Cerulean Documentation

Release 0.3.8.dev

Lourens Veen

Mar 12, 2021

Contents:

Cerulean is a Python 3 library for connecting to HPC compute resources, such as compute clusters and supercomputers. It lets you copy files between local and SFTP filesystems using a `pathlib`-like API, it lets you start processes locally and remotely via SSH, and it lets you submit jobs to schedulers such as Slurm and Torque/PBS.

Cerulean supports Python 3.4 and later.

CHAPTER 1

Tutorial

Welcome to the Cerulean tutorial. This tutorial demonstrates the basics of using Cerulean: using local and remote file systems, running processes locally and remotely, and using schedulers.

To install Cerulean, use

```
pip install cerulean
```

If you're using Cerulean in a program, you will probably want to use a virtualenv and install Cerulean into that, together with your other dependencies.

1.1 Accessing files

The file access functions of Cerulean use a `pathlib`-like API, but unlike in `pathlib`, Cerulean supports remote file systems. That means that there is no longer just the local file system, but multiple file systems, and that Path objects have a particular file system that they are on.

Of course, Cerulean also supports the local file system. To make an object representing the local file system, you use this:

```
import cerulean

fs = cerulean.LocalFileSystem()
```

And then you can make a path on the file system using:

```
import cerulean

fs = cerulean.LocalFileSystem()
my_home_dir = fs / 'home' / 'username'
```

In this example, `my_home_dir` will be a *`cerulean.Path`* object, which is very similar to a normal Python `pathlib.PosixPath`. For example, you can read the contents of a file through it:

```
import cerulean

fs = cerulean.LocalFileSystem()
passwd_file = fs / 'etc' / 'passwd'

users = passwd_file.read_text()
print(users)
```

Note that *cerulean.Path* does not support `open()`. Cerulean can copy files and stream data from and to them, but it does not offer random access, as not all remote file access protocols support this.

You can use the `/` operator to build paths from components as with `pathlib`, and there's a wide variety of supported operations. See the API documentation for *cerulean.Path* for details.

1.1.1 Remote filesystems

Cerulean supports remote file systems through the SFTP protocol. (It uses the Paramiko library internally for this.) Accessing a remote file system through SFTP goes like this:

```
import cerulean

credential = cerulean.PasswordCredential('username', 'password')
with cerulean.SshTerminal('remotehost.example.com', 22, credential) as term
    with SftpFileSystem(term) as fs:
        my_home_dir = fs / 'home' / 'username'
        test_txt = (my_home_dir / 'test.txt').read_text()
        print(test_txt)
```

Since we are going to connect to a remote system, we need a credential. Cerulean has two types of credentials, `PasswordCredential` and `PubKeyCredential`. They are what you expect, one holds a username and a password, the other a username, a local path to a public key file, and optionally a passphrase for the key.

Once we have a credential, we can open a terminal. Like a terminal window on your desktop, a `Terminal` object lets you run commands. Cerulean supports local terminals and remote terminals through SSH. Since the SFTP protocol is an extension to the SSH protocol, we need an SSH terminal connection first, so we make one, connecting to a host, on a port, with our credential. This terminal holds an SSH connection, which needs to be closed when we are done with it. `SshTerminal` is therefore a context manager and needs to be used in a `with` statement. Note that `LocalTerminal` is not a context manager, as it does not hold any resources.

Once we have the terminal, we can make an `SftpFileSystem` object, and from there it works just like a local file system. Just like `SshTerminal`, `SftpFileSystem` is a context manager, so we need another `with`-statement.

1.1.2 Copying files

When running jobs on HPC machines, you often start with copying the input files from the local system to the HPC machine, and finish with copying the results back. Cerulean's `copy()` function takes care of this for you, and works as you would expect:

```
import cerulean


local_fs = cerulean.LocalFileSystem()

credential = cerulean.PasswordCredential('username', 'password')
with cerulean.SshTerminal('remotehost.example.com', 22, credential) as term
```

(continues on next page)

(continued from previous page)

```
    with SftpFileSystem(term) as remote_fs:
        input_file = local_fs / 'home' / 'username' / 'input.txt'
        job_dir = remote_fs / 'home' / 'username' / 'my_job'
        cerulean.copy(input_file, job_dir)

        # run job and wait for it to finish

        output_file = local_fs / 'home' / 'username' / 'output.txt'
        cerulean.copy(job_dir / 'output.txt', output_file)
```

1.2 Running commands

If you have read the above, then the secret is already out: running commands using Cerulean is done using a `Terminal`. For example, you can run a command locally using:

```
import cerulean

term = cerulean.LocalTerminal()

exit_code, stdout_text, stderr_text = term.run(
        10.0, 'ls', ['-l'], None, '/home/username')
```

The first argument to `Terminal.run()` is a timeout value in seconds, which determines how long Cerulean will wait for the command to finish. The second argument is the command to run, followed by a list of arguments. Next is an optional string that, if you specify it, will be fed into the standard input of the program you are starting. The final argument is a string specifying the working directory in which to execute the command.

