
DAPT Documentation

Release 0.9.2

Ben Duggan

Mar 27, 2021

Contents:

1 Overview	3
1.1 Install	3
1.2 Usage	4
1.3 DAPT API Documentation	4
1.4 Supported Online Services	27
1.5 Examples	31
1.6 Development Guide	32
Python Module Index	35
Index	37

A library to assist with running parameter sets across multiple systems. The goal of this library is to provide a tool set and pipeline that make organizing, running and analyzing a large amount of parameter easier. Some of the highlights include:

- Provide an easy way to run parameter sets.
- Protocol for allowing teams to run parameter sets concurrently.
- Use Google Sheets as a database to host and manage parameter sets.
- Access to the Box API which allows files to be uploaded to box.

When working on a project with or without access to high performance computing (HPC), there is often a need to perform large parameter sweeps. Before developing DAPT, there were several problems the ECM team in Dr. Paul Macklin’s research lab identified. First, it was difficult to manage a large number of parameter sets with a large number of parameters. Second, it would be nice to use Google Sheets to run the parameters for easier collaboration and management. Third, only one person in the group would be running all the parameters, making their computer useless for the duration of the runs. Finally, we needed to upload the data to Box for permanent storage and to allow the rest of the team to view the data.

DAPT was written to solve these problems. A “database” (CSV or Google Sheet) is used to store a list of parameter sets. This database is managed by the *Param* class and provides methods to interact with and manage parameter sets. the *Box* class allows data to be uploaded to [Box.com](#). Sensitive API credentials can be stored in a config file (via the *Config* class) which can also be accessed by users to get other variables.

Future versions of the project will work to improve documentation, add examples, cleanup current functionality and add more features. While most of the *dapt* module is documented, the intended way of using each method is not clearly explained. There are examples given for the main features, however, again there is not a satisfactory amount of documentation. Some of the exciting new features to come will be notification and logging integration. For example, we would like to add Slack notification so teams can be notified if there is an error with a test.

1.1 Install

The easiest way to install DAPT is using pip. To do so type:

```
pip install dapt
```

Alternatively, you can download a version the project. It is recommended to download a [release](#) of the project from GitHub for improved stability. If you would like to download the most up to date version, then download the [repo](#) or clone it on your machine `git clone https://github.com/BenSDuggan/DAPT`. Once downloaded navigate to the root of the project (DAPT) and run `pip install -r requirements.txt` to install all of the dependencies. If you use this method of installation, you will need to write all of your Python scripts using DAPT in the root directory of the project. For these reasons, it’s recommended to only use this method if would like to contribute to the project.

You can then test to make sure everything installed by starting a python session and then running:

```
import dapt
dapt.__version__
```

You should see a version looking like 0.9.*.

DAPT is maintained for Python >= 3.6 and a full list of requirements is given in [requirements.txt](#)

You can now use the library! However, the functionality can be greatly increased by connecting some other services such as Google Sheets or Box. Checkout the full list of [Supported Online Services](#).

1.2 Usage

Once you have [installed](#) DAPT and have verified that it's installed correctly you can start setting it up for actual parameter runs. There are several ways to run DAPT but the basic philosophy is outlined below. You can also look at specific [examples](#). To use DAPT, start by importing it.

```
import dapt
```

DAPT can be run with or without a configuration file. The code is easier to use with a config file but it is not strictly necessary. If you would like to create a config file, you should consult the [Config](#) class documentation. Assuming you have created a config file called `config.json`, you can create a Config object.

```
config = dapt.Config(path='config.json')
```

Next, you need to pick a [Database](#). A Database is a class that allows you get access a list of parameter sets. There are currently two Databases: a [Delimited file](#) and [Google sheets](#). Below shows how to create the database objects.

```
db = data.Delimited_file('csv_file.csv', delimiter=',') # Create a Delimited file DB
# or
db = data.Sheet(config=config) # Create a Sheet DB with a config file
# or
spreadsheet_id = 'xxxxxx' # Google Sheet spreadsheet id
creds = 'credentials.json' # Path to Google Sheets API credentials
db = data.Sheet(spreadsheet_id=spreadsheet_id, creds=creds) # Create a Sheet DB with
↳ a config file
```

Now you can create the [Param](#) object to start processing parameters. Create a Param object with the code below.

```
param = dapt.Param(db, config=config)
```

You can now use the methods in the Param class to get the next parameter set and manage the parameter set.

1.3 DAPT API Documentation

This is the DAPT reference guide that shows how the APIs work.

1.3.1 Config

The Config class allows user and API settings to be saved and updated using a configuration file. A config class is not required by DAPT but using one provides several advantages. First, it makes initializing a class much easier as each class can pull required attributes from a config. Second, API credentials can be stored in a config, allowing

credentials to be kept in one place. Third, by allowing API tokens to be stored, there is no need to reauthenticate a service (assuming the tokens are still valid). Finally, it provides a way for users to have their own settings file.

Configuration files use **JSON** (JavaScript Object Notation) format. A detailed understanding of JSON is not required, but the basics should be understood. There are two main components of JSON files: key/value pairs (objects) and arrays/lists. When using key/value pairs, the pairs must be surrounded by curly braces and separated with commas. Objects are separated by colons (:) and keys must be surrounded by quotes. Values can be objects, arrays, strings, numbers, booleans, or null. Bellow is a sample JSON file that could be used by DAPT.

Listing 1: Example of a simple JSON file.

```
{
  "performed-by": "Ben",
  "num-of-runs": -1,
  "testing-variables":
  {
    "executable-path": "./main",
    "output-path": "output/"
  }
}
```

The `performed-by` and `num-of-runs` keys are reserved DAPT *fields*. These cause DAPT to add additional information during tests, initiate classes automatically, and change the testing behavior. The list of reserved fields and their behaviors are shown bellow. The `testing-variables` key has an object in it that might be used for a specific testing parameters. The name of this key does not matter as long as it is not a reserved field. To see how the Config class is used checkout the *usage* section or class documentation.

Fields

There are many key-value pairs which can be used in the configuration to make DAPT behave in a particular way. These keys are called fields. These fields are reserved and should not be used in your config file unless you expect DAPT to use them. A list of top level fields is provided below.

Fields	Description
<code>num-of-runs</code> (int)	The number of parameter sets to run.
<code>performed-by</code> (str)	The username of the person that ran the parameter set.
<code>last-test</code> (str)	The last test id that was run. If a test exits before completing, it will be re-ran.
<code>computer-strength</code> (int)	Only run tests on computers with sufficient power. The parameter set will only be run if this value is greater than or equal that of the parameter sets <code>computer-strength</code> .
<code>google-sheets</code> (str)	Values used by the <i>Google Sheets</i> storage API.
<code>delimited-file</code> (str)	Values used by the <i>Delimited file</i> database class.
<code>box</code> (str)	Values used by the <i>Box</i> storage API.
<code>pretty-save</code> (bool)	Output the config in a ~pretty~ way. True by default.

Some of these fields are used by other DAPT classes to store values. For example, the `google-sheets` field has many sub-fields that set parameters in the class automatically. The `spreadsheet-id` sub-field sets the spreadsheet ID that should be used as the database. These sub-fields are not listed above. They are notable, however, because you may accidentally find one of these sub-fields if you recursively search a config file. If you are worried about

accidentally using one of these fields, the `FULL_CONFIG` variable in the `config` module contains all of the config fields.

Usage

For these examples, the *example JSON* shown above is used, stored in a file named `example.json`. To create a `Config` object the path to the JSON file must be provided.

```
>>> config = dapt.Config(path="example.json")
```

The configuration should be accessed using the `get_value()` method. This method will returned the value of the associated key. Keys can be provided as a string or a list where elements are the path to the value. The `num-or-runs` attribute can be accessed as shown bellow.

```
>>> config.get_value("num-of-runs")
-1
```

If you wanted to find the value of `output-path` then you specify the path to it.

```
>>> config.get_value(["testing-variables", "output-path"])
'output/'
```

Alternatively, the `output-path` key can be accessed by using the `recursive` flag. This flag makes the `get_value()` method recursively search the JSON tree for the first occupance of the specified key. This flag will increase the look-up time and may not return the value you expect if multiple keys with that name are present.

The advantage of using the `get_value()` method is that `None` will be returned if the value is not found.

The configuration dictionary can be accessed indirectly by treating the `Config` object as a dictionary.

```
>>> config["num-of-runs"]
-1
>>> config["testing-variables"]["output-path"]
'output/'
```

Using this approach, the length of the dictionary can be accessed using Python's internal `len()` function or any other *dict* method. The keys of the dictionary can be accessed using the `keys()` method.

Before accessing a value in the config, it is good to check that it exists. This can be done using the `has_value()` method. This method returns `True` if there is a non-`None` value in the config for the given key. The key and recursive attributes behave the same as with the `get_value()` method. For example, to check that the `output-path` key exists you could run the following and expect a return value of `True`:

```
>>> config.has_value(["testing-variables", "output-path"])
True
```

If you checked for the key `foo`, then `has_value()` would return `False`.

