsely connected graph), where only the second is valid.

```
>>> from dwave.cloud import Client
>>> print((0, 1) in solver.edges)   # doctest: +SKIP
False
>>> print((0, 4) in solver.edges)   # doctest: +SKIP
True
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     print(solver.check_problem({0: -1, 1: 1},{(0, 1):0.5}))
...     print(solver.check_problem({0: -1, 4: 1},{(0, 4):0.5}))
...
False
True
```

dwave.cloud.solver.Solver.sample_ising

`Solver.`**`sample_ising`**(*linear*, *quadratic*, ***params*)

Sample from the specified Ising model.

Parameters

- **`linear`** (*list/dict*) – Linear terms of the model (h).
- **`quadratic`** (*dict of (int, int)* – float): Quadratic terms of the model (J).
- ****`params`** – Parameters for the sampling method, specified per solver.

Returns `Future`

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a simple *Ising* problem (opposite linear biases on two coupled qubits), and samples 5 times.

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     computation = solver.sample_ising({u: -1, v: 1},{}, num_reads=5)   #␣
↪doctest: +SKIP
...     for i in range(5):
...         print(computation.samples[i][u], computation.samples[i][v])
...
...
(1, -1)
(1, -1)
(1, -1)
(1, -1)
(1, -1)
```

dwave.cloud.solver.Solver.sample_qubo

Solver.**sample_qubo**(*qubo*, ***params*)
Sample from the specified QUBO.

Parameters

- **qubo** (*dict of (int, int)* – float): Coefficients of a quadratic unconstrained binary optimization (QUBO) model.
- ****params** – Parameters for the sampling method, specified per solver.

Returns Future

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a *QUBO* problem (a Boolean NOT gate represented by a penalty model), and samples 5 times.

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
...     computation = solver.sample_qubo(Q, num_reads=5)
...     for i in range(5):
...         print(computation.samples[i][u], computation.samples[i][v])
...
...
(0, 1)
(1, 0)
(1, 0)
(0, 1)
(1, 0)
```

2.2.4 Computation

Computation manages the interactions between your code and a *solver*, which manages interactions between the remote resource and your submitted problems.

Your solver instantiates a *Future* object for its calls, via D-Wave Sampler API (SAPI) servers, to the remote resource.

You can interact through the *Future* object with pending (running) or completed computation—sampling on a QPU or software solver—executed remotely, monitoring problem status, waiting for and retrieving results, cancelling enqueued jobs, etc.

Some *Future* methods are blocking.

Class

class dwave.cloud.computation.**Future**(*solver*, *id_*, *return_matrix*, *submission_data*)
Class for interacting with jobs submitted to SAPI.

Solver uses *Future* to construct objects for pending SAPI calls that can wait for requests to complete and parse returned messages.

Objects are blocked for the duration of any data accessed on the remote resource.

Warning: *Future* objects are not intended to be directly created. Problem submittal is initiated by *Solver* and executed by the client.

Parameters

- **solver** – Solver responsible for this *Future* object.
- **id** – Identification for a query submitted by a solver to SAPI. May be None following submission until an identification number is set.
- **return_matrix** – Return values for this *Future* object are NumPy matrices.

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and checks a couple of times whether the sampling is completed.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = solver.sample_qubo(Q, num_reads=100)    # doctest: +SKIP
>>> computation.done()  # doctest: +SKIP
False
>>> computation.id   # doctest: +SKIP
u'1cefeb6d-ebd5-4592-87c0-4cc43ec03e27'
>>> computation.done()   # doctest: +SKIP
True
>>> client.close()
```

Methods

Future.result()	Results for a submitted job.
Future.as_completed(fs[, timeout])	Yield Futures objects as they complete.
Future.wait([timeout])	Wait for the solver to receive a response for a submitted problem.
Future.wait_multiple(futures[, min_done, ...])	Wait for multiple *Future* objects to complete.
Future.done()	Check whether the solver received a response for a submitted problem.
Future.cancel()	Try to cancel the problem corresponding to this result.

dwave.cloud.computation.Future.result

Future.**result**()

Results for a submitted job.

Retrives result data in a *Future* object that the solver submitted to a remote resource. First calls to access this data are blocking.

Returns Results of the submitted job. Should be considered read-only.

Return type dict

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem (representing a Boolean NOT gate by a penalty function) to a remote D-Wave resource for 5 samples, and prints part of the returned result (the relevant samples).

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
...     computation = solver.sample_qubo(Q, num_reads=5)
...     for i in range(5):
...         print(computation.result()['samples'][i][u], computation.result()[
↪'samples'][i][v])
...
...
(0, 1)
(1, 0)
(1, 0)
(0, 1)
(0, 1)
```

dwave.cloud.computation.Future.as_completed

static Future.**as_completed**(*fs*, *timeout=None*)

Yield Futures objects as they complete.

