Rez Documentation

Release 2.109.0

Allan Johns

Apr 21, 2022

CONTENTS

CHAPTER

ONE

API DOCUMENTATION

1.1 Rez

1.1.1 Subpackages

rez._sys package

Submodules

rez._sys._setup module

Module contents

rez.bind package

Submodules

rez.bind.arch module

Creates the system architecture package.

rez.bind.arch.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.cmake module

Binds a cmake executable as a rez package.

rez.bind.cmake.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.cmake.**commands**()

rez.bind.cmake.**setup_parser**(*parser*)

rez.bind.hello_world module

Creates the 'hello_world' testing package.

Note: Even though this is a python-based package, it does not list python as a requirement. This is not typical! This package is intended as a very simple test case, and for that reason we do not want any dependencies.

rez.bind.hello_world.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.hello_world.**commands**()

rez.bind.hello_world.**hello_world_source**()

rez.bind.os module

Creates the operating system package.

rez.bind.os.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.platform module

Creates the system platform package.

rez.bind.platform.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.python module

Binds a python executable as a rez package.

rez.bind.python.**bind**(*path*, *version_range=None*, *opts=None*, *parser=None*)

rez.bind.python.**commands**()

rez.bind.python.**post_commands**()

rez.bind.python.**setup_parser**(*parser*)

Module contents

rez.cli package

Submodules

rez.cli._bez module

rez.cli._main module

The main command-line entry point.

class rez.cli._main.**InfoAction**(*option_strings*, *dest*, *default=False*, *required=False*, *help=None*)

Bases: argparse._StoreTrueAction

class rez.cli._main.**SetupRezSubParser**(*module_name*)

Bases: object

Callback class for lazily setting up rez sub-parsers.

get_module()

rez.cli._main.**is_hyphened_command**()

rez.cli._main.**run**(*command=None*)

rez.cli._main.**setup_parser**()

Create and setup parser for given rez command line interface.

Returns Argument parser for rez command.

Return type *LazyArgumentParser*

rez.cli._util module

class rez.cli._util.**LazyArgumentParser**(**args*, ***kwargs*)

Bases: argparse.ArgumentParser

ArgumentParser sub-class which accepts an additional *setup_subparser* argument for lazy setup of sub-parsers.

setup_subparser is passed 'parser_name', 'parser', and can return a help string.

format_help()

Sets up all sub-parsers when help is requested.

class rez.cli._util.**LazySubParsersAction**(*option_strings*, *prog*, *parser_class*, *dest='==SUPPRESS=='*, *required=False*, *help=None*, *metavar=None*)

Bases: argparse._SubParsersAction

Argparse Action which calls the *setup_subparser* function provided to *LazyArgumentParser*.

rez.cli._util.**load_plugin_cmd**()

Load subcommand from command type plugin

The command type plugin module should have attribute *command_behavior*, and the value must be a dict if provided. For example:

in your command plugin module command_behavior = {

"hidden": False, # (bool): default False "arg_mode": None, # (str): "passthrough", "grouped", default None

}

If the attribute not present, default behavior will be given.

rez.cli._util.**print_items**(*items*, *stream=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

rez.cli._util.**sigbase_handler**(*signum*, *frame*)

rez.cli._util.**sigint_handler**(*signum*, *frame*)

Exit gracefully on ctrl-C.

rez.cli._util.**sigterm_handler**(*signum*, *frame*)

Exit gracefully on terminate.

rez.cli.bind module

Create a Rez package for existing software.

rez.cli.bind.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.bind.**setup_parser**(*parser*, *completions=False*)

rez.cli.bootstrap module

rez.cli.build module

Build a package from source.

rez.cli.build.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.build.**get_build_args**(*opts*, *parser*, *extra_arg_groups*)

rez.cli.build.**get_current_developer_package**()

rez.cli.build.**setup_parser**(*parser*, *completions=False*)

rez.cli.build.**setup_parser_common**(*parser*)

Parser setup common to both rez-build and rez-release.

rez.cli.context module

Print information about the current rez context, or a given context file.

rez.cli.context.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.context.**setup_parser**(*parser*, *completions=False*)

rez.cli.env module

Open a rez-configured shell, possibly interactive.

rez.cli.env.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.env.**setup_parser**(*parser*, *completions=False*)

rez.cli.forward module

==SUPPRESS==

rez.cli.forward.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.forward.**setup_parser**(*parser*, *completions=False*)

rez.cli.interpret module

Execute some Rex code and print the interpreted result.

rez.cli.interpret.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.interpret.**setup_parser**(*parser*, *completions=False*)

rez.cli.release module

Build a package from source and deploy it.

rez.cli.release.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.release.**setup_parser**(*parser*, *completions=False*)

rez.cli.settings module

rez.cli.suite module

Manage a suite or print information about an existing suite.

rez.cli.suite.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.suite.**setup_parser**(*parser*, *completions=False*)

rez.cli.test module

Run tests listed in a package's definition file.

rez.cli.test.**command**(*opts*, *parser*, *extra_arg_groups=None*)

rez.cli.test.**setup_parser**(*parser*, *completions=False*)

rez.cli.tools module

Module contents

rez.tests package

Submodules

rez.tests.test_build module

test the build system

class rez.tests.test_build.**TestBuild**(**nargs*, ***kwargs*)

Bases: *rez.tests.util.TestBase*, *rez.tests.util.TempdirMixin*

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

test_build_cmake()

Test a cmake-based package.

test_build_custom()

Test a make-based package that uses the custom_build attribute.

test_build_whack(*shell*)

Test that a broken build fails correctly.

test_builds(*shell*)

Test an interdependent set of builds.

test_builds_anti(*shell*)

Test we can build packages that contain anti packages

rez.tests.test_commands module

test package commands

class rez.tests.test_commands.**TestCommands**(*fn*)

Bases: *rez.tests.util.TestBase*

classmethod get_packages_path()

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_2()

Resolve a package with a dependency, see that their commands are concatenated as expected.

test_new_yaml()

Resolve a yaml-based package with new rex commands.

test_old_yaml()

Resolve a yaml-based package with old-style bash commands.

test_py()

Resolve a new py-based package with rex commands.

rez.tests.test_context module

test resolved contexts

class rez.tests.test_context.**TestContext**(**nargs*, ***kwargs*)

Bases: *rez.tests.util.TestBase*, *rez.tests.util.TempdirMixin*

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

classmethod tearDownClass()

Hook method for deconstructing the class fixture after running all tests in the class.

test_apply()

Test apply() function.

test_bundled()

Test that a bundled context behaves identically.

test_create_context()

Test creation of context.

test_execute_command()

Test command execution in context.

test_execute_command_environ()

Test that execute_command properly sets environ dict.

test_retarget()

Test that a retargeted context behaves identically.

test_serialize()

Test context serialization.

rez.tests.test_formatter module

test rex string formatting

class rez.tests.test_formatter.**TestFormatter**(**nargs*, ***kwargs*)

Bases: *rez.tests.util.TestBase*

assert_formatter_equal(*format*, *expected*, **args*, ***kwargs*)

assert_formatter_raises(*format*, *error*, **args*, ***kwargs*)

setUp()

Hook method for setting up the test fixture before exercising it.

test_formatter_recurse()

test_formatter_rex()

test_formatter_stdlib()

string.Formatter.format tests from the Python standard library used to ensure we haven't broken functionality preset in the standard implementation of the Formatter object.

rez.tests.test_rex module

test the rex command generator API

class rez.tests.test_rex.**TestRex**(**nargs*, ***kwargs*)

Bases: *rez.tests.util.TestBase*

test_1()

Test simple use of every available action.

test_10()
Test env __contains__ and __bool__

test_2()
Test simple setenvs and assignments.

test_3()
Test appending/prepending.

test_4()
Test control flow using internally-set env vars.

test_5()
Test control flow using externally-set env vars.

test_6()
Test variable expansion.

test_7()
Test exceptions.

test_8()
Custom environment variable separators.

test_9()
Test literal and expandable strings.

test_intersects_ephemerals()
Test intersects with ephemerals object

test_intersects_request()
Test intersects with request object

test_intersects_resolve()
Test intersects with resolve object

test_old_style_commands()
Convert old style commands to rex

test_version_binding()
Test the Rex binding of the Version class.

rez.tests.test_shells module

test shell invocation

class rez.tests.test_shells.**TestShells**(**nargs*, ***kwargs*)
Bases: *rez.tests.util.TestBase*, *rez.tests.util.TempdirMixin*

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

classmethod tearDownClass()
Hook method for deconstructing the class fixture after running all tests in the class.

test_aaa_shell_presence()

Ensure specific shell types are present as loaded plugins.

The env var _REZ_ENSURE_TEST_SHELLS should be set by a CI system (such as github actions) to make sure the shells we expect to be installed, are installed, and are getting tested.

Note 'aaa' forces unittest to run this test first.

test_alias_command(*shell*)

Testing alias can be passed in as command

This is important for Windows CMD shell because the doskey.exe isn't executed yet when the alias is being passed.

test_alias_command_with_args(*shell*)

Testing alias can be passed in as command with args

This is important for Windows CMD shell because the doskey.exe isn't executed yet when the alias is being passed.

test_command(*shell*)

test_command_returncode(*shell*)

test_create_executable_script()

test_no_output(*shell*)

test_norc(*shell*)

test_rcfile(*shell*)

test_rex_code(*shell*)

Test that Rex code run in the shell creates the environment variable values that we expect.

test_rex_code_alias(*shell*)

Ensure PATH changes do not influence the alias command.

This is important for Windows because the doskey.exe might not be on the PATH anymore at the time it's executed. That's why we figure out the absolute path to doskey.exe before we modify PATH and continue to use the absolute path after the modifications.

test_rez_command(*shell*)

test_rez_env_output(*shell*)

test_stdin(*shell*)

rez.tests.test_solver module

test dependency resolving algorithm

class rez.tests.test_solver.**TestSolver**(**nargs*, ***kwargs*)

Bases: *rez.tests.util.TestBase*

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

test_01()

Extremely basic solves involving a single package.

test_02()

Basic solves involving a single package.

test_03()

Failures in the initial request.

test_04()

Basic failures.

test_05()

More complex failures.

test_06()

Basic solves involving multiple packages.

test_07()

More complex solves.

test_08()

Cyclic failures.

test_09_version_priority_mode()

test_10_intersection_priority_mode()

test_11_variant_splitting()

rez.tests.util module

class rez.tests.util.**TempdirMixin**

Bases: object

Mixin that adds tmpdir create/delete.

classmethod setUpClass()

classmethod tearDownClass()

class rez.tests.util.**TestBase**(**nargs*, ***kwargs*)

Bases: unittest.case.TestCase

Unit test base class.

classmethod data_path(**dirs*)

Get path to test data.

get_settings_env()

Get an environ dict that applies the current settings.

This is required for cases where a subproc has to pick up the same config settings that the test case has set.

setUp()

Hook method for setting up the test fixture before exercising it.

classmethod setUpClass()

Hook method for setting up class fixture before running tests in the class.

setup_config()

setup_once()

tearDown()

Hook method for deconstructing the test fixture after testing it.

teardown_config()

update_settings(*new_settings*, *override=False*)

Can be called within test methods to modify settings on a per-test basis (as opposed cls.settings, which modifies it for all tests on the class)

Note that multiple calls will not "accumulate" updates, but will instead patch the class's settings with the new_settings each time.

new_settings [dict] the updated settings to override the config with

override [bool] normally, the resulting config will be the result of merging the base cls.settings with the new_settings - ie, like doing cls.settings.update(new_settings). If this is True, however, then the cls.settings will be ignored entirely, and the new_settings will be the only configuration settings applied

rez.tests.util.**find_file_in_path**(*to_find*, *path_str*, *pathsep=None*, *reverse=True*)

Attempts to find the given relative path to_find in the given path

rez.tests.util.**install_dependent**()

Function decorator that skips tests if not run via 'rez-selftest' tool, from a production install

rez.tests.util.**per_available_shell**(*exclude=None*)

Function decorator that runs the function over all available shell types.

rez.tests.util.**program_dependent**(*program_name*, **program_names*)

Function decorator that skips the function if not all given programs are visible.

rez.tests.util.**restore_os_environ**()

Encapsulate changes to os.environ and return to the original state.

This context manager lets you wrap modifications of os.environ and not worry about reverting back to the original.

Examples

```
>>> key = 'ARBITRARY_KEY'
>>> value = 'arbitrary_value'
>>> with os_environ():
>>>     os.environ[key] = value
>>>     assert key in os.environ
True
```

```
>>> assert key in os.environ
False
```

Yields *dict* – The original os.environ.

rez.tests.util.**restore_sys_path**()

Encapsulate changes to sys.path and return to the original state.

This context manager lets you wrap modifications of sys.path and not worry about reverting back to the original.

Examples

```
>>> path = '/arbitrary/path'
>>> with sys_path():
>>>     sys.path.insert(0, '/arbitrary/path')
>>>     assert path in sys.path
True
```

```
>>> assert path in sys.path
False
```

Yields *list* – The original sys.path.

Module contents

1.1.2 Submodules

1.1.3 rez.bind_utils module

1.1.4 rez.bootstrap module

1.1.5 rez.build_process module

class rez.build_process.**BuildProcess**(*working_dir*, *build_system*, *package=None*, *vcs=None*, *ensure_latest=True*, *skip_repo_errors=False*, *ignore_existing_tag=False*, *verbose=False*, *quiet=False*)

Bases: object

A BuildProcess builds and possibly releases a package.

A build process iterates over the variants of a package, creates the correct build environment for each variant, builds that variant using a build system (or possibly creates a script so the user can do that independently), and then possibly releases the package with the nominated VCS. This is an abstract base class, you should use a BuildProcess subclass.

build(*install_path=None*, *clean=False*, *install=False*, *variants=None*)

Perform the build process.

Iterates over the package's variants, resolves the environment for each, and runs the build system within each resolved environment.