The function returns a tuple containing three values: the exit code of the process (or *None* if it didn't finish in time), a string containing text printed to standard output, and a string containing text printed to standard error by the command you ran.

Running commands remotely through SSH of course works in exactly the same way, except you use an `SshTerminal`, as above:

```
import cerulean

credential = cerulean.PasswordCredential('username', 'password')
with cerulean.SshTerminal('remotehost.example.com', 22, credential) as term
    exit_code, stdout_text, stderr_text = term.run(
            10.0, 'ls', ['-l'], None, '/home/username')
```

1.3 Submitting jobs

On High Performance Computing machines, you don't run commands directly. Instead, you submit batch jobs to a scheduler, which will place them in a queue, and run them when everyone else in line before you is done. The most popular scheduler at the moment seems to be Slurm, but Cerulean also supports Torque/PBS.

The usual way of working with a scheduler is to use `ssh` to connect to the cluster, where you run commands that submit jobs and check on their status. Cerulean works in the same way:

```
import cerulean
import time

credential = cerulean.PasswordCredential('username', 'password')
with cerulean.SshTerminal('remotehost.example.com', 22, credential) as term
    sched = cerulean.SlurmScheduler(term)

    job = cerulean.JobDescription()
    job.name = 'cerulean_test'
    job.command = 'ls'
    job.arguments = ['-l']

    job_id = sched.submit_job(job)

    time.sleep(5)
    status = sched.get_status(job_id)

    if status == cerulean.JobStatus.DONE:
        exit_code = sched.get_exit_code()
        print('Job exited with code {}'.format(exit_code))
```

Of course, if you intend to run your submission script on the head node, then the scheduler is local, and you want to use a `LocalTerminal` with your `SlurmScheduler`. If your HPC machine runs Torque/PBS, use a `TorqueScheduler` instead.

1.4 More information

To find all the details of what Cerulean can do and how to do it, please refer to the *API documentation*.

CHAPTER 2

API Reference

2.1 cerulean package

The *cerulean* module is the main API for Cerulean.

This module contains all the functions you need to use Cerulean.

Below, you will also find documentation for submodules. That is developer documentation, you do not need it to use Cerulean.

cerulean.**copy**(*source_path: cerulean.path.Path, target_path: cerulean.path.Path, overwrite: str = 'never', copy_into: bool = True, copy_permissions: bool = False, callback: Optional[Callable[[int, int], None]] = None*) → None

Copy a file or directory from one path to another.

Note that source_path and target_path may be paths on different file systems.

The overwrite parameter decides what to do if a file is encountered on the target side that would be overwritten. If overwrite equals 'raise', then a FileExistsError is raised. If overwrite equals 'always', then the file is overwritten (removed and replaced if needed). If overwrite equals 'never', then the existing file is kept, and the source file not copied.

If the target is a directory and copy_into is True (the default), the source will be copied into the target directory. If an entry with the same name already exists within the target directory, then the overwrite parameter decides what happens. If copy_into is False, the source will be copied on top of the directory, subject to the setting for overwrite.

If copy_permissions is True, this function will make the target's permissions match those of the source, *including* SETUID, SETGID and sticky bits. If copy_permissions is False, the target's permissions are left at their default values (according to the umask, on Unix-like systems), less any permissions that the source file does not have.

If callback is provided, it should be a function taking two arguments, the current count of bytes copied and the total number of bytes to be copied. It will be called once at the beginning of the copy operation (with count == 0), once at the end (with count == total), and in between about once per second, if the copy takes long enough. Note that the total number of bytes passed to the callback is approximate, and that the count may be larger than the total if the estimate was off. To abort the copy, raise an exception from the callback function.

Parameters

- **source_path** – The path to the source file.
- **target_path** – A path to copy it to.
- **overwrite** – Selects behaviour when the target exists.
- **copy_into** – Whether to copy into target directories.
- **copy_permissions** – Whether to copy permissions along.
- **callback** – A callback function to call regularly with progress reports.

cerulean.**logger = <Logger cerulean (WARNING)>**

The Cerulean root logger. Use this to set Cerulean's log level.

