duplicity

Kenneth Loafman

Mar 21, 2023

TABLE OF CONTENTS

Introduction

Duplicity backs directories by producing encrypted tar-format volumes and uploading them to a remote or local file server. Because duplicity uses librsync, the incremental archives are space efficient and only record the parts of files that have changed since the last backup. Because duplicity uses GnuPG to encrypt and/or sign these archives, they will be safe from spying and/or modification by the server.

CHAPTER

ONE

DUPLICITY

1.1 duplicity package

1.1.1 Subpackages

1.1.1.1 duplicity.backends package

Submodules

duplicity.backends._boto_multi module

duplicity.backends._boto_single module

class duplicity.backends._boto_single.**BotoBackend**(*parsed_url*)

Bases: *Backend*

Backend for Amazon's Simple Storage System, (aka Amazon S3), though the use of the boto module, (http://code.google.com/p/boto/).

To make use of this backend you must set aws_access_key_id and aws_secret_access_key in your ~/.boto or /etc/boto.cfg with your Amazon Web Services key id and secret respectively. Alternatively you can export the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY.

__init__(*parsed_url*)

_close()

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

_retry_cleanup()

list_filenames_in_bucket()

pre_process_download(*remote_filename*, *wait=False*)

pre_process_download_batch(*remote_filenames*)

resetConnection()

upload(*filename*, *key*, *headers*)

duplicity.backends._boto_single.**get_connection**(*scheme*, *parsed_url*, *storage_uri*)

duplicity.backends._cf_cloudfiles module

class duplicity.backends._cf_cloudfiles.**CloudFilesBackend**(*parsed_url*)

Bases: *Backend*

Backend for Rackspace's CloudFiles

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

duplicity.backends._cf_pyrax module

class duplicity.backends._cf_pyrax.**PyraxBackend**(*parsed_url*)

Bases: *Backend*

Backend for Rackspace's CloudFiles using Pyrax

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

duplicity.backends.adbackend module

class duplicity.backends.adbackend.**ADBackend**(*parsed_url*)

Bases: *Backend*

Backend for Amazon Drive. It communicates directly with Amazon Drive using their RESTful API and does not rely on externally setup software (like acd_cli).

CLIENT_ID = 'amzn1.application-oa2-client.791c9c2d78444e85a32eb66f92eb6bcc'

CLIENT_SECRET = '5b322c6a37b25f16d848a6a556eddcc30314fc46ae65c87068ff1bc4588d715b'

MULTIPART_BOUNDARY = 'DuplicityFormBoundaryd66364f7f8924f7e9d478e19cf4b871d114a1e00262542'

OAUTH_AUTHORIZE_URL = 'https://www.amazon.com/ap/oa'

OAUTH_REDIRECT_URL = 'https://breunig.xyz/duplicity/copy.html'

OAUTH_SCOPE = ['clouddrive:read_other', 'clouddrive:write']

OAUTH_TOKEN_PATH = '/home/docs/.duplicity_ad_oauthtoken.json'

OAUTH_TOKEN_URL = 'https://api.amazon.com/auth/o2/token'

__init__(*parsed_url*)

_delete(*remote_filename*)

Delete file from Amazon Drive

_get(*remote_filename*, *local_path*)

Download file from Amazon Drive

_list()

List files in Amazon Drive backup folder

_put(*source_path*, *remote_filename*)

Upload a local file to Amazon Drive

_query(*remote_filename*)

Retrieve file size info from Amazon Drive

get_file_id(*remote_filename*)

Find id of remote file in backup target folder

initialize_oauth2_session()

Setup or refresh oauth2 session with Amazon Drive

mkdir(*parent_node_id*, *folder_name*)

Create a new folder as a child of a parent node

multipart_stream(*metadata*, *source_path*)

Generator for multipart/form-data file upload from source file

raise_for_existing_file(*remote_filename*)

Report error when file already existed in location and delete it

read_all_pages(*url*)

Iterates over nodes API URL until all pages were read

resolve_backup_target()
Resolve node id for remote backup target folder

duplicity.backends.azurebackend module

class duplicity.backends.azurebackend.**AzureBackend**(*parsed_url*)
Bases: *Backend*

Backend for Azure Blob Storage Service

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*remote_filename*, *local_path*)

_get_or_create_container()

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

_set_tier(*remote_filename*)

duplicity.backends.azurebackend.**_is_valid_container_name**(*name*)
Check, whether the given name conforms to the rules as defined in https://docs.microsoft.com/en-us/rest/api/storageservices/naming-and-referencing-containers–blobs–and-metadata for valid names.

duplicity.backends.b2backend module

class duplicity.backends.b2backend.**B2Backend**(*parsed_url*)
Bases: *Backend*

Backend for BackBlaze's B2 storage service

__init__(*parsed_url*)
Authorize to B2 api and set up needed variables

_delete(*filename*)
Delete filename from remote server

_get(*remote_filename*, *local_path*)
Download remote_filename to local_path

_list()
List files on remote server

_put(*source_path*, *remote_filename*)
Copy source_path to remote_filename

_query(*filename*)
Get size info of filename

file_info(*filename*)

class duplicity.backends.b2backend.**B2ProgressListener**

Bases: object

bytes_completed(*byte_count*)

close()

set_total_bytes(*total_byte_count*)

duplicity.backends.boxbackend module

class duplicity.backends.boxbackend.**BoxBackend**(*parsed_url*)

Bases: *Backend*

__init__(*parsed_url*)

_delete(*filename*)

Deletes file from the specified remote path

_get(*remote_filename*, *local_path*)

Downloads file from the specified remote path

_list()

Lists files in the specified remote path

_put(*source_path*, *remote_filename*)

Uploads file to the specified remote folder (tries to delete it first to make sure the new one can be uploaded)

_query_list(*filename_list*)

Query metadata for a list of file

delete(*remote_file*)

Delete file in box folder

download(*remote_file*, *local_file*)

Download file in box folder

folder_contents()

Lists files of a remote box path

get_box_client(*parsed_url*)

get_file_id_from_filename(*remote_filename*)

Get the fild id by its file name

get_id_from_path(*remote_path*, *parent_id='0'*)

Get the folder or file id from its path

makedirs(*remote_path*)

Create folder(s) in a path if necessary

upload(*remote_file*, *local_file*)

Upload local file to the box folder

duplicity.backends.cfbackend module

duplicity.backends.dpbxbackend module

class duplicity.backends.dpbxbackend.**DPBXBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using Dr*pB*x service

__init__(*parsed_url*)

_close(**args*)

close backend session? no! just "flush" the data

_delete(**args*)

_error_code(*operation*, *e*)

_get(**args*)

_list(**args*)

_put(**args*)

_query(**args*)

check_renamed_files(*file_list*)

load_access_token()

login()

obtain_access_token()

put_file_chunked(*source_path*, *remote_path*)

put_file_small(*source_path*, *remote_path*)

save_access_token(*access_token*)

user_authenticated()

duplicity.backends.dpbxbackend.**command**(*login_required=True*)

a decorator for handling authentication and exceptions

duplicity.backends.dpbxbackend.**log_exception**(*e*)

duplicity.backends.gdocsbackend module

class duplicity.backends.gdocsbackend.**GDocsBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using Google Google Documents List API

BACKUP_DOCUMENT_TYPE = 'application/binary'

ROOT_FOLDER_ID = 'folder%3Aroot'

__init__(*parsed_url*)

_authorize(*email*, *password*, *captcha_token=None*, *captcha_response=None*)

_delete(*filename*)

_fetch_entries(*folder_id*, *type*, *title=None*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.gdrivebackend module

class duplicity.backends.gdrivebackend.**GDriveBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using Google Drive API V3

MIN_RESUMABLE_UPLOAD = 5242880

PAGE_SIZE = 100

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *error*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

file_by_name(*filename*)

id_by_name(*filename*)

duplicity.backends.giobackend module

class duplicity.backends.giobackend.**GIOBackend**(*parsed_url*)

Bases: *Backend*

Use this backend when saving to a GIO URL. This is a bit of a meta-backend, in that it can handle multiple schemas. URLs look like schema://user@server/path.

__copy_file(*source*, *target*)

__copy_progress(**args*, ***kwargs*)

__done_with_mount(*fileobj*, *result*, *loop*)

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

duplicity.backends.giobackend.ensure_dbus()

duplicity.backends.hsibackend module

class duplicity.backends.hsibackend.HSIBackend(*parsed_url*)

Bases: *Backend*

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.hubicbackend module

class duplicity.backends.hubicbackend.HubicBackend(*parsed_url*)

Bases: *PyraxBackend*

Backend for Hubic using Pyrax

__init__(*parsed_url*)

duplicity.backends.idrivedbackend module

class duplicity.backends.idrivedbackend.IDriveBackend(*parsed_url*)

Bases: *Backend*

__init__(*parsed_url*)

_close()

_delete(*remote_filename*)

_delete_list(*filename_list*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

_query_list(*filename_list*)

connect()

list_raw()

request(*commandline*)

user_connected()

duplicity.backends.imapbackend module

class duplicity.backends.imapbackend.**ImapBackend**(*parsed_url*)

Bases: *Backend*

__init__(*parsed_url*)

_close()

_delete_list(*filename_list*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

delete_single_mail(*i*)

expunge()

imapf(*fun*, **args*)

prepareBody(*f*, *rname*)

resetConnection()

duplicity.backends.jottacloudbackend module

class duplicity.backends.jottacloudbackend.**JottaCloudBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using JottaCloud API

__init__(*parsed_url*)

_close()

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

Get size of filename

get_or_create_directory(*directory_name*)

duplicity.backends.jottacloudbackend.**get_duplicity_log_level**()

Get the current duplicity log level as a stdlib-compatible logging level

duplicity.backends.jottacloudbackend.**get_jotta_device**(*jfs*)

duplicity.backends.jottacloudbackend.**get_root_dir**(*jfs*)

duplicity.backends.jottacloudbackend.**set_jottalib_log_handlers**(*handlers*)

duplicity.backends.jottacloudbackend.**set_jottalib_logging_level**(*log_level*)

duplicity.backends.lftpbackend module

class duplicity.backends.lftpbackend.**LFTPBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using File Transfer Protocol

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.localbackend module

class duplicity.backends.localbackend.**LocalBackend**(*parsed_url*)

Bases: *Backend*

Use this backend when saving to local disk

Urls look like file://testfiles/output. Relative to root can be gotten with extra slash (file:///usr/local).

__init__(*parsed_url*)

_delete(*filename*)

_delete_list(*filenames*)

_get(*filename*, *local_path*)

_list()

_move(*source_path*, *remote_filename*)

_put(*source_path*, *remote_filename*)

_query(*filename*)

duplicity.backends.mediafirebackend module

MediaFire Duplicity Backend

class duplicity.backends.mediafirebackend.**MediafireBackend**(*parsed_url*)

Bases: *Backend*

Use this backend when saving to MediaFire

URLs look like mf:/root/folder.

__init__(*parsed_url*)

_build_uri(*filename=''*)

Build relative URI

_delete(*filename*)

Delete single file

_delete_list(*filename_list*)

Delete list of files

_get(*filename*, *local_path*)

Download file

_list()

List files in backup directory

_put(*source_path*, *remote_filename=None*)

Upload file

_query(*filename*)

Stat the remote file

duplicity.backends.megabackend module

class duplicity.backends.megabackend.**MegaBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using Mega.co.nz API

__init__(*parsed_url*)

_check_binary_exists(*cmd*)

checks that a specified command exists in the current path

_delete(*filename*)

deletes remote

_get(*remote_filename*, *local_path*)

downloads file from Mega

_list()

list files in the backup folder

_makedir(*path*)

creates a remote directory

_makedir_recursive(*path*)

creates a remote directory (recursively the whole path), ingores errors

_put(*source_path*, *remote_filename*)

uploads file to Mega (deletes it first, to ensure it does not exist)

delete(*remote_file*)

download(*remote_file*, *local_file*)

folder_contents(*files_only=False*)

lists contents of a folder, optionally ignoring subdirectories

upload(*local_file*, *remote_file*)

duplicity.backends.megav2backend module

class duplicity.backends.megav2backend.**Megav2Backend**(*parsed_url*)

Bases: *Backend*

Backend for MEGA.nz cloud storage, only one that works for accounts created since Nov. 2018 See https://github.com/megous/megatools/issues/411 for more details

This MEGA backend resorts to official tools (MEGAcmd) as available at https://mega.nz/cmd MEGAcmd works through a single binary called "mega-cmd", which talks to a backend server "mega-cmd-server", which keeps state (for example, persisting a session). Multiple "mega-*" shell wrappers (ie. "mega-ls") exist as the user interface to "mega-cmd" and MEGA API The full MEGAcmd User Guide can be found in the software's GitHub page below : https://github.com/meganz/MEGAcmd/blob/master/UserGuide.md

__init__(*parsed_url*)

_check_binary_exists(*cmd*)

Checks that a specified command exists in the running user command path

_close()

Function called when backend is done being used

_delete(*filename*)

Deletes file from the specified remote path

_get(*remote_filename*, *local_path*)

Downloads file from the specified remote path

_list()

Lists files in the specified remote path

_makedir(*path*)

Creates a remote directory (recursively if necessary)

_put(*source_path*, *remote_filename*)

Uploads file to the specified remote folder (tries to delete it first to make sure the new one can be uploaded)

delete(*remote_file*)

Deletes a file from a remote MEGA path

download(*remote_file*, *local_file*)

Downloads a file from a remote MEGA path

folder_contents(*files_only=False*)

Lists contents of a remote MEGA path, optionally ignoring subdirectories

mega_login()

Helper function to call from each method interacting with MEGA to make sure a session already exists or one is created to start with

upload(*local_file*, *remote_file*)

Uploads a file to a remote MEGA path

duplicity.backends.megav3backend module

class duplicity.backends.megav3backend.**Megav3Backend**(*parsed_url*)

Bases: *Backend*

Backend for MEGA.nz cloud storage, only one that works for accounts created since Nov. 2018 See https://github.com/megous/megatools/issues/411 for more details

This MEGA backend resorts to official tools (MEGAcmd) as available at https://mega.nz/cmd MEGAcmd works through a single binary called "mega-cmd", which keeps state (for example, persisting a session). Multiple "mega-*" shell wrappers (ie. "mega-ls") exist as the user interface to "mega-cmd" and MEGA API The full MEGAcmd User Guide can be found in the software's GitHub page below : https://github.com/meganz/MEGAcmd/blob/master/UserGuide.md

__init__(*parsed_url*)

_check_binary_exists(*cmd*)

Checks that a specified command exists in the running user command path

_close()

Function called when backend is done being used

_delete(*filename*)

Deletes file from the specified remote path

_get(*remote_filename*, *local_path*)

Downloads file from the specified remote path

_list()

Lists files in the specified remote path

_makedir(*path*)

Creates a remote directory (recursively if necessary)

_put(*source_path*, *remote_filename*)

Uploads file to the specified remote folder (tries to delete it first to make sure the new one can be uploaded)

delete(*remote_file*)

Deletes a file from a remote MEGA path

download(*remote_file*, *local_file*)

Downloads a file from a remote MEGA path

ensure_mega_cmd_running()

Trigger any mega command to ensure mega-cmd server is running

folder_contents(*files_only=False*)

Lists contents of a remote MEGA path, optionally ignoring subdirectories

mega_login()

Helper function to check existing session exists

upload(*local_file*, *remote_file*)

Uploads a file to a remote MEGA path

duplicity.backends.multibackend module

class duplicity.backends.multibackend.**MultiBackend**(*parsed_url*)

Bases: *Backend*

Store files across multiple remote stores. URL is a path to a local file containing URLs/other config defining the remote store

__affinities = {}

__init__(*parsed_url*)

__knownQueryParameters = frozenset({'mode', 'onfail', 'subpath'})

__mode = 'stripe'

__mode_allowedSet = frozenset({'mirror', 'stripe'})

__onfail_mode = 'continue'

__onfail_mode_allowedSet = frozenset({'abort', 'continue'})

__stores = []

__subpath = ''

__write_cursor = 0

_delete(*filename*)

_delete_list(*filenames*)

_eligible_stores(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

static get_query_params(*parsed_url*)

pre_process_download(*filename*)

pre_process_download_batch(*filenames*)

duplicity.backends.ncftpbackend module

class duplicity.backends.ncftpbackend.**NCFTPBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using File Transfer Protocol

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.onedrivebackend module

class duplicity.backends.onedrivebackend.**DefaultOAuth2Session**(*api_uri*)

Bases: *OneDriveOAuth2Session*

A possibly-interactive console session using a built-in API key

CLIENT_ID = '1612f841-ae01-46ab-9535-43ba6ec04029'

OAUTH_AUTHORIZE_URI = 'https://login.microsoftonline.com/common/oauth2/v2.0/authorize'

OAUTH_REDIRECT_URI = 'https://login.microsoftonline.com/common/oauth2/nativeclient'

OAUTH_SCOPE = ['Files.Read', 'Files.ReadWrite', 'Files.Read.All', 'Files.ReadWrite.All', 'User.Read', 'offline_access']

OAUTH_TOKEN_PATH = '/home/docs/.duplicity_onedrive_oauthtoken.json'

__init__(*api_uri*)

token_updater(*token*)

class duplicity.backends.onedrivebackend.**ExternalOAuth2Session**(*client_id*, *refresh_token*)

Bases: *OneDriveOAuth2Session*

Caller is managing tokens and provides an active refresh token.