Returns an iterator over the specified list of *Future* objects that yields those objects as they complete. Completion occurs when the submitted job is finished or cancelled.

Emulates the behavior of the *concurrent.futures.as_completed()* function.

Parameters

- **fs** (*list*) – List of *Future* objects to iterate over.
- **timeout** (*float, optional, default=None*) – Maximum number of seconds to await completion. If None, awaits indefinitely.

Returns Listed *Future* objects as they complete.

Return type Generator (*Future* objects)

Raises *concurrent.futures.TimeoutError* is raised if per-future timeout is exceeded.

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource 3 times for differing numers of samples, and yields timing information for each job as it completes.

```
>>> import dwave.cloud as dc
>>> client = dc.Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = [solver.sample_qubo(Q, num_reads=1000),
...                solver.sample_qubo(Q, num_reads=50),
...                solver.sample_qubo(Q, num_reads=10)]   # doctest: +SKIP
>>> for tasks in dc.computation.Future.as_completed(computation, timeout=10)
...     print(tasks.timing)   # doctest: +SKIP
...
{u'total_real_time': 17318, ... u'qpu_readout_time_per_sample': 123}
{u'total_real_time': 10816, ... u'qpu_readout_time_per_sample': 123}
{u'total_real_time': 26285, ... u'qpu_readout_time_per_sample': 123}
>>> # Snipped above response for brevity
>>> client.close()
```

dwave.cloud.computation.Future.wait

`Future.`**`wait`**(*timeout=None*)

Wait for the solver to receive a response for a submitted problem.

Blocking call that waits for a *`Future`* object to complete.

Parameters **`timeout`** (*`float, optional, default=None`*) – Maximum number of seconds to await completion. If None, waits indefinitely.

Returns True if solver received a response.

Return type Boolean

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and tries waiting for 10 seconds for sampling to complete.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = solver.sample_qubo(Q, num_reads=100)   # doctest: +SKIP
>>> computation.wait(timeout=10)    # doctest: +SKIP
False
>>> computation.remote_status
u'IN_PROGRESS'
>>> computation.wait(timeout=10)  # doctest: +SKIP
True
>>> computation.remote_status   # doctest: +SKIP
u'COMPLETED'
>>> client.close()
```

dwave.cloud.computation.Future.wait_multiple

static Future.**wait_multiple**(*futures*, *min_done=None*, *timeout=None*)

Wait for multiple *Future* objects to complete.

Blocking call that uses an event object to emulate multi-wait for Python.

Parameters

- **futures** (*list of Futures*) – List of *Future* objects to await.
- **min_done** (*int, optional, default=None*) – Minimum required completions to end the waiting. The wait is terminated when this number of results are ready. If None, waits for all the *Future* objects to complete.
- **timeout** (*float, optional, default=None*) – Maximum number of seconds to await completion. If None, waits indefinitely.

Returns completed and not completed submitted tasks. Similar to *concurrent.futures.wait()* method's returned two-tuple of *done* and *not_done* sets.

Return type Two-tuple of *Future* objects

See also:

as_completed() for a blocking iterable of resolved futures similar to *concurrent.futures.as_completed()* method.

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource 3 times for differing numers of samples, and waits for sampling to complete on any two of the submissions. The wait ends with the completion of two submissions while the third is still in progress. (A more typical approach would use something like `first = next(Future.as_completed(computation))` instead.)