In particular, if something goes wrong and you want more debug output, you can do:

```
import logging

cerulean.logger.setLevel(logging.INFO)
```

or for even more:

```
cerulean.logger.setLevel(logging.DEBUG)
```

cerulean.**make_file_system**(*protocol: str*, *location: Optional[str] = None*, *credential: Optional[cerulean.credential.Credential] = None*) → cerulean.file_system.FileSystem

Make a file system object.

This is a factory function for FileSystem objects. It will instantiate a FileSystem implementation according to the parameters you give it.

FileSystems may hold resources, so you should either use this function with a `with` statement, or call `close()` on the returned object when you are done with it.

Parameters

- **protocol** – The protocol to use to connect to the file system. Can be *local*, *sftp* or *webdav*. For *local*, location and credential can be omitted. For *webdav*, credential can be omitted.
- **location** – The location in the form *hostname*, *hostname:port* or *http(s)://hostname:port/base_path* to connect to.
- **credential** – The *Credential* to use to connect with.

Returns An instance of a FileSystem representing the described file system.

cerulean.**make_terminal**(*protocol: str*, *location: Optional[str] = None*, *credential: Optional[cerulean.credential.Credential] = None*) → cerulean.terminal.Terminal

Make a terminal object.

This is a factory function for Terminal objects. It will instantiate a Terminal implementation according to the parameters you give it.

Terminals may hold resources, so you should either use this function with a `with` statement, or call `close()` on the returned object when you are done with it.

Parameters

- **protocol** – The protocol to use to connect to the file system. Can be *local* or *sftp*. For *local*, location and credential can be omitted.

- **location** – The location in the form *hostname* or *hostname:port* to connect to.
- **credential** – The *Credential* to use to connect with.

Returns An instance of a FileSystem representing the described file system.

cerulean.**make_scheduler**(*name: str*, *terminal: cerulean.terminal.Terminal*, *prefix: str = ''*) → cerulean.scheduler.Scheduler

Make a scheduler object.

This is a factory function for Scheduler objects. It will instantiate a Scheduler implementation according to the parameters you give it, which talks to the supplied Terminal.

Parameters

- **name** – The name of the scheduler. One of `directgnu`, `slurm`, or `torque`.
- **terminal** – The terminal this Scheduler will communicate on.
- **prefix** – A string to prefix any shell commands with.

Returns The Scheduler.

class cerulean.**Credential**

Bases: `abc.ABC`

A credential for connecting to remote machines.

Credentials don't have much in common other than a username, which is best modelled as a public attribute. So this interface is empty, and only here to provide a generic type to represent any credential in the API.

username
The name of the user to connect as.

class cerulean.**EntryType**

Bases: `enum.Enum`

An enumeration.

BLOCK_DEVICE = 5

CHARACTER_DEVICE = 4

DIRECTORY = 1

FIFO = 6

FILE = 2

SOCKET = 7

SYMBOLIC_LINK = 3

class cerulean.**PasswordCredential**(*username: str*, *password: str*)

Bases: `cerulean.credential.Credential`

A credential comprising a username and password.

username
The name of the user to connect as

password
The password to authenticate with

class cerulean.**Permission**

Bases: `enum.Enum`

An enumeration.

GROUP_EXECUTE = 8

GROUP_READ = 32

GROUP_WRITE = 16

OTHERS_EXECUTE = 1

OTHERS_READ = 4

OTHERS_WRITE = 2

OWNER_EXECUTE = 64

OWNER_READ = 256

OWNER_WRITE = 128

SGID = 1024

STICKY = 512

SUID = 2048

class cerulean.**PubKeyCredential**(*username: str*, *public_key: str*, *passphrase: str = None*)
Bases: cerulean.credential.Credential

A credential using a public/private key pair.

username
The name of the user to connect as

public_key
The (local) path to a key file

passphrase
The passphrase to decrypt the key with; optional.

class cerulean.**DirectGnuScheduler**(*terminal: cerulean.terminal.Terminal*, *prefix: str = ''*)
Bases: cerulean.scheduler.Scheduler

A scheduler that runs processes directly on a GNU system.

This scheduler does not have a queue, instead it launches each job immediately as a process, and uses ps and kill to manage it. This should work fine on any normal GNU/Linux system, but in some cases you may need an extra command to make bash, ps and/or kill available (e.g. setting a PATH). If so, you can specify prefix, and it will be prepended onto these commands. Note that this is a simple string concatenation, so you may need a semicolon at the end depending your exact prefix command.

cancel(*job_id: str*) → None
Cancel a running job.