To add key-value pairs to the configuration or update values, the `update()` method should be used. This method will allow the configuration to change and save it to the JSON file. The configuration can be changed in four different ways. First, by providing the key as a string. Second, by providing the key as an array representing a path to the value. The third method uses a `str` for the string and recursively finds the first occurrence of the key in the config. Lastly, the configuration can be updated by accessing the dictionary directly. Then `update()` can be ran without parameters to save the config. The second and last methods are required to access nested key-value pairs. All of these methods work to add new data or change values in the configuration.

```
>>> config.update(key="performed-by", value="John", recursive=False)
{'performed-by': 'John', 'num-of-runs': -1,
 'testing-variables': {'executable-path': './main', 'output-path': 'output/'}}
>>> config.update(key=["testing-variables", "executable-path"], value="main.exe",
                  recursive=False)
{'performed-by': 'John', 'num-of-runs': -1,
 'testing-variables': {'executable-path': 'main.exe', 'output-path': 'output/'}}
>>> config.update(key="output-path", value="save/", recursive=True)
{'performed-by': 'John', 'num-of-runs': -1,
 'testing-variables': {'executable-path': 'main.exe', 'output-path': 'save/'}}
```

```
>>> config["num-of-runs"] = 3
>>> config.update()
{'performed-by': 'John', 'num-of-runs': 3,
 'testing-variables': {'executable-path': 'main.exe', 'output-path': 'save/'}}
```

When creating a new configuration file, the `create()` method can be used. This static method will create a default configuration file at the path provided. This file contains all of the possible fields used by DAPT.

```
>>> dapt.config.Config.create(path="new-config.json")
```

Configuration files can contain sensitive API credentials or passwords. Storing these in plane text or publishing configuration files online is unsecure as people can then gain access to your online services. To combat this you can “safe” the configuration file. The `safe()` method will remove all API credentials from the configuration so the file cannot be used to access your APIs. Currently, this this process is one-way and the credentials cannot be recovered. However, in the future this will encrypt the file can be distributed online and unlocked by people with the correct password.

```
class dapt.config.Config(path='config.json')
```

Bases: object

Class which loads and allows for editing of a config file.

Parameters `path` (*string*) – path to config file

```
static create (path='config.json')
```

Creates a config file with the reserved keys inserted. The `DEFAULT_CONFIG` will be used.

Parameters `path` (*string*) – path where config file will be written

Returns A `Config` object with the newly created default configuration

```
get_value (key, recursive=False, default=None)
```

Get the first value of the given key or return `None` if one doesn't exist.

Parameters

- **key** (*str or list*) – the key (given as a string) or List containing the path to the value
- **recursive** (*bool*) – recursively look through the config for the given key. False by default. If recursive is set to True then key must be a string.
- **default** (*obj*) – A default value to use if no value can be found. This is `None` by default.

Returns The value associated to the given key or `None` if the key is not in the dictionary.

```
has_value (key, recursive=False)
```

Checks to see if the config contains the key and a value other than `None`.

Parameters

- **key** (*str or list*) – the key (given as a string) or List containing the path to the value
- **recursive** (*bool*) – recursively look through the config for the given key. False by default. If recursive is set to True then key must be a string.

Returns True if the key has a value and it's not None, False otherwise.

keys ()

Get the keys from the configuration. This method only returns the keys at the top of the dictionary. It will not return any nested keys.

Returns A list containing the keys in the dictionary.

read ()

Reads the file with path set to self.path

Returns Dictionary of config file

static safe (path='config.json')

Safe config file by removing accessToken and refreshToken.

Parameters **path** (*string*) – path where config file will be written

update (key=None, value=None, recursive=False)

Given a key and associated value, updated the config file. Alternatively, you can give no arguments and the config dict will be saved. You can also do both.

Parameters

- **key** (*str or list*) – the key (given as a string) or List containing the path to the value. If *None* is given then nothing will be updated in the dictionary.
- **value** (*str*) – the value associated of the key.
- **recursive** (*bool*) – recursively look through the config for the given key. False by default. If recursive is set to True then key must be a string.

Returns Dictionary of config file

1.3.2 Databases

Database Overview

Databases are the places where parameter spaces live. You must use a database in DAPT and the *Param class* requires one to be provided. DAPT views databases similarly to a spreadsheet. Databases can be local or remote. If a database is local (e.g. *Delimited file*), then only one person can run the parameters. When a remote database (e.g. *Google Sheets*) is used, then multiple people can run the tests simultaneously.

The databases provided in DAPT are all built off of the *Base Database*. This ensures that databases can be interchanged easily. For example, if you were using the *Delimited file* database, you could provide the *Google Sheets* to the *Param class* instead. This works because all databases must support the same core functions within the *Base Database* class. Some databases may have additional methods which work better with its design.

The only difference between these classes, from the user level, is the `init` and `connect()` methods. Database classes can be initialized using only a *Config* class. This makes it easy to swap between and initialize databases. Because some databases required users to login, you must connect to it before it can be accessed. This should be done before trying to access the data.

Schematic

The database is made up of tables, identified by a key (`str` or `int`), which contain columns and rows. The tables of database hold a parameter space and each row is a parameter set. A column contains particular fields (attributes) within the space. The column names are called fields and should be strings. Rows are identified by an index, similar to a list. The indexing starts from zero and increments.

Cell holds the value of a particular row and column. Currently, all cells are considered strings, however, some databases allow for other types to be inserted, or automatically inference the type. For this reason, you might need to cast cells to different type.

Config

It is recommend to use a *Config* with the database classes. While all classes can be instantiated without a `Config`, using one greatly increases useability and simplifies switching between databases. Each database has a reserved `Config` key (listed below). The value will be a dictionary with API credentials and database settings. The the structure of configuration is similar between databases, but specific to the API connection requirements. Specific database classes have more information on required configuration contents.

Fields	Description
<code>delimited-file (str)</code>	Reserved word for <code>Delimited_file</code> database.
<code>sheets (str)</code>	Reserved word for <code>Sheets</code> database.

Usage

Because the usage for each database is almost identical, it will be explained here instead of in the submodules. More explications on the methods, checkout the *Base Database*. The connection steps for each database will be explained within the respective documentation.

For this example, the *Sample database* will be used. By calling the `sample_db()` method, an example *Delimited file* class is created.

```
>>> db = dapt.tools.sample_db(file_name='sample_db.csv', delimiter=',')
```

This method returns an instance of the database, but the line below shows how a new database instance can be created.

```
>>> db = dapt.db.Delimited_file(path='sample_db.csv', delimiter=',')
```

The *Delimited file* class doesn't need to connect to anything, but most databases will so you should always run `connect()`.

```
>>> db.connect()
True
```

The table can simply be viewed by running:

```
>>> db.get_table()
[{'id': 't1', 'start-time': '2019-09-06 17:23', 'end-time': '2019-09-06 17:36',
  'status': 'finished', 'a': '2', 'b': '4', 'c': '6'},
 {'id': 't2', 'start-time': '', 'end-time': '', 'status': '', 'a': '10', 'b': '10', 'c
  ↳': ''},
 {'id': 't3', 'start-time': '', 'end-time': '', 'status': '', 'a': '10', 'b': '-10', 'c
  ↳': ''}]
```

Tables are represented as an array of dictionaries. Each element is a parameter set. The keys in the dictionary are fields and values are specific cells in the table. The fields of the table can be retrieved using the `fields()` method.

```
>>> db.fields()
['id', 'start-time', 'status', 'a', 'b', 'c']
```

A specific cell can be changed using the `update_cell()` method. This method requires the row index (starting from 0), field, and updated value. For example, we can update field `c` with id “t2” to 20.

```
>>> db.update_cell(1, 'c', 20)
True
```

An entire row can be updated with the `update_row()` method. This method only requires the row index (starting from 0) and the updated row (given as a dictionary).

```
>>> db.update_row(1, {'id':'t2', 'start-time':'2019-09-06 17:37',
'end-time':'2019-09-06 17:55', 'status':'finished', 'a':'10', 'b':'10', 'c':'20'})
[{'id':'t1', 'start-time':'2019-09-06 17:23', 'end-time':'2019-09-06 17:36',
'status':'finished', 'a':'2', 'b':'4', 'c':'6'},
 {'id':'t2', 'start-time':'2019-09-06 17:37', 'end-time':'2019-09-06 17:55',
'status':'finished', 'a':'10', 'b':'10', 'c':'20'},
 {'id':'t3', 'start-time':'', 'end-time':'', 'status':'', 'a':'10', 'b':'-10', 'c':
↪ ''}]
```

Base Database

The `Database` class is the basic interface for adding parameter set hosting services. The idea is that the core methods stay the same so that the inner workings can use multiple sources to access the parameter sets. These methods should be overridden when making a class that inherits `Database`. You shouldn’t expect that any other method will be called by the `Parameter` class, the main class that uses databases. It may be beneficial to add helper methods though (e.g. `get_worksheet()` in *Google Sheets*).

Databases should give key-value pairs, where the keys are the “ids” of the table and the values are the values in that given row. When getting the table, the result should be an array of dictionaries that contain the contents of the row.

class `dapt.db.base.Database`

Bases: `object`

An interface for accessing and setting parameter set data.

connect ()

The method used to connect to the database and log the user in. Some databases won’t need to use the `connect` method, but it should be called regardless to prevent problems.

Returns True if the database connected successfully and False otherwise.

connected ()

Check to see if the API is connected to the server and working.

Returns True if the API is connected to the server and False otherwise.

fields ()

Get the fields(attributes) of the parameter set

Returns Array of strings with each element being a field (order is preserved if possible)

get_keys ()

Deprecated since version 0.9.3.

This method is being deprecated in favor of the `fields` method. It will be removed in version 0.9.5.

Get the keys of the parameter set

Returns Array of strings with each element being a key (order is preserved if possible)

get_table()

Get the table from the database.

Returns An array with each element being a dictionary of the key-value pairs for the row in the database.

update_cell(row_id, field, value)

Update the cell specified by the `row_id` and `field`.