Parameters

- **install_path** (*str*) – The package repository path to install the package to, if installing. If None, defaults to *config.local_packages_path*.
- **clean** (*bool*) – If True, clear any previous build first. Otherwise, rebuild over the top of a previous build.

- **install** (*bool*) – If True, install the build.
- **variants** (*list of int*) – Indexes of variants to build, all if None.

Raises *BuildError* – If the build failed.

Returns Number of variants successfully built.

Return type int

get_changelog()

Get the changelog since last package release.

Returns Changelog.

Return type str

classmethod name()

property package

release(*release_message=None*, *variants=None*)

Perform the release process.

Iterates over the package's variants, building and installing each into the release path determined by *config.release_packages_path*.

Parameters

- **release_message** (*str*) – Message to associate with the release.
- **variants** (*list of int*) – Indexes of variants to release, all if None.

Raises *ReleaseError* – If the release failed.

Returns Number of variants successfully released.

Return type int

property working_dir

class rez.build_process.**BuildProcessHelper**(*working_dir*, *build_system*, *package=None*, *vcs=None*, *ensure_latest=True*, *skip_repo_errors=False*, *ignore_existing_tag=False*, *verbose=False*, *quiet=False*)

Bases: *rez.build_process.BuildProcess*

A BuildProcess base class with some useful functionality.

create_build_context(*variant*, *build_type*, *build_path*)

Create a context to build the variant within.

get_changelog()

Get the changelog since last package release.

Returns Changelog.

Return type str

get_current_tag_name()

get_package_install_path(*path*)

Return the installation path for a package (where its payload goes).

Parameters **path** (*str*) – Package repository path.

get_previous_release()

get_release_data()

Get release data for this release.

Returns dict.

post_release(*release_message=None*)

pre_release()

repo_operation()

run_hooks(*hook_event*, ***kwargs*)

visit_variants(*func*, *variants=None*, ***kwargs*)

Iterate over variants and call a function on each.

class rez.build_process.**BuildType**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to represent the type of build.

rez.build_process.**create_build_process**(*process_type*, *working_dir*, *build_system*, *package=None*, *vcs=None*, *ensure_latest=True*, *skip_repo_errors=False*, *ignore_existing_tag=False*, *verbose=False*, *quiet=False*)

Create a *BuildProcess* instance.

rez.build_process.**get_build_process_types**()

Returns the available build process implementations.

1.1.6 rez.build_system module

class rez.build_system.**BuildSystem**(*working_dir*, *opts=None*, *package=None*, *write_build_scripts=False*, *verbose=False*, *build_args=[]*, *child_build_args=[]*)

Bases: object

A build system, such as cmake, make, Scons etc.

classmethod add_pre_build_commands(*executor*, *variant*, *build_type*, *install*, *build_path*, *install_path=None*)

Execute pre_build_commands function if present.

classmethod add_standard_build_actions(*executor*, *context*, *variant*, *build_type*, *install*, *build_path*, *install_path=None*)

Perform build actions common to every build system.

classmethod bind_cli(*parser*, *group*)

Expose parameters to an argparse.ArgumentParser that are specific to this build system.

Parameters

- **parser** (*ArgumentParser*) – Arg parser.
- **group** (*ArgumentGroup*) – Arg parser group - you should add args to this, NOT to *parser*.

build(*context*, *variant*, *build_path*, *install_path*, *install=False*, *build_type=<BuildType.local: 0>*)

Implement this method to perform the actual build.

Parameters

- **context** – A ResolvedContext object that the build process must be executed within.
- **variant** (*Variant*) – The variant being built.
- **build_path** – Where to write temporary build files. May be absolute or relative to working_dir.
- **install_path** (*str*) – The package repository path to install the package to, if installing. If None, defaults to *config.local_packages_path*.
- **install** – If True, install the build.
- **build_type** – A BuildType (i.e local or central).

Returns

- success: Bool indicating if the build was successful.
- **extra_files: List of created files of interest, not including** build targets. A good example is the interpreted context file, usually named 'build.rxt.sh' or similar. These files should be located under build_path. Rez may install them for debugging purposes.
- **build_env_script: If this instance was created with write_build_scripts** as True, then the build should generate a script which, when run by the user, places them in the build environment.

Return type A dict containing the following information

classmethod child_build_system()

Returns the child build system.

Some build systems, such as cmake, don't build the source directly. Instead, they build an interim set of build scripts that are then consumed by a second build system (such as make). You should implement this method if that's the case.

Returns Name of build system (corresponding to the plugin name) if this system has a child system, or None otherwise.

classmethod is_valid_root(*path*)

Return True if this build system can build the source in path.

classmethod name()

Return the name of the build system, eg 'make'.

classmethod set_standard_vars(*executor*, *context*, *variant*, *build_type*, *install*, *build_path*, *install_path=None*)

Set some standard env vars that all build systems can rely on.

rez.build_system.**create_build_system**(*working_dir*, *buildsys_type=None*, *package=None*, *opts=None*, *write_build_scripts=False*, *verbose=False*, *build_args=[]*, *child_build_args=[]*)

Return a new build system that can build the source in working_dir.

rez.build_system.**get_buildsys_types**()

Returns the available build system implementations - cmake, make etc.

rez.build_system.**get_valid_build_systems**(*working_dir*, *package=None*)

Returns the build system classes that could build the source in given dir.

Parameters

- **working_dir** (*str*) – Dir containing the package definition and potentially build files.
- **package** (*Package*) – Package to be built. This may or may not be needed to determine the build system. For eg, cmake just has to look for a CMakeLists.txt file, whereas the 'build_command' package field must be present for the 'custom' build system type.

Returns Valid build system class types.

Return type List of class

1.1.7 rez.build_utils module

1.1.8 rez.dot module

1.1.9 rez.env module

1.1.10 rez.exceptions module

Exceptions.

exception rez.exceptions.**BuildContextResolveError**(*context*)

Bases: *rez.exceptions.BuildError*

Raised if unable to resolve the required context when creating the environment for a build process.

exception rez.exceptions.**BuildError**(*value=None*)

Bases: *rez.exceptions.RezError*

Base class for any build-related error.

exception rez.exceptions.**BuildProcessError**(*value=None*)

Bases: *rez.exceptions.RezError*

Base class for build process-related errors.

exception rez.exceptions.**BuildSystemError**(*value=None*)

Bases: *rez.exceptions.BuildError*

Base class for buildsys-related errors.

exception rez.exceptions.**ConfigurationError**(*value=None*)

Bases: *rez.exceptions.RezError*

A misconfiguration error.

exception rez.exceptions.**ContextBundleError**(*value=None*)

Bases: *rez.exceptions.RezError*

There was a problem bundling a context.

exception rez.exceptions.**InvalidPackageError**(*value=None*)

Bases: *rez.exceptions.RezError*

A special case exception used in package 'preprocess function'.

exception rez.exceptions.**PackageCacheError**(*value=None*)

Bases: *rez.exceptions.RezError*

There was an error related to a package cache.

exception rez.exceptions.**PackageCommandError**(*value=None*)

Bases: *rez.exceptions.RezError*

There is an error in a command or list of commands

exception rez.exceptions.**PackageCopyError**(*value=None*)

Bases: *rez.exceptions.RezError*

There was a problem copying a package.

exception rez.exceptions.**PackageFamilyNotFoundError**(*value=None*)

Bases: *rez.exceptions.RezError*

A package could not be found on disk.

exception rez.exceptions.**PackageMetadataError**(*value=None*, *path=None*, *resource_key=None*)

Bases: *rez.exceptions.ResourceContentError*

There is an error in a package's definition file.

type_name = 'package definition file'

exception rez.exceptions.**PackageMoveError**(*value=None*)

Bases: *rez.exceptions.RezError*

There was a problem moving a package.

exception rez.exceptions.**PackageNotFoundError**(*value=None*)

Bases: *rez.exceptions.RezError*

A package could not be found on disk.

exception rez.exceptions.**PackageRepositoryError**(*value=None*)

Bases: *rez.exceptions.RezError*

Base class for package repository- related errors.

exception rez.exceptions.**PackageRequestError**(*value=None*)

Bases: *rez.exceptions.RezError*

There is an error related to a package request.

exception rez.exceptions.**PackageTestError**(*value=None*)

Bases: *rez.exceptions.RezError*

There was a problem running a package test.

exception rez.exceptions.**ReleaseError**(*value=None*)

Bases: *rez.exceptions.RezError*

Any release-related error.

exception rez.exceptions.**ReleaseHookCancellingError**(*value=None*)

Bases: *rez.exceptions.RezError*

A release hook error that asks to cancel the release as a result.

exception rez.exceptions.**ReleaseHookError**(*value=None*)

Bases: *rez.exceptions.RezError*

Base class for release-hook- related errors.

exception rez.exceptions.**ReleaseVCSError**(*value=None*)

Bases: *rez.exceptions.ReleaseError*

Base class for release VCS-related errors.

exception rez.exceptions.**ResolveError**(*value=None*)

Bases: *rez.exceptions.RezError*

A resolve-related error.

exception rez.exceptions.**ResolvedContextError**(*value=None*)

Bases: *rez.exceptions.RezError*

An error occurred in a resolved context.

exception rez.exceptions.**ResourceContentError**(*value=None*, *path=None*, *resource_key=None*)

Bases: *rez.exceptions.ResourceError*

A resource contains incorrect data.

type_name = 'resource file'

exception rez.exceptions.**ResourceError**(*value=None*)

Bases: *rez.exceptions.RezError*

Resource-related exception base class.

exception rez.exceptions.**ResourceNotFoundError**(*value=None*)

Bases: *rez.exceptions.ResourceError*

A resource could not be found.

exception rez.exceptions.**RexError**(*value=None*)

Bases: *rez.exceptions.RezError*

There is an error in Rex code.

exception rez.exceptions.**RexStopError**(*value=None*)

Bases: *rez.exceptions.RexError*

Special error raised when a package commands uses the 'stop' command.

exception rez.exceptions.**RexUndefinedVariableError**(*value=None*)

Bases: *rez.exceptions.RexError*

There is a reference to an undefined variable.

exception rez.exceptions.**RezBindError**(*value=None*)

Bases: *rez.exceptions.RezError*

A bind-related error.

exception rez.exceptions.**RezError**(*value=None*)

Bases: Exception

Base-class Rez error.

exception rez.exceptions.**RezGuiQTImportError**

Bases: ImportError

A special case - see cli/gui.py

exception rez.exceptions.**RezPluginError**(*value=None*)

Bases: *rez.exceptions.RezError*

An error related to plugin or plugin load.

exception rez.exceptions.**RezSystemError**(*value=None*)

Bases: *rez.exceptions.RezError*

Rez system/internal error.

exception rez.exceptions.**SuiteError**(*value=None*)

Bases: *rez.exceptions.RezError*

Any suite-related error.

rez.exceptions.**convert_errors**(*from_*, *to*, *msg=None*)

1.1.11 rez.formulae_manager module

1.1.12 rez.package_maker module

class rez.package_maker.**PackageMaker**(*name*, *data=None*, *package_cls=None*)

Bases: rez.utils.data_utils.AttrDictWrapper

Utility class for creating packages.

get_package()

Create the analogous package.

Returns *Package* object.

rez.package_maker.**make_package**(*name*, *path*, *make_base=None*, *make_root=None*, *skip_existing=True*, *warn_on_skip=True*)

Make and install a package.

Example

```
>>> def make_root(variant, path):
>>>     os.symlink("/foo_payload/misc/python27", "ext")
>>>
>>> with make_package('foo', '/packages', make_root=make_root) as pkg:
>>>     pkg.version = '1.0.0'
>>>     pkg.description = 'does foo things'
>>>     pkg.requires = ['python-2.7']
```

Parameters

- **name** (*str*) – Package name.
- **path** (*str*) – Package repository path to install package into.
- **make_base** (*callable*) – Function that is used to create the package payload, if applicable.

- **make_root** (*callable*) – Function that is used to create the package variant payloads, if applicable.
- **skip_existing** (*bool*) – If True, detect if a variant already exists, and skip with a warning message if so.
- **warn_on_skip** (*bool*) – If True, print warning when a variant is skipped.

Yields *PackageMaker* object.

Note: Both *make_base* and *make_root* are called once per variant install, and have the signature (variant, path).

Note: The 'installed_variants' attribute on the *PackageMaker* instance will be appended with variant(s) created by this function, if any.

1.1.13 rez.package_maker_ module

1.1.14 rez.packages module

class rez.packages.**Package**(*resource*, *context=None*)

Bases: *rez.packages.PackageBaseResourceWrapper*

A package.

Note: Do not instantiate this class directly, instead use the function *iter_packages* or *PackageFamily.iter_packages*.

---

arbitrary_keys()

Get the arbitrary keys present in this package.

These are any keys not in the standard list ('name', 'version' etc).

Returns Arbitrary keys.

Return type set of str

as_exact_requirement()

Get the package, as an exact requirement string.

Returns Equivalent requirement string, eg "maya==2016.1"

property authors

property base

property build_requires

property cachable

property changelog

property commands

property description

get_variant(*index=None*)

Get the variant with the associated index.

Returns *Variant* object, or None if no variant with the given index exists.

property has_plugins

property hashed_variants

property help

property is_cachable

True if the package and its payload is safe to cache locally.

is_package = True

property is_relocatable

True if the package and its payload is safe to copy.

is_variant = False

iter_variants()

Iterate over the variants within this package, in index order.

Returns *Variant* iterator.

keys = {'authors', 'base', 'build_requires', 'cachable', 'changelog', 'commands', 'config', 'description', 'has_plugins', 'hashed_variants', 'help', 'name', 'plugin_for', 'post_commands', 'pre_build_commands', 'pre_commands', 'pre_test_commands', 'previous_revision', 'previous_version', 'private_build_requires', 'release_message', 'relocatable', 'requires', 'revision', 'tests', 'timestamp', 'tools', 'uuid', 'variants', 'vcs', 'version'}

property name

num_variants

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

parent

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property plugin_for

property post_commands

property pre_build_commands

property pre_commands

property pre_test_commands

property previous_revision

property previous_version

property private_build_requires

qualified_name

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property release_message

property relocatable

property requires

property revision

property tests

property timestamp

property tools

property uuid

property variants

property vcs

property version

class rez.packages.**PackageBaseResourceWrapper**(*resource*, *context=None*)

Bases: *rez.packages.PackageRepositoryResourceWrapper*

Abstract base class for *Package* and *Variant*.

arbitrary_keys()

property config

Returns the config for this package.

