LIBRBD (PYTHON)

The *rbd* python module provides file-like access to RBD images.

EXAMPLE: CREATING AND WRITING TO AN IMAGE

To use *rbd*, you must first connect to RADOS and open an IO context:

```
cluster = rados.Rados(conffile='my_ceph.conf')
cluster.connect()
ioctx = cluster.open_ioctx('mypool')
```

Then you instantiate an :class:rbd.RBD object, which you use to create the image:

```
rbd_inst = rbd.RBD()
size = 4 * 1024**3  # 4 GiB
rbd_inst.create(ioctx, 'myimage', size)
```

To perform I/O on the image, you instantiate an :class:rbd.Image object:

```
image = rbd.Image(ioctx, 'myimage')
data = 'foo' * 200
image.write(data, 0)
```

This writes ‘foo’ to the first 600 bytes of the image. Note that data cannot be :type:unicode - *Librbd* does not know how to deal with characters wider than a :c:type:char.

In the end, you will want to close the image, the IO context and the connection to RADOS:

```
image.close()
ioctx.close()
cluster.shutdown()
```

To be safe, each of these calls would need to be in a separate :finally block:

```
cluster = rados.Rados(conffile='my_ceph_conf')
try:
    cluster.connect()
    ioctx = cluster.open_ioctx('my_pool')
    try:
        rbd_inst = rbd.RBD()
        size = 4 * 1024**3  # 4 GiB
        rbd_inst.create(ioctx, 'myimage', size)
        image = rbd.Image(ioctx, 'myimage')
        try:
            data = 'foo' * 200
            image.write(data, 0)
        finally:
            image.close()
    finally:
        ioctx.close()
finally:
    cluster.shutdown()
```

This can be cumbersome, so the **Rados**, **Ioctx**, and **Image** classes can be used as context managers that close/shutdown automatically (see **PEP 343**). Using them as context managers, the above example becomes:

```
with rados.Rados(conffile='my_ceph.conf') as cluster:
    with cluster.open_ioctx('mypool') as ioctx:
        rbd_inst = rbd.RBD()
```

```
        size = 4 * 1024**3  # 4 GiB
        rbd_inst.create(ioctx, 'myimage', size)
        with rbd.Image(ioctx, 'myimage') as image:
            data = 'foo' * 200
            image.write(data, 0)
```

API REFERENCE

This module is a thin wrapper around librbd.

It currently provides all the synchronous methods of librbd that do not use callbacks.

Error codes from librbd are turned into exceptions that subclass **Error**. Almost all methods may raise **Error** (the base class of all rbd exceptions), **PermissionError** and **IOError**, in addition to those documented for the method.

class rbd.**RBD**

This class wraps librbd CRUD functions.

clone(*self*, *p_ioctx*, *p_name*, *p_snapname*, *c_ioctx*, *c_name*, *features=None*, *order=None*, *stripe_unit=None*, *stripe_count=None*, *data_pool=None*)

Clone a parent rbd snapshot into a COW sparse child.

Parameters:
- **p_ioctx** – the parent context that represents the parent snap
- **p_name** – the parent image name
- **p_snapname** – the parent image snapshot name
- **c_ioctx** – the child context that represents the new clone
- **c_name** – the clone (child) name
- **features** (*int*) – bitmask of features to enable; if set, must include layering
- **order** (*int*) – the image is split into (2**order) byte objects
- **stripe_unit** (*int*) – stripe unit in bytes (default None to let librbd decide)
- **stripe_count** (*int*) – objects to stripe over before looping
- **data_pool** (*str*) – optional separate pool for data blocks

Raises: **TypeError**

Raises: **InvalidArgument**

Raises: **ImageExists**

Raises: **FunctionNotSupported**

Raises: **ArgumentOutOfRange**

create(*self*, *ioctx*, *name*, *size*, *order=None*, *old_format=True*, *features=None*, *stripe_unit=None*, *stripe_count=None*, *data_pool=None*)

Create an rbd image.