Parameters

- **row_id** (*int*) – the row id to replace
- **field** (*str*) – the field of the value to replace
- **value** (*object*) – the value to insert into the cell

Returns A boolean that is True if successfully inserted and False otherwise.

update_row(row_index, values)

Update the row at the `row_index` with the values given.

Parameters

- **row_index** (*int*) – the index of the row to replace
- **values** (*Dict*) – the key-value pairs that should be inserted

Returns A boolean that is True if successfully inserted and False otherwise.

Delimited file

This class uses a local [delimited file](#) (e.g. CSV, TSV) as a database for parameter testing. Delimited files represent a table where the row is a line in the file, and each column is separated by a delimiter. The delimiter is often a comma (comma-separated file) or a tab (tab-separated file). A CSV file might look like this:

```
id,start-time,end-time,status,a,b,c
t1,2019-09-06 17:23,2019-09-06 17:36,finished,2,4,6
t2,,,,10,10,
t3,,,,10,-10,
```

and represent a table that looks like this:

id	start-time	end-time	status	a	b	c
t1	2019-09-06 17:23	2019-09-06 17:36	finished	2	4	6
t2				10	10	
t3				10	-10	

Because these files are stored on the users computer, there is no way for a team to work on the parameter set distributively (without manually dividing the parameter sets up).

Delimited files can have a header which gives the columns names. The header is the first row of the table. Headers must be included with DAPT's `Delimited_file` class.

DAPT provides a method named `sample_db()` which creates the sample CSV above. You can create this file by running that method and then use the `Delimited_file` class with it.

```
>>> db = dapt.tools.sample_db(file_name='sample_db.csv', delimiter=',')
>>> db.fields()
['id', 'start-time', 'end-time', 'status', 'a', 'b', 'c']
```

Config

Delimited file can accept a *Config* class. The values listed in the table below are the same attributes used to instantiate the class. These values should be placed inside a JSON object named `delimited-file`.

Fields	Description
<code>path</code> (str)	The path, from the execution directory, to the delimited file.
<code>delimiter</code> (str)	How the columns of the file are separated.

The default configuration looks like this:

Listing 2: Sample JSON configuration for `Delimited_file`

```
{
  "delimited-file" : {
    "path" : "parameters.csv",
    "delimiter" : ",",
  }
}
```

class `dapt.db.delimited_file.Delimited_file` (*args, **kwargs)

Bases: `dapt.db.base.Database`

An interface for accessing and setting parameter set data.

Keyword Arguments

- **path** (str) – path to delimited file
- **delimiter** (str) – the delimiter of the CSV. `,` by default.
- **config** (Config object) – an Config instance

connect ()

The method used to connect to the database and log the user in. Some databases won't need to use the connect method, but it should be called regardless to prevent problems.

Returns True if the database connected successfully and False otherwise.

connected ()

Check to see if the API is connected to the server and working.

Returns True if the API is connected to the server and False otherwise.

fields ()

Get the fields(attributes) of the parameter set

Returns Array of strings with each element being a field (order is preserved if possible) or None if the file is empty.

get_row_index (column_key, row_value)

Get the row index given the column to look through and row value to match to.

Parameters

- **column_key** (*str*) – the column to use.
- **row_value** (*str*) – the row value to match with in the file and determine the row index.

Returns The index or -1 if it could not be determined

get_table ()

Get the table from the database.

Returns An array with each element being a dictionary of the key-value pairs for the row in the database.

update_cell (*row_index*, *field*, *value*)

Update the cell specified by the `row_id` and `field`.

Parameters

- **row_id** (*int*) – the row id to replace
- **field** (*str*) – the field of the value to replace
- **value** (*object*) – the value to insert into the cell

Returns A boolean that is True if successfully inserted and False otherwise.

update_row (*row_index*, *values*)

Update the row at the `row-index` with the values given.

Parameters

- **row_index** (*int*) – the index of the row to replace
- **values** (*Dict*) – the key-value pairs that should be inserted

Returns A boolean that is True if successfully inserted and False otherwise.

Google Sheets

Class which allows for Google Sheets to be used as parameter set database.

Note: If you have data in the first row, you must have entries in some other row.

Authentication

TODO

Config

The Google Sheets class can be instantiated using a *Config* class. There are several options in the config which are redundant (e.g. `worksheet-id` and `worksheet-title`). They are marked with flags in the table below. These values should be placed inside a JSON object named `google` or `google-sheets`. If the keys are placed inside the `google` key, the values will be shared with other Google APIs (e.g. *Google Drive*).

Fields	Description
<code>spreadsheet-id</code> (str)	The Google spreadsheet ID being used. Found in the URL.
<code>*creds-path</code> (str)	Path to the Google Sheets credentials JSON file.
<code>*creds</code> (dict)	The Google credentials provided from the developer console.
<code>#worksheet-id</code> (int)	The Google Sheets worksheet id. Sheets are indexed at 0.
<code>#worksheet-title</code> (str)	The Google Sheets worksheet title.

* fields should not be used together. If you use them together, `creds` will be used over `creds-path`. # fields should also not be used together and `worksheet-id` will be used.

The default configuration looks like this:

Listing 3: Sample JSON configuration for Sheets

```
{
  "google-sheets" : {
    "spreadsheet-id" : "",
    "creds-path" : "",
    "creds" : {},
    "worksheet-id" : "",
    "worksheet-title" : ""
  }
}
```

class `dapt.db.sheets.Sheet` (*args, **kwargs)

Bases: `dapt.db.base.Database`

An interface for accessing and setting parameter set data. You must either provide a Config object or `client_id` and `client_secret`.

Keyword Arguments

- **config** (`Config`) – A Config object which contains the `client_id` and `client_secret`.
- **spreadsheet_id** (`str`) – the Google Sheets ID `creds` (`str`): the path to the file containing the Google API credentials. Default is `credentials.json`.
- **sheet_id** (`int`) – the the sheet id to use. 0 is used if no value is given for `sheet_title`, `sheet_id` or in the Config
- **sheet_title** (`str`) – the title of the sheet to use

connect ()

The method used to connect to the database and log the user in. Some databases won't need to use the connect method, but it should be called regardless to prevent problems.

Returns gspread client if the database connected successfully and False otherwise.

connected ()

Check to see if the API is connected to the server and working.

Returns True if the API is connected to the server and False otherwise.

fields ()

Get the fields(attributes) of the parameter set

Returns Array of strings with each element being a field (order is preserved if possible)

get_key_index (`column_key`)

Get the column index given the key.

Parameters `column_key` (*str*) – the key to find the index of

Returns The index or -1 if it could not be determined.

`get_row_index` (*column_key*, *row_value*)

Get the row index given the column to look through and row value to match to.

Parameters

- `column_key` (*str*) – the key to find the index of
- `row_value` (*str*) – the value of the cell to find

Returns The index or -1 if it could not be determined.

`get_table` ()

Get the table from the database.

Returns An array with each element being a dictionary of the key-value pairs for the row in the database.

`update_cell` (*row_id*, *field*, *value*)

Update the cell specified by the `row_id` and `field`.

Parameters

- `row_id` (*int*) – the row id to replace
- `field` (*str*) – the field of the value to replace
- `value` (*object*) – the value to insert into the cell

Returns A boolean that is True if successfully inserted and False otherwise.

`update_row` (*row_index*, *values*)

Get the row of the parameter set.

Parameters

- `row_index` (*int*) – the index of the row to replace (starting from 1). Indices less than 1 will return False. Indices greater than the table length will be appended.
- `values` (*Dict*) – the key-value pairs that should be inserted. If the dictionary contains more values than number of columns, the table will be extended.

Returns A boolean that is True if successfully inserted and False otherwise.

`worksheet` (**args*, ***kwargs*)

Get a Google Sheet object. The worksheet id or title are obtained from the Config file or initialization.

Returns A Google Sheet worksheet

1.3.3 Parameter

The parameter module contains the `Param` class that interact with the database to get and manage the parameter spaces. This is the main module that you should interact with.

Database

In order to get the parameters, the `Param` class needs to be given a `Database` instance (e.g. *Google Sheets*, *Delimited file*). The database is where the parameters to be tested live. The database has a couple required fields (attributes) and many optional fields. The *Fields* section provides more information on how the database should be configured.

Each time a new parameter set is requested, the database will be downloaded again. This means that the database can be changed as DAPT is running to add or remove the number of tests. An important note regarding database is that they can be ran local or on the internet. This means that multiple people can work on the parameter set at the same time, thus distributing the computational work load.

Fields

A field is the key (or identifier) used to get the value when a parameter set is returned. Each database is required to have an `id` and `status` field. There are many optional fields which can be used to give additionally information about the run such as start time and who performed the run. Below are the fields that are used with parsing parameter sets. Required parameters are marked with an asterisk(*).

Fields	Description
<code>id</code> * (str)	Unique parameter set installed
<code>status</code> * (str)	The current status of the parameter set. Blank values(default) have not been ran, <code>successful</code> have finished and <code>failed</code> have failed.
<code>start-time</code> (str)	The time that the parameter set began. Times are in UTC time format.
<code>end-time</code> (str)	The time that the parameter set finished. Times are in UTC time format.
<code>performed-by</code> (str)	The username of the person that ran the parameter set.
<code>comments</code> (str)	Any comments such as error messages relating to the parameter set.
<code>computer-strength</code> (int)	The minimum strength that the computer running the test should have. The <code>computer-strength</code> in the <code>Config</code> must be greater than or equal to this value for the test to be ran

The `id` field is a unique identifier for that test. This attribute is used to identify the parameter set and must be given to most of the methods in the `Param` class. The `status` field gives the current status of the test.