Defaults to global config if this package did not provide a 'config' section.

is_local

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

late_bind_schemas = {'requires': Schema([Or(<class 'rez.utils.formatting.PackageRequest'>, And(<class 'str'>, Use(<class 'rez.utils.formatting.PackageRequest'>)))])}

print_info(*buf=None*, *format_=<FileFormat.yaml: ('yaml',)>*, *skip_attributes=None*, *include_release=False*)

Print the contents of the package.

Parameters

- **buf** (*file-like object*) – Stream to write to.

- **format** (*FileFormat*) – Format to write in.
- **skip_attributes** (*list of str*) – List of attributes to not print.
- **include_release** (*bool*) – If True, include release-related attributes, such as 'timestamp' and 'changelog'

set_context(*context*)

property uri

class rez.packages.**PackageFamily**(*resource*)

Bases: *rez.packages.PackageRepositoryResourceWrapper*

A package family.

Note: Do not instantiate this class directly, instead use the function *iter_package_families*.

iter_packages()

Iterate over the packages within this family, in no particular order.

Returns *Package* iterator.

keys = {'name'}

property name

class rez.packages.**PackageRepositoryResourceWrapper**(*resource*)

Bases: rez.utils.resources.ResourceWrapper, rez.utils.formatting.StringFormatMixin

format_expand = <StringFormatType.unchanged: 3>

property repository

The package repository this resource comes from.

Returns *PackageRepository*.

validated_data()

class rez.packages.**PackageSearchPath**(*packages_path*)

Bases: object

A list of package repositories.

For example, $REZ_PACKAGES_PATH refers to a list of repositories.

iter_packages(*name*, *range_=None*)

See *iter_packages*.

Returns *Package* iterator.

class rez.packages.**Variant**(*resource*, *context=None*, *parent=None*)

Bases: *rez.packages.PackageBaseResourceWrapper*

A package variant.

Note: Do not instantiate this class directly, instead use the function *Package.iter_variants*.

arbitrary_keys()

property authors

property base

property build_requires

property cachable

property changelog

property commands

property description

get_requires(*build_requires=False*, *private_build_requires=False*)

Get the requirements of the variant.

Parameters

- **build_requires** (*bool*) – If True, include build requirements.
- **private_build_requires** (*bool*) – If True, include private build requirements.

Returns List of *Requirement* objects.

property has_plugins

property hashed_variants

property help

property index

install(*path*, *dry_run=False*, *overrides=None*)

Install this variant into another package repository.

If the package already exists, this variant will be correctly merged into the package. If the variant already exists in this package, the existing variant is returned.

Parameters

- **path** (*str*) – Path to destination package repository.
- **dry_run** (*bool*) – If True, do not actually install the variant. In this mode, a *Variant* instance is only returned if the equivalent variant already exists in this repository; otherwise, None is returned.
- **overrides** (*dict*) – Use this to change or add attributes to the installed variant.

Returns *Variant* object - the (existing or newly created) variant in the specified repository. If *dry_run* is True, None may be returned.

is_package = False

is_variant = True

keys = {'authors', 'base', 'build_requires', 'cachable', 'changelog', 'commands', 'config', 'description', 'has_plugins', 'hashed_variants', 'help', 'index', 'name', 'plugin_for', 'post_commands', 'pre_build_commands', 'pre_commands', 'pre_test_commands', 'previous_revision', 'previous_version', 'private_build_requires', 'release_message', 'relocatable', 'requires', 'revision', 'root', 'subpath', 'tests', 'timestamp', 'tools', 'uuid', 'vcs', 'version'}

property name

parent

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property plugin_for

property post_commands

property pre_build_commands

property pre_commands

property pre_test_commands

property previous_revision

property previous_version

property private_build_requires

qualified_name

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

qualified_package_name

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property release_message

property relocatable

property requires

Get variant requirements.

This is a concatenation of the package requirements and those of this specific variant.

Returns List of *Requirement* objects.

property revision

property root

property subpath

property tests

property timestamp

property tools

property uuid

property variant_requires

Get the subset of requirements specific to this variant.

Returns List of *Requirement* objects.

property vcs

property version

rez.packages.**create_package**(*name*, *data*, *package_cls=None*)

Create a package given package data.

Parameters

- **name** (*str*) – Package name.

- **data** (*dict*) – Package data. Must conform to *package_maker.package_schema*.

Returns *Package* object.

rez.packages.**get_completions**(*prefix*, *paths=None*, *family_only=False*)

Get autocompletion options given a prefix string.

Example

```
>>> get_completions("may")
set(["maya", "maya_utils"])
>>> get_completions("maya-")
set(["maya-2013.1", "maya-2015.0.sp1"])
```

Parameters

- **prefix** (*str*) – Prefix to match.
- **paths** (*list of str*) – paths to search for packages, defaults to *config.packages_path*.
- **family_only** (*bool*) – If True, only match package names, do not include version component.

Returns Set of strings, may be empty.

rez.packages.**get_developer_package**(*path*, *format=None*)

Create a developer package.

Parameters

- **path** (*str*) – Path to dir containing package definition file.
- **format** (*str*) – Package definition file format, detected if None.

Returns *DeveloperPackage*.

rez.packages.**get_last_release_time**(*name*, *paths=None*)

Returns the most recent time this package was released.

Note that releasing a variant into an already-released package is also considered a package release.

Parameters

- **name** (*str*) – Package family name.
- **paths** (*list of str*) – paths to search for packages, defaults to *config.packages_path*.

Returns Epoch time of last package release, or zero if this cannot be determined.

Return type int

rez.packages.**get_latest_package**(*name*, *range_=None*, *paths=None*, *error=False*)

Get the latest package for a given package name.

Parameters

- **name** (*str*) – Package name.
- **range** (*VersionRange*) – Version range to search within.
- **paths** (*list of str, optional*) – paths to search for package families, defaults to *config.packages_path*.

- **error** (*bool*) – If True, raise an error if no package is found.

Returns *Package* object, or None if no package is found.

rez.packages.**get_latest_package_from_string**(*txt*, *paths=None*, *error=False*)

Get the latest package found within the given request string.

Parameters

- **txt** (*str*) – Request, eg 'foo-1.2+'
- **paths** (*list of str, optional*) – paths to search for packages, defaults to *config.packages_path*.
- **error** (*bool*) – If True, raise an error if no package is found.

Returns *Package* object, or None if no package is found.

rez.packages.**get_package**(*name*, *version*, *paths=None*)

Get a package by searching a list of repositories.

Parameters

- **name** (*str*) – Name of the package, eg 'maya'.
- **version** (*Version or str*) – Version of the package, eg '1.0.0'
- **paths** (*list of str, optional*) – paths to search for package, defaults to *config.packages_path*.

Returns *Package* object, or None if the package was not found.

rez.packages.**get_package_family_from_repository**(*name*, *path*)

Get a package family from a repository.

Parameters **name** (*str*) – Name of the package, eg 'maya'.

Returns *PackageFamily* object, or None if the family was not found.

rez.packages.**get_package_from_handle**(*package_handle*)

Create a package given its handle (or serialized dict equivalent)

Parameters **package_handle** (*ResourceHandle* or dict) – Resource handle, or equivalent serialized dict representation from ResourceHandle.to_dict

Returns *Package*.

rez.packages.**get_package_from_repository**(*name*, *version*, *path*)

Get a package from a repository.

Parameters

- **name** (*str*) – Name of the package, eg 'maya'.
- **version** (*Version or str*) – Version of the package, eg '1.0.0'

Returns *Package* object, or None if the package was not found.

rez.packages.**get_package_from_string**(*txt*, *paths=None*)

Get a package given a string.

Parameters

- **txt** (*str*) – String such as 'foo', 'bah-1.3'.
- **paths** (*list of str, optional*) – paths to search for package, defaults to *config.packages_path*.

Returns *Package* instance, or None if no package was found.

rez.packages.**get_package_from_uri**(*uri*, *paths=None*)

Get a package given its URI.

Parameters

- **uri** (*str*) – Variant URI
- **paths** (*list of str*) – paths to search for packages, defaults to *config.packages_path*. If None, attempts to find a package that may have come from any package repo.

Returns *Package*, or None if the package could not be found.

rez.packages.**get_variant**(*variant_handle*, *context=None*)

Create a variant given its handle (or serialized dict equivalent)

Parameters

- **variant_handle** (*ResourceHandle* or dict) – Resource handle, or equivalent serialized dict representation from ResourceHandle.to_dict
- **context** (*ResolvedContext*) – The context this variant is associated with, if any.

Returns *Variant*.

rez.packages.**get_variant_from_uri**(*uri*, *paths=None*)

Get a variant given its URI.

Parameters

- **uri** (*str*) – Variant URI
- **paths** (*list of str*) – paths to search for variants, defaults to *config.packages_path*. If None, attempts to find a variant that may have come from any package repo.

Returns *Variant*, or None if the variant could not be found.

rez.packages.**iter_package_families**(*paths=None*)

Iterate over package families, in no particular order.

Note that multiple package families with the same name can be returned. Unlike packages, families later in the searchpath are not hidden by earlier families.

Parameters **paths** (*list of str, optional*) – paths to search for package families, defaults to *config.packages_path*.

Returns *PackageFamily* iterator.

rez.packages.**iter_packages**(*name*, *range_=None*, *paths=None*)

Iterate over *Package* instances, in no particular order.

Packages of the same name and version earlier in the search path take precedence - equivalent packages later in the paths are ignored. Packages are not returned in any specific order.

Parameters

- **name** (*str*) – Name of the package, eg 'maya'.
- **range** (*VersionRange or str*) – If provided, limits the versions returned to those in *range_*.
- **paths** (*list of str, optional*) – paths to search for packages, defaults to *config.packages_path*.

Returns *Package* iterator.

1.1.15 rez.platform_ module

1.1.16 rez.plugin_managers module

Manages loading of all types of Rez plugins.

class rez.plugin_managers.**BuildProcessPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different build and release processes.

type_name = 'build_process'

class rez.plugin_managers.**BuildSystemPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different build systems when building packages.

type_name = 'build_system'

class rez.plugin_managers.**CommandPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different custom Rez applications/subcommands.

type_name = 'command'

class rez.plugin_managers.**PackageRepositoryPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different package repositories for loading packages.

type_name = 'package_repository'

class rez.plugin_managers.**ReleaseHookPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different version control systems when releasing packages.

type_name = 'release_hook'

class rez.plugin_managers.**ReleaseVCSPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different version control systems when releasing packages.

type_name = 'release_vcs'

class rez.plugin_managers.**RezPluginManager**

Bases: `object`

Primary interface for working with registered plugins.

Custom plugins are organized under a python package named 'rezplugins'. The direct sub-packages of 'rezplugins' are the plugin types supported by rez, and the modules below that are individual custom plugins extending that type.

For example, rez provides plugins of type 'build_system': 'cmake' and 'make':

```
rezplugins/
  __init__.py
  build_system/
    __init__.py
    cmake.py
    make.py
  ...
```

Here is an example of how to provide your own plugin. In the example, we'll be adding a plugin for the SCons build system.

1. Create the 'rezplugins/build_system' directory structure, add the empty '__init__.py' files, and then place your new 'scons.py' plugin module into the 'build_system' sub-package:

   ```
   rezplugins/
     __init__.py
     build_system/
       __init__.py
       scons.py
   ```

2. Write your 'scons.py' plugin module, sub-classing your *SConsBuildSystem* class from *rez.build_systems.BuildSystem* base class.

   At the bottom of the module add a *register_plugin* function that returns your plugin class:

   ```
   def register_plugin():
       return SConsBuildSystem
   ```

3 Set or append the rez config setting *plugin_path* to point to the directory **above** your 'rezplugins' directory.

All 'rezplugin' packages found on the search path will all be merged into a single python package.

Note: Even though 'rezplugins' is a python package, your sparse copy of it should not be on the *PYTHONPATH*, just the *REZ_PLUGIN_PATH*. This is important because it ensures that rez's copy of 'rezplugins' is always found first.

create_instance(*plugin_type*, *plugin_name*, ***instance_kwargs*)

Create and return an instance of the given plugin.

get_failed_plugins(*plugin_type*)

Return a list of plugins for the given type that failed to load.

Returns name (str): Name of the plugin. reason (str): Error message.

Return type List of 2-tuples

get_plugin_class(*plugin_type*, *plugin_name*)

Return the class registered under the given plugin name.

get_plugin_config_data(*plugin_type*)

Return the merged configuration data for the plugin type.

get_plugin_config_schema(*plugin_type*)

get_plugin_module(*plugin_type*, *plugin_name*)

Return the module defining the class registered under the given plugin name.

get_plugin_types()

Return a list of the registered plugin types.

get_plugins(*plugin_type*)

Return a list of the registered names available for the given plugin type.

get_summary_string()

Get a formatted string summarising the plugins that were loaded.

register_plugin_type(*type_class*)

rezplugins_module_paths

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

class rez.plugin_managers.**RezPluginType**

Bases: object

An abstract base class representing a single type of plugin.

'type_name' must correspond with one of the source directories found under the 'plugins' directory.

config_schema

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

create_instance(*plugin*, ***instance_kwargs*)

Create and return an instance of the given plugin.

get_plugin_class(*plugin_name*)

Returns the class registered under the given plugin name.

get_plugin_module(*plugin_name*)

Returns the module containing the plugin of the given name.

load_plugins()

register_plugin(*plugin_name*, *plugin_class*, *plugin_module*)

type_name = None

class rez.plugin_managers.**ShellPluginType**

Bases: *rez.plugin_managers.RezPluginType*

Support for different types of target shells, such as bash, tcsh.

type_name = 'shell'

rez.plugin_managers.**extend_path**(*path*, *name*)

Extend a package's path.