Submits a cancellation request for a job to the scheduler.

Parameters **job_id** – Id of the job to be cancelled.

get_exit_code(*job_id: str*) → Optional[int]
Get the exit code of a finished job.

Once a job is done, its exit code may be requested using this method. If the job is still running or failed to start, then there is no exit code, and None will be returned.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The exit code, or None if there is none.

get_status (*job_id: str*) → cerulean.job_status.JobStatus
Look up the status of a job.

This method is used to check if a job is still in the queue, running, or done.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The status of the job as a *JobStatus*

submit (*job_description: cerulean.job_description.JobDescription*) → str
Submit a job for execution.

Parameters **job_description** – A description of the job to run.

Returns A job id that can be used to keep track of it.

class cerulean.**FileSystem**
Bases: abc.ABC

Represents a file system.

This is a generic interface class that all file systems inherit from, so you can use it wherever any file system will do.

In order to do something useful, you'll want an actual file system, like a *LocalFileSystem* or an *SftpFileSystem*.

FileSystems may hold resources, so you should either use them with a `with` statement, or call *close()* on the returned object when you are done with it.

Beyond that, file systems support a single operation:

```
fs / 'path'
```

which produces a *Path*, through which you can do things with files.

close() → None
Close connections and free resources, if any.

FileSystem objects may hold resources that need to be freed when you are done with the object. You can free them by calling this function, or you can use the FileSystem as a context manager using a `with` statement.

root() → cerulean.path.Path
Returns a Path representing the root of the file system.

class cerulean.**JobDescription**
Bases: object

Describes a job to submit to a scheduler.

name
The name of the job, with which it will show up in the scheduler's queue. Cerulean does not use the name, but it may be useful if you manually check the queue.

Type str

working_directory
The working directory to execute in.

Type str

environment
A dictionary of environment variables to define, and their values.

Type Dict[str, str]

command
The command to execute.

Type str

arguments
A list of arguments to pass. If needed, you need to add quotes yourself, the arguments will not be escaped by cerulean.

Type list of str

stdout_file
File to direct standard output to.

Type str

stderr_file
File to direct standard error to.

Type str

queue_name
Name of the queue to submit to.

Type str

time_reserved
Time to reserve, in seconds.

Type int

num_nodes
The number of nodes to reserve.

Type int

mpi_processes_per_node
Number of MPI processes to start per node.

Type int

system_out_file
File to direct the standard output of the scheduler to.

Type str

system_err_file
File to direct the standard error of the scheduler to.

Type str

extra_scheduler_options
Additional options to add to the scheduler command line on job submission. Note that these are scheduler-specific!

Type str

Note that stdout_file and stderr_file will receive the output of the process y

Type e.g. that the job ran out of its time limit and was killed

class cerulean.**JobStatus**
Bases: enum.Enum

An enumeration.

DONE = 3

RUNNING = 2

WAITING = 1

class cerulean.**LocalFileSystem**

Bases: cerulean.file_system_impl.FileSystemImpl

Represents the local file system.

To create an instance, just call *LocalFileSystem()*.

LocalFileSystem support a single operation:

```
fs / 'path'
```

which produces a *Path*, through which you can do things with local files.

LocalFileSystem is a context manager, so you can use it in a `with` statement, and it has a `close()` method, but since it doesn't hold any resources, you do not need to use them. It may be good to do so anyway, to avoid leaks if you ever replace it with a different *FileSystem* that does.

root() → cerulean.path.Path

Returns a Path representing the root of the file system.

class cerulean.**LocalTerminal**

Bases: cerulean.terminal.Terminal

A Terminal for running commands on the local machine.

To create one, just do `term = LocalTerminal()`.

run(*timeout: float, command: str, args: List[str], stdin_data: str = None, workdir: str = None*) → Tuple[Optional[int], str, str]

Run a shell command.

The command will be run in the default shell, and arguments are **not** quoted automatically. If you have untrusted or unknown input, be sure to quote it using *quote()* from the *shlex* module of the Python standard library.

Parameters

- **timeout** – How long to wait for the result(s)
- **command** – The command to run.
- **args** – A list of arguments to pass
- **stdin_data** – Data to pass to standard input
- **workdir** – Working directory to execute in

Returns A tuple containing the exit code, standard output, and standard error output.

class cerulean.**Path**(*filesystem: FileSystemImpl, path: Union[pathlib.Path, pathlib.PurePosixPath, pathlib.PureWindowsPath]*)

Bases: object

A path on a file system.