There are five main status values: empty, “successful”, “failed”, “in progress”, and other text. When a test has an empty status it indicates that the test has not been ran yet. A status of “successful” indicates that the test has finished successfully, and a “failed” status shows that the test failed.

When you request another parameter set by running `next_parameters()`, the status will automatically be set to “in progress”. If the status is not empty, then DAPT will not offer it when the `next_parameters()` method is called. You can update the status to something you want by calling the `update_status()` method.

Config

The `Config` fields will only be used if they are included in the `Config`. If the fields are excluded, then the the fields will not be added.

Fields	Description
num-of-runs (int)	The number of parameter sets to run.
performed-by (str)	The name of the person that ran the parameter set.
last-test (str)	The last test id that was run. If a test exits before completing, it will be re-ran.
computer-strength (int)	Only run tests on computers with sufficient power. The parameter set will only be run if this value is greater than or equal that of the parameter sets computer-strength.

Usage

To initiate the `Param` class, you must provide a database object. The database used in this example is the `dapt.tools.sample_db()`. A config object can additionally be provided to enable advanced control.

```
>>> param = dapt.Param(db, config=conf)
```

The `param` object is used to interact with parameter sets in the parameter space. To get the next parameter set, you use the `next_parameters()` method. This will return a JSON object containing the parameter set.

```
>>> p = param.next_parameters()
>>> p
{'id': 't2', 'start-time': '2020-12-27 17:21:00', 'end-time': '', 'status': 'in_
↳progress',
 'a': '10', 'b': '10', 'c': ''}
```

The status of the parameter set will automatically be set to “in progress”. To change the status, you can use the `update_status()` method. This method requires the `id` of the parameter set and the new status to be provided. In this case, the `id` is “t2”.

```
>>> p = param.update_status(p['id'], 'adding')
>>> p
{'id': 't2', 'start-time': '2020-12-28 21:11:10', 'end-time': '', 'status': 'adding',
 'a': '10', 'b': '10', 'c': ''}
```

The status can be updated as many times as you’d like. Once you have finished running the test, you can mark it as successful or failed using the respective method. These methods require the `id` of the parameter set to be specified.

```
>>> param.successful(p['id'])
{'id': 't2', 'start-time': '2020-12-28 21:11:10', 'end-time': '2020-12-28 21:24:50',
 'status': 'successful', 'a': '10', 'b': '10', 'c': ''}
```

If you mark the test as failed, the reason can optionally be provided.

```
class dapt.param.Param(database, config=None)
    Bases: object
```

Create a `Param` instance with a database and optional config file.

Parameters

- **database** (`Database`) – a Database instance (such as *Google Sheets*, *Delimited file*)
- **config** (`Config`) – a config object which allows for more features. This is optional.

```
failed(id, err=)
```

Mark a parameter set as failed to completed.

Parameters

- **id** (*str*) – the id of the parameter set to use
- **err** (*str*) – the error message. Empty by default.

Returns The new parameter set that has been updated or False if not able to update.

next_parameters ()

Get the next parameter set if one exists

Returns An OrderedDict containing the key-value pairs from that parameter set or None if there are no more to sets.

successful (*id*)

Mark a parameter set as successfully completed.

Parameters **id** (*str*) – the id of the parameter set to use

Returns The new parameter set that has been updated or False if not able to update.

update_status (*id, status*)

Update the status of the selected parameter. If status is not included in the parameter set keys then nothing will be updated.

Parameters **id** (*str*) – the id of the parameter set to use

Returns The new parameter set that has been updated or False if not able to update.

1.3.4 Storage

Storage Overview

This module contains classes and functions that assist with the storage APIs. It includes the `Storage` class and methods to deal with overwriting files/folders.

Because the APIs of services are all different, DAPT calls the resource identification a `file_id`. Even if the resource is not a file, it is called a `file_id`. This is similar to everything is a file in Linux.

To attempt to make paths easier to navigate, the download and upload methods include a `folder` and `name` attribute. So if you wanted to upload a file in `foo/bar/file.py`, you would set `folder` to `foo/bar` and `name` to `file.py`. You can omit the `folder` attribute and the current directory will be used. The motivation for this is to 1) make the file name and save location explicit, and 2) standardize these variables across the download and upload functions. When downloading a resource, you may want to keep the file name from the service, or rename it. By setting the `name` attribute to `None`, the name of the resource will be used.

Storage base

The `Storage` class is designed to provide a standard interface for adding APIs that enable storage. This class defines the basic required functions that must be implemented for two classes inheriting this class to work in the same workflow, assuming the correct API keys are used. Switching storage objects should work seamlessly, if a `Config` object is used to initialize the `Storage` object. If the API credentials are folderectly provided, this cannot be guaranteed because different services had different methods of initialization.

Different APIs might have different methods for identifying files. For example, Box uses IDs for files and folders, but another service might use a path from the root directory. The method of identifying files or folders is called a `fid` (file/folder identification) in DAPT. Different implementations might use different protocols for files and folders, so the `Storage` methods should take care of this.

Required methods

There are four required methods that all Storage objects must implement. The required methods are download, delete, rename, and upload. These methods are based off REST APIs, although the underlying implementation do not need to use REST.

class `dapt.storage.base.Storage`

Bases: `object`

connect ()

The method used to connect to the database and log the user in. Some databases won't need to use the connect method, but it should be called regardless to prevent problems.

Returns True if the database connected successfully and False otherwise.

connected ()

Check to see if the API is connected to the server and working.

Returns True if the API is connected to the server and False otherwise.

delete_file (*file_id*)

Delete the the given file.

Parameters `file_id` (*str*) – The file identification to be downloaded

Returns True if successful and False otherwise

delete_folder (*file_id*)

Delete the given folder.

Parameters `file_id` (*str*) – The folder identification to be downloaded

Returns True if successful and False otherwise

download_file (*file_id*, *folder='.'*, *name=None*, *overwrite=True*)

Download the file at the given `file_id` to the given path.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **folder** (*str*) – The directory where the file should be saved
- **name** (*str*) – The name that the file should be saved as. If None is given (default), then the name of the file on the resource will be used.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

download_folder (*file_id*, *folder='.'*, *name=None*, *overwrite=True*)

Download the folder at the given `file_id` to the given path.

Parameters

- **file_id** (*str*) – The folder identification to be downloaded
- **folder** (*str*) – The directory where the file should be saved
- **name** (*str*) – The name that the file should be saved as. If None is given (default), then the name of the file on the resource will be used.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

rename_file (*file_id*, *name*)

Rename the given file.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

rename_folder (*file_id*, *name*)

Rename the given folder.

Parameters

- **file_id** (*str*) – The folder identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

upload_file (*file_id*, *name*, *folder*='.', *overwrite*=True)

Upload a file to the given folder.

Parameters

- **file_id** (*str*) – The folder where the file should be saved.
- **name** (*str*) – The name that the file should be uploaded.
- **folder** (*str*) – The directory where the file is stored.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

upload_folder (*file_id*, *name*, *folder*='.', *overwrite*=True)

Upload a folder to the given folder.

Parameters

- **file_id** (*str*) – The folder where the folder should be saved.
- **name** (*str*) – The name that the file should be uploaded.
- **folder** (*str*) – The directory where the file is stored.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

`dapt.storage.base.check_overwrite_file` (*folder*, *name*, *overwrite*, *remove_existing*)

This method checks to see if the file at the path specified should be overwritten.

Parameters

- **folder** (*str*) – The directory where the file might be.
- **name** (*str*) – The name of the file
- **overwrite** (*bool*) – Should the file be overwritten
- **remove_existing** – (bool): Should the file be deleted if it already exists

Returns True if the file should be overwritten and False otherwise.

`dapt.storage.base.check_overwrite_folder` (*folder*, *name*, *overwrite*, *make_folder*)

This method checks to see if the file at the path specified should be overwritten.

Parameters

- **folder** (*str*) – The directory where the file might be.
- **name** (*str*) – The name of the folder
- **overwrite** (*bool*) – Should the file be overwritten
- **make_folder** (*bool*) – Should the directory be made if it passes the overwrite test.

Returns True if the folder can be overwritten and was created (if `make_folder` was True) and False otherwise.

`dapt.storage.base.get_mime_type(name)`

Get the MIME type of the given file based on it's file extension.

Parameters **name** (*str*) – the name of the file including the extension

Returns The MIME type of the file and None if the MIME type cannot be found.

Box

Class that allows for access to the box API and methods to directly upload files. If you wish to use the Box API you should view the [install](#).

Authentication

In order for the Box API to work, it needs to get a user specific access and refresh token. Box provides access tokens to users which are a session key. They remain active for one hour at which time they must be refreshed using the refresh token. Once a new access and refresh token has been given, the old one will no longer work.

The tokens can be provided in three ways. First, you can run `Box(...).connect()` which will start a flask webserver. You can then proceed to `127.0.0.1:5000` and log in with your Box username and password. This is done securely through Box and your username and password cannot be extracted. Second, you can insert the access and refresh token in the config file. Then the Box class will use these tokens. The final way to provide the tokens is by directly passign them to `Box(...).connect(access_token=<your access token>, refresh_token=<your refresh token>)`.

On a server, where you have no access to a web browser, you will need to get the tokens using a computer which has a web browser. You can then place those tokens in the config file or directly pass them to the `connect()` method.