Intended use is to place the following code in a package's __init__.py:

from pkgutil import extend_path __path__ = extend_path(__path__, __name__)

This will add to the package's __path__ all subdirectories of directories on 'config.plugin_path' named after the package. This is useful if one wants to distribute different parts of a single logical package as multiple directories.

If the input path is not a list (as is the case for frozen packages) it is returned unchanged. The input path is not modified; an extended copy is returned. Items are only appended to the copy at the end.

It is assumed that 'plugin_path' is a sequence. Items of 'plugin_path' that are not (unicode or 8-bit) strings referring to existing directories are ignored. Unicode items of sys.path that cause errors when used as filenames may cause this function to raise an exception (in line with os.path.isdir() behavior).

rez.plugin_managers.**uncache_rezplugins_module_paths**(*instance=None*)

1.1.17 rez.py_dist module

1.1.18 rez.release_hook module

class rez.release_hook.**ReleaseHook**(*source_path*)

Bases: object

An object that allows for custom behaviour during releases.

A release hook provides methods that you implement to inject custom behaviour during parts of the release process. For example, the builtin 'email' hook sends a post-release email to a configured address.

classmethod name()

Return name of source retriever, eg 'git'

post_release(*user*, *install_path*, *variants*, *release_message=None*, *changelog=None*, *previous_version=None*, *previous_revision=None*, ***kwargs*)

Post-release hook.

This is called after all package variants have been released.

Parameters

- **user** – Name of person who did the release.
- **install_path** – Directory the package was installed into.
- **variants** (list of *Variant*) – The variants that have been released.
- **release_message** – User-supplied release message.
- **changelog** – List of strings describing changes since last release.
- **previous_version** – Version of previously-release package, None if no previous release.
- **previous_revision** – Revision of previously-releaved package (type depends on repo - see ReleaseVCS.get_current_revision().
- **kwargs** – Reserved.

pre_build(*user*, *install_path*, *variants=None*, *release_message=None*, *changelog=None*, *previous_version=None*, *previous_revision=None*, ***kwargs*)

Pre-build hook.

Parameters

- **user** – Name of person who did the release.
- **install_path** – Directory the package was installed into.
- **variants** – List of variant indices we are attempting to build, or None
- **release_message** – User-supplied release message.
- **changelog** – List of strings describing changes since last release.
- **previous_version** – Version object - previously-release package, or None if no previous release.
- **previous_revision** – Revision of previously-released package (type depends on repo - see ReleaseVCS.get_current_revision().
- **kwargs** – Reserved.

Note: This method should raise a *ReleaseHookCancellingError* if the release process should be cancelled.

---

pre_release(*user*, *install_path*, *variants=None*, *release_message=None*, *changelog=None*, *previous_version=None*, *previous_revision=None*, ***kwargs*)

Pre-release hook.

This is called before any package variants are released.

Parameters

- **user** – Name of person who did the release.
- **install_path** – Directory the package was installed into.
- **variants** – List of variant indices we are attempting to release, or None

- **release_message** – User-supplied release message.
- **changelog** – List of strings describing changes since last release.
- **previous_version** – Version object - previously-release package, or None if no previous release.
- **previous_revision** – Revision of previously-releaved package (type depends on repo - see ReleaseVCS.get_current_revision().
- **kwargs** – Reserved.

Note: This method should raise a *ReleaseHookCancellingError* if the release process should be cancelled.

class rez.release_hook.**ReleaseHookEvent**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to help manage release hooks.

rez.release_hook.**create_release_hook**(*name*, *source_path*)

Return a new release hook of the given type.

rez.release_hook.**create_release_hooks**(*names*, *source_path*)

rez.release_hook.**get_release_hook_types**()

Returns the available release hook implementations.

1.1.19 rez.release_vcs module

class rez.release_vcs.**ReleaseVCS**(*pkg_root*, *vcs_root=None*)

Bases: object

A version control system (VCS) used to release Rez packages.

create_release_tag(*tag_name*, *message=None*)

Create a tag in the repo.

Create a tag in the repository representing the release of the given version.

Parameters

- **tag_name** (*str*) – Tag name to write to the repo.
- **message** (*str*) – Message string to associate with the release.

classmethod export(*revision*, *path*)

Export the repository to the given path at the given revision.

Note: The directory at *path* must not exist, but the parent directory must exist.

Parameters

- **revision** (*object*) – Revision to export; current revision if None.
- **path** (*str*) – Directory to export the repository to.

classmethod find_executable(*name*)

classmethod find_vcs_root(*path*)

Try to find a version control root directory of this type for the given path.

If successful, returns (vcs_root, levels_up), where vcs_root is the path to the version control root directory it found, and levels_up is an integer indicating how many parent directories it had to search through to find it, where 0 means it was found in the indicated path, 1 means it was found in that path's parent, etc. If not sucessful, returns None

get_changelog(*previous_revision=None*, *max_revisions=None*)

Get the changelog text since the given revision.

If previous_revision is not an ancestor (for example, the last release was from a different branch) you should still return a meaningful changelog - perhaps include a warning, and give changelog back to the last common ancestor.

Parameters

- **previous_revision** – The revision to give the changelog since. If
- **None** –
- **changelog.** (*give the entire*) –

Returns Changelog, as a string.

get_current_revision()

Get the current revision, this can be any type (str, dict etc) appropriate to your VCS implementation.

Note: You must ensure that a revision contains enough information to clone/export/checkout the repo elsewhere - otherwise you will not be able to implement *export*.

---

classmethod is_valid_root(*path*)

Return True if the given path is a valid root directory for this version control system.

Note that this is different than whether the path is under the control of this type of vcs; to answer that question, use find_vcs_root

classmethod name()

Return the name of the VCS type, eg 'git'.

classmethod search_parents_for_root()

Return True if this vcs type should check parent directories to find the root directory

tag_exists(*tag_name*)

Test if a tag exists in the repo.

Parameters **tag_name** (*str*) – Tag name to check for.

Returns True if the tag exists, False otherwise.

Return type bool

validate_repostate()

Ensure that the VCS working copy is up-to-date.

rez.release_vcs.**create_release_vcs**(*path*, *vcs_name=None*)

Return a new release VCS that can release from this source path.

rez.release_vcs.**get_release_vcs_types**()

Returns the available VCS implementations - git, hg etc.

1.1.20 rez.resolved_context module

class rez.resolved_context.**PatchLock**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to represent the 'lock type' used when patching context objects.

class rez.resolved_context.**ResolvedContext**(*package_requests*, *verbosity=0*, *timestamp=None*, *building=False*, *caching=None*, *package_paths=None*, *package_filter=None*, *package_orderers=None*, *max_fails=-1*, *add_implicit_packages=True*, *time_limit=-1*, *callback=None*, *package_load_callback=None*, *buf=None*, *suppress_passive=False*, *print_stats=False*, *package_caching=None*)

Bases: object

A class that resolves, stores and spawns Rez environments.

The main Rez entry point for creating, saving, loading and executing resolved environments. A ResolvedContext object can be saved to file and loaded at a later date, and it can reconstruct the equivalent environment at that time. It can spawn interactive and non-interactive shells, in any supported shell plugin type, such as bash and tcsh. It can also run a command within a configured python namespace, without spawning a child shell.

class Callback(*max_fails*, *time_limit*, *callback*, *buf=None*)

Bases: object

apply(*parent_environ=None*)

Apply the context to the current python session.

Note that this updates os.environ and possibly sys.path, if *parent_environ* is not provided.

Parameters parent_environ – Environment to interpret the context within, defaults to os.environ if None.

context_tracking_lock = <unlocked _thread.lock object>

context_tracking_payload = None

copy()

Returns a shallow copy of the context.

execute_command(*args*, *parent_environ=None*, ***Popen_args*)

Run a command within a resolved context.

This applies the context to a python environ dict, then runs a subprocess in that namespace. This is not a fully configured subshell - shell-specific commands such as aliases will not be applied. To execute a command within a subshell instead, use execute_shell().

Warning: This runs a command in a configured environ dict only, not in a true shell. To do that, call *execute_shell* using the *command* keyword argument.

Parameters

- **args** – Command arguments, can be a string.
- **parent_environ** – Environment to interpret the context within, defaults to os.environ if None.
- **Popen_args** – Args to pass to subprocess.Popen.

Returns A subprocess.Popen object.

Note: This does not alter the current python session.

---

execute_rex_code(*code*, *filename=None*, *shell=None*, *parent_environ=None*, ***Popen_args*)

Run some rex code in the context.

Note: This is just a convenience form of *execute_shell*.

Parameters

- **code** (*str*) – Rex code to execute.
- **filename** (*str*) – Filename to report if there are syntax errors.
- **shell** – Shell type, for eg 'bash'. If None, the current shell type is used.
- **parent_environ** – Environment to run the shell process in, if None then the current environment is used.
- **Popen_args** – args to pass to the shell process object constructor.

Returns *subprocess.Popen* object for the shell process.

execute_shell(*shell=None*, *parent_environ=None*, *rcfile=None*, *norc=False*, *stdin=False*, *command=None*, *quiet=False*, *block=None*, *actions_callback=None*, *post_actions_callback=None*, *context_filepath=None*, *start_new_session=False*, *detached=False*, *pre_command=None*, ***Popen_args*)

Spawn a possibly-interactive shell.

Parameters

- **shell** – Shell type, for eg 'bash'. If None, the current shell type is used.
- **parent_environ** – Environment to run the shell process in, if None then the current environment is used.
- **rcfile** – Specify a file to source instead of shell startup files.
- **norc** – If True, skip shell startup files, if possible.
- **stdin** – If True, read commands from stdin, in a non-interactive shell.
- **command** – If not None, execute this command in a non-interactive shell. If an empty string or list, don't run a command, but don't open an interactive shell either. Can be a list of args.
- **quiet** – If True, skip the welcome message in interactive shells.
- **block** – If True, block until the shell is terminated. If False, return immediately. If None, will default to blocking if the shell is interactive.

- **actions_callback** – Callback with signature (RexExecutor). This lets the user append custom actions to the context, such as setting extra environment variables. Callback is run prior to context Rex execution.
- **post_actions_callback** – Callback with signature (RexExecutor). This lets the user append custom actions to the context, such as setting extra environment variables. Callback is run after context Rex execution.
- **context_filepath** – If provided, the context file will be written here, rather than to the default location (which is in a tempdir). If you use this arg, you are responsible for cleaning up the file.
- **start_new_session** – If True, change the process group of the target process. Note that this may override the Popen_args keyword 'preexec_fn'.
- **detached** – If True, open a separate terminal. Note that this may override the *pre_command* argument.
- **pre_command** – Command to inject before the shell command itself. This is for internal use.
- **Popen_args** – args to pass to the shell process object constructor.

Returns A 3-tuple of (returncode, stdout, stderr); If non-blocking - A subprocess.Popen object for the shell process.

Return type If blocking

classmethod from_dict(*d*, *identifier_str=None*)

Load a *ResolvedContext* from a dict.

Parameters

- **d** (*dict*) – Dict containing context data.
- **identifier_str** (*str*) – String identifying the context, this is only used to display in an error string if a serialization version mismatch is detected.

Returns *ResolvedContext* object.

get_actions(*parent_environ=None*)

Get the list of rex.Action objects resulting from interpreting this context. This is provided mainly for testing purposes.

Parameters **within** (*parent_environ Environment to interpret the context*) – defaults to os.environ if None.

:param : defaults to os.environ if None.

Returns A list of rex.Action subclass instances.

get_conflicting_tools(*request_only=False*)

Returns tools of the same name provided by more than one package.

Parameters **request_only** – If True, only return the key from resolved packages that were also present in the request.

Returns set([Variant])}.

Return type Dict of {tool-name

classmethod get_current()

Get the context for the current env, if there is one.

Returns Current context, or None if not in a resolved env.

Return type *ResolvedContext*

`get_dependency_graph`(*as_dot=False*)

Generate the dependency graph.

The dependency graph is a simpler subset of the resolve graph. It contains package name nodes connected directly to their dependencies. Weak references and conflict requests are not included in the graph. The dependency graph does not show conflicts.

Returns *pygraph.digraph* object.

`get_environ`(*parent_environ=None*)

Get the environ dict resulting from interpreting this context.

@param parent_environ Environment to interpret the context within, defaults to os.environ if None.

@returns The environment dict generated by this context, when interpreted in a python rex interpreter.

`get_key`(*key*, *request_only=False*)

Get a data key value for each resolved package.

Parameters

- **key** (*str*) – String key of property, eg 'tools'.
- **request_only** (*bool*) – If True, only return the key from resolved packages that were also present in the request.

Returns (variant, value)}.

Return type Dict of {pkg-name

`get_patched_request`(*package_requests=None*, *package_subtractions=None*, *strict=False*, *rank=0*)

Get a 'patched' request.

A patched request is a copy of this context's request, but with some changes applied. This can then be used to create a new, 'patched' context.

New package requests override original requests based on the type - normal, conflict or weak. So 'foo-2' overrides 'foo-1', '!foo-2' overrides '!foo-1' and '~foo-2' overrides '~foo-1', but a request such as '!foo-2' would not replace 'foo-1' - it would be added instead.

Note that requests in *package_requests* can have the form '^foo'. This is another way of supplying package subtractions.

Any new requests that don't override original requests are appended, in the order that they appear in *package_requests*.

Parameters

- **package_requests** (list of str or list of *PackageRequest*) – Overriding requests.
- **package_subtractions** (*list of str*) – Any original request with a package name in this list is removed, before the new requests are added.
- **strict** (*bool*) – If True, the current context's resolve is used as the original request list, rather than the request.
- **rank** (*int*) – If > 1, package versions can only increase in this rank and further - for example, rank=3 means that only version patch numbers are allowed to increase, major and minor versions will not change. This is only applied to packages that have not been explicitly overridden in *package_requests*. If rank <= 1, or *strict* is True, rank is ignored.