This class implements the pathlib.PurePosixPath interface fully, and a pathlib.PosixPath-like interface, although it has some omissions, additions, and improvements to make it more compatible with remote and non-standard file systems.

To make a Path, create a FileSystem first, then use the / operator on it, e.g. fs / 'home' / 'user'. Do not construct objects of this class directly.

Paths can be compared for (non-)equality using == and !=. Paths that compare unequal could still refer to the same file, if it is accessible in multiple ways. For instance, a local path /tmp would compare unequal to a path /tmp on an SftpFileSystem, even if the SftpFileSystem is connected to localhost, and the paths do in fact refer to the same directory.

filesystem
The file system that this path is on.

anchor
The concatenation of the drive and the root.

as_posix() → str
Returns the path as a string with forward slashes.

as_uri() → str
Returns a URI representing the path.

This is not yet implemented, please file an issue if you need it.

chmod(*mode: int*) → None
Sets permissions.

Parameters **mode** – The numerical mode describing the permissions to set. This uses standard POSIX mode definitions, see man chmod.

drive
The drive letter (including the colon), if any.

entry_type() → cerulean.path.EntryType
Returns the kind of directory entry type the path points to.

Returns An *EntryType* enum value describing the filesystem entry.

Raises FileNotFoundError – If there is no file here.

exists() → bool
Returns true iff a filesystem object exists at this path.

If the path denotes a symlink, returns whether the symlink points to an existing filesystem object, recursively. If the symlink is part of a link loop, returns False.

Returns True iff the path exists on the filesystem.

gid() → Optional[int]
Returns the group id associated with the object.

Returns An integer with the id, or None of not supported.

has_permission(*permission: cerulean.path.Permission*) → bool
Checks permissions.

Parameters **permission** – A particular file permission, see Permission

Returns True iff the object exists and has the given permission.

is_absolute() → bool
Returns whether the path is absolute.

is_dir() → bool
Returns whether the path is a directory.

Returns True iff the path exists and is a directory, or a symbolic link pointing to a directory.

is_file() → bool
Returns whether the path is a file.

Returns True iff the path exists and is a file, or a symbolic link pointing to a file.

is_reserved() → bool
Return whether the path is reserved.

This can only happen on Windows on a LocalFileSystem.

is_symlink() → bool
Returns whether the path is a symlink.

Returns True iff the path exists and is a symbolic link.

iterdir() → Generator[cerulean.path.Path, None, None]
Iterates through a directory's contents.

Yields Paths of entries in the directory.

joinpath(**other*) → cerulean.path.Path
Joins another path or string onto the back of this one.

Parameters **other** – The other path to append to this one.

Returns The combined path.

mkdir(*mode: Optional[int] = None*, *parents: bool = False*, *exists_ok: bool = False*) → None
Makes a directory with the given access rights.

If mode is not set or None, assigns permissions according to the current umask. If parents is True, makes parent directories as needed. If exists_ok is True, silently ignores if the directory already exists.

Parameters

- **mode** – A numerical Posix access permissions mode.
- **parents** – Whether to make parent directories.
- **exists_ok** – Don't raise if target already exists.

name
The name of the file or directory.

This excludes parents but includes the suffix.

parent
The logical parent of the path.

parents
A sequence containing the logical ancestors of the path.

parts
A tuple containing the path's components.

read_bytes() → bytes
Reads file contents as a bytes object.

Returns The contents of the file.

read_text(*encoding: str = 'utf-8'*) → str
Reads file contents as a string.

Assumes UTF-8 encoding.

Parameters **encoding** – The encoding to assume.

Returns The contents of the file.

readlink(*recursive: bool = False*) → cerulean.path.Path
Reads the target of a symbolic link.

Note that the result may be a relative path, which should then be taken relative to the directory containing the link.

If recursive is True, this function will follow a chain of symlinks until it reaches something that is not a symlink, or until the maximum recursion depth is reached and a RunTimeError is raised.

Parameters **recursive** – Whether to resolve recursively.

Returns The path that the symlink points to.

Raises RunTimeError – The recursion depth was reached, probably as a result of a link loop.

relative_to(**other*) → cerulean.path.Path
Returns a version of this path relative to another path.

Both paths must be on the same file system.

Parameters **other** – The path to use as a reference.

remove() → None
Removes a file, link, device or directory.

Directories are removed recursively, links, devices and files are deleted. Link targets are left in place. Returns without error if there is nothing there already.

Use this method to ensure that there is no entry at this path.

rename(*target: cerulean.path.Path*) → None
Renames a file.