Config

The best way to use Box is with a configuration file. Box attributes can be added to the config file as a JSON object which is the value for the key `box`. An sample config file for box is shown bellow.

```
{
  "box" : {
    "client_id" : "xxx",
    "client_secret" : "xxx",
    "access_token" : "xxx",
    "refresh_token" : "xxx",
    "refresh_time" : "xxx"
  }
}
```

class `dapt.storage.box.Box` (*args, **kwargs)

Bases: `dapt.storage.base.Storage`

Class which allows for connection to box API. You must either provide a Config object or client_id and client_secret.

Keyword Arguments

- **config** (`Config`) – A Config object which contains the client_id and client_secret.
- **client_id** (`str`) – The Box client ID.
- **client_secret** (`str`) – The Box client secret.

connect (`access_token=None, refresh_token=None`)

Tries to connect to box using arguments provided in Config and starts server for authorization if not.

Parameters

- **access_token** (`str`) – Optional argument that allows DAPT to connect to box without going through web authentication (assuming refresh_token is given and not expired).
- **refresh_token** (`str`) – Optional argument that allows DAPT to connect to box without going through web authentication (assuming access_token is given and not expired).

Returns Box client if successful

delete_file (`file_id`)

Delete the the given file.

Parameters **file_id** (`str`) – The file identification to be downloaded

Returns True if successful and False otherwise

delete_folder (`folder_id`)

Delete the given folder.

Parameters **folder_id** (`str`) – The folder identification to be downloaded

Returns True if successful and False otherwise

download_file (`file_id, path='.', overwrite=True`)

Download the file at the given file_id to the given path.

Parameters

- **file_id** (`str`) – The file identification to be downloaded
- **path** (`str`) – The path where the file should be saved
- **overwrite** (`bool`) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

download_folder (`folder_id, path='.', overwrite=True`)

Download the folder at the given file_id to the given path.

Parameters

- **folder_id** (`str`) – The folder identification to be downloaded
- **path** (`str`) – The path where the file should be saved
- **overwrite** (`bool`) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

rename_file (*file_id, name*)

Rename the given file.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

rename_folder (*folder_id, name*)

Rename the given folder.

Parameters

- **folder_id** (*str*) – The folder identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

update_tokens (*access_token*)

Refresh the access and refresh token given a valid access token

Parameters **access_token** (*string*) – box access token to be refreshed

Returns Box client

upload_file (*folder_id, path, name=None, overwrite=True*)

Upload a file to the given folder.

Parameters

- **folder_id** (*str*) – The folder identification to be downloaded
- **path** (*str*) – The path to the file or folder to be uploaded
- **name** (*str*) – The name the file or folder should be saved with. If None then the leaf of the path is used as the name.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

upload_folder (*folder_id, path, name=None, overwrite=True*)

Upload a folder to the given folder.

Parameters

- **folder_id** (*str*) – The folder identification to be downloaded
- **path** (*str*) – The path to the file or folder to be uploaded
- **name** (*str*) – The name the file or folder should be saved with. If None then the leaf of the path is used as the name.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

Google Drive

Authentication

Config

Usage

class `dapt.storage.google_drive.Google_Drive` (**kwargs)

Bases: `dapt.storage.base.Storage`

Download, upload, move, and delete files or folders from Google Drive.

Keyword Arguments

- **creds_path** (*str*) – the path to the file containing the Google API credentials. Default is `credentials.json`.
- **config** (*Config*) – a *Config* object with the associated config file to be used

connect ()

Allows you to sign into your Google account through the internet browser. This should automatically open the browser up.

Returns True if the connection was successful and False otherwise.

create_folder (*file_id*, *name*)

Create a folder named *name* in the folder with the *file_id* given.

Parameters

- **file_id** (*str*) – The file id of the parent folder to create the new folder in
- **name** (*str*) – What the name of the new folder should be

Returns The file metadata if successful and None otherwise.

delete_file (*file_id*)

Delete the the given file.

Parameters **file_id** (*str*) – The file identification to be downloaded

Returns True if successful and False otherwise

delete_folder (*file_id*)

Delete the given folder.

Parameters **file_id** (*str*) – The folder identification to be downloaded

Returns True if successful and False otherwise

download_file (*file_id*, *folder='.'*, *name=None*, *overwrite=True*)

Download the file at the given *file_id* to the given path. This will only download binary files such as Microsoft Docs, PDFs, PNGs, MP4, etc. This method is not capable of downloading Google products such as Google Docs and Google Sheets.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **folder** (*str*) – The directory where the file should be saved
- **name** (*str*) – The name that the file should be saved as. If None is given (default), then the name of the file on the resource will be used.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

download_folder (*file_id*, *folder='.'*, *name=None*, *overwrite=True*)

Download the folder at the given *file_id* to the given path.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **folder** (*str*) – The directory where the file should be saved
- **name** (*str*) – The name that the file should be saved as. If None is given (default), then the name of the file on the resource will be used.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if successful and False otherwise

rename_file (*file_id*, *name*)

Rename the given file.

Parameters

- **file_id** (*str*) – The file identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

rename_folder (*file_id*, *name*)

Rename the given folder.

Parameters

- **file_id** (*str*) – The folder identification to be downloaded
- **name** (*str*) – The new name of the file or folder

Returns True if the file or folder was renamed, False otherwise.

upload_file (*file_id*, *name*, *folder='.'*, *overwrite=True*)

Upload a file to the given folder.

Parameters

- **file_id** (*str*) – The folder where the file should be saved.
- **name** (*str*) – The name that the file should be uploaded.
- **folder** (*str*) – The directory where the file is stored.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

upload_folder (*file_id*, *name*, *folder='.'*, *overwrite=True*)

Upload a folder to the given folder.

Parameters

- **file_id** (*str*) – The folder where the folder should be saved.
- **name** (*str*) – The name that the file should be uploaded.
- **folder** (*str*) – The directory where the folder is stored.
- **overwrite** (*bool*) – Should the data on your machine be overwritten. True by default.

Returns True if the upload was successful and False otherwise.

1.3.5 Tools

A collection of tools that make DAPT easy to use, especially with `PhysiCell`. The `sample_db` and `create_settings_file()` methods are helpful with anyone using DAPT. The rest of the methods are used specifically for `PhysiCell` pipelines.

```
dapt.tools.create_XML(parameters, default_settings='PhysiCell_settings_default.xml',
                      save_settings='PhysiCell_settings.xml', off_limits=[])
```

Create a `PhysiCell` XML settings file given a dictionary of parameters. This function works by having a `default_settings` file which contains the generic XML structure. Each key in `parameters` then contains the paths to each XML tag in the `default_settings` file. The value of that tag is then set to the value in the associated key. If a key in `parameters` does not exist in the `default_settings` XML file then it is ignored. If a key in `parameters` also exists in `off_limits` then it is ignored.

Parameters

- **parameters** (*dict*) – A dictionary of parameters where the key is the path to the xml variable and the value is the desired value in the XML file.
- **default_settings** (*str*) – the path to the default xml file
- **save_settings** (*str*) – the path to the output xml file
- **off_limits** (*list*) – a list of keys that should not be inserted into the XML file.

```
dapt.tools.create_settings_file(parameters, pid=None)
```

Creates a file where each line contains a key from the parameters and its associated key, separated by a semi-colon.

Parameters

- **parameters** (*dict*) – the parameters to be saved in the file
- **pid** (*str*) – the parameter id of the current parameter run. If you don't give an id then the id in `parameters` will be used.

```
dapt.tools.create_zip(pid)
```

Zip all of the important `PhysiCell` items.

Parameters `pid` (*str*) – the id of the current parameter run

Returns The name of the zipped file

```
dapt.tools.data_cleanup(config=None)
```

Emulating `make data-cleanup-light`: remove `.mat`, `.xml`, `.svg`, `.txt`, `.pov`. You can optionally remove zipped files by setting `remove_zip` equal to `True` or remove `*.mp4` by setting `remove_movie` to `True` in the config file.

Parameters `config` (*Config*) – A config object, optionally given.

```
dapt.tools.sample_db(file_name='sample_db.csv', delimiter=',')
```

Create a sample *Delimited_file* database. The sample table is shown below. This method will create a file specified in the `file_name` attribute using the delimiter specified by `delimiter`.

id	start-time	end-time	status	a	b	c
t1	2019-09-06 17:23	2019-09-06 17:36	finished	2	4	6
t2				10	10	
t3				10	-10	

Parameters

- **file_name** (*str*) – the file name of the file to create and use for the database. The default value is *sample_db.csv*.
- **delimiter** (*str*) – the delimiter to use for the file. The default is a ,.

Returns A *Delimited_file* object using the file_name specified.