__init__(*client_id*, *refresh_token*)

class duplicity.backends.onedrivebackend.**OneDriveBackend**(*parsed_url*)

Bases: *Backend*

Uses Microsoft OneDrive (formerly SkyDrive) for backups.

API_URI = 'https://graph.microsoft.com/v1.0/'

REQUIRED_FRAGMENT_SIZE_MULTIPLE = 327680

__init__(*parsed_url*)

_delete(*remote_filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*remote_filename*)

_retry_cleanup()

initialize_oauth2_session()

class duplicity.backends.onedrivebackend.**OneDriveOAuth2Session**

Bases: object

A tiny wrapper for OAuth2Session that handles some OneDrive details.

OAUTH_TOKEN_URI = 'https://login.microsoftonline.com/common/oauth2/v2.0/token'

__init__()

delete(**args*, ***kwargs*)

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

post(**args*, ***kwargs*)

put(**args*, ***kwargs*)

duplicity.backends.par2backend module

class duplicity.backends.par2backend.**Par2Backend**(*parsed_url*)

Bases: *Backend*

This backend wrap around other backends and create Par2 recovery files before the file and the Par2 files are transfered with the wrapped backend.

If a received file is corrupt it will try to repair it on the fly.

__init__(*parsed_url*)

close()

delete(*filename*)

delete given filename and its .par2 files

delete_list(*filename_list*)

delete given filename_list and all .par2 files that belong to them

error_code(*operation*, *e*)

get(*remote_filename*, *local_path*)

transfer remote_filename and the related .par2 file into a temp-dir. remote_filename will be renamed into local_path before finishing.

If "par2 verify" detect an error transfer the Par2-volumes into the temp-dir and try to repair.

list()

Return list of filenames (byte strings) present in backend

Files ending with ".par2" will be excluded from the list.

move(*local*, *remote*)

put(*local*, *remote*)

query(*filename*)

query_list(*filename_list*)

retry_cleanup()

transfer(*method*, *source_path*, *remote_filename*)

create Par2 files and transfer the given file and the Par2 files with the wrapped backend.

Par2 must run on the real filename or it would restore the temp-filename later on. So first of all create a tempdir and symlink the soure_path with remote_filename into this.

unfiltered_list()

duplicity.backends.pcabackend module

class duplicity.backends.pcabackend.**PCABackend**(*parsed_url*)

Bases: *Backend*

Backend for OVH PCA

__init__(*parsed_url*)

__list_objs(*ffilter=None*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

pre_process_download_batch(*remote_filenames*)

This is called before downloading volumes from this backend by main engine. For PCA, volumes passed as argument need to be unsealed. This method is blocking, showing a status at regular interval

unseal(*remote_filename*)

unseal_status(*u_remote_filenames*)

Shows unsealing status for input volumes

duplicity.backends.pydrivebackend module

class duplicity.backends.pydrivebackend.**PyDriveBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using PyDrive API

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *error*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

file_by_name(*filename*)

id_by_name(*filename*)

duplicity.backends.rclonebackend module

class duplicity.backends.rclonebackend.**RcloneBackend**(*parsed_url*)

Bases: *Backend*

__init__(*parsed_url*)

_delete(*remote_filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_subprocess_safe_popen(*commandline*)

duplicity.backends.rsyncbackend module

class duplicity.backends.rsyncbackend.**RsyncBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using rsync

rsync backend contributed by Sebastian Wilhelmi <seppi@seppi.de> rsyncd auth, alternate port support Copyright 2010 by Edgar Soldin <edgar.soldin@web.de>

__init__(*parsed_url*)

rsyncBackend initializer

_delete_list(*filename_list*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

get_rsync_path()

over_rsyncd()

duplicity.backends.s3_boto3_backend module

class duplicity.backends.s3_boto3_backend.**S3Boto3Backend**(*parsed_url*)

Bases: *Backend*

Backend for Amazon's Simple Storage System, (aka Amazon S3), though the use of the boto3 module. (See https://boto3.amazonaws.com/v1/documentation/api/latest/index.html for information on boto3.)

Pursuant to Amazon's announced deprecation of path style S3 access, this backend only supports virtual host style bucket URIs. See the man page for full details.

To make use of this backend, you must provide AWS credentials. This may be done in several ways: through the environment variables AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY, by the ~/.aws/credentials file, by the ~/.aws/config file, or by using the boto2 style ~/.boto or /etc/boto.cfg files.

__init__(*parsed_url*)

_delete(*remote_filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*local_source_path*, *remote_filename*)

_query(*remote_filename*)

reset_connection()

class duplicity.backends.s3_boto3_backend.**UploadProgressTracker**

Bases: `object`

__init__()

progress_cb(*fresh_byte_count*)

duplicity.backends.s3_boto_backend module

duplicity.backends.slatebackend module

class duplicity.backends.slatebackend.**SlateBackend**(*parsed_url*)

Bases: *Backend*

Backend for Slate

__init__(*parsed_url*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.ssh_paramiko_backend module

class duplicity.backends.ssh_paramiko_backend.SSHParamikoBackend(*parsed_url*)

Bases: *Backend*

This backend accesses files using the sftp or scp protocols. It does not need any local client programs, but an ssh server and the sftp program must be installed on the remote side (or with scp, the programs scp, ls, mkdir, rm and a POSIX-compliant shell).

Authentication keys are requested from an ssh agent if present, then ~/.ssh/id_rsa/dsa are tried. If -oIdentityFile=path is present in –ssh-options, then that file is also tried. The passphrase for any of these keys is taken from the URI or FTP_PASSWORD. If none of the above are available, password authentication is attempted (using the URI or FTP_PASSWORD).

Missing directories on the remote side will be created.

If scp is active then all operations on the remote side require passing arguments through a shell, which introduces unavoidable quoting issues: directory and file names that contain single quotes will not work. This problem does not exist with sftp.

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

gethostconfig(*file*, *host*)

runremote(*cmd*, *ignoreexitcode=False*, *errorprefix=''*)

small convenience function that opens a shell channel, runs remote command and returns stdout of command. throws an exception if exit code!=0 and not ignored

duplicity.backends.ssh_pexpect_backend module

class duplicity.backends.ssh_pexpect_backend.SSHPExpectBackend(*parsed_url*)

Bases: *Backend*

This backend copies files using scp. List not supported. Filenames should not need any quoting or this will break.

__init__(*parsed_url*)

scpBackend initializer

_delete(*filename*)

_delete_list(*filename_list*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

get_scp(*remote_filename*, *local_path*)

get_sftp(*remote_filename*, *local_path*)

put_scp(*source_path*, *remote_filename*)

put_sftp(*source_path*, *remote_filename*)

run_scp_command(*commandline*)

Run an scp command, responding to password prompts

run_sftp_command(*commandline*, *commands*)

Run an sftp command, responding to password prompts, passing commands from list

duplicity.backends.swiftbackend module

class duplicity.backends.swiftbackend.**SwiftBackend**(*parsed_url*)

Bases: *Backend*

Backend for Swift

__init__(*parsed_url*)

_delete(*filename*)

_error_code(*operation*, *e*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_query(*filename*)

duplicity.backends.sxbackend module

class duplicity.backends.sxbackend.**SXBackend**(*parsed_url*)

Bases: *Backend*

Connect to remote store using Skylable Protocol

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

duplicity.backends.tahoebackend module

class duplicity.backends.tahoebackend.**TAHOEBackend**(*parsed_url*)

Bases: *Backend*

Backend for the Tahoe file system

__init__(*parsed_url*)

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

get_remote_path(*filename=None*)

run(**args*)

duplicity.backends.webdavbackend module

class duplicity.backends.webdavbackend.**CustomMethodRequest**(*method*, **args*, ***kwargs*)

Bases: Request

This request subclass allows explicit specification of the HTTP request method. Basic urllib.request.Request class chooses GET or POST depending on self.has_data()

__init__(*method*, **args*, ***kwargs*)

get_method()

Return a string indicating the HTTP request method.

class duplicity.backends.webdavbackend.**VerifiedHTTPSConnection**(**args*, ***kwargs*)

Bases: HTTPSConnection

__init__(**args*, ***kwargs*)

connect()

Connect to a host on a given (SSL) port.

request(**args*, ***kwargs*)

Send a complete request to the server.

class duplicity.backends.webdavbackend.**WebDAVBackend**(*parsed_url*)

Bases: *Backend*

Backend for accessing a WebDAV repository.

webdav backend contributed in 2006 by Jesper Zedlitz <jesper@zedlitz.de>

__init__(*parsed_url*)

_close()

_delete(*filename*)

_get(*remote_filename*, *local_path*)

_list()

_put(*source_path*, *remote_filename*)

_retry_cleanup()

connect(*forced=False*)

Connect or re-connect to the server, updates self.conn # reconnect on errors as a precaution, there are errors e.g. # "[Errno 32] Broken pipe" or SSl errors that render the connection unusable

getText(*nodelist*)

get_authorization(*response*, *path*)

Fetches the auth header based on the requested method (basic or digest)

get_basic_authorization()

Returns the basic auth header

get_digest_authorization(*path*)

Returns the digest auth header

get_kerberos_authorization()

listbody = '<?xml version="1.0"?><D:propfind xmlns:D="DAV:"><D:prop><D:resourcetype/></D:prop></D:propfind>'

Connect to remote store using WebDAV Protocol

makedir()

Make (nested) directories on the server.

parse_digest_challenge(*challenge_string*)

request(*method*, *path*, *data=None*, *redirected=0*)

Wraps the connection.request method to retry once if authentication is required

sanitize_path(*path*)

taste_href(*href*)

Internal helper to taste the given href node and, if it is a duplicity file, collect it as a result file.

@return: A matching filename, or None if the href did not match.

Module contents

Imports of backends should not be done directly in this module. All backend imports are done via import_backends() in backend.py. This file is only to instantiate the duplicity.backends module itself.

1.1.2 Submodules

1.1.2.1 duplicity.asyncscheduler module

Asynchronous job scheduler, for concurrent execution with minimalistic dependency guarantees.

class duplicity.asyncscheduler.**AsyncScheduler**(*concurrency*)

Bases: object

Easy-to-use scheduler of function calls to be executed concurrently. A very simple dependency mechanism exists in the form of barriers (see insert_barrier()).

Each instance has a concurrency level associated with it. A concurrency of 0 implies that all tasks will be executed synchronously when scheduled. A concurrency of 1 indicates that a task will be executed asynchronously, but never concurrently with other tasks. Both 0 and 1 guarantee strict ordering among all tasks (i.e., they will be executed in the order scheduled).

At concurrency levels above 1, the tasks will end up being executed in an order undetermined except insofar as is enforced by calls to insert_barrier().

An AsynchScheduler should be created for any independent process; the scheduler will assume that if any background job fails (raises an exception), it makes further work moot.

__execute_caller(*caller*)

__init__(*concurrency*)

Create an asynchronous scheduler that executes jobs with the given level of concurrency.

__run_asynchronously(*fn*, *params*)

__run_synchronously(*fn*, *params*)

__start_worker(*caller*)

Start a new worker.

insert_barrier()

Proclaim that any tasks scheduled prior to the call to this method MUST be executed prior to any tasks scheduled after the call to this method.

The intended use case is that if task B depends on A, a barrier must be inserted in between to guarantee that A happens before B.

schedule_task(*fn*, *params*)

Schedule the given task (callable, typically function) for execution. Pass the given parameters to the function when calling it. Returns a callable which can optionally be used to wait for the task to complete, either by returning its return value or by propagating any exception raised by said task.

This method may block or return immediately, depending on the configuration and state of the scheduler.

This method may also raise an exception in order to trigger failures early, if the task (if run synchronously) or a previous task has already failed.

NOTE: Pay particular attention to the scope in which this is called. In particular, since it will execute concurrently in the background, assuming fn is a closure, any variables used most be properly bound in the closure. This is the reason for the convenience feature of being able to give parameters to the call, to avoid having to wrap the call itself in a function in order to "fixate" variables in, for example, an enclosing loop.

wait()

Wait for the scheduler to become entirely empty (i.e., all tasks having run to completion).

IMPORTANT: This is only useful with a single caller scheduling tasks, such that no call to schedule_task() is currently in progress or may happen subsequently to the call to wait().

1.1.2.2 duplicity.backend module

Provides a common interface to all backends and certain sevices intended to be used by the backends themselves.

class duplicity.backend.**Backend**(*parsed_url*)

Bases: object

See README in backends directory for information on how to write a backend.

__init__(*parsed_url*)

__subprocess_popen(*args*)

For internal use. Execute the given command line, interpreted as a shell command. Returns int Exitcode, string StdOut, string StdErr

get_password()

Return a password for authentication purposes. The password will be obtained from the backend URL, the environment, by asking the user, or by some other method. When applicable, the result will be cached for future invocations.

munge_password(*commandline*)

Remove password from commandline by substituting the password found in the URL, if any, with a generic place-holder.

This is intended for display purposes only, and it is not guaranteed that the results are correct (i.e., more than just the ':password@' may be substituted.

popen_breaks = {}

subprocess_popen(*commandline*)

Execute the given command line with error check. Returns int Exitcode, string StdOut, string StdErr

Raise a BackendException on failure.

use_getpass = True

class duplicity.backend.**BackendWrapper**(*backend*)

Bases: object

Represents a generic duplicity backend, capable of storing and retrieving files.

__do_put(*source_path*, *remote_filename*)

__init__(*backend*)

_do_delete(**args*)

_do_delete_list(**args*)

_do_query(**args*)

_do_query_list(**args*)

close()
Close the backend, releasing any resources held and invalidating any file objects obtained from the backend.

delete(*filename_list*)
Delete each filename in filename_list, in order if possible.

get(**args*)

get_data(*filename*, *parseresults=None*)
Retrieve a file from backend, process it, return contents.

get_fileobj_read(*filename*, *parseresults=None*)
Return fileobject opened for reading of filename on backend

The file will be downloaded first into a temp file. When the returned fileobj is closed, the temp file will be deleted.

list(**args*)

move(**args*)

pre_process_download(*remote_filename*)
Manages remote access before downloading files (unseal data in cold storage for instance)

pre_process_download_batch(*remote_filenames*)
Manages remote access before downloading files (unseal data in cold storage for instance)

put(**args*)

query_info(*filename_list*)
Return metadata about each filename in filename_list

class duplicity.backend.ParsedUrl(*url_string*)
Bases: object

Parse the given URL as a duplicity backend URL.

Returns the data of a parsed URL with the same names as that of the standard urlparse.urlparse() except that all values have been resolved rather than deferred. There are no get_* members. This makes sure that the URL parsing errors are detected early.