Returns List of *PackageRequest* objects that can be used to construct a new *ResolvedContext* object.

`get_resolve_as_exact_requests()`

Convert to a package request list of exact resolved package versions.

```
>>> r = ResolvedContext(['foo']
>>> r.get_resolve_as_exact_requests()
['foo==1.2.3', 'bah==1.0.1', 'python==2.7.12']
```

Returns Context as a list of exact version requests.

Return type List of *PackageRequest*

`get_resolve_diff(other)`

Get the difference between the resolve in this context and another.

The difference is described from the point of view of the current context - a newer package means that the package in *other* is newer than the package in *self*.

Diffs can only be compared if their package search paths match, an error is raised otherwise.

The diff is expressed in packages, not variants - the specific variant of a package is ignored.

Returns

- 'newer_packages': A dict containing items: - package name (str); - List of *Package* objects. These are the packages up to and

  including the newer package in *self*, in ascending order.

- 'older_packages': A dict containing: - package name (str); - List of *Package* objects. These are the packages down to and

  including the older package in *self*, in descending order.

- **'added_packages': Set of *Package* objects present in *self* but** not in *other*;

- **'removed_packages': Set of *Package* objects present in *other*,** but not in *self*.

If any item ('added_packages' etc) is empty, it is not added to the resulting dict. Thus, an empty dict is returned if there is no difference between contexts.

Return type A dict containing

`get_resolved_package(name)`

Returns a *Variant* object or None if the package is not in the resolve.

`get_shell_code(shell=None, parent_environ=None, style=<OutputStyle.file: ('Code as it would appear in a script file.', )>)`

Get the shell code resulting from intepreting this context.

Parameters

- **shell** (*str*) – Shell type, for eg 'bash'. If None, the current shell type is used.
- **parent_environ** (*dict*) – Environment to interpret the context within, defaults to os.environ if None.
- **()** (*style*) – Style to format shell code in.

get_tool_variants(*tool_name*)

Get the variant(s) that provide the named tool.

If there are more than one variants, the tool is in conflict, and Rez does not know which variant's tool is actually exposed.

Parameters **tool_name** (*str*) – Name of the tool to search for.

Returns Set of *Variant* objects. If no variant provides the tool, an empty set is returned.

get_tools(*request_only=False*)

Returns the commandline tools available in the context.

Parameters **request_only** – If True, only return the tools from resolved packages that were also present in the request.

Returns (variant, [tools])}.

Return type Dict of {pkg-name

graph(*as_dot=False*)

Get the resolve graph.

Parameters **as_dot** – If True, get the graph as a dot-language string. Otherwise, a pygraph.digraph object is returned.

Returns A string or *pygraph.digraph* object, or None if there is no graph associated with the resolve.

property has_graph

Return True if the resolve has a graph.

is_current()

Returns True if this is the currently sourced context, False otherwise.

Return type bool

classmethod load(*path*)

Load a resolved context from file.

local = <_thread._local object>

package_cache_present = True

print_info(*buf=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *verbosity=0*, *source_order=False*, *show_resolved_uris=False*)

Prints a message summarising the contents of the resolved context.

Parameters

- **buf** (*file-like object*) – Where to print this info to.
- **verbosity** (*bool*) – Verbose mode.
- **source_order** (*bool*) – If True, print resolved packages in the order they are sourced, rather than alphabetical order.
- **show_resolved_uris** (*bool*) – By default, resolved packages have their 'root' property listed, or their 'uri' if 'root' is None. Use this option to list 'uri' regardless.

print_resolve_diff(*other*, *heading=None*)

Print the difference between the resolve of two contexts.

Parameters

- **other** (*ResolvedContext*) – Context to compare to.
- **heading** – One of: - None: Do not display a heading; - True: Display the filename of each context as a heading, if

  both contexts have a filepath;

  – 2-tuple: Use the given two strings as headings - the first is the heading for *self*, the second for *other*.

print_tools(*buf=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*)

classmethod read_from_buffer(*buf*, *identifier_str=None*)

Load the context from a buffer.

requested_packages(*include_implicit=False*)

Get packages in the request.

Parameters **include_implicit** (*bool*) – If True, implicit packages are appended to the result.

Returns List of *PackageRequest* objects.

property resolved_ephemerals

Get non-conflict ephemerals in the resolve.

Returns List of *Requirement* objects, or None if the resolve failed.

property resolved_packages

Get packages in the resolve.

Returns List of *Variant* objects, or None if the resolve failed.

retargeted(*package_paths*, *package_names=None*, *skip_missing=False*)

Create a retargeted copy of this context.

Retargeting a context means replacing its variant references with the same variants from other package repositories.

Parameters

- **package_paths** – List of paths to search for pkgs to retarget to.
- **package_names** (*list of str*) – Only retarget these packages. If None, retarget all packages.
- **skip_missing** (*bool*) – If True, skip retargeting of variants that cannot be found in *package_paths*. By default, a *PackageNotFoundError* is raised.

Returns The retargeted context.

Return type ResolvecContext`

save(*path*)

Save the resolved context to file.

serialize_version = (4, 7)

set_load_path(*path*)

Set the path that this context was reportedly loaded from.

You may want to use this method in cases where a context is saved to disk, but you need to associate this new path with the context while it is still in use.

property status

Return the current status of the context.

Returns ResolverStatus.

property success

True if the context has been solved, False otherwise.

tmpdir_manager = <rez.utils.filesystem.TempDirs object>

to_dict(*fields=None*)

Convert context to dict containing only builtin types.

Parameters **fields** (*list of str*) – If present, only write these fields into the dict. This can be used to avoid constructing expensive fields (such as 'graph') for some cases.

Returns Dictified context.

Return type dict

validate()

Validate the context.

which(*cmd*, *parent_environ=None*, *fallback=False*)

Find a program in the resolved environment.

Parameters

- **cmd** – String name of the program to find.
- **parent_environ** – Environment to interpret the context within, defaults to os.environ if None.
- **fallback** – If True, and the program is not found in the context, the current environment will then be searched.

Returns Path to the program, or None if the program was not found.

write_to_buffer(*buf*)

Save the context to a buffer.

class rez.resolved_context.**RezToolsVisibility**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Determines if/how rez cli tools are added back to PATH within a resolved environment.

class rez.resolved_context.**SuiteVisibility**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Defines what suites on $PATH stay visible when a new rez environment is resolved.

rez.resolved_context.**get_lock_request**(*name*, *version*, *patch_lock*, *weak=True*)

Given a package and patch lock, return the equivalent request.

For example, for object 'foo-1.2.1' and lock type 'lock_3', the equivalent request is '~foo-1.2'. This restricts updates to foo to patch-or-lower version changes only.

For objects not versioned down to a given lock level, the closest possible lock is applied. So 'lock_3' applied to 'foo-1' would give '~foo-1'.

Parameters

- **name** (*str*) – Package name.
- **version** (*Version*) – Package version.
- **patch_lock** (PatchLock) – Lock type to apply.

Returns *PackageRequest* object, or None if there is no equivalent request.

1.1.21 rez.resolver module

class rez.resolver.**Resolver**(*context*, *package_requests*, *package_paths*, *package_filter=None*, *package_orderers=None*, *timestamp=0*, *callback=None*, *building=False*, *verbosity=False*, *buf=None*, *package_load_callback=None*, *caching=True*, *suppress_passive=False*, *print_stats=False*)

Bases: object

The package resolver.

The Resolver uses a combination of Solver(s) and cache(s) to resolve a package request as quickly as possible.

property graph

Return the resolve graph.

The resolve graph shows unsuccessful as well as successful resolves.

Returns A pygraph.digraph object, or None if the solve has not completed.

property resolved_ephemerals

Get the list of resolved ewphemerals.

Returns List of *Requirement* objects, or None if the resolve has not completed.

property resolved_packages

Get the list of resolved packages.

Returns List of *PackageVariant* objects, or None if the resolve has not completed.

solve()

Perform the solve.

property status

Return the current status of the resolve.

Returns ResolverStatus.

class rez.resolver.**ResolverStatus**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to represent the current state of a resolver instance. The enum also includes a human readable description of what the state represents.

1.1.22 rez.resources module

1.1.23 rez.rex module

class rez.rex.**Action**(**args*)

Bases: object

classmethod get_command_types()

classmethod register()

classmethod register_command_type(*name*, *klass*)

class rez.rex.**ActionInterpreter**

Bases: object

Abstract base class that provides callbacks for rex Actions. This class should not be used directly. Its methods are called by the *ActionManager* in response to actions issued by user code written using the rex python API.

Sub-classes should override the *get_output* method to return implementation-specific data structure. For example, an interpreter for a shell language like bash would return a string of shell code. An interpreter for an active python session might return a dictionary of the modified environment.

Sub-classes can override the *expand_env_vars* class variable to instruct the *ActionManager* whether or not to expand the value of environment variables which reference other variables (e.g. "this-${THAT}").

ENV_VAR_REGEX = re.compile('\\${([^\\{\\}]+?)}|\\$([a-zA-Z_]+[a-zA-Z0-9_]*?)')

alias(*key*, *value*)

appendenv(*key*, *value*)

This is optional, but if it is not implemented, you must implement setenv.

command(*value*)

comment(*value*)

error(*value*)

escape_string(*value*, *is_path=False*)

Escape a string.

Escape the given string so that special characters (such as quotes and whitespace) are treated properly. If *value* is a string, assume that this is an expandable string in this interpreter.

Note that *is_path* provided because of the special case where a path-like envvar is set. In this case, path normalization, if it needs to occur, has to be part of the string escaping process.

Note: This default implementation returns the string with no escaping applied.

Parameters

- **value** (str or *EscapedString*) – String to escape.
- **is_path** (*bool*) – True if the value is path-like.

Returns The escaped string.

Return type str

expand_env_vars = False

get_output(*style=<OutputStyle.file: ('Code as it would appear in a script file.',)>*)

Returns any implementation specific data.

Parameters **style** (*OutputStyle*) – Style affecting output format.

Returns Depends on implementation, but usually a code string.

info(*value*)

normalize_path(*path*)

Normalize a path.

Change *path* to a valid filepath representation for this interpreter.

IMPORTANT: Because var references like ${THIS} might be passed to funcs like appendvar, *path* might be in this form. You need to take that into account (ie, ensure normalization doesn't break such a var reference).

Parameters **path** (str) – A filepath which may be in posix format, or windows format, or some combination of the two. For eg, a string like *{root}/bin* on windows will evaluate to *C:.../bin* - in this case, the *cmd* shell would want to normalize this and convert to all forward slashes.

Returns The normalized path.

Return type str

normalize_paths(*value*)

Normalize value if it's a path(s).

Note that *value* may be more than one pathsep-delimited paths.

pathsep = ':'

prependenv(*key*, *value*)

This is optional, but if it is not implemented, you must implement setenv.

resetenv(*key*, *value*, *friends=None*)

setenv(*key*, *value*)

shebang()

source(*value*)

unsetenv(*key*)

class rez.rex.**ActionManager**(*interpreter*, *parent_environ=None*, *parent_variables=None*, *formatter=None*, *verbose=False*, *env_sep_map=None*)

Bases: object

Handles the execution book-keeping. Tracks env variable values, and triggers the callbacks of the *ActionInterpreter*.

alias(*key*, *value*)

appendenv(*key*, *value*)

command(*value*)

comment(*value*)

defined(*key*)

error(*value*)

expandvars(*value*, *format=True*)

get_action_methods()

return a list of methods on this class for executing actions. methods are return as a list of (name, func) tuples

get_output(*style=<OutputStyle.file: ('Code as it would appear in a script file.',)>*)

get_public_methods()

return a list of methods on this class which should be exposed in the rex API.

getenv(*key*)

info(*value=''*)

prependenv(*key*, *value*)

resetenv(*key*, *value*, *friends=None*)

setenv(*key*, *value*)

shebang()

source(*value*)

stop(*msg*, **nargs*)

undefined(*key*)

unsetenv(*key*)

class rez.rex.**Alias**(**args*)

Bases: *rez.rex.Action*

name = 'alias'

class rez.rex.**Appendenv**(**args*)

Bases: *rez.rex.Setenv*

name = 'appendenv'

class rez.rex.**Command**(**args*)

Bases: *rez.rex.Action*

name = 'command'

class rez.rex.**Comment**(**args*)

Bases: *rez.rex.Action*

name = 'comment'

class rez.rex.**EnvAction**(**args*)

Bases: *rez.rex.Action*

property key

property value

class rez.rex.**EnvironmentDict**(*manager*)

Bases: collections.abc.MutableMapping

Provides a mapping interface to *EnvironmentVariable* instances, which provide an object-oriented interface for recording environment variable manipulations.

__getitem__ is always guaranteed to return an *EnvironmentVariable* instance: it will not raise a KeyError.

keys() → a set-like object providing a view on D's keys

class rez.rex.**EnvironmentVariable**(*name*, *environ_map*)

Bases: object

class representing an environment variable

combined with EnvironmentDict class, records changes to the environment

append(*value*)

get()

property name

prepend(*value*)

reset(*value*, *friends=None*)

set(*value*)

setdefault(*value*)

set value if the variable does not yet exist

unset()

value()

class rez.rex.**Error**(**args*)

Bases: *rez.rex.Action*

name = 'error'

class rez.rex.**EscapedString**(*value*, *is_literal=False*)

Bases: object

Class for constructing literal or expandable strings, or a combination of both.