1.4 Supported Online Services

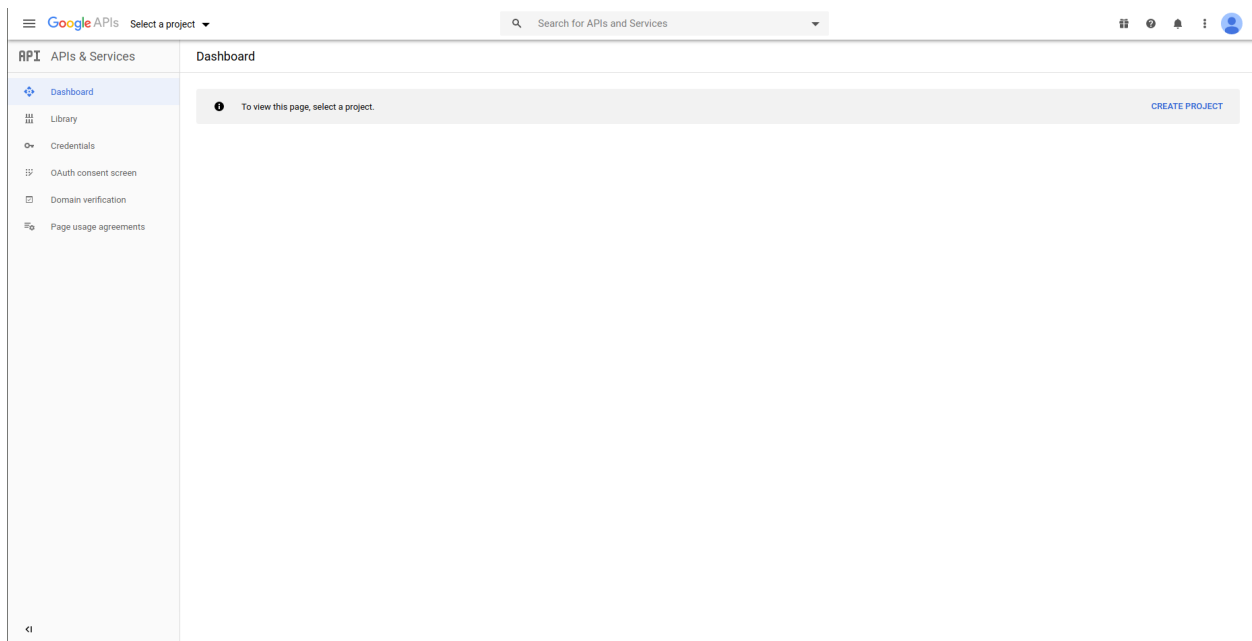
1.4.1 Google Sheets Installation

Google Sheets can be used as a database to store your parameter sets. The advantage to using this “database” over a file containing the parameters is that a team can work on the set more collaboratively and update the parameter list on the fly. The bigger advantage is that the parameter list can be run dynamically. Meaning that people, running the library simultaneously, can connect to Google Sheets and get the next parameter set in the list.

Get API Credentials

To use Google Sheets you will need to use the Google Sheets API and generate the proper credentials.

1. Start by going to the [Google Developer Console](#) and login using a Google account.
2. Create a new project by selecting the button labeled “CREATE PROJECT”. If you have already created a project or do not see the button, selecting the down arrow next in the top left corner of the page next to the “Google API” logo and clicking “New Project”.



3. Give the project a name and press “CREATE”.

Google APIs Search for APIs and Services

New Project

You have 12 projects remaining in your quota. Request an increase or delete projects. [Learn more](#)

[MANAGE QUOTAS](#)

Project name *
dapt-database

Project ID: dapt-database. It cannot be changed later. [EDIT](#)

Location *
No organization [BROWSE](#)


Parent organization or folder

[CREATE](#) [CANCEL](#)

- Click “ENABLE APIS AND SERVICES”, search for “Google Sheets API” and click it. Then click “Enable”.

Google APIs dapt-database Search for APIs and Services

API Library

 **Google Sheets API**
Google

The Sheets API gives you full control over the content and appearance of your spreadsheet data.

[ENABLE](#) [TRY THIS API](#)

Type
[APIs & services](#)

Last updated
12/9/19, 7:36 PM

Category
[G Suite](#)
[CRM](#)

Service name
sheets.googleapis.com

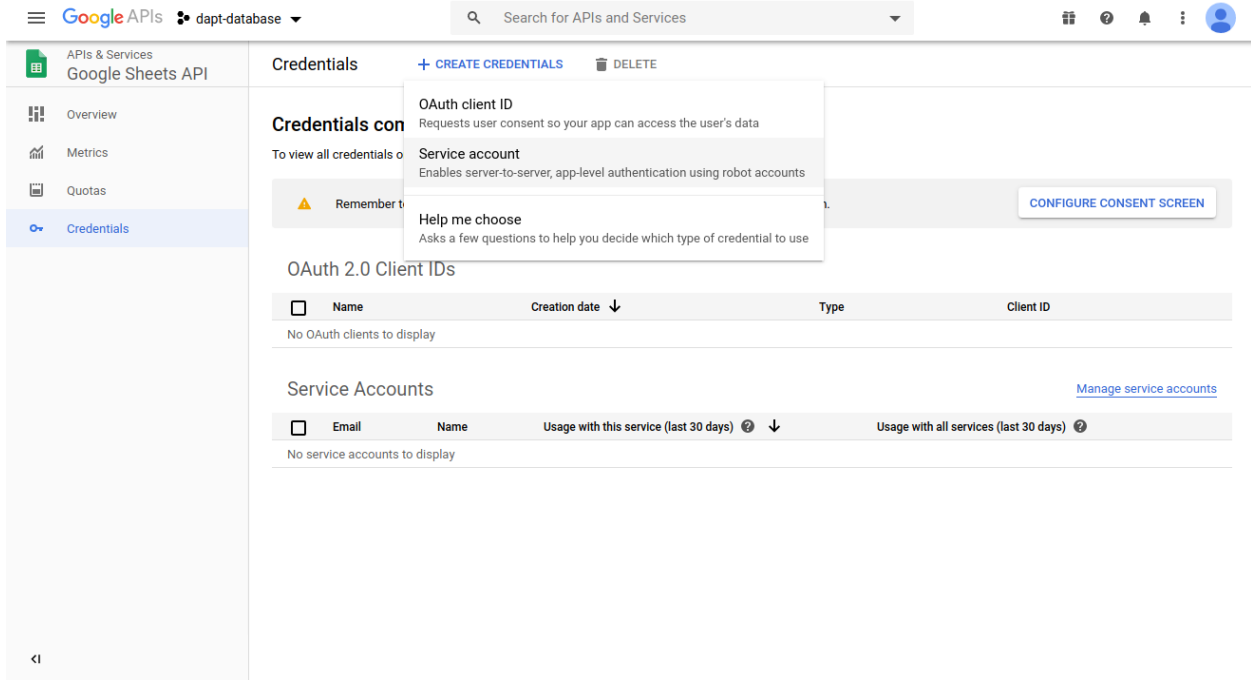
Overview
Reads and writes Google Sheets.

About Google
Google's mission is to organize the world's information and make it universally accessible and useful. Through products and platforms like Search, Maps, Gmail, Android, Google Play, Chrome and YouTube, Google plays a meaningful role in the daily lives of billions of people.

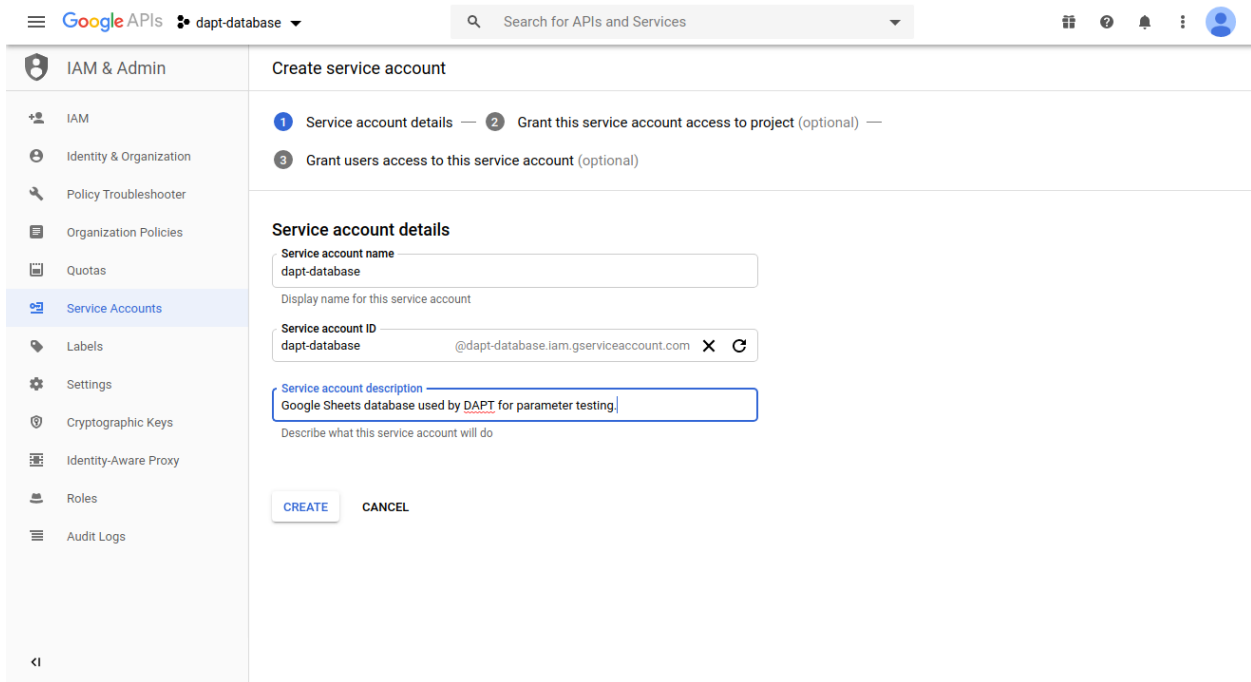
Tutorials and documentation
[Learn more](#)