Raise InvalidBackendURL on invalid URL's

__init__(*url_string*)

geturl()

strip_auth()

duplicity.backend._get_code_from_exception(*backend*, *operation*, *e*)

duplicity.backend.get_backend(*url_string*)
Instantiate a backend suitable for the given URL, or return None if the given string looks like a local path rather than a URL.

Raise InvalidBackendURL if the URL is not a valid URL.

duplicity.backend.get_backend_object(*url_string*)
Find the right backend class instance for the given URL, or return None if the given string looks like a local path rather than a URL.

Raise InvalidBackendURL if the URL is not a valid URL.

duplicity.backend.**import_backends**()

Import files in the duplicity/backends directory where the filename ends in 'backend.py' and ignore the rest.

@rtype: void @return: void

duplicity.backend.**is_backend_url**(*url_string*)

@return Whether the given string looks like a backend URL.

duplicity.backend.**register_backend**(*scheme*, *backend_factory*)

Register a given backend factory responsible for URL:s with the given scheme.

The backend must be a callable which, when called with a URL as the single parameter, returns an object implementing the backend protocol (i.e., a subclass of Backend).

Typically the callable will be the Backend subclass itself.

This function is not thread-safe and is intended to be called during module importation or start-up.

duplicity.backend.**register_backend_prefix**(*scheme*, *backend_factory*)

Register a given backend factory responsible for URL:s with the given scheme prefix.

The backend must be a callable which, when called with a URL as the single parameter, returns an object implementing the backend protocol (i.e., a subclass of Backend).

Typically the callable will be the Backend subclass itself.

This function is not thread-safe and is intended to be called during module importation or start-up.

duplicity.backend.**retry**(*operation*, *fatal=True*)

duplicity.backend.**strip_auth_from_url**(*parsed_url*)

Return a URL from a urlparse object without a username or password.

duplicity.backend.**strip_prefix**(*url_string*, *prefix_scheme*)

strip the prefix from a string e.g. par2+ftp://... -> ftp://...

1.1.2.3 duplicity.cached_ops module

Cache-wrapped functions for grp and pwd lookups.

class duplicity.cached_ops.**CachedCall**(*f*)

Bases: object

Decorator for caching the results of function calls.

__init__(*f*)

1.1.2.4 duplicity.commandline module

Parse command line, check for consistency, and set config

class duplicity.commandline.**DupOption**(**opts*, ***attrs*)

Bases: Option

ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count', 'callback', 'help', 'version', 'extend')

ALWAYS_TYPED_ACTIONS = ('store', 'append', 'extend')

STORE_ACTIONS = ('store', 'store_const', 'store_true', 'store_false', 'append', 'append_const', 'count', 'extend')

TYPED_ACTIONS = ('store', 'append', 'callback', 'extend')

TYPES = ('string', 'int', 'long', 'float', 'complex', 'choice', 'file', 'time', 'verbosity')

TYPE_CHECKER = {'choice': <function check_choice>, 'complex': <function check_builtin>, 'file': <function check_file>, 'float': <function check_builtin>, 'int': <function check_builtin>, 'long': <function check_builtin>, 'time': <function check_time>, 'verbosity': <function check_verbosity>}

take_action(*action*, *dest*, *opt*, *value*, *values*, *parser*)

duplicity.commandline.**ProcessCommandLine**(*cmdline_list*)

Process command line, set config, return action

action will be "list-current", "collection-status", "cleanup", "remove-old", "restore", "verify", "full", or "inc".

duplicity.commandline.**args_to_path_backend**(*arg1*, *arg2*)

Given exactly two arguments, arg1 and arg2, figure out which one is the backend URL and which one is a local path, and return (local, backend).

duplicity.commandline.**check_consistency**(*action*)

Final consistency check, see if something wrong with command line

duplicity.commandline.**check_file**(*option*, *opt*, *value*)

duplicity.commandline.**check_time**(*option*, *opt*, *value*)

duplicity.commandline.**check_verbosity**(*option*, *opt*, *value*)

duplicity.commandline.**command_line_error**(*message*)

Indicate a command line error and exit

duplicity.commandline.**expand_archive_dir**(*archdir*, *backname*)

Return expanded version of archdir joined with backname.

duplicity.commandline.**expand_fn**(*filename*)

duplicity.commandline.**generate_default_backup_name**(*backend_url*)

@param backend_url: URL to backend. @returns A default backup name (string).

duplicity.commandline.**noop**()

duplicity.commandline.**old_fn_deprecation**(*opt*)

duplicity.commandline.**old_globbing_filelist_deprecation**(*opt*)

duplicity.commandline.**parse_cmdline_options**(*arglist*)

Parse argument list

duplicity.commandline.**process_local_dir**(*action*, *local_pathname*)

Check local directory, set config.local_path

duplicity.commandline.**set_archive_dir**(*dirstring*)

Check archive dir and set global

duplicity.commandline.**set_backend**(*arg1*, *arg2*)

Figure out which arg is url, set backend

Return value is pair (path_first, path) where is_first is true iff path made from arg1.

duplicity.commandline.**set_selection**()

Return selection iter starting at filename with arguments applied

duplicity.commandline.**set_sign_key**(*sign_key*)

Set config.sign_key assuming proper key given

duplicity.commandline.**stdin_deprecation**(*opt*)

duplicity.commandline.**usage**()

Returns terse usage info. The code is broken down into pieces for ease of translation maintenance. Any comments that look extraneous or redundant should be assumed to be for the benefit of translators, since they can get each string (paired with its preceding comment, if any) independently of the others.

1.1.2.5 duplicity.config module

Store global configuration information

1.1.2.6 duplicity.diffdir module

Functions for producing signatures and deltas of directories

Note that the main processes of this module have two parts. In the first, the signature or delta is constructed of a ROPath iterator. In the second, the ROPath iterator is put into tar block form.

class duplicity.diffdir.**DeltaTarBlockIter**(*input_iter*)

Bases: *TarBlockIter*

TarBlockIter that yields parts of a deltatar file

Unlike SigTarBlockIter, the argument to __init__ is a delta_path_iter, so the delta information has already been calculated.

get_data_block(*fp*)

Return pair (next data block, boolean last data block)

process(*delta_ropath*)

Get a tarblock from delta_ropath

process_continued()

Return next volume in multivol diff or snapshot

exception duplicity.diffdir.**DiffDirException**

Bases: Exception

duplicity.diffdir.**DirDelta**(*path_iter*, *dirsig_fileobj_list*)

Produce tarblock diff given dirsig_fileobj_list and pathiter

dirsig_fileobj_list should either be a tar fileobj or a list of those, sorted so the most recent is last.

duplicity.diffdir.**DirDelta_WriteSig**(*path_iter*, *sig_infp_list*, *newsig_outfp*)

Like DirDelta but also write signature into sig_fileobj

Like DirDelta, sig_infp_list can be a tar fileobj or a sorted list of those. A signature will only be written to newsig_outfp if it is different from (the combined) sig_infp_list.

duplicity.diffdir.**DirFull**(*path_iter*)

Return a tarblock full backup of items in path_iter

A full backup is just a diff starting from nothing (it may be less elegant than using a standard tar file, but we can be sure that it will be easy to split up the tar and make the volumes the same sizes).

duplicity.diffdir.**DirFull_WriteSig**(*path_iter*, *sig_outfp*)

Return full backup like above, but also write signature to sig_outfp

duplicity.diffdir.**DirSig**(*path_iter*)

Alias for SigTarBlockIter below

class duplicity.diffdir.**DummyBlockIter**(*input_iter*)

Bases: *TarBlockIter*

TarBlockIter that does no file reading

process(*delta_ropath*)

Get a fake tarblock from delta_ropath

class duplicity.diffdir.**FileWithReadCounter**(*infile*)

Bases: `object`

File-like object which also computes amount read as it is read

__init__(*infile*)

FileWithReadCounter initializer

close()

read(*length=-1*)

class duplicity.diffdir.**FileWithSignature**(*infile*, *callback*, *filelen*, **extra_args*)

Bases: `object`

File-like object which also computes signature as it is read

__init__(*infile*, *callback*, *filelen*, **extra_args*)

FileTee initializer

The object will act like infile, but whenever it is read it add infile's data to a SigGenerator object. When the file has been read to the end the callback will be called with the calculated signature, and any extra_args if given.

filelen is used to calculate the block size of the signature.

blocksize = 32768

close()

read(*length=-1*)

class duplicity.diffdir.**SigTarBlockIter**(*input_iter*)

Bases: *TarBlockIter*

TarBlockIter that yields blocks of a signature tar from path_iter

process(*path*)

Return associated signature TarBlock from path

class duplicity.diffdir.**TarBlock**(*index*, *data*)

Bases: object

Contain information to add next file to tar

__init__(*index*, *data*)

TarBlock initializer - just store data

class duplicity.diffdir.**TarBlockIter**(*input_iter*)

Bases: object

A bit like an iterator, yield tar blocks given input iterator

Unlike an iterator, however, control over the maximum size of a tarblock is available by passing an argument to next(). Also the get_footer() is available.

__init__(*input_iter*)

TarBlockIter initializer

get_footer()

Return closing string for tarfile, reset offset

get_previous_index()

Return index of last tarblock, or None if no previous index

get_read_size()

process(*val*)

Turn next value of input_iter into a TarBlock

process_continued()

Get more tarblocks

If processing val above would produce more than one TarBlock, get the rest of them by calling process_continue.

queue_index_data(*data*)

Next time next() is called, we will return data instead of processing

recall_index()

Retrieve index remembered with remember_next_index

remember_next_index()

When called, remember the index of the next block iterated

tarinfo2tarblock(*index*, *tarinfo*, *file_data=b''*)

Make tarblock out of tarinfo and file data

duplicity.diffdir.**collate2iters**(*riter1*, *riter2*)

Collate two iterators.

The elements yielded by each iterator must be have an index variable, and this function returns pairs (elem1, elem2), (elem1, None), or (None, elem2) two elements in a pair will have the same index, and earlier indicies are yielded later than later indicies.

duplicity.diffdir.**combine_path_iters**(*path_iter_list*)

Produce new iterator by combining the iterators in path_iter_list

This new iter will iterate every path that is in path_iter_list in order of increasing index. If multiple iterators in path_iter_list yield paths with the same index, combine_path_iters will discard all paths but the one yielded by the last path_iter.

This is used to combine signature iters, as the output will be a full up-to-date signature iter.

duplicity.diffdir.**delta_iter_error_handler**(*exc*, *new_path*, *sig_path*, *sig_tar=None*)

Called by get_delta_iter, report error in getting delta

duplicity.diffdir.**get_block_size**(*file_len*)

Return a reasonable block size to use on files of length file_len

If the block size is too big, deltas will be bigger than is necessary. If the block size is too small, making deltas and patching can take a really long time.

duplicity.diffdir.**get_combined_path_iter**(*sig_infp_list*)

Return path iter combining signatures in list of open sig files

duplicity.diffdir.**get_delta_iter**(*new_iter*, *sig_iter*, *sig_fileobj=None*)

Generate delta iter from new Path iter and sig Path iter.

For each delta path of regular file type, path.difftype with be set to "snapshot", "diff". sig_iter will probably iterate ROPaths instead of Paths.

If sig_fileobj is not None, will also write signatures to sig_fileobj.

duplicity.diffdir.**get_delta_path**(*new_path*, *sig_path*, *sigTarFile=None*)

Return new delta_path which, when read, writes sig to sig_fileobj, if sigTarFile is not None

duplicity.diffdir.**log_delta_path**(*delta_path*, *new_path=None*, *stats=None*)

Look at delta path and log delta. Add stats if new_path is set

duplicity.diffdir.**sigtar2path_iter**(*sigtarobj*)

Convert signature tar file object open for reading into path iter

duplicity.diffdir.**write_block_iter**(*block_iter*, *out_obj*)

Write block_iter to filename, path, or file object

1.1.2.7 duplicity.dup_collections module

Classes and functions on collections of backup volumes

class duplicity.dup_collections.**BackupChain**(*backend*)

Bases: `object`

BackupChain - a number of linked BackupSets

A BackupChain always starts with a full backup set and continues with incremental ones.

__init__(*backend*)

Initialize new chain, only backend is required at first

add_inc(*incset*)

Add incset to self. Return False if incset does not match

delete(*keep_full=False*)

Delete all sets in chain, in reverse order

get_all_sets()

Return list of all backup sets in chain

get_first()

Return first BackupSet in chain (ie the full backup)

get_last()

Return last BackupSet in chain

get_num_volumes()

Return the total number of volumes in the chain

get_sets_at_time(*time*)

Return a list of sets in chain earlier or equal to time

set_full(*fullset*)

Add full backup set

short_desc()

Return a short one-line description of the chain, suitable for log messages.

to_log_info(*prefix=''*)

Return summary, suitable for printing to log

class duplicity.dup_collections.**BackupSet**(*backend*, *action*)

Bases: object

Backup set - the backup information produced by one session

__init__(*backend*, *action*)

Initialize new backup set, only backend is required at first

add_filename(*filename*, *pr=None*)

Add a filename to given set. Return true if it fits.

The filename will match the given set if it has the right times and is of the right type. The information will be set from the first filename given.

@param filename: name of file to add @type filename: string

@param pr: pre-computed result of file_naming.parse(filename) @type pr: Optional[ParseResults]

check_manifests(*check_remote=True*)

Make sure remote manifest is equal to local one

delete()

Remove all files in set, both local and remote

get_filenames()

Return sorted list of (remote) filenames of files in set

get_files_changed()

get_local_manifest()

Return manifest object by reading local manifest file

get_manifest()

Return manifest object, showing preference for local copy

get_remote_manifest()

Return manifest by reading remote manifest on backend

get_time()

Return time if full backup, or end_time if incremental

get_timestr()

Return time string suitable for log statements

is_complete()

Assume complete if found manifest file

set_files_changed()

set_info(*pr*)

Set BackupSet information from ParseResults object

@param pr: parse results @type pf: ParseResults

set_manifest(*remote_filename*)

Add local and remote manifest filenames to backup set

class duplicity.dup_collections.**BackupSetChangesStatus**(*backup_set*)

Bases: object

__init__(*backup_set*)

exception duplicity.dup_collections.**CollectionsError**

Bases: Exception

class duplicity.dup_collections.**CollectionsStatus**(*backend*, *archive_dir_path*, *action*)

Bases: object

Hold information about available chains and sets

__init__(*backend*, *archive_dir_path*, *action*)

Make new object. Does not set values

get_all_file_changed_records(*set_index*)

Returns file changes in the specific backup set

get_backup_chain_at_time(*time*)

Return backup chain covering specified time

Tries to find the backup chain covering the given time. If there is none, return the earliest chain before, and failing that, the earliest chain.

get_backup_chains(*filename_list*)

Split given filename_list into chains

Return value will be tuple (list of chains, list of sets, list of incomplete sets), where the list of sets will comprise sets not fitting into any chain, and the incomplete sets are sets missing files.

get_chains_older_than(*t*)

Returns a list of backup chains older than the given time t

All of the times will be associated with an intact chain. Furthermore, none of the times will be of a chain which a newer set may depend on. For instance, if set A is a full set older than t, and set B is an incremental based on A which is newer than t, then the time of set A will not be returned.

get_extraneous()

Return list of the names of extraneous duplicity files

A duplicity file is considered extraneous if it is recognizable as a duplicity file, but isn't part of some complete backup set, or current signature chain.

get_file_changed_record(*filepath*)

Returns time line of specified file changed

get_last_backup_chain()

Return the last full backup of the collection, or None if there is no full backup chain.

get_last_full_backup_time()

Return the time of the last full backup, or 0 if there is none.

get_nth_last_backup_chain(*n*)

Return the nth-to-last full backup of the collection, or None if there is less than n backup chains.