Parameters:
- **ioctx** (**rados.Ioctx**) – the context in which to create the image
- **name** (*str*) – what the image is called
- **size** (*int*) – how big the image is in bytes
- **order** (*int*) – the image is split into (2**order) byte objects
- **old_format** (*bool*) – whether to create an old-style image that is accessible by old clients, but can't use more advanced features like layering.
- **features** (*int*) – bitmask of features to enable
- **stripe_unit** (*int*) – stripe unit in bytes (default None to let librbd decide)
- **stripe_count** (*int*) – objects to stripe over before looping
- **data_pool** (*str*) – optional separate pool for data blocks

Raises: **ImageExists**

Raises: **TypeError**

Raises: **InvalidArgument**

Raises: **FunctionNotSupported**

group_create(*self*, *ioctx*, *name*)

Create a group.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is used
- **name** (*str*) – the name of the group

Raises: **ObjectExists**

Raises: **InvalidArgument**
Raises: **FunctionNotSupported**

group_list(*self*, *ioctx*)

List groups.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read
Returns: list – a list of groups names
Raises: **FunctionNotSupported**

group_remove(*self*, *ioctx*, *name*)

Delete an RBD group. This may take a long time, since it does not return until every image in the group has been removed from the group.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool the group is in
- **name** (*str*) – the name of the group to remove

Raises: **ObjectNotFound**
Raises: **InvalidArgument**
Raises: **FunctionNotSupported**

group_rename(*self*, *ioctx*, *src*, *dest*)

Rename an RBD group.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool the group is in
- **src** (*str*) – the current name of the group
- **dest** (*str*) – the new name of the group

Raises: **ObjectExists**
Raises: **ObjectNotFound**
Raises: **InvalidArgument**
Raises: **FunctionNotSupported**

list(*self*, *ioctx*)

List image names.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read
Returns: list – a list of image names

mirror_image_status_list(*self*, *ioctx*)

Iterate over the mirror image statuses of a pool.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read
Returns: **MirrorImageStatus**

mirror_image_status_summary(*self*, *ioctx*)

Get mirror image status summary of a pool.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read
Returns: list - a list of (state, count) tuples

mirror_mode_get(*self*, *ioctx*)

Get pool mirror mode.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read
Returns: int - pool mirror mode

mirror_mode_set(*self*, *ioctx*, *mirror_mode*)

Set pool mirror mode.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is written
- **mirror_mode** (*int*) – mirror mode to set

mirror_peer_add(*self*, *ioctx*, *cluster_name*, *client_name*)

Add mirror peer.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is used
- **cluster_name** (*str*) – mirror peer cluster name
- **client_name** (*str*) – mirror peer client name

Returns: str - peer uuid

mirror_peer_list(*self*, *ioctx*)

Iterate over the peers of a pool.

Parameters: **ioctx** (**rados.Ioctx**) – determines which RADOS pool is read

Returns: **MirrorPeerIterator**

mirror_peer_remove(*self*, *ioctx*, *uuid*)

Remove mirror peer.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is used
- **uuid** (*str*) – peer uuid

mirror_peer_set_client(*self*, *ioctx*, *uuid*, *client_name*)

Set mirror peer client name

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is written
- **uuid** (*str*) – uuid of the mirror peer
- **client_name** (*str*) – client name of the mirror peer to set

mirror_peer_set_cluster(*self*, *ioctx*, *uuid*, *cluster_name*)

Set mirror peer cluster name

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool is written
- **uuid** (*str*) – uuid of the mirror peer
- **cluster_name** (*str*) – cluster name of the mirror peer to set

remove(*self*, *ioctx*, *name*)

Delete an RBD image. This may take a long time, since it does not return until every object that comprises the image has been deleted. Note that all snapshots must be deleted before the image can be removed. If there are snapshots left, **ImageHasSnapshots** is raised. If the image is still open, or the watch from a crashed client has not expired, **ImageBusy** is raised.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool the image is in
- **name** (*str*) – the name of the image to remove

Raises: **ImageNotFound**, **ImageBusy**, **ImageHasSnapshots**

rename(*self*, *ioctx*, *src*, *dest*)

Rename an RBD image.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool the image is in
- **src** (*str*) – the current name of the image
- **dest** (*str*) – the new name of the image

Raises: **ImageNotFound**, **ImageExists**

trash_get(*self*, *ioctx*, *image_id*)

Retrieve RBD image info from trash :param ioctx: determines which RADOS pool the image is in :type ioctx: **rados.Ioctx** :param image_id: the id of the image to restore :type image_id: str :returns: dict - contains the following keys:

- `id` (str) - image id
- `name` (str) - image name
- `source` (str) - source of deletion
- `deletion_time` (datetime) - time of deletion
- `deferment_end_time` (datetime) - time that an image is allowed to be removed from trash

Raises: **ImageNotFound**

trash_list(*self*, *ioctx*)

List all entries from trash. :param ioctx: determines which RADOS pool the image is in :type ioctx: **rados.Ioctx** :returns: **TrashIterator**

trash_move(*self*, *ioctx*, *name*, *delay=0*)

Move an RBD image to the trash.