This determines how a string is escaped in an interpreter. For example, the following rex commands may result in the bash code shown:

```
>>> env.FOO = literal('oh "noes"')
>>> env.BAH = expandable('oh "noes"')
export FOO='oh "noes"'
export BAH="oh "noes""
```

You do not need to use *expandable* - a string by default is interpreted as expandable. However you can mix literals and expandables together, like so:

```
>>> env.FOO = literal("hello").expandable(" ${DUDE}")
export FOO='hello'" ${DUDE}"
```

Shorthand methods *e* and *l* are also supplied, for better readability:

```
>>> env.FOO = literal("hello").e(" ${DUDE}").l(", and welcome!")
export FOO='hello'" ${DUDE}"', and welcome!'
```

Note: you can use the *literal* and *expandable* free functions, rather than constructing a class instance directly.

copy()

classmethod demote(*value*)

classmethod disallow(*value*)

e(*value*)

expandable(*value*)

expanduser()

Analogous to os.path.expanduser.

Returns *EscapedString* object with expanded '~' references.

formatted(*func*)

Return the string with non-literal parts formatted.

Parameters **func** (*callable*) – Callable that translates a string into a formatted string.

Returns *EscapedString* object.

classmethod join(*sep*, *values*)

l(*value*)

literal(*value*)

classmethod promote(*value*)

split(*delimiter=None*)

Same as string.split(), but retains literal/expandable structure.

Returns List of *EscapedString*.

class rez.rex.**Info**(**args*)

Bases: *rez.rex.Action*

name = 'info'

class rez.rex.**NamespaceFormatter**(*namespace*)

Bases: string.Formatter

String formatter that, as well as expanding '{variable}' strings, also protects environment variable references such as ${THIS} so they do not get expanded as though {THIS} is a formatting target. Also, environment variable references such as $THIS are converted to ${THIS}, which gives consistency across shells, and avoids some problems with non-curly-braced variables in some situations.

format(*format_string*, **args*, ***kwargs*)

format_field(*value*, *format_spec*)

get_value(*key*, *args*, *kwds*)

class rez.rex.**OutputStyle**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to represent the style of code output when using Rex.

class rez.rex.**Prependenv**(**args*)

Bases: *rez.rex.Setenv*

name = 'prependenv'

class rez.rex.**Python**(*target_environ=None*, *passive=False*)

Bases: *rez.rex.ActionInterpreter*

Execute commands in the current python session

adjust_env_for_platform(*env*)

Make required platform-specific adjustments to env.

alias(*key*, *value*)

appendenv(*key*, *value*)

This is optional, but if it is not implemented, you must implement setenv.

apply_environ()

Apply changes to target environ.

command(*value*)

comment(*value*)

error(*value*)

expand_env_vars = True

get_key_token(*key*)

get_output(*style=<OutputStyle.file: ('Code as it would appear in a script file.',)>*)

Returns any implementation specific data.

Parameters **style** (*OutputStyle*) – Style affecting output format.

Returns Depends on implementation, but usually a code string.

info(*value*)

prependenv(*key*, *value*)

This is optional, but if it is not implemented, you must implement setenv.

resetenv(*key*, *value*, *friends=None*)

set_manager(*manager*)

setenv(*key*, *value*)

shebang()

source(*value*)

subprocess(*args*, ***subproc_kwargs*)

unsetenv(*key*)

class rez.rex.**Resetenv**(**args*)

Bases: *rez.rex.EnvAction*

property friends

name = 'resetenv'

post_exec(*interpreter*, *result*)

pre_exec(*interpreter*)

class rez.rex.**RexExecutor**(*interpreter=None*, *globals_map=None*, *parent_environ=None*, *parent_variables=None*, *shebang=True*, *add_default_namespaces=True*)

Bases: object

Runs an interpreter over code within the given namespace. You can also access namespaces and rex functions directly in the executor, like so:

RexExecutor ex() ex.setenv('FOO', 'BAH') ex.env.FOO_SET = 1 ex.alias('foo','foo -l')

property actions

List of Action objects that will be executed.

append_rez_path()

Append rez path to $PATH.

append_system_paths()

Append system paths to $PATH.

bind(*name*, *obj*)

Binds an object to the execution context.

Parameters

- **name** (*str*) –
- **obj** (*object*) – Object to bind.

classmethod compile_code(*code*, *filename=None*, *exec_namespace=None*)

Compile and possibly execute rex code.

Parameters

- **code** (*str or SourceCode*) – The python code to compile.
- **filename** (*str*) – File to associate with the code, will default to '<string>'.
- **exec_namespace** (*dict*) – Namespace to execute the code in. If None, the code is not executed.

Returns Compiled code object.

execute_code(*code*, *filename=None*, *isolate=False*)

Execute code within the execution context.

Parameters

- **code** (*str or SourceCode*) – Rex code to execute.
- **filename** (*str*) – Filename to report if there are syntax errors.

- **isolate** (*bool*) – If True, do not affect *self.globals* by executing this code. DEPRECATED - use *self.reset_globals* instead.

execute_function(*func*, **nargs*, ***kwargs*)

Execute a function object within the execution context. @returns The result of the function call.

expand(*value*)

get_output(*style=<OutputStyle.file: ('Code as it would appear in a script file.',)>*)

Returns the result of all previous calls to execute_code.

property interpreter

normalize_path(*path*)

Normalize a path.

Note that in many interpreters this will be unchanged.

Returns The normalized path.

Return type str

prepend_rez_path()

Prepend rez path to $PATH.

reset_globals()

Remove changes to globals dict post-context.

Any bindings (self.bind) will only be visible during this context.

unbind(*name*)

Unbind an object from the execution context.

Has no effect if the binding does not exist.

Parameters **name** (*str*) –

class rez.rex.**Setenv**(**args*)

Bases: *rez.rex.EnvAction*

name = 'setenv'

post_exec(*interpreter*, *result*)

pre_exec(*interpreter*)

class rez.rex.**Shebang**(**args*)

Bases: *rez.rex.Action*

name = 'shebang'

class rez.rex.**Source**(**args*)

Bases: *rez.rex.Action*

name = 'source'

class rez.rex.**Stop**(**args*)

Bases: *rez.rex.Action*

name = 'stop'

class rez.rex.**Unsetenv**(**args*)

Bases: *rez.rex.EnvAction*

name = 'unsetenv'

rez.rex.**expandable**(*value*)

Creates an expandable string.

rez.rex.**literal**(*value*)

Creates a literal string.

rez.rex.**optionvars**(*name*, *default=None*)

Access arbitrary data from rez config setting 'optionvars'.

Parameters

- **name** (*str*) – Name of the optionvar. Use dot notation for values in nested dicts.
- **default** (*object*) – Default value if setting is missing.

1.1.24 rez.rex_bindings module

Provides wrappers for various types for binding to Rex. We do not want to bind object instances directly in Rex, because this would create an indirect dependency between rex code in package.py files, and versions of Rez.

The classes in this file are intended to have simple interfaces that hide unnecessary data from Rex, and provide APIs that will not change.

class rez.rex_bindings.**Binding**(*data=None*)

Bases: object

Abstract base class.

class rez.rex_bindings.**EphemeralsBinding**(*ephemerals*)

Bases: *rez.rex_bindings.RO_MappingBinding*

Binds a list of resolved ephemeral packages.

Note that the leading '.' is implied when referring to ephemerals. Eg:

in package.py def commands():

if "foo.cli" in ephemerals: # will match '.foo.cli-*' request

get_range(*name*, *default=None*)

Returns ephemeral version range object

class rez.rex_bindings.**RO_MappingBinding**(*data*)

Bases: *rez.rex_bindings.Binding*

A read-only, dict-like object.

get(*name*, *default=None*)

class rez.rex_bindings.**RequirementsBinding**(*requirements*)

Bases: *rez.rex_bindings.RO_MappingBinding*

Binds a list of version.Requirement objects.

get_range(*name*, *default=None*)

Returns requirement version range object

class rez.rex_bindings.**VariantBinding**(*variant*, *cached_root=None*, *interpreter=None*)

Bases: *rez.rex_bindings.Binding*

Binds a packages.Variant object.

property root

This is here to support package caching. This ensures that references such as 'resolve.mypkg.root' resolve to the cached payload location, if the package is cached.

class rez.rex_bindings.**VariantsBinding**(*variant_bindings*)

Bases: *rez.rex_bindings.RO_MappingBinding*

Binds a list of packages.VariantBinding objects, under the package name of each variant.

class rez.rex_bindings.**VersionBinding**(*version*)

Bases: *rez.rex_bindings.Binding*

Binds a version.Version object.

```
>>> v = VersionBinding(Version("1.2.3alpha"))
>>> v.major
1
>>> v.patch
'3alpha'
>>> len(v)
3
>>> v[1]
2
>>> v[:3]
(1, 2, '3alpha')
>>> str(v)
'1.2.3alpha'
>>> print(v[5])
None
>>> v.as_tuple():
(1, 2, '3alpha')
```

as_tuple()

property major

property minor

property patch

rez.rex_bindings.**intersects**(*obj*, *range_*)

Test if an object intersects with the given version range.

Examples

in package.py def commands():

test a request if intersects(request.maya, '2019+'):

info('requested maya allows >=2019.*')

tests if a resolved version intersects with given range if intersects(resolve.maya, '2019+')

...

same as above if intersects(resolve.maya.version, '2019+')

...

disable my cli tools if .foo.cli-0 was specified def commands():

if intersects(ephemerals.get('foo.cli', '1'), '1'): env.PATH.append('{root}/bin')

Parameters

- **obj** (*VariantBinding or str*) – Object to test, either a variant, or requirement string (eg 'foo-1.2.3+').
- **range** (*str*) – Version range, eg '1.2+<2'

Returns True if the object intersects the given range.

Return type bool

1.1.25 rez.settings module

1.1.26 rez.shells module

Pluggable API for creating subshells using different programs, such as bash.

class rez.shells.**Shell**

Bases: *rez.rex.ActionInterpreter*

Class representing a shell, such as bash or tcsh.

classmethod convert_tokens(*value*)

Converts any token like ${VAR} and $VAR to shell specific form. Uses the ENV_VAR_REGEX to correctly parse tokens.

Parameters **value** – str to convert

Returns str with shell specific variables

property executable

classmethod executable_filepath()

Get full filepath to executable, or raise if not found.

classmethod executable_name()

Name of executable to create shell instance.

classmethod file_extension()

Get the file extension associated with the shell.

Returns Shell file extension.

Return type str

classmethod find_executable(*name*, *check_syspaths=False*)

Find an executable.

Parameters

- **name** (*str*) – Program name.
- **check_syspaths** (*bool*) – If True, check the standard system paths as well, if program was not found on current $PATH.

Returns Full filepath of executable.

Return type str

classmethod get_all_key_tokens(*key*)

Encodes the environment variable into the shell specific forms. Shells might implement multiple forms, but the most common/safest should be always returned at index 0.

Parameters **key** – Variable name to encode

Returns list of str with encoded token forms

classmethod get_key_token(*key*)

Encodes the environment variable into the shell specific form. Shells might implement multiple forms, but the most common/safest should be returned here.

Parameters **key** – Variable name to encode

Returns str of encoded token form

get_output(*style=<OutputStyle.file: ('Code as it would appear in a script file.',)>*)

Returns any implementation specific data.

Parameters **style** (*OutputStyle*) – Style affecting output format.

Returns Depends on implementation, but usually a code string.

classmethod get_syspaths()

classmethod is_available()

Determine if the shell is available to instantiate.

Returns True if the shell can be created.

Return type bool

classmethod join(*command*)

Note: Default to unix sh/bash- friendly behaviour.

Parameters **command** – A sequence of program arguments to be joined into a single string that can be executed in the current shell.

Returns A string object representing the command.

classmethod line_terminator()

Returns default line terminator

Return type str

classmethod name()

Plugin name.

new_shell()

Returns A new, reset shell of the same type.

schema_dict = {'prompt': <class 'str'>}

spawn_shell(*context_file*, *tmpdir*, *rcfile=None*, *norc=False*, *stdin=False*, *command=None*, *env=None*, *quiet=False*, *pre_command=None*, *add_rez=True*, *package_commands_sourced_first=None*, ***Popen_args*)

Spawn a possibly interactive subshell. :param context: _file File that must be sourced in the new shell, this

configures the Rez environment.

Parameters

- **tmpdir** – Tempfiles, if needed, should be created within this path.
- **rcfile** – Custom startup script.
- **norc** – Don't run startup scripts. Overrides rcfile.
- **stdin** – If True, read commands from stdin in a non-interactive shell. If a different non-False value, such as subprocess.PIPE, the same occurs, but stdin is also passed to the resulting subprocess.Popen object.
- **command** – If not None, execute this command in a non-interactive shell. If an empty string, don't run a command, but don't open an interactive shell either.
- **env** – Environ dict to execute the shell within; uses the current environment if None.
- **quiet** – If True, don't show the configuration summary, and suppress any stdout from startup scripts.
- **pre_command** – Command to inject before the shell command itself. This is for internal use.
- **add_rez** – If True, assume this shell is being used with rez, and do things such as set the prompt etc.
- **package_commands_sourced_first** – If True, source the context file before sourcing startup scripts (such as .bashrc). If False, source the context file AFTER. If None, use the configured setting.
- **popen_args** – args to pass to the shell process object constructor.

Returns A subprocess.Popen object representing the shell process.

classmethod startup_capabilities(*rcfile=False*, *norc=False*, *stdin=False*, *command=False*)

Given a set of options related to shell startup, return the actual options that will be applied. @returns 4-tuple representing applied value of each option.

class rez.shells.**UnixShell**

Bases: *rez.shells.Shell*

A base class for common *nix shells, such as bash and tcsh.

command(*value*)

command_arg = '-c'

comment(*value*)

error(*value*)

classmethod get_all_key_tokens(*key*)

Encodes the environment variable into the shell specific forms. Shells might implement multiple forms, but the most common/safest should be always returned at index 0.

Parameters **key** – Variable name to encode

Returns list of str with encoded token forms

classmethod get_startup_sequence(*rcfile*, *norc*, *stdin*, *command*)

Return a dict containing: - 'stdin': resulting stdin setting. - 'command': resulting command setting. - 'do_rcfile': True if a file should be sourced directly. - 'envvar': Env-var that points at a file to source at startup. Can be None. - 'files': Existing files that will be sourced (non-user-expanded), in source

order. This may also incorporate rcfile, and file pointed at via envvar. Can be empty.