NOTE: n = 1 -> time of latest available chain (n = 0 is not a valid input). Thus the second-to-last is obtained with n=2 rather than n=1.

get_nth_last_full_backup_time(*n*)

Return the time of the nth to last full backup, or 0 if there is none.

get_older_than(*t*)

Returns a list of backup sets older than the given time t

All of the times will be associated with an intact chain. Furthermore, none of the times will be of a set which a newer set may depend on. For instance, if set A is a full set older than t, and set B is an incremental based on A which is newer than t, then the time of set A will not be returned.

get_older_than_required(*t*)

Returns list of old backup sets required by new sets

This function is similar to the previous one, but it only returns the times of sets which are old but part of the chains where the newer end of the chain is newer than t.

get_signature_chain_at_time(*time*)

Return signature chain covering specified time

Tries to find the signature chain covering the given time. If there is none, return the earliest chain before, and failing that, the earliest chain.

get_signature_chains(*local*, *filelist=None*)

Find chains in archive_dir_path (if local is true) or backend

Use filelist if given, otherwise regenerate. Return value is pair (list of chains, list of signature paths not in any chains).

get_signature_chains_older_than(*t*)

Returns a list of signature chains older than the given time t

All of the times will be associated with an intact chain. Furthermore, none of the times will be of a chain which a newer set may depend on. For instance, if set A is a full set older than t, and set B is an incremental based on A which is newer than t, then the time of set A will not be returned.

get_sorted_chains(*chain_list*)

Return chains sorted by end_time. If tie, local goes last

get_sorted_sets(*set_list*)

Sort set list by end time, return (sorted list, incomplete)

set_matched_chain_pair(*sig_chains*, *backup_chains*)

Set self.matched_chain_pair and self.other_sig/backup_chains

The latest matched_chain_pair will be set. If there are both remote and local signature chains capable of matching the latest backup chain, use the local sig chain (it does not need to be downloaded).

set_values(*sig_chain_warning=1*)

Set values from archive_dir_path and backend.

Returns self for convenience. If sig_chain_warning is set to None, do not warn about unnecessary sig chains. This is because there may naturally be some unecessary ones after a full backup.

sort_sets(*setlist*)

Return new list containing same elems of setlist, sorted by time

to_log_info()

Return summary of the collection, suitable for printing to log

warn(*sig_chain_warning*)

Log various error messages if find incomplete/orphaned files

class duplicity.dup_collections.**FileChangedStatus**(*filepath*, *fileinfo_list*)

Bases: object

__init__(*filepath*, *fileinfo_list*)

class duplicity.dup_collections.**SignatureChain**(*local*, *location*)

Bases: object

A number of linked SignatureSets

Analog to BackupChain - start with a full-sig, and continue with new-sigs.

__init__(*local*, *location*)

Return new SignatureChain.

local should be true iff the signature chain resides in config.archive_dir_path and false if the chain is in config.backend.

@param local: True if sig chain in config.archive_dir_path @type local: Boolean

@param location: Where the sig chain is located @type location: config.archive_dir_path or config.backend

add_filename(*filename*, *pr=None*)

Add new sig filename to current chain. Return true if fits

check_times(*time_list*)

Check to make sure times are in whole seconds

delete(*keep_full=False*)

Remove all files in signature set

get_filenames(*time=None*)

Return ordered list of filenames in set, up to a provided time

get_fileobjs(*time=None*)

Return ordered list of signature fileobjs opened for reading, optionally at a certain time

islocal()

Return true if represents a signature chain in archive_dir_path

1.1.2.8 duplicity.dup_main module

class duplicity.dup_main.**Restart**(*last_backup*)

Bases: object

Class to aid in restart of inc or full backup. Instance in config.restart if restart in progress.

__init__(*last_backup*)

checkManifest(*mf*)

setLastSaved(*mf*)

setParms(*last_backup*)

duplicity.dup_main.**check_last_manifest**(*col_stats*)

Check consistency and hostname/directory of last manifest

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**check_resources**(*action*)

Check for sufficient resources: - temp space for volume build - enough max open files Put out fatal error if not sufficient to run

@type action: string @param action: action in progress

@rtype: void @return: void

duplicity.dup_main.**check_sig_chain**(*col_stats*)

Get last signature chain for inc backup, or None if none available

@type col_stats: CollectionStatus object @param col_stats: collection status

duplicity.dup_main.**cleanup**(*col_stats*)

Delete the extraneous files in the current backend

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**do_backup**(*action*)

duplicity.dup_main.**dummy_backup**(*tarblock_iter*)

Fake writing to backend, but do go through all the source paths.

@type tarblock_iter: tarblock_iter @param tarblock_iter: iterator for current tar block

@rtype: int @return: constant 0 (zero)

duplicity.dup_main.**full_backup**(*col_stats*)

Do full backup of directory to backend, using archive_dir_path

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**get_man_fileobj**(*backup_type*)

Return a fileobj opened for writing, save results as manifest

Save manifest in config.archive_dir_path gzipped. Save them on the backend encrypted as needed.

@type man_type: string @param man_type: either "full" or "new"

@rtype: fileobj @return: fileobj opened for writing

duplicity.dup_main.**get_passphrase**(*n*, *action*, *for_signing=False*)

Check to make sure passphrase is indeed needed, then get the passphrase from environment, from gpg-agent, or user

If n=3, a password is requested and verified. If n=2, the current password is verified. If n=1, a password is requested without verification for the time being.

@type n: int @param n: verification level for a passphrase being requested @type action: string @param action: action to perform @type for_signing: boolean @param for_signing: true if the passphrase is for a signing key, false if not @rtype: string @return: passphrase

duplicity.dup_main.**get_sig_fileobj**(*sig_type*)

Return a fileobj opened for writing, save results as signature

Save signatures in config.archive_dir gzipped. Save them on the backend encrypted as needed.

@type sig_type: string @param sig_type: either "full-sig" or "new-sig"

@rtype: fileobj @return: fileobj opened for writing

duplicity.dup_main.**getpass_safe**(*message*)

duplicity.dup_main.**incremental_backup**(*sig_chain*)

Do incremental backup of directory to backend, using archive_dir_path

@rtype: void @return: void

duplicity.dup_main.**list_current**(*col_stats*)

List the files current in the archive (examining signature only)

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**log_startup_parms**(*verbosity=5*)

log Python, duplicity, and system versions

duplicity.dup_main.**main**()

Start/end here

duplicity.dup_main.**print_statistics**(*stats*, *bytes_written*)

If config.print_statistics, print stats after adding bytes_written

@rtype: void @return: void

duplicity.dup_main.**remove_all_but_n_full**(*col_stats*)

Remove backup files older than the last n full backups.

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**remove_old**(*col_stats*)

Remove backup files older than config.remove_time from backend

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**replicate**()

Replicate backup files from one remote to another, possibly encrypting or adding parity.

@rtype: void @return: void

duplicity.dup_main.**restart_position_iterator**(*tarblock_iter*)

Fake writing to backend, but do go through all the source paths. Stop when we have processed the last file and block from the last backup. Normal backup will proceed at the start of the next volume in the set.

@type tarblock_iter: tarblock_iter @param tarblock_iter: iterator for current tar block

@rtype: int @return: constant 0 (zero)

duplicity.dup_main.**restore**(*col_stats*)

Restore archive in config.backend to config.local_path

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**restore_add_sig_check**(*fileobj*)

Require signature when closing fileobj matches sig in gpg_profile

@rtype: void @return: void

duplicity.dup_main.**restore_check_hash**(*volume_info*, *vol_path*)

Check the hash of vol_path path against data in volume_info

@rtype: boolean @return: true (verified) / false (failed)

duplicity.dup_main.**restore_get_enc_fileobj**(*backend*, *filename*, *volume_info*)

Return plaintext fileobj from encrypted filename on backend

If volume_info is set, the hash of the file will be checked, assuming some hash is available. Also, if config.sign_key is set, a fatal error will be raised if file not signed by sign_key.

with –ignore-errors set continue on hash mismatch

duplicity.dup_main.**restore_get_patched_rop_iter**(*col_stats*)

Return iterator of patched ROPaths of desired restore data

@type col_stats: CollectionStatus object @param col_stats: collection status

duplicity.dup_main.**sync_archive**(*col_stats*)

Synchronize local archive manifest file and sig chains to remote archives. Copy missing files from remote to local as needed to make sure the local archive is synchronized to remote storage.

@rtype: void @return: void

duplicity.dup_main.**verify**(*col_stats*)

Verify files, logging differences

@type col_stats: CollectionStatus object @param col_stats: collection status

@rtype: void @return: void

duplicity.dup_main.**write_multivol**(*backup_type*, *tarblock_iter*, *man_outfp*, *sig_outfp*, *backend*)

Encrypt volumes of tarblock_iter and write to backend

backup_type should be "inc" or "full" and only matters here when picking the filenames. The path_prefix will determine the names of the files written to backend. Also writes manifest file. Returns number of bytes written.

@type backup_type: string @param backup_type: type of backup to perform, either 'inc' or 'full' @type tarblock_iter: tarblock_iter @param tarblock_iter: iterator for current tar block @type backend: callable backend object @param backend: I/O backend for selected protocol

@rtype: int @return: bytes written

1.1.2.9 duplicity.dup_temp module

Manage temporary files

class duplicity.dup_temp.**Block**(*data*)

Bases: object

Data block to return from SrcIter

__init__(*data*)

class duplicity.dup_temp.**FileobjHooked**(*fileobj*, *tdp=None*, *dirpath=None*, *partname=None*, *permname=None*, *remname=None*)

Bases: object

Simulate a file, but add hook on close

__init__(*fileobj*, *tdp=None*, *dirpath=None*, *partname=None*, *permname=None*, *remname=None*)

Initializer. fileobj is the file object to simulate

addhook(*hook*)

Add hook (function taking no arguments) to run upon closing

close()

Close fileobj, running hooks right afterwards

flush()

Flush fileobj and force sync.

get_name()

Return the name of the file

property name

Return the name of the file

read(*length=-1*)

Read fileobj, return result of read()

seek(*offset*)

Seeks to a location of fileobj

tell()

Returns current location of fileobj

to_final()

We are finished, rename to final, gzip if needed.

to_partial()

We have achieved the first checkpoint, make file visible and permanent.

to_remote()

We have written the last checkpoint, now encrypt or compress and send a copy of it to the remote for final storage.

write(*buf*)

Write fileobj, return result of write()

class duplicity.dup_temp.**SrcIter**(*src*)

Bases: object

Iterate over source and return Block of data.

__init__(*src*)

get_footer()

get_read_size()

class duplicity.dup_temp.**TempDupPath**(*base*, *index=()*, *parseresults=None*)

Bases: *DupPath*

Like TempPath, but build around DupPath

delete()

Forget and delete

filtered_open_with_delete(*mode*)

Returns a filtered fileobj. When that is closed, delete file

open_with_delete(*mode='rb'*)

Returns a fileobj. When that is closed, delete file

class duplicity.dup_temp.**TempPath**(*base*, *index=()*)

Bases: *Path*

Path object used as a temporary file

delete()

Forget and delete

open_with_delete(*mode*)

Returns a fileobj. When that is closed, delete file

duplicity.dup_temp.**get_fileobj_duppath**(*dirpath*, *partname*, *permname*, *remname*, *overwrite=False*)

Return a file object open for writing, will write to filename

Data will be processed and written to a temporary file. When the return fileobject is closed, rename to final position. filename must be a recognizable duplicity data file.

duplicity.dup_temp.**new_tempduppath**(*parseresults*)

Return a new TempDupPath, using settings from parseresults

duplicity.dup_temp.**new_temppath**()

Return a new TempPath

1.1.2.10 duplicity.dup_threading module

Duplicity specific but otherwise generic threading interfaces and utilities.

(Not called "threading" because we do not want to conflict with the standard threading module, and absolute imports require at least python 2.5.)

class duplicity.dup_threading.**Value**(*value=None*)

Bases: object

A thread-safe container of a reference to an object (but not the object itself).

In particular this means it is safe to:

value.set(1)

But unsafe to:

value.get()['key'] = value

Where the latter must be done using something like:

def _setprop():
    value.get()['key'] = value

with_lock(value, _setprop)

Operations such as increments are best done as:

value.transform(lambda val: val + 1)

__init__(*value=None*)

Initialuze with the given value.

acquire()

Acquire this Value for mutually exclusive access. Only ever needed when calling code must perform operations that cannot be done with get(), set() or transform().

get()

Returns the value protected by this Value.

release()

Release this Value for mutually exclusive access.

set(*value*)

Resets the value protected by this Value.

transform(*fn*)

Call fn with the current value as the parameter, and reset the value to the return value of fn.

During the execution of fn, all other access to this Value is prevented.

If fn raised an exception, the value is not reset.

Returns the value returned by fn, or raises the exception raised by fn.

duplicity.dup_threading.**async_split**(*fn*)

Splits the act of calling the given function into one front-end part for waiting on the result, and a back-end part for performing the work in another thread.

Returns (waiter, caller) where waiter is a function to be called in order to wait for the results of an asynchronous invokation of fn to complete, returning fn's result or propagating it's exception.

Caller is the function to call in a background thread in order to execute fn asynchronously. Caller will return (success, waiter) where success is a boolean indicating whether the function suceeded (did NOT raise an exception), and waiter is the waiter that was originally returned by the call to async_split().

duplicity.dup_threading.**interruptably_wait**(*cv*, *waitFor*)

cv - The threading.Condition instance to wait on test - Callable returning a boolean to indicate whether

the criteria being waited on has been satisfied.

Perform a wait on a condition such that it is keyboard interruptable when done in the main thread. Due to Python limitations as of <= 2.5, lock acquisition and conditions waits are not interruptable when performed in the main thread.

Currently, this comes at a cost additional CPU use, compared to a normal wait. Future implementations may be more efficient if the underlying python supports it.

The condition must be acquired.

This function should only be used on conditions that are never expected to be acquired for extended periods of time, or the lock-acquire of the underlying condition could cause an uninterruptable state despite the efforts of this function.

There is no equivalent for acquireing a lock, as that cannot be done efficiently.

Example:

Instead of:

cv.acquire() while not thing_done:

cv.wait(someTimeout)

cv.release()

do:

cv.acquire() interruptable_condwait(cv, lambda: thing_done) cv.release()

duplicity.dup_threading.**require_threading**(*reason=None*)

Assert that threading is required for operation to continue. Raise an appropriate exception if this is not the case.

Reason specifies an optional reason why threading is required, which will be used for error reporting in case threading is not supported.

duplicity.dup_threading.**thread_module**()

Returns the thread module, or dummy_thread if threading is not supported.

duplicity.dup_threading.**threading_module**()

Returns the threading module, or dummy_thread if threading is not supported.

duplicity.dup_threading.**threading_supported**()

Returns whether threading is supported on the system we are running on.

duplicity.dup_threading.**with_lock**(*lock*, *fn*)

Call fn with lock acquired. Guarantee that lock is released upon the return of fn.

Returns the value returned by fn, or raises the exception raised by fn.

(Lock can actually be anything responding to acquire() and release().)

1.1.2.11 duplicity.dup_time module

Provide time related exceptions and functions

exception duplicity.dup_time.**TimeException**

Bases: Exception

duplicity.dup_time.**cmp**(*time1*, *time2*)

Compare time1 and time2 and return -1, 0, or 1

duplicity.dup_time.**genstrtotime**(*timestr*, *override_curtime=None*)

Convert a generic time string to a time in seconds

duplicity.dup_time.**gettzd**(*dstflag*)

Return w3's timezone identification string.

Expresed as [+/-]hh:mm. For instance, PST is -08:00. Zone is coincides with what localtime(), etc., use.

duplicity.dup_time.**intstringtoseconds**(*interval_string*)

Convert a string expressing an interval (e.g. "4D2s") to seconds

duplicity.dup_time.**inttopretty**(*seconds*)

Convert num of seconds to readable string like "2 hours".

duplicity.dup_time.**setcurtime**(*time_in_secs=None*)

Sets the current time in curtime and curtimestr

duplicity.dup_time.**setprevtime**(*time_in_secs*)

Sets the previous time in prevtime and prevtimestr

duplicity.dup_time.**stringtopretty**(*timestring*)

Return pretty version of time given w3 time string

duplicity.dup_time.**stringtotime**(*timestring*)

Return time in seconds from w3 or duplicity timestring

If there is an error parsing the string, or it doesn't look like a valid datetime string, return None.

duplicity.dup_time.**timetopretty**(*timeinseconds*)

Return pretty version of time

duplicity.dup_time.**timetostring**(*timeinseconds*)

Return w3 or duplicity datetime compliant listing of timeinseconds

duplicity.dup_time.**tzdtoseconds**(*tzd*)

Given w3 compliant TZD, return how far ahead UTC is

1.1.2.12 duplicity.errors module

Error/exception classes that do not fit naturally anywhere else.

exception duplicity.errors.**BackendException**(*msg*, *code=50*)

Bases: *DuplicityError*

Raised to indicate a backend specific problem.