Parameters:
- **ioctx** (**rados.Ioctx**) – determines which RADOS pool the image is in
- **name** (*str*) – the name of the image to remove
- **delay** (*int*) – time delay in seconds before the image can be deleted from trash

Raises: **ImageNotFound**

trash_remove(*self*, *ioctx*, *image_id*, *force=False*)

Delete an RBD image from trash. If image deferment time has not expired **PermissionError** is raised. :param ioctx: determines which RADOS pool the image is in :type ioctx: **rados.Ioctx** :param image_id: the id of the image to remove :type image_id: str :param force: force remove even if deferment time has not expired :type force: bool :raises: **ImageNotFound**, **PermissionError**

trash_restore(*self*, *ioctx*, *image_id*, *name*)

Restore an RBD image from trash. :param ioctx: determines which RADOS pool the image is in :type ioctx: **rados.Ioctx** :param image_id: the id of the image to restore :type image_id: str :param name: the new name of the restored image :type name: str :raises: **ImageNotFound**

version(*self*)

Get the version number of the `librbd` C library.

Returns: a tuple of `(major, minor, extra)` components of the librbd version

class rbd.**Image**(*ioctx*, *name=None*, *snapshot=None*, *read_only=False*, *image_id=None*)

This class represents an RBD image. It is used to perform I/O on the image and interact with snapshots.

Note: Any method of this class may raise **ImageNotFound** if the image has been deleted.

aio_discard(*self*, *offset*, *length*, *oncomplete*)

Asynchronously trim the range from the image. It will be logically filled with zeroes.

aio_flush(*self*, *oncomplete*)

Asyncronously wait until all writes are fully flushed if caching is enabled.

aio_read(*self*, *offset*, *length*, *oncomplete*, *fadvise_flags=0*)

Asynchronously read data from the image

Raises **InvalidArgument** if part of the range specified is outside the image.

oncomplete will be called with the returned read value as well as the completion:

oncomplete(completion, data_read)

Parameters:
- **offset** (*int*) – the offset to start reading at
- **length** (*int*) – how many bytes to read
- **oncomplete** (*completion*) – what to do when the read is complete
- **fadvise_flags** (*int*) – fadvise flags for this read

Returns: **Completion** - the completion object

Raises: **InvalidArgument**, **IOError**

aio_write(*self*, *data*, *offset*, *oncomplete*, *fadvise_flags=0*)

Asynchronously write data to the image

Raises **InvalidArgument** if part of the write would fall outside the image.

oncomplete will be called with the completion:

oncomplete(completion)

Parameters:
- **data** (*bytes*) – the data to be written
- **offset** (*int*) – the offset to start writing at
- **oncomplete** (*completion*) – what to do when the write is complete
- **fadvise_flags** (*int*) – fadvise flags for this write

Returns: `Completion` - the completion object

Raises: `InvalidArgument`, `IOError`

`block_name_prefix`(*self*)

Get the RBD block name prefix

Returns: str - block name prefix

`break_lock`(*self*, *client*, *cookie*)

Release a lock held by another rados client.

`close`(*self*)

Release the resources used by this image object.

After this is called, this object should not be used.

`copy`(*self*, *dest_ioctx*, *dest_name*, *features=None*, *order=None*, *stripe_unit=None*, *stripe_count=None*, *data_pool=None*)

Copy the image to another location.