- **'bind_files': Files to inject Rez binding into, even if that file doesn't** already exist.
- 'source_bind_files': Whether to source bind files, if they exist.

histfile = None

histvar = None

info(*value*)

last_command_status = '$?'

classmethod line_terminator()

Returns default line terminator

Return type str

norc_arg = None

rcfile_arg = None

resetenv(*key*, *value*, *friends=None*)

shebang()

spawn_shell(*context_file*, *tmpdir*, *rcfile=None*, *norc=False*, *stdin=False*, *command=None*, *env=None*, *quiet=False*, *pre_command=None*, *add_rez=True*, *package_commands_sourced_first=None*, ***Popen_args*)

Spawn a possibly interactive subshell. :param context: _file File that must be sourced in the new shell, this

configures the Rez environment.

Parameters

- **tmpdir** – Tempfiles, if needed, should be created within this path.
- **rcfile** – Custom startup script.
- **norc** – Don't run startup scripts. Overrides rcfile.
- **stdin** – If True, read commands from stdin in a non-interactive shell. If a different non-False value, such as subprocess.PIPE, the same occurs, but stdin is also passed to the resulting subprocess.Popen object.

- **command** – If not None, execute this command in a non-interactive shell. If an empty string, don't run a command, but don't open an interactive shell either.
- **env** – Environ dict to execute the shell within; uses the current environment if None.
- **quiet** – If True, don't show the configuration summary, and suppress any stdout from startup scripts.
- **pre_command** – Command to inject before the shell command itself. This is for internal use.
- **add_rez** – If True, assume this shell is being used with rez, and do things such as set the prompt etc.
- **package_commands_sourced_first** – If True, source the context file before sourcing startup scripts (such as .bashrc). If False, source the context file AFTER. If None, use the configured setting.
- **popen_args** – args to pass to the shell process object constructor.

Returns A subprocess.Popen object representing the shell process.

stdin_arg = '-s'

classmethod supports_command()

classmethod supports_norc()

classmethod supports_stdin()

syspaths = None

rez.shells.**create_shell**(*shell=None*, ***kwargs*)

Returns a Shell of the given or current type.

Returns Instance of given shell.

Return type *Shell*

rez.shells.**get_shell_class**(*shell=None*)

Get the plugin class associated with the given or current shell.

Returns Plugin class for shell.

Return type class

rez.shells.**get_shell_types**()

Returns the available shell types: bash, tcsh etc.

Returns: List of str: Shells.

1.1.27 rez.sigint module

1.1.28 rez.solver module

The dependency resolving module.

This gives direct access to the solver. You should use the resolve() function in resolve.py instead, which will use cached data where possible to provide you with a faster resolve.

See SOLVER.md for an in-depth description of how this module works.

class rez.solver.**Cycle**(*packages*)

Bases: *rez.solver.FailureReason*

The solve contains a cyclic dependency.

description()

involved_requirements()

class rez.solver.**DependencyConflict**(*dependency*, *conflicting_request*)

Bases: rez.solver._Common

A common dependency shared by all variants in a scope, conflicted with another scope in the current phase.

class rez.solver.**DependencyConflicts**(*conflicts*)

Bases: *rez.solver.FailureReason*

A common dependency in a scope conflicted with another scope in the current phase.

description()

involved_requirements()

class rez.solver.**FailureReason**

Bases: rez.solver._Common

description()

involved_requirements()

class rez.solver.**PackageVariant**(*variant*, *building*)

Bases: rez.solver._Common

A variant of a package.

property conflict_request_fams

get(*pkg_name*)

property handle

property index

property name

property request_fams

requires_list

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
```

(continues on next page)

(continued from previous page)

```
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property version

class rez.solver.**PackageVariantCache**(*solver*)

Bases: object

get_variant_slice(*package_name*, *range_*)

Get a list of variants from the cache.

Parameters

- **package_name** (*str*) – Name of package.
- **range** (*VersionRange*) – Package version range.

Returns *_PackageVariantSlice* object.

class rez.solver.**Reduction**(*name*, *version*, *variant_index*, *dependency*, *conflicting_request*)

Bases: rez.solver._Common

A variant was removed because its dependencies conflicted with another scope in the current phase.

involved_requirements()

reducee_str()

class rez.solver.**Solver**(*package_requests*, *package_paths*, *context=None*, *package_filter=None*, *package_orderers=None*, *callback=None*, *building=False*, *optimised=True*, *verbosity=0*, *buf=None*, *package_load_callback=None*, *prune_unfailed=True*, *suppress_passive=False*, *print_stats=False*)

Bases: rez.solver._Common

Solver.

A package solver takes a list of package requests (the 'request'), then runs a resolve algorithm in order to determine the 'resolve' - the list of non-conflicting packages that include all dependencies.

property cyclic_fail

Return True if the solve failed due to a cycle, False otherwise.

dump()

Print a formatted summary of the current solve state.

failure_description(*failure_index=None*)

Get a description of the failure.

This differs from *failure_reason* - in some cases, such as when a callback forces a failure, there is more information in the description than there is from *failure_reason*.

failure_packages(*failure_index=None*)

Get packages involved in a failure.

Parameters **failure_index** – See *failure_reason*.

Returns A list of Requirement objects.

`failure_reason`(*failure_index=None*)

Get the reason for a failure.

Parameters `failure_index` – Index of the fail to return the graph for (can be negative). If None, the most appropriate failure is chosen according to these rules: - If the fail is cyclic, the most recent fail (the one containing

the cycle) is used;

- If a callback has caused a failure, the most recent fail is used;
- Otherwise, the first fail is used.

Returns A *FailureReason* subclass instance describing the failure.

`get_fail_graph`(*failure_index=None*)

Returns a graph showing a solve failure.

Parameters `failure_index` – See *failure_reason*

Returns A pygraph.digraph object.

`get_graph`()

Returns the most recent solve graph.

This gives a graph showing the latest state of the solve. The specific graph returned depends on the solve status. When status is: unsolved: latest unsolved graph is returned; solved: final solved graph is returned; failed: most appropriate failure graph is returned (see *failure_reason*); cyclic: last failure is returned (contains cycle).

Returns A pygraph.digraph object.

`max_verbosity = 3`

`property num_fails`

Return the number of failed solve steps that have been executed. Note that num_solves is inclusive of failures.

`property num_solves`

Return the number of solve steps that have been executed.

`reset`()

Reset the solver, removing any current solve.

`property resolved_ephemerals`

Return the list of final ephemeral package ranges.

Note that conflict ephemerals are not included.

Returns Final non-conflict ephemerals, or None if the resolve did not complete or was unsuccessful.

Return type List of *Requirement*

`property resolved_packages`

Return a list of resolved variants.

Returns Resolved variants, or None if the resolve did not complete or was unsuccessful.

Return type list of *PackageVariant*

solve()

Attempt to solve the request.

property solve_stats

solve_step()

Perform a single solve step.

property status

Return the current status of the solve.

Returns Enum representation of the state of the solver.

Return type *SolverStatus*

timed(*target*)

class rez.solver.**SolverCallbackReturn**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum returned by the *callback* callable passed to a *Solver* instance.

class rez.solver.**SolverState**(*num_solves*, *num_fails*, *phase*)

Bases: object

Represent the current state of the solver instance for use with a callback.

class rez.solver.**SolverStatus**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Enum to represent the current state of a solver instance. The enum also includes a human readable description of what the state represents.

class rez.solver.**TotalReduction**(*reductions*)

Bases: *rez.solver.FailureReason*

All of a scope's variants were reduced away.

description()

involved_requirements()

class rez.solver.**VariantSelectMode**(*value*, *names=None*, *module=None*, *type=None*)

Bases: rez.vendor.enum.Enum

Variant selection mode.

1.1.29 rez.source_retrieval module

1.1.30 rez.system module

class rez.system.**System**

Bases: object

Access to underlying system data.

arch

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

clear_caches(*hard=False*)

Clear all caches in Rez.

Rez caches package contents and iteration during a python session. Thus newly released packages, and changes to existing packages, may not be picked up. You need to clear the cache for these changes to become visible.

Parameters **hard** (*bool*) – Perform a 'hard' cache clear. This just means that the memcached cache is also cleared. Generally this is not needed - this option is for debugging purposes.

domain

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

fqdn

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

get_summary_string()

Get a string summarising the state of Rez as a whole.

Returns String.

home

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

hostname

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
```

(continues on next page)

(continued from previous page)

```
bah
1
>>> f.bah
1
```

property is_production_rez_install

Return True if this is a production rez install.

os

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

platform

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

rez_bin_path

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

property rez_version

Returns the current version of Rez.

property selftest_is_running

Return True if tests are running via rez-selftest tool.

shell

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

user

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
```

(continues on next page)

(continued from previous page)

```
>>> f.bah
bah
1
>>> f.bah
1
```

variant

Simple property caching descriptor.

Example

```
>>> class Foo(object):
>>>     @cached_property
>>>     def bah(self):
>>>         print('bah')
>>>         return 1
>>>
>>> f = Foo()
>>> f.bah
bah
1
>>> f.bah
1
```

1.1.31 rez.util module

Misc useful stuff. TODO: Move this into rez.utils.?

class rez.util.**ProgressBar**(*label*, *max*)

Bases: rez.vendor.progress.bar.Bar

rez.util.**dedup**(*seq*)

Remove duplicates from a list while keeping order.

rez.util.**find_last_sublist**(*list_*, *sublist*)

Given a list, find the last occurance of a sublist within it.

Returns Index where the sublist starts, or None if there is no match.

rez.util.**get_close_matches**(*term*, *fields*, *fuzziness=0.4*, *key=None*)

rez.util.**get_close_pkgs**(*pkg*, *pkgs*, *fuzziness=0.4*)

rez.util.**is_non_string_iterable**(*arg*)

Python 2 and 3 compatible non-string iterable identifier

rez.util.**shlex_join**(*value*, *unsafe_regex=None*, *replacements=None*, *enclose_with=''*)

Join args into a valid shell command.

rez.util.**which**(**programs*, ***shutilwhich_kwargs*)

1.1.32 Module contents

1.2 Rez Plugins

1.2.1 Subpackages

rezplugins.build_system package

Submodules

rezplugins.build_system.bez module

rezplugins.build_system.cmake module

CMake-based build system

class rezplugins.build_system.cmake.**CMakeBuildSystem**(*working_dir*, *opts=None*, *package=None*, *write_build_scripts=False*, *verbose=False*, *build_args=[]*, *child_build_args=[]*)

Bases: *rez.build_system.BuildSystem*

The CMake build system.

The 'cmake' executable is run within the build environment. Rez supplies a library of cmake macros in the 'cmake_files' directory; these are added to cmake's searchpath and are available to use in your own CMakeLists.txt file.

The following CMake variables are available: - REZ_BUILD_TYPE: One of 'local', 'central'. Describes whether an install

is going to the local packages path, or the release packages path.

- REZ_BUILD_INSTALL: One of 0 or 1. If 1, an installation is taking place; if 0, just a build is occurring.

classmethod bind_cli(*parser*, *group*)

Expose parameters to an argparse.ArgumentParser that are specific to this build system.

Parameters

- **parser** (*ArgumentParser*) – Arg parser.
- **group** (*ArgumentGroup*) – Arg parser group - you should add args to this, NOT to *parser*.

build(*context*, *variant*, *build_path*, *install_path*, *install=False*, *build_type=<BuildType.local: 0>*)

Implement this method to perform the actual build.

Parameters

- **context** – A ResolvedContext object that the build process must be executed within.
- **variant** (*Variant*) – The variant being built.
- **build_path** – Where to write temporary build files. May be absolute or relative to working_dir.
- **install_path** (*str*) – The package repository path to install the package to, if installing. If None, defaults to *config.local_packages_path*.

- **install** – If True, install the build.
- **build_type** – A BuildType (i.e local or central).

Returns

- success: Bool indicating if the build was successful.
- **extra_files: List of created files of interest, not including** build targets. A good example is the interpreted context file, usually named 'build.rxt.sh' or similar. These files should be located under build_path. Rez may install them for debugging purposes.
- **build_env_script: If this instance was created with write_build_scripts** as True, then the build should generate a script which, when run by the user, places them in the build environment.

Return type A dict containing the following information

build_systems = {'codeblocks': 'CodeBlocks - Unix Makefiles', 'eclipse': 'Eclipse CDT4 - Unix Makefiles', 'make': 'Unix Makefiles', 'mingw': 'MinGW Makefiles', 'ninja': 'Ninja', 'nmake': 'NMake Makefiles', 'xcode': 'Xcode'}

build_targets = ['Debug', 'Release', 'RelWithDebInfo']

classmethod child_build_system()

Returns the child build system.

Some build systems, such as cmake, don't build the source directly. Instead, they build an interim set of build scripts that are then consumed by a second build system (such as make). You should implement this method if that's the case.

Returns Name of build system (corresponding to the plugin name) if this system has a child system, or None otherwise.

classmethod is_valid_root(*path*, *package=None*)

Return True if this build system can build the source in path.

classmethod name()

Return the name of the build system, eg 'make'.

schema_dict = {'build_system': Or('eclipse', 'codeblocks', 'make', 'nmake', 'mingw', 'xcode', 'ninja'), 'build_target': Or('Debug', 'Release', 'RelWithDebInfo'), 'cmake_args': [<class 'str'>], 'cmake_binary': Or(None, <class 'str'>), 'make_binary': Or(None, <class 'str'>)}

exception rezplugins.build_system.cmake.**RezCMakeError**(*value=None*)

Bases: *rez.exceptions.BuildSystemError*

rezplugins.build_system.cmake.**register_plugin**()

rezplugins.build_system.make module

Make-based build system

class rezplugins.build_system.make.**MakeBuildSystem**(*working_dir*, *opts=None*, *package=None*, *write_build_scripts=False*, *verbose=False*, *build_args=[]*, *child_build_args=[]*)

Bases: *rez.build_system.BuildSystem*

classmethod is_valid_root(*path*, *package=None*)

Return True if this build system can build the source in path.

classmethod name()

Return the name of the build system, eg 'make'.

rezplugins.build_system.make.**register_plugin**()

Module contents

rezplugins.release_hook package

Submodules

rezplugins.release_hook.emailer module

Sends a post-release email

class rezplugins.release_hook.emailer.**EmailReleaseHook**(*source_path*)

Bases: *rez.release_hook.ReleaseHook*

get_recipients()

load_recipients(*filepath*)

classmethod name()

Return name of source retriever, eg 'git'

post_release(*user*, *install_path*, *variants*, *release_message=None*, *changelog=None*, *previous_version=None*, ***kwargs*)

Post-release hook.