The new path must be in the same filesystem. If the new path exists, then it will be overwritten.

Parameters **target** – The new path of the file.

rmdir(*recursive: bool = False*) → None
Removes a directory.

If recursive is True, remove all files and directories inside as well. If recursive is False, the directory must be empty.

root
A string representing the root of the filesystem.

set_permission(*permission: cerulean.path.Permission*, *value: bool = True*) → None
Sets permissions.

Parameters

- **permission** – The permission to set.
- **value** – Whether to enable or disable the permission.

size() → int
Returns the size of the file.

Returns An integer with the number of bytes in the file.

stem
The stem of this path.

This is the name of the file or directory, excluding parents and excluding the suffix.

streaming_read() → Generator[bytes, None, None]
Streams data from a file.

This is a generator function that generates bytes objects containing consecutive chunks of the file.

streaming_write(*data: Iterable[bytes]*) → None
Streams data to a file.

Creates a new file (overwriting any existing file) at the current path, and writes data to it from the given iterable.

Parameters **data** – An iterable of bytes containing data to be written.

suffix
The file extension of the file or directory, if any.

suffixes
A list of all the extensions in the file name.

symlink_to(*target: cerulean.path.Path*) → None
Makes a symlink from the current path to the target.

If this raises an OSError with the message Failed, then the problem may be that the target does not exist.

Parameters **target** – The path to symlink to.

Raises FileExistsError – if you try to overwrite an existing entry with a symlink.

touch() → None
Updates the access and modification times of file.

If the file does not exist, it will be created, which is often what this function is used for.

uid() → Optional[int]
Returns the user id of the owner of the object.

Returns An integer with the id, or None if not supported.

unlink() → None
Removes a file or device node.

For removing directories, see rmdir().

walk(*topdown: bool = True, onerror: Optional[Callable[[OSError], None]] = None, followlinks: bool = False*) → Generator[Tuple[cerulean.path.Path, List[str], List[str]], None, None]
Walks a directory hierarchy recursively.

This is a version of Python's os.walk() function adapted to be a little bit more pathlib-like. It walks the directory and its subdirectories, yielding a tuple (dirpath, dirnames, filenames) for each directory. These are the path of the directory, as a *Path*, a list of strings containing the names of the subdirectories inside this directory, and a list of strings containing the names of the non-directories in this directory respectively.

If topdown is True, the triple for a directory will be produced before the triples of its subdirectories; if it is False, it will be produced after (pre- and post-order traversal respectively).

If onerror is set, the function it is set to will be called if an error occurs, and passed an instance of OSError. The callback can handle the error in some way, or raise it to end the traversal. The OSError object will have an attribute filename containing the name of the file that triggered the problem as a string.

If followlinks is True, this function will recurse into symlinks that point to directories, if it is False, it will silently skip them.

Yields Tuples (dirpath, dirnames, filenames), as above.

with_name (*name: str*) → cerulean.path.Path
Return a new path with the last component set to *name*.

Parameters **name** – The new name to use.

with_suffix (*suffix: str*) → cerulean.path.Path
Return a new path with the suffix set to *suffix*

Parameters **suffix** – The new suffix to use.

write_bytes (*data: bytes*) → None
Writes bytes to the file.

Overwrites the file if it exists.

Parameters **data** – The data to be written.

write_text (*text: str*, *encoding: str = 'utf-8'*) → None
Writes text to a file.

Overwrites the file if it exists.

Parameters

- **text** – The text to be written.
- **encoding** – The encoding to use.

class cerulean.**Scheduler**
Bases: abc.ABC

Interface for job schedulers.

To run jobs using a scheduler, you will want to use *SlurmScheduler* or *TorqueScheduler*.

cancel (*job_id: str*) → None
Cancel a running job.

Submits a cancellation request for a job to the scheduler.

Parameters **job_id** – Id of the job to be cancelled.

get_exit_code (*job_id: str*) → Optional[int]
Get the exit code of a finished job.

Once a job is done, its exit code may be requested using this method. If the job is still running or failed to start, then there is no exit code, and None will be returned.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The exit code, or None if there is none.

get_status (*job_id: str*) → cerulean.job_status.JobStatus
Look up the status of a job.

This method is used to check if a job is still in the queue, running, or done.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The status of the job as a *JobStatus*

submit (*job_description: cerulean.job_description.JobDescription*) → str
Submit a job for execution.

Parameters **job_description** – A description of the job to run.