```
>>> import dwave.cloud as dc
>>> client = dc.Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = [solver.sample_qubo(Q, num_reads=1000),
...                solver.sample_qubo(Q, num_reads=50),
...                solver.sample_qubo(Q, num_reads=10)]   # doctest: +SKIP
>>> dc.computation.Future.wait_multiple(computation, min_done=1)    # doctest:␣
↪+SKIP
([<dwave.cloud.computation.Future at 0x17dde518>,
  <dwave.cloud.computation.Future at 0x17ddee80>],
 [<dwave.cloud.computation.Future at 0x15078080>])
>>> print(computation[0].done())   # doctest: +SKIP
False
>>> print(computation[1].done())  # doctest: +SKIP
True
>>> print(computation[2].done())   # doctest: +SKIP
True
>>> client.close()
```

dwave.cloud.computation.Future.done

Future.**done**()

Check whether the solver received a response for a submitted problem.

Non-blocking call that checks whether the solver has received a response from the remote resource.

Returns True if solver received a response.

Return type Boolean

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and checks a couple of times whether sampling is completed.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = solver.sample_qubo(Q, num_reads=100)    # doctest: +SKIP
>>> computation.done()  # doctest: +SKIP
False
>>> computation.done()    # doctest: +SKIP
True
>>> client.close()
```

dwave.cloud.computation.Future.cancel

Future.**cancel**()

Try to cancel the problem corresponding to this result.

Non-blocking call to the remote resource in a best-effort attempt to prevent execution of a problem.

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem to a remote D-Wave resource for 100 samples, and tries (and in this case succeeds) to cancel it.

```
>>> from dwave.cloud import Client
>>> client = Client.from_config()
>>> solver = client.get_solver()
>>> u, v = next(iter(solver.edges))
>>> Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
>>> computation = solver.sample_qubo(Q, num_reads=100)    # doctest: +SKIP
>>> computation.cancel()  # doctest: +SKIP
>>> computation.done()    # doctest: +SKIP
True
>>> computation.remote_status    # doctest: +SKIP
u'CANCELLED'
>>> client.close()
```

Properties

Future.samples	State buffer for the submitted job.
Future.energies	Energy buffer for the submitted job.
Future.occurrences	Occurrences buffer for the submitted job.
Future.timing	Timing information about a solver operation.

dwave.cloud.computation.Future.samples

Future.**samples**

State buffer for the submitted job.

First calls to access data of a *Future* object are blocking; subsequent access to this property is non-blocking.

Returns Samples on the nodes of solver's graph.

Return type list of lists or NumPy matrix

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple QUBO problem (representing a Boolean NOT gate by a penalty function) to a remote D-Wave resource for 5 samples, and prints part of the returned result (the relevant samples).

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     Q = {(u, u): -1, (u, v): 0, (v, u): 2, (v, v): -1}
...     computation = solver.sample_qubo(Q, num_reads=5)
...     for i in range(5):
...         print(computation.samples[i][u], computation.samples[i][v])
...
...
(1, 0)
(0, 1)
(0, 1)
(1, 0)
(0, 1)
```

dwave.cloud.computation.Future.energies

Future.**energies**

Energy buffer for the submitted job.

First calls to access data of a *Future* object are blocking; subsequent access to this property is non-blocking.

Returns Energies for each set of samples.

Return type list or NumPy matrix of doubles

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a random Ising problem (+1 or -1 values of linear and quadratic biases on all nodes and edges, respectively, of the solver's garph) to a remote D-Wave resource for 10 samples, and prints the returned energies.