Maintenance & support

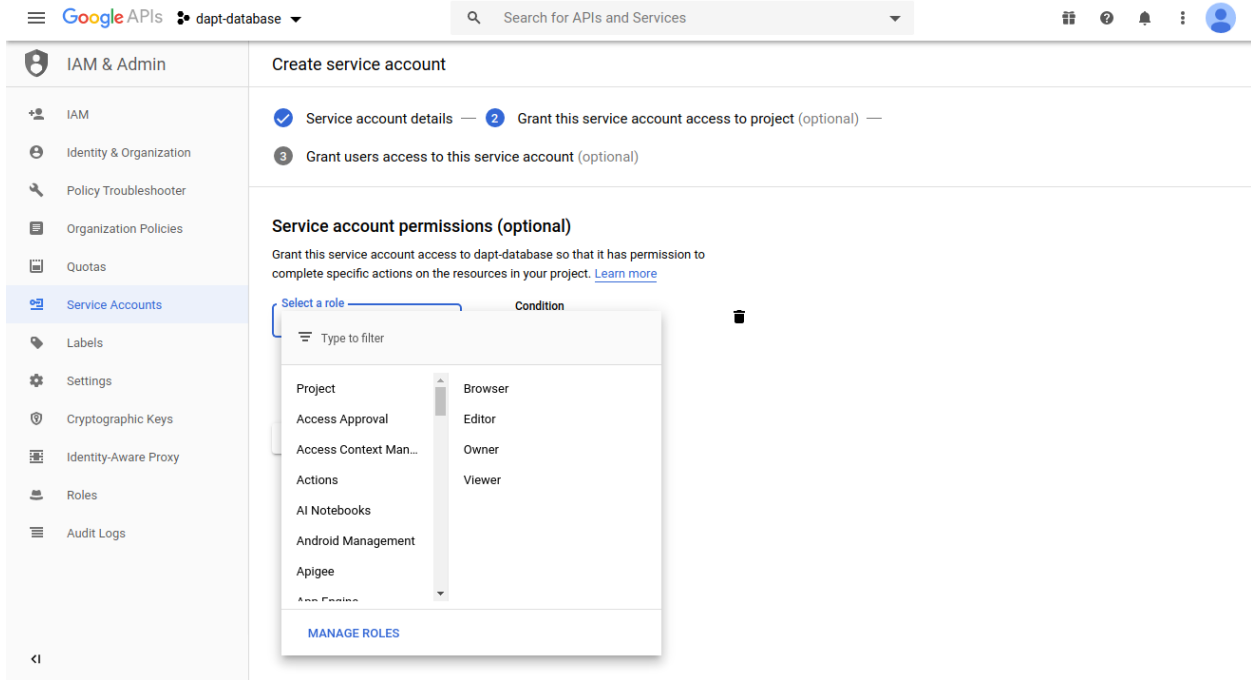
- Click the “Credentials” tab from the menu on the left side of the page. Click the dropdown at the top of the page that says “CREATE CREDENTIALS” and select “Service account”.



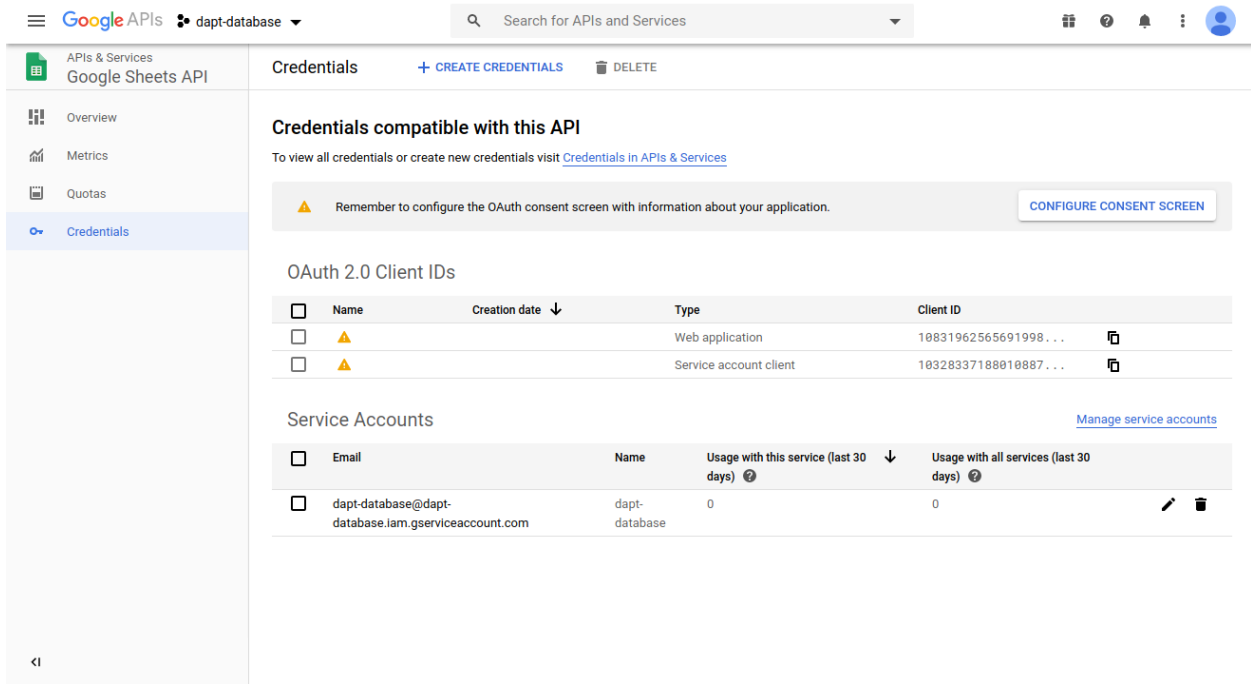
6. Give the service a name and click “Create”.



7. In the next section asking about service account permissions, create a role with by selecting “Project” then “Editor”. Then select “Continue”.

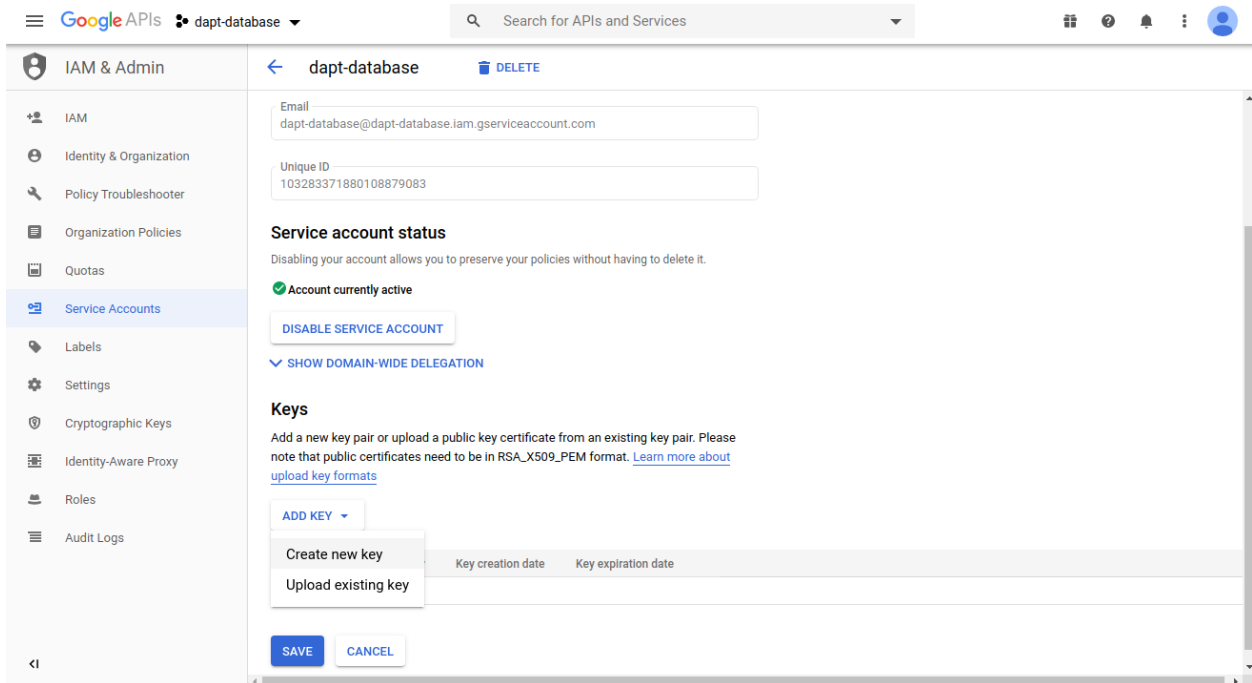


8. On step three of creating the credentials, click “DONE”.
9. You should now be on the credentials page of the Google Sheets API. Under Service Account you should see an entry with the name of the account you just created. Record the email address given there. You will need to share the Google Sheet acting as a database with it. The email address should end in `.iam.gserviceaccount.com`.



10. To get the credentials needed by DAPT click on the pen on the right side of the Service Account.
11. Under the “Keys” section, select “ADD KEY”, then “Create new key”. Ensure the key type of “JSON” is selected and click “CREATE”. A JSON file should then start downloading to your computer. You will give

DAPT the path to this JSON file when using Google Sheets. Then click “DONE”.



1.4.2 Box Installation

Box is a cloud storage service that many universities allow students, faculty and staff to use. The advantage of box is that it allows a large amount of data to be uploaded to a common place where team members can observe data. In order to allow DAPT to upload to box, you must create some API credentials.