This is called after all package variants have been released.

Parameters

- **user** – Name of person who did the release.
- **install_path** – Directory the package was installed into.
- **variants** (list of *Variant*) – The variants that have been released.
- **release_message** – User-supplied release message.
- **changelog** – List of strings describing changes since last release.
- **previous_version** – Version of previously-release package, None if no previous release.
- **previous_revision** – Revision of previously-releaved package (type depends on repo - see ReleaseVCS.get_current_revision().

- **kwargs** – Reserved.

schema_dict = {'body': <class 'str'>, 'recipients': Or(<class 'str'>, [<class 'str'>]), 'sender': <class 'str'>, 'smtp_host': <class 'str'>, 'smtp_port': <class 'int'>, 'subject': <class 'str'>}

send_email(*subject*, *body*)

rezplugins.release_hook.emailer.**register_plugin**()

Module contents

rezplugins.release_vcs package

Submodules

rezplugins.release_vcs.git module

Git version control

class rezplugins.release_vcs.git.**GitReleaseVCS**(*pkg_root*, *vcs_root=None*)

Bases: *rez.release_vcs.ReleaseVCS*

create_release_tag(*tag_name*, *message=None*)

Create a tag in the repo.

Create a tag in the repository representing the release of the given version.

Parameters

- **tag_name** (*str*) – Tag name to write to the repo.
- **message** (*str*) – Message string to associate with the release.

classmethod export(*revision*, *path*)

Export the repository to the given path at the given revision.

Note: The directory at *path* must not exist, but the parent directory must exist.

Parameters

- **revision** (*object*) – Revision to export; current revision if None.
- **path** (*str*) – Directory to export the repository to.

get_changelog(*previous_revision=None*, *max_revisions=None*)

Get the changelog text since the given revision.

If previous_revision is not an ancestor (for example, the last release was from a different branch) you should still return a meaningful changelog - perhaps include a warning, and give changelog back to the last common ancestor.

Parameters

- **previous_revision** – The revision to give the changelog since. If
- **None** –

- **changelog.** (*give the entire*) –

Returns Changelog, as a string.

get_current_revision()

Get the current revision, this can be any type (str, dict etc) appropriate to your VCS implementation.

Note: You must ensure that a revision contains enough information to clone/export/checkout the repo elsewhere - otherwise you will not be able to implement *export*.

---

get_local_branch()

Returns the label of the current local branch.

get_relative_to_remote()

Return the number of commits we are relative to the remote. Negative is behind, positive in front, zero means we are matched to remote.

get_tracking_branch()

Returns (remote, branch) tuple, or None,None if there is no remote.

git(**nargs*)

classmethod is_valid_root(*path*)

Return True if the given path is a valid root directory for this version control system.

Note that this is different than whether the path is under the control of this type of vcs; to answer that question, use find_vcs_root

classmethod name()

Return the name of the VCS type, eg 'git'.

schema_dict = {'allow_no_upstream': <class 'bool'>}

classmethod search_parents_for_root()

Return True if this vcs type should check parent directories to find the root directory

tag_exists(*tag_name*)

Test if a tag exists in the repo.

Parameters **tag_name** (*str*) – Tag name to check for.

Returns True if the tag exists, False otherwise.

Return type bool

validate_repostate()

Ensure that the VCS working copy is up-to-date.

exception rezplugins.release_vcs.git.**GitReleaseVCSError**(*value=None*)

Bases: *rez.exceptions.ReleaseVCSError*

rezplugins.release_vcs.git.**register_plugin**()

rezplugins.release_vcs.hg module

Mercurial version control

class rezplugins.release_vcs.hg.**HgReleaseVCS**(*pkg_root*, *vcs_root=None*)

Bases: *rez.release_vcs.ReleaseVCS*

create_release_tag(*tag_name*, *message=None*)

Create a tag in the repo.

Create a tag in the repository representing the release of the given version.

Parameters

- **tag_name** (*str*) – Tag name to write to the repo.
- **message** (*str*) – Message string to associate with the release.

get_changelog(*previous_revision=None*, *max_revisions=None*)

Get the changelog text since the given revision.

If previous_revision is not an ancestor (for example, the last release was from a different branch) you should still return a meaningful changelog - perhaps include a warning, and give changelog back to the last common ancestor.

Parameters

- **previous_revision** – The revision to give the changelog since. If
- **None** –
- **changelog.** (*give the entire*) –

Returns Changelog, as a string.

get_current_revision()

Get the current revision, this can be any type (str, dict etc) appropriate to your VCS implementation.

Note: You must ensure that a revision contains enough information to clone/export/checkout the repo elsewhere - otherwise you will not be able to implement *export*.

---

get_default_url(*patch=False*)

get_paths(*patch=False*)

get_tags(*patch=False*)

hg(**nargs*, ***kwargs*)

is_ancestor(*commit1*, *commit2*, *patch=False*)

Returns True if commit1 is a direct ancestor of commit2, or False otherwise.

This method considers a commit to be a direct ancestor of itself

classmethod is_valid_root(*path*)

Return True if the given path is a valid root directory for this version control system.

Note that this is different than whether the path is under the control of this type of vcs; to answer that question, use find_vcs_root

classmethod name()

Return the name of the VCS type, eg 'git'.

classmethod search_parents_for_root()

Return True if this vcs type should check parent directories to find the root directory

tag_exists(*tag_name*)

Test if a tag exists in the repo.

Parameters **tag_name** (*str*) – Tag name to check for.

Returns True if the tag exists, False otherwise.

Return type bool

validate_repostate()

Ensure that the VCS working copy is up-to-date.

exception rezplugins.release_vcs.hg.**HgReleaseVCSError**(*value=None*)

Bases: *rez.exceptions.ReleaseVCSError*

rezplugins.release_vcs.hg.**register_plugin()**

rezplugins.release_vcs.svn module

Module contents

rezplugins.shell package

Submodules

rezplugins.shell.bash module

Bash shell

class rezplugins.shell.bash.**Bash**

Bases: *rezplugins.shell.sh.SH*

alias(*key*, *value*)

classmethod get_startup_sequence(*rcfile*, *norc*, *stdin*, *command*)

Return a dict containing: - 'stdin': resulting stdin setting. - 'command': resulting command setting. - 'do_rcfile': True if a file should be sourced directly. - 'envvar': Env-var that points at a file to source at startup. Can be None. - 'files': Existing files that will be sourced (non-user-expanded), in source

order. This may also incorporate rcfile, and file pointed at via envvar. Can be empty.

- **'bind_files': Files to inject Rez binding into, even if that file doesn't** already exist.
- 'source_bind_files': Whether to source bind files, if they exist.

classmethod name()

Plugin name.

norc_arg = '--norc'

rcfile_arg = '--rcfile'

classmethod startup_capabilities(*rcfile=False*, *norc=False*, *stdin=False*, *command=False*)

Given a set of options related to shell startup, return the actual options that will be applied. @returns 4-tuple representing applied value of each option.

rezplugins.shell.bash.**register_plugin**()

rezplugins.shell.csh module

CSH shell

class rezplugins.shell.csh.**CSH**

Bases: *rez.shells.UnixShell*

alias(*key*, *value*)

escape_string(*value*, *is_path=False*)

Escape a string.

Escape the given string so that special characters (such as quotes and whitespace) are treated properly. If *value* is a string, assume that this is an expandable string in this interpreter.

Note that *is_path* provided because of the special case where a path-like envvar is set. In this case, path normalization, if it needs to occur, has to be part of the string escaping process.

Note: This default implementation returns the string with no escaping applied.

Parameters

- **value** (str or *EscapedString*) – String to escape.
- **is_path** (*bool*) – True if the value is path-like.

Returns The escaped string.

Return type str

classmethod file_extension()

Get the file extension associated with the shell.

Returns Shell file extension.

Return type str

classmethod get_startup_sequence(*rcfile*, *norc*, *stdin*, *command*)

Return a dict containing: - 'stdin': resulting stdin setting. - 'command': resulting command setting. - 'do_rcfile': True if a file should be sourced directly. - 'envvar': Env-var that points at a file to source at startup. Can be None. - 'files': Existing files that will be sourced (non-user-expanded), in source

order. This may also incorporate rcfile, and file pointed at via envvar. Can be empty.

- **'bind_files': Files to inject Rez binding into, even if that file doesn't** already exist.
- 'source_bind_files': Whether to source bind files, if they exist.

classmethod get_syspaths()

histfile = '~/.history'

histvar = 'histfile'

classmethod join(*command*)

Note: Default to unix sh/bash- friendly behaviour.

Parameters **command** – A sequence of program arguments to be joined into a single string that can be executed in the current shell.

Returns A string object representing the command.

last_command_status = '$status'

classmethod name()

Plugin name.

norc_arg = '-f'

setenv(*key*, *value*)

source(*value*)

classmethod startup_capabilities(*rcfile=False*, *norc=False*, *stdin=False*, *command=False*)

Given a set of options related to shell startup, return the actual options that will be applied. @returns 4-tuple representing applied value of each option.

unsetenv(*key*)

rezplugins.shell.csh.**register_plugin**()

rezplugins.shell.sh module

SH shell

class rezplugins.shell.sh.**SH**

Bases: *rez.shells.UnixShell*

alias(*key*, *value*)

escape_string(*value*, *is_path=False*)

Escape a string.

Escape the given string so that special characters (such as quotes and whitespace) are treated properly. If *value* is a string, assume that this is an expandable string in this interpreter.

Note that *is_path* provided because of the special case where a path-like envvar is set. In this case, path normalization, if it needs to occur, has to be part of the string escaping process.

Note: This default implementation returns the string with no escaping applied.

Parameters

- **value** (str or *EscapedString*) – String to escape.
- **is_path** (*bool*) – True if the value is path-like.

Returns The escaped string.

Return type str

classmethod file_extension()

Get the file extension associated with the shell.

Returns Shell file extension.

Return type str

classmethod get_startup_sequence(*rcfile*, *norc*, *stdin*, *command*)

Return a dict containing: - 'stdin': resulting stdin setting. - 'command': resulting command setting. - 'do_rcfile': True if a file should be sourced directly. - 'envvar': Env-var that points at a file to source at startup. Can be None. - 'files': Existing files that will be sourced (non-user-expanded), in source

order. This may also incorporate rcfile, and file pointed at via envvar. Can be empty.

- **'bind_files': Files to inject Rez binding into, even if that file doesn't** already exist.
- 'source_bind_files': Whether to source bind files, if they exist.

classmethod get_syspaths()

histfile = '~/.bash_history'

histvar = 'HISTFILE'

classmethod name()

Plugin name.

norc_arg = '--noprofile'

setenv(*key*, *value*)

source(*value*)

classmethod startup_capabilities(*rcfile=False*, *norc=False*, *stdin=False*, *command=False*)

Given a set of options related to shell startup, return the actual options that will be applied. @returns 4-tuple representing applied value of each option.

unsetenv(*key*)

rezplugins.shell.sh.**register_plugin()**

rezplugins.shell.tcsh module

TCSH shell

class rezplugins.shell.tcsh.**TCSH**

Bases: *rezplugins.shell.csh.CSH*

escape_string(*value*, *is_path=False*)

Escape a string.

Escape the given string so that special characters (such as quotes and whitespace) are treated properly. If *value* is a string, assume that this is an expandable string in this interpreter.

Note that *is_path* provided because of the special case where a path-like envvar is set. In this case, path normalization, if it needs to occur, has to be part of the string escaping process.

Note: This default implementation returns the string with no escaping applied.

Parameters

- **value** (str or *EscapedString*) – String to escape.
- **is_path** (*bool*) – True if the value is path-like.

Returns The escaped string.

Return type str

classmethod **name**()

Plugin name.

rezplugins.shell.tcsh.**register_plugin**()

rezplugins.shell.windows module

Module contents

rezplugins.source_retriever package

Submodules

rezplugins.source_retriever.archive module

rezplugins.source_retriever.git module

rezplugins.source_retriever.hg module

Module contents

1.2.2 Module contents

CHAPTER

TWO

ONE-LINERS

A list of useful one-liners for rez-config and related tools

Display info about the package foo:

```
rez-info foo
```

List the packages that foo depends on:

```
rez-config --print-packages foo
```

Jump into an environment containing foo-5(.x.x.x...):

```
rez-env foo-5
```

Run a command inside a configured shell

```
rez-run foo-5 bah-1.2 -- my-command
```

Show the resolve dot-graph for a given shell:

```
rez-run foo-5 bah-1.2 fee -- rez-context-image
```

Display a dot-graph showing the first failed attempt of the given configuration PKGS:

```
rez-config --max-fails=0 --dot-file=/tmp/dot.jpg PKGS ; firefox /tmp/dot.jpg
```

Show a dot-graph of all the packages dependent on foo:

```
rez-depends show-dot foo
```

List every package in the system, and the description of each

```
rez-config-list --desc
```

Show the resolve dot-graph for a given shell, but just show that part of the graph that contains packages dependent (directly or indirectly) on fee:

```
rez-run foo-5 bah-1.2 fee -- rez-context-image --package=fee
```

Run a command inside a toolchain wrapper:

```
rez-run mytoolchain -- sometool -- some-command
```

Jump into a toolchain, and then into a wrapper's env:

```
rez-run mytoolchain
sometool ---i
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W