```
>>> import random
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     linear = {index: random.choice([-1, 1]) for index in solver.nodes}
...     quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}
...     computation = solver.sample_ising(linear, quad, num_reads=10)
...     print(computation.energies)
...
[-3976.0, -3974.0, -3972.0, -3970.0, -3968.0, -3968.0, -3966.0,
 -3964.0, -3964.0, -3960.0]
```

dwave.cloud.computation.Future.occurrences

`Future.`**`occurrences`**

Occurrences buffer for the submitted job.

First calls to access data of a *`Future`* object are blocking; subsequent access to this property is non-blocking.

Returns Occurrences. When returned results are ordered in a histogram, *occurrences* indicates the number of times a particular solution recurred.

Return type list or NumPy matrix of doubles

Examples

This example creates a solver using the local system's default D-Wave Cloud Client configuration file, submits a simple Ising problem with several ground states to a remote D-Wave resource for 20 samples, and prints the returned results, which are ordered as a histogram. The problem's ground states tend to recur frequently, and so those solutions have *occurrences* greater than 1.

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:  # doctest: +SKIP
...     solver = client.get_solver()
...     quad = {(16, 20): -1, (17, 20): 1, (16, 21): 1, (17, 21): 1}
...     computation = solver.sample_ising({}, quad, num_reads=500, answer_mode=
↪'histogram')
...     for i in range(len(computation.occurrences)):
...         print(computation.samples[i][16],computation.samples[i][17],␣
↪computation.samples[i][20], computation.samples[i][21], ' --> ', computation.
↪energies[i], computation.occurrences[i])
...
(-1, 1, -1, -1, ' --> ', -2.0, 41)
(-1, -1, -1, 1, ' --> ', -2.0, 53)
(1, -1, 1, 1, ' --> ', -2.0, 55)
(1, 1, -1, -1, ' --> ', -2.0, 52)
(1, 1, 1, -1, ' --> ', -2.0, 60)
(1, -1, 1, -1, ' --> ', -2.0, 196)
(-1, 1, -1, 1, ' --> ', -2.0, 15)
(-1, -1, 1, 1, ' --> ', -2.0, 28)
```

dwave.cloud.computation.Future.timing

Future.**timing**

Timing information about a solver operation.

Mapping from string keys to numeric values representing timing details for a submitted job as returned from the remote resource. Keys are dependant on the particular solver.

First calls to access data of a *Future* object are blocking; subsequent access to this property is non-blocking.

Returns Mapping from string keys to numeric values representing timing information.

Return type dict

Examples

This example creates a client using the local system's default D-Wave Cloud Client configuration file, which is configured to access a D-Wave 2000Q QPU, submits a simple *Ising* problem (opposite linear biases on two coupled qubits) for 5 samples, and prints timing information for the job.

```
>>> from dwave.cloud import Client
>>> with Client.from_config() as client:
...     solver = client.get_solver()
...     u, v = next(iter(solver.edges))
...     computation = solver.sample_ising({u: -1, v: 1},{}, num_reads=5)    #␣
↪doctest: +SKIP
...     print(computation.timing)
...
{u'total_real_time': 10961, u'anneal_time_per_run': 20,
>>> # Snipped above response for brevity
```

2.2.5 Exceptions

exception dwave.cloud.exceptions.**CanceledFutureError**

An exception raised when code tries to read from a canceled future.

exception dwave.cloud.exceptions.**ConfigFileError**

Base exception for all config file processing errors.

exception dwave.cloud.exceptions.**ConfigFileParseError**

Invalid format of config file.

exception dwave.cloud.exceptions.**ConfigFileReadError**

Non-existing or unreadable config file specified or implied.

exception dwave.cloud.exceptions.**InvalidAPIResponseError**

Raised when an invalid/unexpected response from D-Wave Solver API is received.

exception dwave.cloud.exceptions.**PollingTimeout**

Problem polling timed out.

exception dwave.cloud.exceptions.**RequestTimeout**

REST API request timed out.

exception dwave.cloud.exceptions.**SolverAuthenticationError**

An exception raised when there is an authentication error.

exception dwave.cloud.exceptions.**SolverError**

Generic base class for all solver-related errors.

exception dwave.cloud.exceptions.**SolverFailureError**
An exception raised when there is a remote failure calling a solver.

exception dwave.cloud.exceptions.**SolverNotFoundError**
Solver with matching feature set not found / not available.

exception dwave.cloud.exceptions.**SolverOfflineError**
Action attempted on an offline solver.

exception dwave.cloud.exceptions.**Timeout**
General timeout error.

exception dwave.cloud.exceptions.**UnsupportedSolverError**
The solver we received from the API is not supported by the client.

2.3 Installation

Compatible with Python 2 and 3:

```
pip install dwave-cloud-client
```

To install from source (available on GitHub in dwavesystems/dwave-cloud-client repo):

```
pip install -r requirements.txt
python setup.py install
```

2.4 License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative

Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "[]" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

2.5 Bibliography

CHAPTER 3

Indices and tables

- genindex
- modindex
- search
- Glossary

Python Module Index

d

Index

T

U

W