Parameters:
- **dest_ioctx** (`rados.Ioctx`) – determines which pool to copy into
- **dest_name** (*str*) – the name of the copy
- **features** (*int*) – bitmask of features to enable; if set, must include layering
- **order** (*int*) – the image is split into (2**order) byte objects
- **stripe_unit** (*int*) – stripe unit in bytes (default None to let librbd decide)
- **stripe_count** (*int*) – objects to stripe over before looping
- **data_pool** (*str*) – optional separate pool for data blocks

Raises: `TypeError`

Raises: `InvalidArgument`

Raises: `ImageExists`

Raises: `FunctionNotSupported`

Raises: `ArgumentOutOfRange`

`create_snap`(*self*, *name*)

Create a snapshot of the image.

Parameters: **name** (*str*) – the name of the snapshot

Raises: `ImageExists`

`create_timestamp`(*self*)

Return the create timestamp for the image.

`deep_copy`(*self*, *dest_ioctx*, *dest_name*, *features=None*, *order=None*, *stripe_unit=None*, *stripe_count=None*, *data_pool=None*)

Deep copy the image to another location.

Parameters:
- **dest_ioctx** (`rados.Ioctx`) – determines which pool to copy into
- **dest_name** (*str*) – the name of the copy
- **features** (*int*) – bitmask of features to enable; if set, must include layering
- **order** (*int*) – the image is split into (2**order) byte objects
- **stripe_unit** (*int*) – stripe unit in bytes (default None to let librbd decide)
- **stripe_count** (*int*) – objects to stripe over before looping
- **data_pool** (*str*) – optional separate pool for data blocks

Raises: `TypeError`

Raises: `InvalidArgument`

Raises: `ImageExists`

Raises: `FunctionNotSupported`

Raises: `ArgumentOutOfRange`

diff_iterate(*self*, *offset*, *length*, *from_snapshot*, *iterate_cb*, *include_parent=True*, *whole_object=False*)

Iterate over the changed extents of an image.

This will call iterate_cb with three arguments:

(offset, length, exists)

where the changed extent starts at offset bytes, continues for length bytes, and is full of data (if exists is True) or zeroes (if exists is False).

If from_snapshot is None, it is interpreted as the beginning of time and this generates all allocated extents.

The end version is whatever is currently selected (via set_snap) for the image.

iterate_cb may raise an exception, which will abort the diff and will be propagated to the caller.

Raises **InvalidArgument** if from_snapshot is after the currently set snapshot.

Raises **ImageNotFound** if from_snapshot is not the name of a snapshot of the image.

Parameters:
- **offset** (*int*) – start offset in bytes
- **length** (*int*) – size of region to report on, in bytes
- **from_snapshot** (*str or None*) – starting snapshot name, or None
- **iterate_cb** (*function acception arguments for offset, length, and exists*) – function to call for each extent
- **include_parent** (*bool*) – True if full history diff should include parent
- **whole_object** (*bool*) – True if diff extents should cover whole object

Raises: **InvalidArgument**, **IOError**, **ImageNotFound**

discard(*self*, *offset*, *length*)

Trim the range from the image. It will be logically filled with zeroes.

features(*self*)

Get the features bitmask of the image.

Returns: int - the features bitmask of the image

flags(*self*)

Get the flags bitmask of the image.

Returns: int - the flags bitmask of the image

flatten(*self*)

Flatten clone image (copy all blocks from parent to child)

flush(*self*)

Block until all writes are fully flushed if caching is enabled.

get_snap_limit(*self*)

Get the snapshot limit for an image.

get_snap_timestamp(*self*, *snap_id*)

Get the snapshot timestamp for an image. :param snap_id: the snapshot id of a snap shot

group(*self*)

Get information about the image's group.

Returns: dict - contains the following keys:
- `pool` (int) - id of the group pool
- `name` (str) - name of the group

id(*self*)

Get the RBD v2 internal image id

Returns: str - image id

invalidate_cache(*self*)

Drop any cached data for the image.

is_exclusive_lock_owner(*self*)

Get the status of the image exclusive lock.

Returns: bool - true if the image is exclusively locked

is_protected_snap(*self*, *name*)

Find out whether a snapshot is protected from deletion.

Parameters: **name** (*str*) – the snapshot to check
Returns: bool - whether the snapshot is protected
Raises: **IOError**, **ImageNotFound**

list_children(*self*)

List children of the currently set snapshot (set via set_snap()).

Returns: list - a list of (pool name, image name) tuples

list_children2(*self*)

Iterate over the children of a snapshot.