__init__(*msg*, *code=50*)

exception duplicity.errors.**BadVolumeException**

Bases: *DuplicityError*

exception duplicity.errors.**ConflictingScheme**

Bases: *DuplicityError*

Raised to indicate an attempt was made to register a backend for a scheme for which there is already a backend registered.

exception duplicity.errors.**DuplicityError**

Bases: Exception

exception duplicity.errors.**FatalBackendException**(*msg*, *code=50*)

Bases: *BackendException*

Raised to indicate a backend failed fatally.

exception duplicity.errors.**InvalidBackendURL**

Bases: *UserError*

Raised to indicate a URL was not a valid backend URL.

exception duplicity.errors.**NotSupported**

Bases: *DuplicityError*

Exception raised when an action cannot be completed because some particular feature is not supported by the environment.

exception duplicity.errors.**TemporaryLoadException**(*msg*, *code=50*)

Bases: *BackendException*

Raised to indicate a temporary issue on the backend. Duplicity should back off for a bit and try again.

exception duplicity.errors.**UnsupportedBackendScheme**(*url*)

Bases: *InvalidBackendURL*, *UserError*

Raised to indicate that a backend URL was parsed successfully as a URL, but was not supported.

__init__(*url*)

exception duplicity.errors.**UserError**

Bases: *DuplicityError*

Subclasses use this in their inheritance hierarchy to signal that the error is a user generated one, and that it is therefore typically unsuitable to display a full stack trace.

1.1.2.13 duplicity.file_naming module

Produce and parse the names of duplicity's backup files

class duplicity.file_naming.**ParseResults**(*type*, *manifest=None*, *volume_number=None*, *time=None*, *start_time=None*, *end_time=None*, *encrypted=None*, *compressed=None*, *partial=False*)

Bases: object

Hold information taken from a duplicity filename

__init__(*type*, *manifest=None*, *volume_number=None*, *time=None*, *start_time=None*, *end_time=None*, *encrypted=None*, *compressed=None*, *partial=False*)

duplicity.file_naming.**from_base36**(*s*)

Convert string s in base 36 to long int

duplicity.file_naming.**get**(*type*, *volume_number=None*, *manifest=False*, *encrypted=False*, *gzipped=False*, *partial=False*)

Return duplicity filename of specified type

type can be "full", "inc", "full-sig", or "new-sig". volume_number can be given with the full and inc types. If manifest is true the filename is of a full or inc manifest file.

duplicity.file_naming.**get_suffix**(*encrypted*, *gzipped*)

Return appropriate suffix depending on status of encryption, compression, and short_filenames.

duplicity.file_naming.**parse**(*filename*)

Parse duplicity filename, return None or ParseResults object

duplicity.file_naming.**prepare_regex**(*force=False*)

duplicity.file_naming.**to_base36**(*n*)

Return string representation of n in base 36 (use 0-9 and a-z)

1.1.2.14 duplicity.filechunkio module

class duplicity.filechunkio.**FileChunkIO**(*name*, *mode='r'*, *closefd=True*, *offset=0*, *bytes=None*, **args*, ***kwargs*)

Bases: FileIO

A class that allows you reading only a chunk of a file.

__init__(*name*, *mode='r'*, *closefd=True*, *offset=0*, *bytes=None*, **args*, ***kwargs*)

Open a file chunk. The mode can only be 'r' for reading. Offset is the amount of bytes that the chunks starts after the real file's first byte. Bytes defines the amount of bytes the chunk has, which you can set to None to include the last byte of the real file.

read(*n=-1*)

Read and return at most n bytes.

readall()

Read all data from the chunk.

readinto(*b*)

Same as RawIOBase.readinto().

seek(*offset*, *whence=0*)

Move to a new chunk position.

tell()

Current file position.

1.1.2.15 duplicity.globmatch module

exception duplicity.globmatch.**FilePrefixError**

Bases: *GlobbingError*

Signals that a specified file doesn't start with correct prefix

exception duplicity.globmatch.**GlobbingError**

Bases: Exception

Something has gone wrong when parsing a glob string

duplicity.globmatch.**_glob_get_prefix_regexs**(*glob_str*)

Return list of regexps equivalent to prefixes of glob_str

duplicity.globmatch.**glob_to_regex**(*pat*)

Returned regular expression equivalent to shell glob pat

Currently only the ?, , *[], and* * expressions are supported. Ranges like [a-z] are currently unsupported. There is no way to quote these special characters.

This function taken with minor modifications from efnmatch.py by Donovan Baarda.

duplicity.globmatch.**select_fn_from_glob**(*glob_str*, *include*, *ignore_case=False*)

Return a function test_fn(path) which tests whether path matches glob, as per the Unix shell rules, taking as arguments a path, a glob string and include (0 indicating that the glob string is an exclude glob and 1 indicating that it is an include glob, returning:

0 - if the file should be excluded 1 - if the file should be included 2 - if the folder should be scanned for any included/excluded files None - if the selection function has nothing to say about the file

The basic idea is to turn glob_str into a regular expression, and just use the normal regular expression. There is a complication because the selection function should return '2' (scan) for directories which may contain a file which matches the glob_str. So we break up the glob string into parts, and any file which matches an initial sequence of glob parts gets scanned.

Thanks to Donovan Baarda who provided some code which did some things similar to this.

Note: including a folder implicitly includes everything within it.

1.1.2.16 duplicity.gpg module

duplicity's gpg interface, builds upon Frank Tobin's GnuPGInterface which is now patched with some code for iterative threaded execution see duplicity's README for details

exception duplicity.gpg.**GPGError**

Bases: Exception

Indicate some GPG Error

class duplicity.gpg.**GPGFile**(*encrypt*, *encrypt_path*, *profile*)

Bases: object

File-like object that encrypts decrypts another file on the fly

__init__(*encrypt*, *encrypt_path*, *profile*)

GPGFile initializer

If recipients is set, use public key encryption and encrypt to the given keys. Otherwise, use symmetric encryption.

encrypt_path is the Path of the gpg encrypted file. Right now only symmetric encryption/decryption is supported.

If passphrase is false, do not set passphrase - GPG program should prompt for it.

close()

get_signature()

Return keyID of signature, or None if none

gpg_failed()

read(*length=-1*)

seek(*offset*)

set_signature()

Set self.signature to signature keyID

This only applies to decrypted files. If the file was not signed, set self.signature to None.

tell()

write(*buf*)

class duplicity.gpg.**GPGProfile**(*passphrase=None*, *sign_key=None*, *recipients=None*, *hidden_recipients=None*)

Bases: object

Just hold some GPG settings, avoid passing tons of arguments

__init__(*passphrase=None*, *sign_key=None*, *recipients=None*, *hidden_recipients=None*)

Set all data with initializer

passphrase is the passphrase. If it is None (not ""), assume it hasn't been set. sign_key can be blank if no signing is indicated, and recipients should be a list of keys. For all keys, the format should be an hex key like 'AA0E73D2'.

_version_re = re.compile(b'^gpg.*\\(GnuPG(?:/MacGPG2)?\\)
(?P<maj>[0-9]+)\\.(?P<min>[0-9]+)\\.(?P<bug>[0-9]+)(-.+)?$')

get_gpg_version(*binary*)

rc(*flags=0*)

Compile a regular expression pattern, returning a Pattern object.

duplicity.gpg.**GPGWriteFile**(*block_iter*, *filename*, *profile*, *size=209715200*, *max_footer_size=16384*)

Write GPG compressed file of given size

This function writes a gpg compressed file by reading from the input iter and writing to filename. When it has read an amount close to the size limit, it "tops off" the incoming data with incompressible data, to try to hit the limit exactly.

block_iter should have methods .next(size), which returns the next block of data, which should be at most size bytes long. Also .get_footer() returns a string to write at the end of the input file. The footer should have max length max_footer_size.

Because gpg uses compression, we don't assume that putting bytes_in bytes into gpg will result in bytes_out = bytes_in out. However, do assume that bytes_out <= bytes_in approximately.

Returns true if succeeded in writing until end of block_iter.

duplicity.gpg.**GzipWriteFile**(*block_iter*, *filename*, *size=209715200*, *gzipped=True*)

Write gzipped compressed file of given size

This is like the earlier GPGWriteFile except it writes a gzipped file instead of a gpg'd file. This function is somewhat out of place, because it doesn't deal with GPG at all, but it is very similar to GPGWriteFile so they might as well be defined together.

The input requirements on block_iter and the output is the same as GPGWriteFile (returns true if wrote until end of block_iter).

duplicity.gpg.**PlainWriteFile**(*block_iter*, *filename*, *size=209715200*, *gzipped=False*)

Write plain uncompressed file of given size

This is like the earlier GPGWriteFile except it writes a gzipped file instead of a gpg'd file. This function is somewhat out of place, because it doesn't deal with GPG at all, but it is very similar to GPGWriteFile so they might as well be defined together.

The input requirements on block_iter and the output is the same as GPGWriteFile (returns true if wrote until end of block_iter).

duplicity.gpg.**get_hash**(*hash*, *path*, *hex=1*)

Return hash of path

hash should be "MD5" or "SHA1". The output will be in hexadecimal form if hex is true, and in text (base64) otherwise.

1.1.2.17 duplicity.gpginterface module

Interface to GNU Privacy Guard (GnuPG)

!!! This was renamed to gpginterface.py.
Please refer to duplicity's README for the reason. !!!

gpginterface is a Python module to interface with GnuPG which based on GnuPGInterface by Frank J. Tobin. It concentrates on interacting with GnuPG via filehandles, providing access to control GnuPG via versatile and extensible means.

This module is based on GnuPG::Interface, a Perl module by the same author.

Normally, using this module will involve creating a GnuPG object, setting some options in it's 'options' data member (which is of type Options), creating some pipes to talk with GnuPG, and then calling the run() method, which will connect those pipes to the GnuPG process. run() returns a Process object, which contains the filehandles to talk to GnuPG with.

Example code:

```
>>> import gpginterface
>>>
>>> plaintext  = b"Three blind mice"
>>> passphrase = "This is the passphrase"
>>>
>>> gnupg = gpginterface.GnuPG()
>>> gnupg.options.armor = 1
>>> gnupg.options.meta_interactive = 0
>>> gnupg.options.extra_args.append('--no-secmem-warning')
>>>
>>> # Normally we might specify something in
>>> # gnupg.options.recipients, like
>>> # gnupg.options.recipients = [ '0xABCD1234', 'bob@foo.bar' ]
>>> # but since we're doing symmetric-only encryption, it's not needed.
>>> # If you are doing standard, public-key encryption, using
>>> # --encrypt, you will need to specify recipients before
>>> # calling gnupg.run()
>>>
>>> # First we'll encrypt the test_text input symmetrically
>>> p1 = gnupg.run(['--symmetric'],
...                create_fhs=['stdin', 'stdout', 'passphrase'])
>>>
>>> ret = p1.handles['passphrase'].write(passphrase)
>>> p1.handles['passphrase'].close()
>>>
>>> ret = p1.handles['stdin'].write(plaintext)
>>> p1.handles['stdin'].close()
>>>
>>> ciphertext = p1.handles['stdout'].read()
>>> p1.handles['stdout'].close()
>>>
>>> # process cleanup
>>> p1.wait()
>>>
>>> # Now we'll decrypt what we just encrypted it,
>>> # using the convience method to get the
>>> # passphrase to GnuPG
>>> gnupg.passphrase = passphrase
>>>
>>> p2 = gnupg.run(['--decrypt'], create_fhs=['stdin', 'stdout'])
>>>
>>> ret = p2.handles['stdin'].write(ciphertext)
>>> p2.handles['stdin'].close()
>>>
>>> decrypted_plaintext = p2.handles['stdout'].read()
>>> p2.handles['stdout'].close()
>>>
>>> # process cleanup
>>> p2.wait()
>>>
>>> # Our decrypted plaintext:
>>> decrypted_plaintext
b'Three blind mice'
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # ...and see it's the same as what we orignally encrypted
>>> assert decrypted_plaintext == plaintext,           "GnuPG decrypted output does not␣
↪match original input"
>>>
>>>
>>> ##################################################
>>> # Now let's trying using run()'s attach_fhs parameter
>>>
>>> # we're assuming we're running on a unix...
>>> infp = open('/etc/manpaths', 'rb')
>>>
>>> p1 = gnupg.run(['--symmetric'], create_fhs=['stdout'],
...                                 attach_fhs={'stdin': infp})
>>>
>>> # GnuPG will read the stdin from /etc/motd
>>> ciphertext = p1.handles['stdout'].read()
>>>
>>> # process cleanup
>>> p1.wait()
>>>
>>> # Now let's run the output through GnuPG
>>> # We'll write the output to a temporary file,
>>> import tempfile
>>> temp = tempfile.TemporaryFile()
>>>
>>> p2 = gnupg.run(['--decrypt'], create_fhs=['stdin'],
...                               attach_fhs={'stdout': temp})
>>>
>>> # give GnuPG our encrypted stuff from the first run
>>> ret = p2.handles['stdin'].write(ciphertext)
>>> p2.handles['stdin'].close()
>>>
>>> # process cleanup
>>> p2.wait()
>>>
>>> # rewind the tempfile and see what GnuPG gave us
>>> ret = temp.seek(0)
>>> decrypted_plaintext = temp.read()
>>>
>>> # compare what GnuPG decrypted with our original input
>>> ret = infp.seek(0)
>>> input_data = infp.read()
>>> assert decrypted_plaintext == input_data,           "GnuPG decrypted output does␣
↪not match original input"
```

To do things like public-key encryption, simply pass do something like:

gnupg.passphrase = 'My passphrase' gnupg.options.recipients = ['bob@foobar.com'] gnupg.run(['–sign', '–encrypt'], create_fhs=..., attach_fhs=...)

Here is an example of subclassing gpginterface.GnuPG, so that it has an encrypt_string() method that returns ciphertext.

```
>>> import gpginterface
>>>
>>> class MyGnuPG(gpginterface.GnuPG):
...
...     def __init__(self):
...         super().__init__()
...         self.setup_my_options()
...
...     def setup_my_options(self):
...         self.options.armor = 1
...         self.options.meta_interactive = 0
...         self.options.extra_args.append('--no-secmem-warning')
...
...     def encrypt_string(self, string, recipients):
...        gnupg.options.recipients = recipients   # a list!
...
...        proc = gnupg.run(['--encrypt'], create_fhs=['stdin', 'stdout'])
...
...        proc.handles['stdin'].write(string)
...        proc.handles['stdin'].close()
...
...        output = proc.handles['stdout'].read()
...        proc.handles['stdout'].close()
...
...        proc.wait()
...        return output
...
>>> gnupg = MyGnuPG()
>>> ciphertext = gnupg.encrypt_string(b"The secret", ['E477C232'])
>>>
>>> # just a small sanity test here for doctest
>>> import types
>>> assert isinstance(ciphertext, bytes),            "What GnuPG gave back is not bytes!"
```

Here is an example of generating a key: >>> import gpginterface >>> gnupg = gpginterface.GnuPG() >>> gnupg.options.meta_interactive = 0 >>> >>> # We will be creative and use the logger filehandle to capture >>> # what GnuPG says this time, instead stderr; no stdout to listen to, >>> # but we capture logger to surpress the dry-run command. >>> # We also have to capture stdout since otherwise doctest complains; >>> # Normally you can let stdout through when generating a key. >>> >>> proc = gnupg.run([’–gen-key’], create_fhs=[‘stdin’, ‘stdout’, … ‘logger’]) >>> >>> ret = proc.handles[‘stdin’].write(b”’Key-Type: DSA … Key-Length: 1024 … # We are only testing syntax this time, so dry-run … %dry-run … Subkey-Type: ELG-E … Subkey-Length: 1024 … Name-Real: Joe Tester … Name-Comment: with stupid passphrase … Name-Email: joe@foo.bar … Expire-Date: 2y … Passphrase: abc … %pubring foo.pub … %secring foo.sec … ‘”) >>> >>> proc.handles[‘stdin’].close() >>> >>> report = proc.handles[‘logger’].read() >>> proc.handles[‘logger’].close() >>> >>> proc.wait()

COPYRIGHT:

Copyright (C) 2001 Frank J. Tobin, ftobin@neverending.org

LICENSE:

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the

implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA or see http://www.gnu.org/copyleft/lesser.html

class duplicity.gpginterface.**GnuPG**

Bases: object

Class instances represent GnuPG.