Returns A job id that can be used to keep track of it.

wait (*job_id: str*, *time_out: float = -1.0*, *interval: float = None*) → Optional[int]
Wait until the job is done.

Will wait approximately time_out seconds for the job to finish. Returns the exit code if the job finished, otherwise None.

The state of the job will be checked every *interval* seconds. If *interval* is None, or not specified, then the interval will be 1s, or time_out / 50, whichever is larger. If time_out is not given, the interval will start at 1s, then increase gradually to about 30s.

Parameters

- **job_id** – The job to wait for.
- **time_out** – Time to wait in seconds. If negative, wait forever.
- **interval** – Time to wait between checks. See above.

Returns The exit code of the job.

class cerulean.**SftpFileSystem**(*terminal: cerulean.ssh_terminal.SshTerminal*, *own_term: bool = False*)
Bases: cerulean.file_system_impl.FileSystemImpl

A FileSystem implementation that connects to an SFTP server.

SftpFileSystem supports the / operation:

```
fs / 'path'
```

which produces a *Path*, through which you can do things with the remote files.

It is also a context manager, so that you can (and should!) use it with a `with` statement, which will ensure that the connection is closed when you are done with the it. Alternatively, you can call *close()* to close the connection.

If *own_term* is True, this class assumes that it owns the terminal you gave it, and that it is responsible for closing it when it's done with it. If you share an SshTerminal between an SftpFileSystem and a scheduler, or use the terminal directly yourself, then you want to use False here, and close the terminal yourself when you don't need it any more.

Parameters

- **terminal** – The terminal to connect through.
- **own_term** – Whether to close the terminal when the file system is closed.

close() → None
Close connections and free resources, if any.

FileSystem objects may hold resources that need to be freed when you are done with the object. You can free them by calling this function, or you can use the FileSystem as a context manager using a `with` statement.

root() → cerulean.path.Path
Returns a Path representing the root of the file system.

class cerulean.**SlurmScheduler**(*terminal: cerulean.terminal.Terminal*, *prefix: str = ''*)
Bases: cerulean.scheduler.Scheduler

Represents a Slurm scheduler.

This class represents a Slurm scheduler, to which it talks through a *Terminal*.

On some machines, an additional command is needed to make Slurm available to the user, e.g. 'module load slurm'. If you specify a prefix, it will be prepended to any Slurm command run by this class. Note that this is a plain string concatenation, so you'll probably need something like 'module load slurm;', with a semicolon to separate the commands.

cancel (*job_id: str*) → None
Cancel a running job.

Submits a cancellation request for a job to the scheduler.

Parameters **job_id** – Id of the job to be cancelled.

get_exit_code (*job_id: str*) → Optional[int]
Get the exit code of a finished job.

Once a job is done, its exit code may be requested using this method. If the job is still running or failed to start, then there is no exit code, and None will be returned.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The exit code, or None if there is none.

get_status (*job_id: str*) → cerulean.job_status.JobStatus
Look up the status of a job.

This method is used to check if a job is still in the queue, running, or done.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The status of the job as a *JobStatus*

submit (*job_description: cerulean.job_description.JobDescription*) → str
Submit a job for execution.

Parameters **job_description** – A description of the job to run.

Returns A job id that can be used to keep track of it.

class cerulean.**SshTerminal** (*host: str*, *port: int*, *credential: cerulean.credential.Credential*)
Bases: cerulean.terminal.Terminal

A terminal that runs commands over SSH.

This terminal connects to a host using SSH, then lets you run commands there.

Parameters

- **host** – The hostname to connect to.
- **port** – The port to connect on.
- **credential** – The credential to authenticate with.

close () → None
Close the terminal.

This closes any connections and frees resources associated with the terminal.

run (*timeout: float, command: str, args: List[str], stdin_data: str = None, workdir: str = None*) → Tuple[Optional[int], str, str]
Run a shell command.

The command will be run in the default shell, and arguments are **not** quoted automatically. If you have untrusted or unknown input, be sure to quote it using *quote()* from the *shlex* module of the Python standard library.

Parameters

- **timeout** – How long to wait for the result(s)
- **command** – The command to run.
- **args** – A list of arguments to pass
- **stdin_data** – Data to pass to standard input
- **workdir** – Working directory to execute in

Returns A tuple containing the exit code, standard output, and standard error output.

class cerulean.**Terminal**
Bases: abc.ABC

Interface for Terminals.

This is a generic interface class that all terminals inherit from, so you can use it wherever any terminal will do.

In order to do something useful, you'll want an actual terminal, like a *LocalTerminal* or an *SshTerminal*.