Returns: **ChildIterator**

list_lockers(*self*)

List clients that have locked the image and information about the lock.

Returns: dict - contains the following keys:

- `tag` - the tag associated with the lock (every additional locker must use the same tag)
- `exclusive` - boolean indicating whether the lock is exclusive or shared
- `lockers` - a list of (client, cookie, address) tuples

list_snaps(*self*)

Iterate over the snapshots of an image.

Returns: **SnapIterator**

lock_acquire(*self*, *lock_mode*)

Acquire a managed lock on the image.

Parameters: **lock_mode** (*int*) – lock mode to set
Raises: **ImageBusy** if the lock could not be acquired

lock_break(*self*, *lock_mode*, *lock_owner*)

Break the image lock held by a another client.

Parameters: **lock_owner** (*str*) – the owner of the lock to break

lock_exclusive(*self*, *cookie*)

Take an exclusive lock on the image.

Raises: **ImageBusy** if a different client or cookie locked it **ImageExists** if the same client and cookie locked it

lock_get_owners(*self*)

Iterate over the lock owners of an image.

Returns: **LockOwnerIterator**

lock_release(*self*)

Release a managed lock on the image that was previously acquired.

lock_shared(*self*, *cookie*, *tag*)

Take a shared lock on the image. The tag must match that of the existing lockers, if any.

Raises: **ImageBusy** if a different client or cookie locked it **ImageExists** if the same client and cookie locked it

metadata_get(*self*, *key*)

Get image metadata for the given key.

Parameters: **key** (*str*) – metadata key
Returns: str - image id

metadata_list(*self*)

List image metadata.

Returns: **MetadataIterator**

metadata_remove(*self*, *key*)

Remove image metadata for the given key.

Parameters: **key** (*str*) – metadata key

metadata_set(*self*, *key*, *value*)

Set image metadata for the given key.

Parameters:
- **key** (*str*) – metadata key
- **value** (*str*) – metadata value

mirror_image_demote(*self*)

Demote the image to secondary for mirroring.

mirror_image_disable(*self*, *force*)

Disable mirroring for the image.

Parameters: **force** (*bool*) – force disabling

mirror_image_enable(*self*)

Enable mirroring for the image.

mirror_image_get_info(*self*)

Get mirror info for the image.

Returns: dict - contains the following keys:
- `global_id` (str) - image global id
- `state` (int) - mirror state
- `primary` (bool) - is image primary

mirror_image_get_status(*self*)

Get mirror status for the image.

Returns: dict - contains the following keys:
- `name` (str) - mirror image name
- *info* (dict) - mirror image info
- `state` (int) - status mirror state
- `description` (str) - status description
- `last_update` (datetime) - last status update time
- up (bool) - is mirroring agent up

mirror_image_promote(*self*, *force*)

Promote the image to primary for mirroring.

Parameters: **force** (*bool*) – force promoting

mirror_image_resync(*self*)

Flag the image to resync.

old_format(*self*)

Find out whether the image uses the old RBD format.

Returns: bool - whether the image uses the old RBD format

op_features(*self*)

Get the op features bitmask of the image.

Returns: int - the op features bitmask of the image

overlap(*self*)

Get the number of overlapping bytes between the image and its parent image. If open to a snapshot, returns the overlap between the snapshot and the parent image.

Returns: int - the overlap in bytes
Raises: **ImageNotFound** if the image doesn't have a parent

parent_id(*self*)

Get image id of a cloned image's parent (if any)

Returns: str - the parent id
Raises: **ImageNotFound** if the image doesn't have a parent

parent_info(*self*)

Get information about a cloned image's parent (if any)

Returns: tuple - `(pool name, image name, snapshot name)` components of the parent image
Raises: **ImageNotFound** if the image doesn't have a parent

protect_snap(*self*, *name*)

Mark a snapshot as protected. This means it can't be deleted until it is unprotected.

Parameters: **name** (*str*) – the snapshot to protect
Raises: **IOError**, **ImageNotFound**

read(*self*, *offset*, *length*, *fadvise_flags=0*)

Read data from the image. Raises **InvalidArgument** if part of the range specified is outside the image.