Instance attributes of a GnuPG object are:

- call – string to call GnuPG with. Defaults to "gpg"
- passphrase – Since it is a common operation to pass in a passphrase to GnuPG, and working with the passphrase filehandle mechanism directly can be mundane, if set, the passphrase attribute works in a special manner. If the passphrase attribute is set, and no passphrase file object is sent in to run(), then GnuPG instnace will take care of sending the passphrase to GnuPG, the executable instead of having the user sent it in manually.
- options – Object of type gpginterface.Options. Attribute-setting in options determines the command-line options used when calling GnuPG.

__init__()

_as_child(*process*, *gnupg_commands*, *args*)

Stuff run after forking in child

_as_parent(*process*)

Stuff run after forking in parent

_attach_fork_exec(*gnupg_commands*, *args*, *create_fhs*, *attach_fhs*)

This is like run(), but without the passphrase-helping (note that run() calls this).

run(*gnupg_commands*, *args=None*, *create_fhs=None*, *attach_fhs=None*)

Calls GnuPG with the list of string commands gnupg_commands, complete with prefixing dashes. For example, gnupg_commands could be '["–sign", "–encrypt"]' Returns a gpginterface.Process object.

args is an optional list of GnuPG command arguments (not options), such as keyID's to export, filenames to process, etc.

create_fhs is an optional list of GnuPG filehandle names that will be set as keys of the returned Process object's 'handles' attribute. The generated filehandles can be used to communicate with GnuPG via standard input, standard output, the status-fd, passphrase-fd, etc.

Valid GnuPG filehandle names are:

- stdin
- stdout
- stderr
- status
- passphase
- command
- logger

The purpose of each filehandle is described in the GnuPG documentation.

attach_fhs is an optional dictionary with GnuPG filehandle names mapping to opened files. GnuPG will read or write to the file accordingly. For example, if 'my_file' is an opened file and 'attach_fhs[stdin] is my_file', then GnuPG will read its standard input from my_file. This is useful if you want GnuPG to read/write to/from an existing file. For instance:

f = open("encrypted.gpg") gnupg.run(["–decrypt"], attach_fhs={'stdin': f})

Using attach_fhs also helps avoid system buffering issues that can arise when using create_fhs, which can cause the process to deadlock.

If not mentioned in create_fhs or attach_fhs, GnuPG filehandles which are a std* (stdin, stdout, stderr) are defaulted to the running process' version of handle. Otherwise, that type of handle is simply not used when calling GnuPG. For example, if you do not care about getting data from GnuPG's status filehandle, simply do not specify it.

run() returns a Process() object which has a 'handles' which is a dictionary mapping from the handle name (such as 'stdin' or 'stdout') to the respective newly-created FileObject connected to the running GnuPG process. For instance, if the call was

process = gnupg.run(["–decrypt"], stdin=1)

after run returns 'process.handles["stdin"]' is a FileObject connected to GnuPG's standard input, and can be written to.

class duplicity.gpginterface.**Options**

Bases: object

Objects of this class encompass options passed to GnuPG. This class is responsible for determining command-line arguments which are based on options. It can be said that a GnuPG object has-a Options object in its options attribute.

Attributes which correlate directly to GnuPG options:

Each option here defaults to false or None, and is described in GnuPG documentation.

Booleans (set these attributes to booleans)

- armor
- no_greeting
- no_verbose
- quiet
- batch
- always_trust
- rfc1991
- openpgp
- force_v3_sigs
- no_options
- textmode

Strings (set these attributes to strings)

- homedir
- default_key

- comment
- compress_algo
- options

Lists (set these attributes to lists)

- recipients (***NOTE*** plural of 'recipient')
- encrypt_to

Meta options

Meta options are options provided by this module that do not correlate directly to any GnuPG option by name, but are rather bundle of options used to accomplish a specific goal, such as obtaining compatibility with PGP 5. The actual arguments each of these reflects may change with time. Each defaults to false unless otherwise specified.

meta_pgp_5_compatible – If true, arguments are generated to try to be compatible with PGP 5.x.

meta_pgp_2_compatible – If true, arguments are generated to try to be compatible with PGP 2.x.

meta_interactive – If false, arguments are generated to try to help the using program use GnuPG in a non-interactive environment, such as CGI scripts. Default is true.

extra_args – Extra option arguments may be passed in via the attribute extra_args, a list.

```
>>> import gpginterface
>>>
>>> gnupg = gpginterface.GnuPG()
>>> gnupg.options.armor = 1
>>> gnupg.options.recipients = ['Alice', 'Bob']
>>> gnupg.options.extra_args = ['--no-secmem-warning']
>>>
>>> # no need for users to call this normally; just for show here
>>> gnupg.options.get_args()
['--armor', '--recipient', 'Alice', '--recipient', 'Bob', '--no-secmem-warning']
```

__init__()

get_args()

Generate a list of GnuPG arguments based upon attributes.

get_meta_args()

Get a list of generated meta-arguments

get_standard_args()

Generate a list of standard, non-meta or extra arguments

class duplicity.gpginterface.**Pipe**(*parent*, *child*, *direct*)

Bases: object

simple struct holding stuff about pipes we use

__init__(*parent*, *child*, *direct*)

class duplicity.gpginterface.**Process**

Bases: object

Objects of this class encompass properties of a GnuPG process spawned by GnuPG.run().

gnupg is a GnuPG object process = gnupg.run(['–decrypt'], stdout = 1) out = process.handles['stdout'].read() ... os.waitpid(process.pid, 0)

Data Attributes

handles – This is a map of filehandle-names to the file handles, if any, that were requested via run() and hence are connected to the running GnuPG process. Valid names of this map are only those handles that were requested.

pid – The PID of the spawned GnuPG process. Useful to know, since once should call os.waitpid() to clean up the process, especially if multiple calls are made to run().

__init__()

wait()

Wait on threaded_waitpid to exit and examine results. Will raise an IOError if the process exits non-zero.

duplicity.gpginterface.**threaded_waitpid**(*process*)

When started as a thread with the Process object, thread will execute an immediate waitpid() against the process pid and will collect the process termination info. This will allow us to reap child processes as soon as possible, thus freeing resources quickly.

1.1.2.18 duplicity.lazy module

Define some lazy data structures and functions acting on them

class duplicity.lazy.**ITRBranch**

Bases: object

Helper class for IterTreeReducer above

There are five stub functions below: start_process, end_process, branch_process, fast_process, and can_fast_process. A class that subclasses this one will probably fill in these functions to do more.

base_index = None

branch_process(*branch*)

Process a branch right after it is finished (stub)

call_end_proc()

Runs the end_process on self, checking for errors

can_fast_process(**args*)

True if object can be processed without new branch (stub)

caught_exception = None

end_process()

Do any final processing before leaving branch (stub)

fast_process(**args*)

Process args without new child branch (stub)

finished = None

index = None

log_prev_error(*index*)

Call function if no pending exception

on_error(*exc*, **args*)

This is run on any exception in start/end-process

start_process(**args*)

Do some initial processing (stub)

start_successful = None

class duplicity.lazy.**Iter**

Bases: object

Hold static methods for the manipulation of lazy iterators

static And(*iter*)

True if all elements in iterator are true. Short circuiting

static Or(*iter*)

True if any element in iterator is true. Short circuiting

static cat(**iters*)

Lazily concatenate iterators

static cat2(*iter_of_iters*)

Lazily concatenate iterators, iterated by big iterator

static empty(*iter*)

True if iterator has length 0

static equal(*iter1*, *iter2*, *verbose=None*, *operator=<function Iter.<lambda>>*)

True if iterator 1 has same elements as iterator 2

Use equality operator, or == if it is unspecified.

static filter(*predicate*, *iterator*)

Like filter in a lazy functional programming language

static foldl(*f*, *default*, *iter*)

the fundamental list iteration operator..

static foldr(*f*, *default*, *iter*)

foldr the "fundamental list recursion operator"?

static foreach(*function*, *iterator*)

Run function on each element in iterator

static len(*iter*)

Return length of iterator

static map(*function*, *iterator*)

Like map in a lazy functional programming language

static multiplex(*iter*, *num_of_forks*, *final_func=None*, *closing_func=None*)

Split a single iterater into a number of streams

The return val will be a list with length num_of_forks, each of which will be an iterator like iter. final_func is the function that will be called on each element in iter just as it is being removed from the buffer. closing_func is called when all the streams are finished.

class duplicity.lazy.**IterMultiplex2**(*iter*)

Bases: object

Multiplex an iterator into 2 parts

This is a special optimized case of the Iter.multiplex function, used when there is no closing_func or final_func, and we only want to split it into 2. By profiling, this is a time sensitive class.

__init__(*iter*)

yielda()

Return first iterator

yieldb()

Return second iterator

class duplicity.lazy.**IterTreeReducer**(*branch_class*, *branch_args*)

Bases: object

Tree style reducer object for iterator - stolen from rdiff-backup

The indicies of a RORPIter form a tree type structure. This class can be used on each element of an iter in sequence and the result will be as if the corresponding tree was reduced. This tries to bridge the gap between the tree nature of directories, and the iterator nature of the connection between hosts and the temporal order in which the files are processed.

This will usually be used by subclassing ITRBranch below and then call the initializer below with the new class.

Finish()

Call at end of sequence to tie everything up

__init__(*branch_class*, *branch_args*)

ITR initializer

add_branch()

Return branch of type self.branch_class, add to branch list

finish_branches(*index*)

Run Finish() on all branches index has passed

When we pass out of a branch, delete it and process it with the parent. The innermost branches will be the last in the list. Return None if we are out of the entire tree, and 1 otherwise.

process_w_branch(*index*, *branch*, *args*)

Run start_process on latest branch

1.1.2.19 duplicity.librsync module

Provides a high-level interface to some librsync functions

This is a python wrapper around the lower-level _librsync module, which is written in C. The goal was to use C as little as possible...

class duplicity.librsync.**DeltaFile**(*signature*, *new_file*)

Bases: *LikeFile*

File-like object which incrementally generates a librsync delta

__init__(*signature*, *new_file*)

DeltaFile initializer - call with signature and new file

Signature can either be a string or a file with read() and close() methods. New_file also only needs to have read() and close() methods. It will be closed when self is closed.

class duplicity.librsync.**LikeFile**(*infile*, *need_seek=None*)

Bases: object

File-like object used by SigFile, DeltaFile, and PatchFile

__init__(*infile*, *need_seek=None*)

LikeFile initializer - zero buffers, set eofs off

_add_to_inbuf()

Make sure len(self.inbuf) >= blocksize

_add_to_outbuf_once()

Add one cycle's worth of output to self.outbuf

check_file(*file*, *need_seek=None*)

Raise type error if file doesn't have necessary attributes

close()

Close infile

maker = None

mode = 'rb'

read(*length=-1*)

Build up self.outbuf, return first length bytes

class duplicity.librsync.**PatchedFile**(*basis_file*, *delta_file*)

Bases: *LikeFile*

File-like object which applies a librsync delta incrementally

__init__(*basis_file*, *delta_file*)

PatchedFile initializer - call with basis delta

Here basis_file must be a true Python file, because we may need to seek() around in it a lot, and this is done in C. delta_file only needs read() and close() methods.

class duplicity.librsync.**SigFile**(*infile*, *blocksize=duplicity._librsync.RS_DEFAULT_BLOCK_LEN*)

Bases: *LikeFile*

File-like object which incrementally generates a librsync signature

__init__(*infile*, *blocksize=duplicity._librsync.RS_DEFAULT_BLOCK_LEN*)

SigFile initializer - takes basis file

basis file only needs to have read() and close() methods. It will be closed when we come to the end of the signature.

class duplicity.librsync.**SigGenerator**(*blocksize=duplicity._librsync.RS_DEFAULT_BLOCK_LEN*)

Bases: object

Calculate signature.

Input and output is same as SigFile, but the interface is like md5 module, not filelike object

__init__(*blocksize=duplicity._librsync.RS_DEFAULT_BLOCK_LEN*)

Return new signature instance

getsig()

Return signature over given data

process_buffer()

Run self.buffer through sig_maker, add to self.sig_string

update(*buf*)

Add buf to data that signature will be calculated over

exception duplicity.librsync.**librsyncError**

Bases: Exception

Signifies error in internal librsync processing (bad signature, etc.)

underlying _librsync.librsyncError's are regenerated using this class because the C-created exceptions are by default unPickleable. There is probably a way to fix this in _librsync, but this scheme was easier.

1.1.2.20 duplicity.log module

Log various messages depending on verbosity level

duplicity.log.**Debug**(*s*)

Shortcut used for debug message (verbosity 9).

class duplicity.log.**DetailFormatter**

Bases: Formatter

Formatter that creates messages in a syntax somewhat like syslog.

__init__()

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` ({}) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

duplicity.log.**DupToLoggerLevel**(*verb*)

Convert duplicity level to the logging module's system, where higher is more severe

class duplicity.log.**ErrFilter**(*name=''*)

Bases: Filter

Filter that only allows messages more important than warnings

filter(*record*)

Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

duplicity.log.**Error**(*s*, *code=1*, *extra=None*)

Write error message

class duplicity.log.**ErrorCode**

Bases: object

Enumeration class to hold error code values. These values should never change, as frontends rely upon them. Don't use 0 or negative numbers. This code is returned by duplicity to indicate which error occurred via both exit code and log.

absolute_files_from = 72

backend_code_error = 55

backend_command_error = 54

backend_error = 50

backend_no_space = 53

backend_not_found = 52

backend_permission_denied = 51

backup_dir_doesnt_exist = 13

bad_archive_dir = 9

bad_request = 48

bad_sign_key = 10

bad_url = 8

boto_calling_format = 26

boto_lib_too_old = 25

boto_old_style = 24

cant_open_filelist = 7

command_line = 2

connection_failed = 38

dpbx_nologin = 47

empty_files_from = 73

enryption_mismatch = 45

exception = 30

file_prefix_error = 14

```
ftp_ncftp_missing = 27

ftp_ncftp_too_old = 28

ftps_lftp_missing = 43

generic = 1

get_freespace_failed = 34

get_ulimit_failed = 36

gio_not_available = 40

globbing_error = 15

gpg_failed = 31

hostname_mismatch = 3

inc_without_sigs = 17

maxopen_too_low = 37

mismatched_hash = 21

mismatched_manifests = 5

no_manifests = 4

no_restore_files = 20

no_sigs = 18

not_enough_freespace = 35

not_implemented = 33

pythonoptimize_set = 46

redundant_filter = 70

redundant_inclusion = 16

restart_file_not_found = 39

restore_dir_exists = 11

restore_dir_not_found = 19

s3_bucket_not_style = 32

s3_kms_no_id = 49

source_dir_mismatch = 42

trailing_filter = 71

unreadable_manifests = 6

unsigned_volume = 22
```

user_error = 23

verify_dir_doesnt_exist = 12

volume_wrong_size = 44

duplicity.log.**FatalError**(*s*, *code=1*, *extra=None*)

Write fatal error message and exit

duplicity.log.**Info**(*s*, *code=1*, *extra=None*)

Shortcut used for info messages (verbosity 5).

class duplicity.log.**InfoCode**

Bases: object

Enumeration class to hold info code values. These values should never change, as frontends rely upon them. Don't use 0 or negative numbers.

asynchronous_upload_begin = 12

asynchronous_upload_done = 14

collection_status = 3

diff_file_changed = 5

diff_file_deleted = 6

diff_file_new = 4

file_list = 10

generic = 1

patch_file_patching = 8

patch_file_writing = 7

progress = 2

skipping_socket = 15

synchronous_upload_begin = 11

synchronous_upload_done = 13

upload_progress = 16

duplicity.log.**LevelName**(*level*)

duplicity.log.**Log**(*s*, *verb_level*, *code=1*, *extra=None*, *force_print=False*, *transfer_progress=False*)

Write s to stderr if verbosity level low enough

duplicity.log.**LoggerToDupLevel**(*verb*)

Convert logging module level to duplicity's system, where lower is more severe

class duplicity.log.**MachineFilter**(*name=''*)

Bases: Filter

Filter that only allows levels that are consumable by other processes.

filter(*record*)

Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class duplicity.log.**MachineFormatter**

Bases: Formatter

Formatter that creates messages in a syntax easily consumable by other processes.