Terminals may hold resources, so you should either use them with a `with` statement, or call *close()* on them when you are done with them.

close() → None
Close the terminal.

This closes any connections and frees resources associated with the terminal. *LocalTerminal* does not require this, but terminals that connect to remote machines do. You may want to always either close a Terminal, or use it as a context manager, to avoid problems if you ever change from a local terminal to a remote one.

run(*timeout: float, command: str, args: List[str], stdin_data: str = None, workdir: str = None*) → Tuple[Optional[int], str, str]
Run a shell command.

The command will be run in the default shell, and arguments are **not** quoted automatically. If you have untrusted or unknown input, be sure to quote it using *quote()* from the *shlex* module of the Python standard library.

Parameters

- **timeout** – How long to wait for the result(s)
- **command** – The command to run.
- **args** – A list of arguments to pass
- **stdin_data** – Data to pass to standard input
- **workdir** – Working directory to execute in

Returns A tuple containing the exit code, standard output, and standard error output.

class cerulean.**TorqueScheduler**(*terminal: cerulean.terminal.Terminal*, *prefix: str = ''*)
Bases: cerulean.scheduler.Scheduler

Represents a Torque scheduler.

This class represents a Torque scheduler, to which it talks through a *Terminal*.

On some machines, an additional command is needed to make Torque available to the user, e.g. 'module load torque'. If you specify a prefix, it will be prepended to any Torque command run by this class. Note that this is a plain string concatenation, so you'll probably need something like 'module load torque;', with a semicolon to separate the commands.

Parameters

- **terminal** – The terminal to use to talk to the scheduler.
- **prefix** – A string to prefix the Torque commands with.

cancel (*job_id: str*) → None
Cancel a running job.

Submits a cancellation request for a job to the scheduler.

Parameters **job_id** – Id of the job to be cancelled.

get_exit_code (*job_id: str*) → Optional[int]
Get the exit code of a finished job.

Once a job is done, its exit code may be requested using this method. If the job is still running or failed to start, then there is no exit code, and None will be returned.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The exit code, or None if there is none.

get_status (*job_id: str*) → cerulean.job_status.JobStatus
Look up the status of a job.

This method is used to check if a job is still in the queue, running, or done.

Parameters **job_id** – A job id string obtained from *submit()*.

Returns The status of the job as a *JobStatus*

submit (*job_description: cerulean.job_description.JobDescription*) → str
Submit a job for execution.

Parameters **job_description** – A description of the job to run.

Returns A job id that can be used to keep track of it.

exception cerulean.**UnsupportedOperationError**
Bases: RuntimeError

Raised when an unsupported method is called.

See *WebdavFileSystem*.

class cerulean.**WebdavFileSystem**(*url: str*, *credential: Optional[cerulean.credential.Credential] = None*, *host_ca_cert_file: Optional[str] = None*, *unsupported_methods_raise: Optional[bool] = True*)
Bases: cerulean.file_system_impl.FileSystemImpl

A FileSystem implementation that connects to a WebDAV server.

WebdavFileSystem supports the / operation:

```
fs / 'path'
```

which produces a *Path*, through which you can do things with the remote files.

It is also a context manager, so that you can (and should!) use it with a `with` statement, which will ensure that the connection is closed when you are done with the it. Alternatively, you can call *close()* to close the connection.

The WebDAV protocol does not support all operations specified by the Cerulean API. In particular, symbolic links are not supported, nor are ownership and permissions. Read-access to these properties is emulated, e.g. *is_symlink()* simply always returns false, all files and directories are owned by uid 0 and gid 0, with access permissions determined by whether the server will let us access them.

By default, if you try to run any of the related modifying methods, e.g. *symlink_to()* or *set_permissions()*, an *UnsupportedOperationError* will be raised. If you set *unsupported_methods_raise* to *False* when creating a WebdavFileSystem, then these methods will simply return without doing anything.

WebdavFileSystem supports both HTTP and HTTPS, but not (yet) client-side certificates.

Parameters

- **url** – The server base location, e.g. http://example.com/webdav
- **credential** – The credential to use to connect.
- **host_ca_cert_file** – Path to a certificate file to use for authentication. Useful for servers that use a self-signed certificate.
- **unsupported_methods_raise** – Raise on using an unsupported method, see above.

close() → None

Close connections and free resources, if any.

FileSystem objects may hold resources that need to be freed when you are done with the object. You can free them by calling this function, or you can use the FileSystem as a context manager using a `with` statement.

root() → cerulean.path.Path

Returns a Path representing the root of the file system.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index

W