Parameters:
- **offset** (*int*) – the offset to start reading at
- **length** (*int*) – how many bytes to read
- **fadvise_flags** (*int*) – fadvise flags for this read

Returns: str - the data read
Raises: **InvalidArgument**, **IOError**

rebuild_object_map(*self*)

Rebuild the object map for the image HEAD or currently set snapshot

remove_snap(*self*, *name*)

Delete a snapshot of the image.

Parameters: **name** (*str*) – the name of the snapshot
Raises: **IOError**, **ImageBusy**

remove_snap2(*self*, *name*, *flags*)

Delete a snapshot of the image.

Parameters:
- **name** (*str*) – the name of the snapshot
- **flags** – the flags for removal

Raises: **IOError**, **ImageBusy**

remove_snap_limit(*self*)

Remove the snapshot limit for an image, essentially setting the limit to the maximum size allowed by the implementation.

rename_snap(*self*, *srcname*, *dstname*)

rename a snapshot of the image.

Parameters:
- **srcname** (*str*) – the src name of the snapshot
- **dstname** (*str*) – the dst name of the snapshot

Raises: **ImageExists**

resize(*self*, *size*)

Change the size of the image.

Parameters: **size** (*int*) – the new size of the image

rollback_to_snap(*self*, *name*)

Revert the image to its contents at a snapshot. This is a potentially expensive operation, since it rolls back each object individually.

Parameters: **name** (*str*) – the snapshot to rollback to

Raises: **IOError**

set_snap(*self*, *name*)

Set the snapshot to read from. Writes will raise ReadOnlyImage while a snapshot is set. Pass None to unset the snapshot (reads come from the current image) , and allow writing again.

Parameters: **name** (*str or None*) – the snapshot to read from, or None to unset the snapshot

set_snap_limit(*self*, *limit*)

Set the snapshot limit for an image.

Parameters: **limit** – the new limit to set

size(*self*)

Get the size of the image. If open to a snapshot, returns the size of that snapshot.

Returns: the size of the image in bytes

snap_get_group_namespace(*self*, *snap_id*)

get the group namespace details. :param snap_id: the snapshot id of the group snapshot

snap_get_namespace_type(*self*, *snap_id*)

Get the snapshot namespace type. :param snap_id: the snapshot id of a snap shot

stat(*self*)

Get information about the image. Currently parent pool and parent name are always -1 and ‘’.

Returns: dict - contains the following keys:
- `size` (int) - the size of the image in bytes
- `obj_size` (int) - the size of each object that comprises the image
- `num_objs` (int) - the number of objects in the image
- `order` (int) - log_2(object_size)
- `block_name_prefix` (str) - the prefix of the RADOS objects used to store the image
- `parent_pool` (int) - deprecated
- `parent_name` (str) - deprecated

See also **format()** and **features()**.

stripe_count(*self*)

Return the stripe count used for the image.

stripe_unit(*self*)

Return the stripe unit used for the image.

unlock(*self*, *cookie*)

Release a lock on the image that was locked by this rados client.

unprotect_snap(*self*, *name*)

Mark a snapshot unprotected. This allows it to be deleted if it was protected.

Parameters: **name** (*str*) – the snapshot to unprotect

Raises: **IOError**, **ImageNotFound**

update_features(*self*, *features*, *enabled*)

Update the features bitmask of the image by enabling/disabling a single feature. The feature must support the ability to be dynamically enabled/disabled.

Parameters:
- **features** (*int*) – feature bitmask to enable/disable
- **enabled** (*bool*) – whether to enable/disable the feature

Raises: **InvalidArgument**

watchers_list(*self*)

List image watchers.

Returns: **WatcherIterator**

write(*self*, *data*, *offset*, *fadvise_flags=0*)

Write data to the image. Raises **InvalidArgument** if part of the write would fall outside the image.

Parameters:
- **data** (*bytes*) – the data to be written
- **offset** (*int*) – where to start writing data
- **fadvise_flags** (*int*) – fadvise flags for this write

Returns: int - the number of bytes written

Raises: **IncompleteWriteError**, **LogicError**, **InvalidArgument**, **IOError**

class rbd.**SnapIterator**(*Image image*)

Iterator over snapshot info for an image.

Yields a dictionary containing information about a snapshot.

Keys are:

- id (int) - numeric identifier of the snapshot
- size (int) - size of the image at the time of snapshot (in bytes)
- name (str) - name of the snapshot