__init__()

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` ({}) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

duplicity.log.**Notice**(*s*)

Shortcut used for notice messages (verbosity 3, the default).

class duplicity.log.**OutFilter**(*name=''*)

Bases: Filter

Filter that only allows warning or less important messages

filter(*record*)

Determine if the specified record is to be logged.

Is the specified record to be logged? Returns 0 for no, nonzero for yes. If deemed appropriate, the record may be modified in-place.

class duplicity.log.**PrettyProgressFormatter**

Bases: Formatter

Formatter that overwrites previous progress lines on ANSI terminals

__init__()

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` ({}) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

format(*record*)

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

last_record_was_progress = False

duplicity.log.**PrintCollectionChangesInSet**(*col_stats*, *set_index*, *force_print=False*)

Prints changes in the specified set to the log

duplicity.log.**PrintCollectionFileChangedStatus**(*col_stats*, *filepath*, *force_print=False*)

Prints a collection status to the log

duplicity.log.**PrintCollectionStatus**(*col_stats*, *force_print=False*)

Prints a collection status to the log

duplicity.log.**Progress**(*s*, *current*, *total=None*)

Shortcut used for progress messages (verbosity 5).

duplicity.log.**TransferProgress**(*progress*, *eta*, *changed_bytes*, *elapsed*, *speed*, *stalled*)

Shortcut used for upload progress messages (verbosity 5).

duplicity.log.**Warn**(*s*, *code=1*, *extra=None*)

Shortcut used for warning messages (verbosity 2)

class duplicity.log.**WarningCode**

Bases: `object`

Enumeration class to hold warning code values. These values should never change, as frontends rely upon them. Don't use 0 or negative numbers.

cannot_iterate = 8

cannot_process = 12

cannot_read = 10

cannot_stat = 9

ftp_ncftp_v320 = 7

generic = 1

incomplete_backup = 5

no_sig_for_time = 11

orphaned_backup = 6

orphaned_sig = 2

process_skipped = 13

unmatched_sig = 4

unnecessary_sig = 3

duplicity.log.**_ElapsedSecs2Str**(*secs*)

duplicity.log.**_RemainingSecs2Str**(*secs*)

duplicity.log.**add_fd**(*fd*)
Add stream to which to write machine-readable logging

duplicity.log.**add_file**(*filename*)
Add file to which to write machine-readable logging

duplicity.log.**getverbosity**()
Get the verbosity level

duplicity.log.**setup**()
Initialize logging

duplicity.log.**setverbosity**(*verb*)
Set the verbosity level

duplicity.log.**shutdown**()
Cleanup and flush loggers

1.1.2.21 duplicity.manifest module

Create and edit manifest for session contents

class duplicity.manifest.**Manifest**(*fh=None*)
Bases: object

List of volumes and information about each one

__init__(*fh=None*)
Create blank Manifest

@param fh: fileobj for manifest @type fh: DupPath

@rtype: Manifest @return: manifest

add_volume_info(*vi*)
Add volume info vi to manifest and write to manifest

@param vi: volume info to add @type vi: VolumeInfo

@return: void

check_dirinfo()
Return None if dirinfo is the same, otherwise error message

Does not raise an error message if hostname or local_dirname are not available.

@rtype: string @return: None or error message

del_volume_info(*vol_num*)
Remove volume vol_num from the manifest

@param vol_num: volume number to delete @type vi: int

@return: void

from_string(*s*)
Initialize self from string s, return self

get_containing_volumes(*index_prefix*)

Return list of volume numbers that may contain index_prefix

get_files_changed()

set_dirinfo()

Set information about directory from config, and write to manifest file.

@rtype: Manifest @return: manifest

set_files_changed_info(*files_changed*)

to_string()

Return string version of self (just concatenate vi strings)

@rtype: string @return: self in string form

write_to_path(*path*)

Write string version of manifest to given path

exception duplicity.manifest.**ManifestError**

Bases: Exception

Exception raised when problem with manifest

duplicity.manifest.**Quote**(*s*)

Return quoted version of s safe to put in a manifest or volume info

duplicity.manifest.**Unquote**(*quoted_string*)

Return original string from quoted_string produced by above

class duplicity.manifest.**VolumeInfo**

Bases: object

Information about a single volume

__init__()

VolumeInfo initializer

contains(*index_prefix*, *recursive=1*)

Return true if volume might contain index

If recursive is true, then return true if any index starting with index_prefix could be contained. Otherwise, just check if index_prefix itself is between starting and ending indicies.

from_string(*s*)

Initialize self from string s as created by to_string

get_best_hash()

Return pair (hash_type, hash_data)

SHA1 is the best hash, and MD5 is the second best hash. None is returned if no hash is available.

set_hash(*hash_name*, *data*)

Set the value of hash hash_name (e.g. "MD5") to data

set_info(*vol_number*, *start_index*, *start_block*, *end_index*, *end_block*)

Set essential VolumeInfo information, return self

Call with starting and ending paths stored in the volume. If a multivol diff gets split between volumes, count it as being part of both volumes.

to_string()

Return nicely formatted string reporting all information

exception duplicity.manifest.**VolumeInfoError**

Bases: Exception

Raised when there is a problem initializing a VolumeInfo from string

duplicity.manifest.**maybe_chr**(*ch*)

1.1.2.22 duplicity.patchdir module

class duplicity.patchdir.**IndexedTuple**(*index*, *sequence*)

Bases: object

Like a tuple, but has .index (used previously by collate_iters)

__init__(*index*, *sequence*)

class duplicity.patchdir.**Multivol_Filelike**(*tf*, *tar_iter*, *tarinfo_list*, *index*)

Bases: object

Emulate a file like object from multivols

Maintains a buffer about the size of a volume. When it is read() to the end, pull in more volumes as desired.

__init__(*tf*, *tar_iter*, *tarinfo_list*, *index*)

Initializer. tf is TarFile obj, tarinfo is first tarinfo

addtobuffer()

Add next chunk to buffer

close()

If not at end, read remaining data

read(*length=-1*)

Read length bytes from file

duplicity.patchdir.**Patch**(*base_path*, *difftar_fileobj*)

Patch given base_path and file object containing delta

exception duplicity.patchdir.**PatchDirException**

Bases: Exception

duplicity.patchdir.**Patch_from_iter**(*base_path*, *fileobj_iter*, *restrict_index=()*)

Patch given base_path and iterator of delta file objects

class duplicity.patchdir.**PathPatcher**(*base_path*)

Bases: *ITRBranch*

Used by DirPatch, process the given basis and diff

__init__(*base_path*)

Set base_path, Path of root of tree

can_fast_process(*index*, *basis_path*, *diff_ropath*)

No need to recurse if diff_ropath isn't a directory

end_process()

Copy directory permissions when leaving tree

fast_process(*index*, *basis_path*, *diff_ropath*)

For use when neither is a directory

start_process(*index*, *basis_path*, *diff_ropath*)

Start processing when diff_ropath is a directory

class duplicity.patchdir.**ROPath_IterWriter**(*base_path*)

Bases: *ITRBranch*

Used in Write_ROPaths above

We need to use an ITR because we have to update the permissions/times of directories after we write the files in them.

__init__(*base_path*)

Set base_path, Path of root of tree

can_fast_process(*index*, *ropath*)

Can fast process (no recursion) if ropath isn't a directory

end_process()

Update information of a directory when leaving it

fast_process(*index*, *ropath*)

Write non-directory ropath to destination

start_process(*index*, *ropath*)

Write ropath. Only handles the directory case

class duplicity.patchdir.**TarFile_FromFileobjs**(*fileobj_iter*)

Bases: object

Like a tarfile.TarFile iterator, but read from multiple fileobjs

__init__(*fileobj_iter*)

Make new tarinfo iterator

fileobj_iter should be an iterator of file objects opened for reading. They will be closed at end of reading.

extractfile(*tarinfo*)

Return data associated with given tarinfo

set_tarfile()

Set tarfile from next file object, or raise StopIteration

duplicity.patchdir.**Write_ROPaths**(*base_path*, *rop_iter*)

Write out ropaths in rop_iter starting at base_path

Returns 1 if something was actually written, 0 otherwise.

duplicity.patchdir.**collate_iters**(*iter_list*)

Collate iterators by index

Input is a list of n iterators each of which must iterate elements with an index attribute. The elements must come out in increasing order, and the index should be a tuple itself.

The output is an iterator which yields tuples where all elements in the tuple have the same index, and the tuple has n elements in it. If any iterator lacks an element with that index, the tuple will have None in that spot.

duplicity.patchdir.**difftar2path_iter**(*diff_tarfile*)

Turn file-like difftarobj into iterator of ROPaths

duplicity.patchdir.**empty_iter**()

duplicity.patchdir.**filter_path_iter**(*path_iter*, *index*)

Rewrite path elements of path_iter so they start with index

Discard any that doesn't start with index, and remove the index prefix from the rest.

duplicity.patchdir.**get_index_from_tarinfo**(*tarinfo*)

Return (index, difftype, multivol) pair from tarinfo object

duplicity.patchdir.**integrate_patch_iters**(*iter_list*)

Combine a list of iterators of ropath patches

The iter_list should be sorted in patch order, and the elements in each iter_list need to be orderd by index. The output will be an iterator of the final ROPaths in index order.

duplicity.patchdir.**normalize_ps**(*patch_sequence*)

Given an sequence of ROPath deltas, remove blank and unnecessary

The sequence is assumed to be in patch order (later patches apply to earlier ones). A patch is unnecessary if a later one doesn't require it (for instance, any patches before a "delete" are unnecessary).

duplicity.patchdir.**patch_diff_tarfile**(*base_path*, *diff_tarfile*, *restrict_index=()*)

Patch given Path object using delta tarfile (as in tarfile.TarFile)

If restrict_index is set, ignore any deltas in diff_tarfile that don't start with restrict_index.

duplicity.patchdir.**patch_seq2ropath**(*patch_seq*)

Apply the patches in patch_seq, return single ropath

duplicity.patchdir.**tarfiles2rop_iter**(*tarfile_list*, *restrict_index=()*)

Integrate tarfiles of diffs into single ROPath iter

Then filter out all the diffs in that index which don't start with the restrict_index.

1.1.2.23 duplicity.path module

Wrapper class around a file like "/usr/bin/env"

This class makes certain file operations more convenient and associates stat information with filenames

class duplicity.path.**DupPath**(*base*, *index=()*, *parseresults=None*)

Bases: *Path*

Represent duplicity data files

Based on the file name, files that are compressed or encrypted will have different open() methods.

__init__(*base*, *index=()*, *parseresults=None*)

DupPath initializer

The actual filename (no directory) must be the single element of the index, unless parseresults is given.

filtered_open(*mode='rb'*, *gpg_profile=None*)

Return fileobj with appropriate encryption/compression

If encryption is specified but no gpg_profile, use config.default_profile.

class duplicity.path.**Path**(*base*, *index=()*)

Bases: *ROPath*

Path class - wrapper around ordinary local files

Besides caching stat() results, this class organizes various file code.

__init__(*base*, *index=()*)

Path initializer

append(*ext*)

Return new Path with ext added to index

chmod(*mode*)

Change permissions of the path

compare_recursive(*other*, *verbose=None*)

Compare self to other Path, descending down directories

contains(*child*)

Return true if path is a directory and contains child

delete()

Remove this file

deltree()

Remove self by recursively deleting files under it

get_canonical()

Return string of canonical version of path

Remove ".", and trailing slashes where possible. Note that it's harder to remove "..", as "foo/bar/.." is not necessarily "foo", so we can't use path.normpath()

get_filename()

Return filename of last component

get_parent_dir()

Return directory that self is in

get_temp_in_same_dir()

Return temp non existent path in same directory as self

isemptydir()

Return true if path is a directory and is empty

listdir()

Return list generated by os.listdir

makedev(*type*, *major*, *minor*)

Make a device file with specified type, major/minor nums

mkdir()

Make directory(s) at specified path

move(*new_path*)

Like rename but destination may be on different file system

new_index(*index*)

Return new Path with index index

open(*mode='rb'*)

Return fileobj associated with self

Usually this is just the file data on disk, but can be replaced with arbitrary data using the setfileobj method.

patch_with_attribs(*diff_ropath*)

Patch self with diff and then copy attributes over

quote(*s=None*)

Return quoted version of s (defaults to self.name)

The output is meant to be interpreted with shells, so can be used with os.system.

regex_chars_to_quote = re.compile('[\\\\\\\"\\$`]')

rename(*new_path*)

Rename file at current path to new_path.

rename_index(*index*)

setdata()

Refresh stat cache

touch()

Open the file, write 0 bytes, close

unquote(*s*)

Return unquoted version of string s, as quoted by above quote()

writefileobj(*fin*)

Copy file object fin to self. Close both when done.

class duplicity.path.**PathDeleter**

Bases: *ITRBranch*

Delete a directory. Called by Path.deltree

can_fast_process(*index*, *path*)

True if object can be processed without new branch (stub)

end_process()

Do any final processing before leaving branch (stub)

fast_process(*index*, *path*)

Process args without new child branch (stub)

start_process(*index*, *path*)

Do some initial processing (stub)

exception duplicity.path.**PathException**

Bases: Exception

class duplicity.path.**ROPath**(*index*, *stat=None*)

Bases: object

Read only Path

Objects of this class doesn't represent real files, so they don't have a name. They are required to be indexed though.

__init__(*index*, *stat=None*)

ROPath initializer

blank()

Black out self - set type and stat to None

compare_data(*other*)

Compare data from two regular files, return true if same

compare_verbose(*other*, *include_data=0*)

Compare ROPaths like __eq__, but log reason if different

This is placed in a separate function from __eq__ because __eq__ should be very time sensitive, and logging statements would slow it down. Used when verifying.