API Credentials

1. Start by going to the [Box Development](#) website and clicking on the blue “Console” button. Then log in.
2. Click “Create New App”. Then click “Custom App” and “Next” on the next page.
3. On the “Authentication Method” page click “Standard OAuth 2.0 (User Authentication)” and name your project. Then click “View Your App”.
4. Scroll down to the “OAuth 2.0 Credentials” section and record the Client ID and Secret. You will pass these to the DAPT Box class to allow the Box SDK to work.
5. Lastly, scroll down to the “OAuth 2.0 Credentials” section and change the url to `http://127.0.0.1:5000/return`. Then click “Save Changes”.

1.5 Examples

Examples of DAPT are kept in the [examples](#) folder. There are basic examples of the main features of DAPT including using a delimited file, Google Sheets and Box. There is also an example of how DAPT can be used with [PhysiCell](#). It is recommended that you start with the [csv_example.py](#) script as it is the simplest to use and only requires DAPT to be installed. The other scripts require API keys to be generated.

1.6 Development Guide

Extra cool badges:

1.6.1 Contribute

If you would like to contribute please fork the repo and make a pull request explaining what you added/fixes and why you added it. When you write a new feature please write tests in the `test` directory and documentation in the `docs` folder.

Documentation

Documentation is performed using [Sphinx](#). The `docs` folder holds all of the resources to document the code. If you're not familiar with Sphinx you can read this [Medium tutorial](#) for an introduction. Google docstrings are used for inline commenting inside each file.

To install the required packages, run the following commands:

```
pip install sphinx sphinx_rtd_theme
```

You can compile the docs by running `make build-html`, assuming you have sphinx installed. This will remove the old documentation and create the new html documentation in `docs/_build/html`.

Tests

Tests are located in the `tests` folder and written using [pytest](#). You can run the tests locally by running `python3 -m pytest` in the root DAPT directory. This assumes that you have a configuration file named `test_config.json` in the root directory. The convention used is to name all files and functions in the test directory `test_x`, where `x` is the name/description of the test.

To run tests on the tests for TravisCI, the API keys need to be stored in environment variables so they can be kept private. These values must be escaped in the same way Bash shells must have escaped values. A simple way to do this is python is by using the `json.dump(SECRET_KEY)` method which will automatically escape the values for you.

1.6.2 Updates

Guide for pushing updates

0. Install requirements by running `pip install twine`
1. Test the code locally by running `pip install .` in the root directory.
2. Update the version in `setup.py` file.
3. Run `python3 setup.py sdist bdist_wheel`.
4. Run `twine upload dist/*`.

New way (test):

1. `python3 -m pip install --upgrade build`

2. `python3 -m build`
3. `python3 -m pip install --upgrade twine`
4. `python3 -m twine upload --repository testpypi dist/* uploads to test`
5. `python3 -m pip install --index-url https://test.pypi.org/simple/ --no-deps example-pkg-YOUR-USERNAME-HERE to install from test`

New way (production):

1. `python3 -m pip install --upgrade build`
2. `python3 -m build`
3. `python3 -m pip install --upgrade twine`
4. `twine upload dist/* uploads to production`
5. `python3 -m pip install dapt to install from production`

d

- `dapt.config`, 4
- `dapt.db`, 8
 - `dapt.db.base`, 10
 - `dapt.db.delimited_file`, 11
 - `dapt.db.sheets`, 13
- `dapt.param`, 15
- `dapt.storage`, 18
 - `dapt.storage.base`, 18
 - `dapt.storage.box`, 21
 - `dapt.storage.google_drive`, 23
- `dapt.tools`, 25

B

Box (class in *dapt.storage.box*), 21

C

check_overwrite_file() (in module *dapt.storage.base*), 20

check_overwrite_folder() (in module *dapt.storage.base*), 20

Config (class in *dapt.config*), 7

connect() (*dapt.db.base.Database* method), 10

connect() (*dapt.db.delimited_file.Delimited_file* method), 12

connect() (*dapt.db.sheets.Sheet* method), 14

connect() (*dapt.storage.base.Storage* method), 19

connect() (*dapt.storage.box.Box* method), 22

connect() (*dapt.storage.google_drive.Google_Drive* method), 24

connected() (*dapt.db.base.Database* method), 10

connected() (*dapt.db.delimited_file.Delimited_file* method), 12

connected() (*dapt.db.sheets.Sheet* method), 14

connected() (*dapt.storage.base.Storage* method), 19

create() (*dapt.config.Config* static method), 7

create_folder() (*dapt.storage.google_drive.Google_Drive* method), 24

create_settings_file() (in module *dapt.tools*), 26

create_XML() (in module *dapt.tools*), 26

create_zip() (in module *dapt.tools*), 26

D

dapt.config (module), 4

dapt.db (module), 8

dapt.db.base (module), 10

dapt.db.delimited_file (module), 11

dapt.db.sheets (module), 13

dapt.param (module), 15

dapt.storage (module), 18

dapt.storage.base (module), 18

dapt.storage.box (module), 21

dapt.storage.google_drive (module), 23

dapt.tools (module), 25

data_cleanup() (in module *dapt.tools*), 26

Database (class in *dapt.db.base*), 10

delete_file() (*dapt.storage.base.Storage* method), 19

delete_file() (*dapt.storage.box.Box* method), 22

delete_file() (*dapt.storage.google_drive.Google_Drive* method), 24

delete_folder() (*dapt.storage.base.Storage* method), 19

delete_folder() (*dapt.storage.box.Box* method), 22

delete_folder() (*dapt.storage.google_drive.Google_Drive* method), 24

Delimited_file (class in *dapt.db.delimited_file*), 12

download_file() (*dapt.storage.base.Storage* method), 19

download_file() (*dapt.storage.box.Box* method), 22

download_file() (*dapt.storage.google_drive.Google_Drive* method), 24

download_folder() (*dapt.storage.base.Storage* method), 19

download_folder() (*dapt.storage.box.Box* method), 22

download_folder() (*dapt.storage.google_drive.Google_Drive* method), 24

F

failed() (*dapt.param.Param* method), 17

fields() (*dapt.db.base.Database* method), 10

fields() (*dapt.db.delimited_file.Delimited_file* method), 12

fields() (*dapt.db.sheets.Sheet* method), 14

G

get_key_index() (*dapt.db.sheets.Sheet* method), 14

get_keys() (*dapt.db.base.Database* method), 10

[get_mime_type\(\)](#) (*in module `dapt.storage.base`*), 21
[get_row_index\(\)](#) (*`dapt.db.delimited_file.Delimited_file` method*), 12
[get_row_index\(\)](#) (*`dapt.db.sheets.Sheet` method*), 15
[get_table\(\)](#) (*`dapt.db.base.Database` method*), 11
[get_table\(\)](#) (*`dapt.db.delimited_file.Delimited_file` method*), 13
[get_table\(\)](#) (*`dapt.db.sheets.Sheet` method*), 15
[get_value\(\)](#) (*`dapt.config.Config` method*), 7
[Google_Drive](#) (*class in `dapt.storage.google_drive`*), 24

H

[has_value\(\)](#) (*`dapt.config.Config` method*), 7

K

[keys\(\)](#) (*`dapt.config.Config` method*), 8

N

[next_parameters\(\)](#) (*`dapt.param.Param` method*), 18

P

[Param](#) (*class in `dapt.param`*), 17

R

[read\(\)](#) (*`dapt.config.Config` method*), 8
[rename_file\(\)](#) (*`dapt.storage.base.Storage` method*), 19
[rename_file\(\)](#) (*`dapt.storage.box.Box` method*), 22
[rename_file\(\)](#) (*`dapt.storage.google_drive.Google_Drive` method*), 25
[rename_folder\(\)](#) (*`dapt.storage.base.Storage` method*), 20
[rename_folder\(\)](#) (*`dapt.storage.box.Box` method*), 23
[rename_folder\(\)](#) (*`dapt.storage.google_drive.Google_Drive` method*), 25

S

[safe\(\)](#) (*`dapt.config.Config` static method*), 8
[sample_db\(\)](#) (*in module `dapt.tools`*), 26
[Sheet](#) (*class in `dapt.db.sheets`*), 14
[Storage](#) (*class in `dapt.storage.base`*), 19
[successful\(\)](#) (*`dapt.param.Param` method*), 18

U

[update\(\)](#) (*`dapt.config.Config` method*), 8
[update_cell\(\)](#) (*`dapt.db.base.Database` method*), 11
[update_cell\(\)](#) (*`dapt.db.delimited_file.Delimited_file` method*), 13
[update_cell\(\)](#) (*`dapt.db.sheets.Sheet` method*), 15
[update_row\(\)](#) (*`dapt.db.base.Database` method*), 11
[update_row\(\)](#) (*`dapt.db.delimited_file.Delimited_file` method*), 13
[update_row\(\)](#) (*`dapt.db.sheets.Sheet` method*), 15
[update_status\(\)](#) (*`dapt.param.Param` method*), 18
[update_tokens\(\)](#) (*`dapt.storage.box.Box` method*), 23
[upload_file\(\)](#) (*`dapt.storage.base.Storage` method*), 20
[upload_file\(\)](#) (*`dapt.storage.box.Box` method*), 23
[upload_file\(\)](#) (*`dapt.storage.google_drive.Google_Drive` method*), 25
[upload_folder\(\)](#) (*`dapt.storage.base.Storage` method*), 20
[upload_folder\(\)](#) (*`dapt.storage.box.Box` method*), 23
[upload_folder\(\)](#) (*`dapt.storage.google_drive.Google_Drive` method*), 25

W

[worksheet\(\)](#) (*`dapt.db.sheets.Sheet` method*), 15