Only run if include_data is true.

copy(*other*)

Copy self to other. Also copies data. Other must be Path

copy_attribs(*other*)

Only copy attributes from self to other

exists()

True if corresponding file exists

get_data()

Return contents of associated fileobj in string

get_relative_path()

Return relative path, created from index

get_ropath()

Return ropath copy of self

get_tarinfo()

Generate a tarfile.TarInfo object based on self

Doesn't set size based on stat, because we may want to replace data wiht other stream. Size should be set separately by calling function.

getdevloc()

Return device number path resides on

getmtime()

Return mod time of path in seconds

getperms()

Return permissions mode, owner and group

getsize()

Return length in bytes from stat object

init_from_tarinfo(*tarinfo*)

Set data from tarinfo object (part of tarfile module)

isdev()

True is self is a device file

isdir()

True if self is dir

isfifo()

True if self is fifo

isreg()

True if self corresponds to regular file

issock()

True is self is socket

issym()

True if self is sym

open(*mode*)

Return fileobj associated with self

perms_equal(*other*)

True if self and other have same permissions and ownership

set_from_stat()

Set the value of self.type, self.mode from self.stat

setfileobj(*fileobj*)

Set file object returned by open()

class duplicity.path.**StatResult**

Bases: object

Used to emulate the output of os.stat() and related

st_mode = 0

1.1.2.24 duplicity.progress module

Functions to compute progress of compress & upload files The heuristics try to infer the ratio between the amount of data collected by the deltas and the total size of the changing files. It also infers the compression and encryption ration of the raw deltas before sending them to the backend. With the inferred ratios, the heuristics estimate the percentage of completion and the time left to transfer all the (yet unknown) amount of data to send. This is a forecast based on gathered evidence.

class duplicity.progress.**LogProgressThread**

Bases: Thread

Background thread that reports progress to the log, every –progress-rate seconds

__init__()

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form “Thread-N” where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class duplicity.progress.**ProgressTracker**

Bases: object

__init__()

annotate_written_bytes(*bytecount*)

Annotate the number of bytes that have been added/changed since last time this function was called. bytecount param will show the number of bytes since the start of the current volume and for the current volume

has_collected_evidence()

Returns true if the progress computation is on and duplicity has not yet started the first dry-run pass to collect some information

log_upload_progress()

Aproximative and evolving method of computing the progress of upload

set_evidence(*stats*, *is_full*)

Stores the collected statistics from a first-pass dry-run, to use this information later so as to estimate progress

set_start_volume(*volume*)

snapshot_progress(*volume*)

Snapshots the current progress status for each volume into the disk cache If backup is interrupted, next restart will deserialize the data and try start progress from the snapshot

total_elapsed_seconds()

Elapsed seconds since the first call to log_upload_progress method

class duplicity.progress.**Snapshot**(*iterable=[]*, *maxlen=10*)

Bases: deque

A convenience class for storing snapshots in a space/timing efficient manner Stores up to 10 consecutive progress snapshots, one for each volume

__init__(*iterable=[]*, *maxlen=10*)

clear()

Remove all elements from the deque.

get_snapshot(*volume*)

marshall()

Serializes object to cache

pop_snapshot()

push_snapshot(*volume*, *snapshot_data*)

static unmarshall()

De-serializes cached data it if present

duplicity.progress.**report_transfer**(*bytecount*, *totalbytes*)

Method to call tracker.annotate_written_bytes from outside the class, and to offer the "function(long, long)" signature which is handy to pass as callback

1.1.2.25 duplicity.robust module

duplicity.robust.**check_common_error**(*error_handler*, *function*, *args=()*)

Apply function to args, if error, run error_handler on exception

This only catches certain exceptions which seem innocent enough.

duplicity.robust.**listpath**(*path*)

Like path.listdir() but return [] if error, and sort results

1.1.2.26 duplicity.selection module

class duplicity.selection.**Select**(*path*)

Bases: object

Iterate appropriate Paths in given directory

This class acts as an iterator on account of its next() method. Basically, it just goes through all the files in a directory in order (depth-first) and subjects each file to a bunch of tests (selection functions) in order. The first test that includes or excludes the file means that the file gets included (iterated) or excluded. The default is include, so with no tests we would just iterate all the files in the directory in order.

The one complication to this is that sometimes we don't know whether or not to include a directory until we examine its contents. For instance, if we want to include all the **.py files. If /home/ben/foo.py exists, we should also include /home and /home/ben, but if these directories contain no **.py files, they shouldn't be included. For this reason, a test may not include or exclude a directory, but merely "scan" it. If later a file in the directory gets included, so does the directory.

As mentioned above, each test takes the form of a selection function. The selection function takes a path, and returns:

None - means the test has nothing to say about the related file 0 - the file is excluded by the test 1 - the file is included 2 - the test says the file (must be directory) should be scanned

Also, a selection function f has a variable f.exclude which should be true if f could potentially exclude some file. This is used to signal an error if the last function only includes, which would be redundant and presumably isn't what the user intends.

Iterate(*path*)

Return iterator yielding paths in path

This function looks a bit more complicated than it needs to be because it avoids extra recursion (and no extra function calls for non-directory files) while still doing the "directory scanning" bit.

ParseArgs(*argtuples*, *filelists*)

Create selection functions based on list of tuples

The tuples are created when the initial commandline arguments are read. They have the form (option string, additional argument) except for the filelist tuples, which should be (option-string, (additional argument, filelist_fp)).

Select(*path*)

Run through the selection functions and return dominant val 0/1/2

__init__(*path*)

Initializer, called with Path of root directory

add_selection_func(*sel_func*, *add_to_start=None*)

Add another selection function at the end or beginning

devfiles_get_sf()

Return a selection function to exclude all dev files

exclude_older_get_sf(*date*)

Return selection function based on files older than modification date

filelist_general_get_sfs(*filelist_fp*, *inc_default*, *list_name*, *mode='globbing'*, *ignore_case=False*)

Return list of selection functions by reading fileobj

filelist_fp should be an open file object inc_default is true if this is an include list list_name is just the name of the list, used for logging mode indicates whether to glob, regex, or not

filelist_sanitise_line(*line*, *include_default*)

Sanitises lines of both normal and globbing filelists, returning (line, include) and line=None if blank/comment

The aim is to parse filelists in a consistent way, prior to the interpretation of globbing statements. The function removes whitespace, comment lines and processes modifiers (leading +/-) and quotes.

general_get_sf(*pattern_str*, *include*, *mode='globbing'*, *ignore_case=False*)

Return selection function given by a pattern string

The selection patterns are interpretted in accordance with the mode argument, "globbing", "literal", or "regex".

The 'ignorecase:' prefix is a legacy feature which historically lived on the globbing code path and was only ever documented as working for globs.

glob_get_sf(*glob_str*, *include*, *ignore_case=False*)

Return selection function based on glob_str

literal_get_sf(*lit_str*, *include*, *ignore_case=False*)

Return a selection function that matches a literal string while still including the contents of any folders which are matched

other_filesystems_get_sf(*include*)

Return selection function matching files on other filesystems

parse_catch_error(*exc*)

Deal with selection error exc

parse_files_from(*filelist_fp*, *list_name*)

Loads an explicit list of files to backup from a filelist, building a dictionary of directories and their contents which can be used later to emulate a filesystem walk over the listed files only.

Each specified path is unwound to identify the parents folder(s) as these are implicitly to be included.

Paths read are not to be stripped, checked for comments, etc. Every character on each line is significant and treated as part of the path.

parse_last_excludes()

Exit with error if last selection function isn't an exclude

present_get_sf(*filename*, *include*)

Return selection function given by existence of a file in a directory

regexp_get_sf(*regexp_string*, *include*, *ignore_case=False*)

Return selection function given by regexp_string

select_fn_from_literal(*lit_str*, *include*, *ignore_case=False*)

Return a function test_fn(path) which test where a path matches a literal string. See also select_fn_from_blog() in globmatch.py

This function is separated from literal_get_sf() so that it can be used to test the prefix without creating a loop.

TODO: this doesn't need to be part of the Select class type, but not sure where else to put it?

set_iter()

Initialize generator, prepare to iterate.

1.1.2.27 duplicity.statistics module

Generate and process backup statistics

class duplicity.statistics.**StatsDeltaProcess**

Bases: *StatsObj*

Keep track of statistics during DirDelta process

__init__()

StatsDeltaProcess initializer - zero file attributes

add_changed_file(*path*)

Add stats of file that has changed since last backup

add_deleted_file(*path*)

Add stats of file no longer in source directory

add_delta_entries_file(*path*, *action_type*)

add_new_file(*path*)

Add stats of new file path to statistics

add_unchanged_file(*path*)

Add stats of file that hasn't changed since last backup

close()

End collection of data, set EndTime

get_delta_entries_file()

exception duplicity.statistics.**StatsException**

Bases: Exception

class duplicity.statistics.**StatsObj**

Bases: object

Contains various statistics, provide string conversion functions

__init__()

Set attributes to None

byte_abbrev_list = ((1099511627776, 'TB'), (1073741824, 'GB'), (1048576, 'MB'), (1024, 'KB'))

get_byte_summary_string(*byte_count*)

Turn byte count into human readable string like "7.23GB"

get_filestats_string()

Return portion of statistics string about files and bytes

get_miscstats_string()

Return portion of extended stat string about misc attributes

get_stat(*attribute*)

Get a statistic

get_stats_line(*index*, *use_repr=1*)

Return one line abbreviated version of full stats string

get_stats_logstring(*title*)

Like get_stats_string, but add header and footer

get_stats_string()

Return extended string printing out statistics

get_statsobj_copy()

Return new StatsObj object with same stats as self

get_timestats_string()

Return portion of statistics string dealing with time

increment_stat(*attr*)

Add 1 to value of attribute

read_stats_from_path(*path*)

Set statistics from path, return self for convenience

set_stat(*attr*, *value*)

Set attribute to given value

set_stats_from_line(*line*)

Set statistics from given line

set_stats_from_string(*s*)

Initialize attributes from string, return self for convenience

set_to_average(*statobj_list*)

Set self's attributes to average of those in statobj_list

space_regex = re.compile(' ')

stat_attrs = ('Filename', 'StartTime', 'EndTime', 'ElapsedTime', 'Errors', 'TotalDestinationSizeChange', 'SourceFiles', 'SourceFileSize', 'NewFiles', 'NewFileSize', 'DeletedFiles', 'ChangedFiles', 'ChangedFileSize', 'ChangedDeltaSize', 'DeltaEntries', 'RawDeltaSize')

```
stat_file_attrs = ('SourceFiles', 'SourceFileSize', 'NewFiles', 'NewFileSize',
'DeletedFiles', 'ChangedFiles', 'ChangedFileSize', 'ChangedDeltaSize',
'DeltaEntries', 'RawDeltaSize')

stat_file_pairs = (('SourceFiles', False), ('SourceFileSize', True), ('NewFiles',
False), ('NewFileSize', True), ('DeletedFiles', False), ('ChangedFiles', False),
('ChangedFileSize', True), ('ChangedDeltaSize', True), ('DeltaEntries', False),
('RawDeltaSize', True))

stat_misc_attrs = ('Errors', 'TotalDestinationSizeChange')

stat_time_attrs = ('StartTime', 'EndTime', 'ElapsedTime')
```

stats_equal(*s*)

Return true if s has same statistics as self

write_stats_to_path(*path*)

Write statistics string to given path

1.1.2.28 duplicity.tarfile module

Like system tarfile but with caching.

1.1.2.29 duplicity.tempdir module

Provides temporary file handling cenetered around a single top-level securely created temporary directory.

The public interface of this module is thread-safe.

class duplicity.tempdir.**TemporaryDirectory**(*temproot=None*)

Bases: `object`

A temporary directory.

An instance of this class is backed by a directory in the file system created securely by the use of tempfile.mkdtemp(). Said instance can be used to obtain unique filenames inside of this directory for cases where mktemp()-like semantics is desired, or (recommended) an fd,filename pair for mkstemp()-like semantics.

See further below for the security implications of using it.

Each instance will keep a list of all files ever created by it, to faciliate deletion of such files and rmdir() of the directory itself. It does this in order to be able to clean out the directory without resorting to a recursive delete (ala rm -rf), which would be risky. Calling code can optionally (recommended) notify an instance of the fact that a tempfile was deleted, and thus need not be kept track of anymore.

This class serves two primary purposes:

Firstly, it provides a convenient single top-level directory in which all the clutter ends up, rather than cluttering up the root of the system temp directory itself with many files.

Secondly, it provides a way to get mktemp() style semantics for temporary file creation, with most of the risks gone. Specifically, since the directory itself is created securely, files in this directory can be (mostly) safely created non-atomically without the usual mktemp() security implications. However, in the presence of tmpwatch, tmpreaper, or similar mechanisms that will cause files in the system tempdir to expire, a security risk is still present because the removal of the TemporaryDirectory managed directory removes all protection it offers.

For this reason, use of mkstemp() is greatly preferred above use of mktemp().

In addition, since cleanup is in the form of deletion based on a list of filenames, completely independently of whether someone else already deleted the file, there exists a race here as well. The impact should however be limited to the removal of an 'attackers' file.

__init__(*temproot=None*)

Create a new TemporaryDirectory backed by a unique and securely created file system directory.

tempbase - The temp root directory, or None to use system default (recommended).

cleanup()

Cleanup any files created in the temporary directory (that have not been forgotten), and clean up the temporary directory itself.

On failure they are logged, but this method will not raise an exception.

dir()

Returns the absolute pathname of the temp folder.

forget(*fname*)

Forget about the given filename previously obtained through mktemp() or mkstemp(). This should be called *after* the file has been deleted, to stop a future cleanup() from trying to delete it.

Forgetting is only needed for scaling purposes; that is, to avoid n timefile creations from implying that n filenames are kept in memory. Typically this whould never matter in duplicity, but for niceness sake callers are recommended to use this method whenever possible.

mkstemp()

Returns a filedescriptor and a filename, as per os.mkstemp(), but located in the temporary directory and subject to tracking and automatic cleanup.

mkstemp_file()

Convenience wrapper around mkstemp(), with the file descriptor converted into a file object.

mktemp()

Return a unique filename suitable for use for a temporary file. The file is not created.

Subsequent calls to this method are guaranteed to never return the same filename again. As a result, it is safe to use under concurrent conditions.

NOTE: mkstemp() is greatly preferred.

duplicity.tempdir.**default**()

Obtain the global default instance of TemporaryDirectory, creating it first if necessary. Failures are propagated to caller. Most callers are expected to use this function rather than instantiating TemporaryDirectory directly, unless they explicitly desdire to have their "own" directory for some reason.

This function is thread-safe.

1.1.2.30 duplicity.util module

Miscellaneous utilities.

class duplicity.util.**BlackHoleList**(*iterable=()*, */*)

Bases: list

append(*x*)

Append object to the end of the list.

class duplicity.util.**FakeTarFile**

Bases: object

close()

debug = 0

duplicity.util.**casefold_compat**(*s*)

Compatability function for casefolding which provides an acceptable for older pythons. Can likely be removed once python2 support is no longer o any interest.

duplicity.util.**copyfileobj**(*infp*, *outfp*, *byte_count=-1*)

Copy byte_count bytes from infp to outfp, or all if byte_count < 0

Returns the number of bytes actually written (may be less than byte_count if find eof. Does not close either fileobj.

duplicity.util.**csv_args_to_dict**(*arg*)

Given the string arg in single line csv format, split into pairs (key, val) and produce a dictionary from those key:val pairs.

duplicity.util.**escape**(*string*)

Convert a (bytes) filename to a format suitable for logging (quoted utf8)

duplicity.util.**exception_traceback**(*limit=50*)

@return A string representation in typical Python format of the
currently active/raised exception.

duplicity.util.**get_tarinfo_name**(*ti*)

duplicity.util.**ignore_missing**(*fn*, *filename*)

Execute fn on filename. Ignore ENOENT errors, otherwise raise exception.

@param fn: callable @param filename: string

duplicity.util.**make_tarfile**(*mode*, *fp*)

duplicity.util.**maybe_ignore_errors**(*fn*)

Execute fn. If the global configuration setting ignore_errors is set to True, catch errors and log them but do continue (and return None).

@param fn: A callable. @return Whatever fn returns when called, or None if it failed and ignore_errors is true.

duplicity.util.**merge_dicts**(**dict_args*)

Given any number of dictionaries, shallow copy and merge into a new dict, precedence goes to key-value pairs in latter dictionaries.

duplicity.util.**release_lockfile**()

duplicity.util.**start_debugger**()

duplicity.util.**uexc**(*e*)

Returns the exception message in Unicode

duplicity.util.**uindex**(*index*)

Convert an index (a tuple of path parts) to unicode for printing

duplicity.util.**which**(*program*)

Return absolute path for program name. Returns None if program not found.

1.1.3 Module contents

1.2 testing package

1.2.1 Subpackages

1.2.1.1 testing.functional package

Submodules

testing.functional.test_baduploa module

testing.functional.test_cleanup module

testing.functional.test_final module

testing.functional.test_log module

testing.functional.test_rdiffdir module

testing.functional.test_replicate module

testing.functional.test_restart module

testing.functional.test_selection module

testing.functional.test_verify module

Module contents

1.2.1.2 testing.unit package

Submodules

testing.unit.test_backend module

testing.unit.test_backend_instance module

testing.unit.test_collections module

testing.unit.test_diffdir module

testing.unit.test_dup_temp module

testing.unit.test_dup_time module

1.2.2 Submodules

1.2.3 Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

Y