anise - Manual

Release 4.3.6278

Josef Hahn

Mar 17, 2021

CONTENTS

CHAPTER

ONE

LICENSE

anise is written by Josef Hahn under the terms of the AGPLv3.

Please read the *LICENSE* file from the package and the *Dependencies* section for included third-party stuff.

CHAPTER

TWO

ABOUT

Anise is a Python-based execution engine for automation tasks.

Automation tasks exist in software development, and probably all kinds of other sectors. They typically require the execution of different smaller and larger tools. Complex tasks often need a sequence of many steps to execute, with some steps having dependencies to each other. Manually triggering all these steps in the graphical interfaces of all the involved tools is possible in theory, but will generate errors and frustration after some cycles.

The automation interfaces of those tools are sometimes easier, but sometimes they are error-prone. Some tasks may also need to ask the user for some information in an interactive way. Some smaller parts might also be machine-specific (e.g. filesystem paths or the code how to access a password vault), while the entire task must be runnable on some different machines. In some situations, this can lead to a rather intransparent forest of different tools, with unique oddnesses and special conventions. As the number of different project increases, you will see more and more different tools, often doing a similar job, but for different platforms or frameworks and, of course, with different usage conventions. Spontaneously written glue scripts help in the beginning, but will explode as the complexity exceeds some threshold.

Typical tasks in software development could be:

- Generating documentation
- Testing
- Automatic code generation
- Creating packages
- Creating a homepage, automatically built from the available version information, the packages, the documentation and so on
- Deploying this homepage to a web server
- Handling version information - e.g. print it in the manual
- and many more

The Anise framework allows you to implement all those tasks in a structured but generic way in a combination of XML and Python code. Once you have created this stuff at a defined place in your project, Anise lets you easily execute your tasks from command line (or from any editor if you embed it somehow). This gives you a common and easy interface to all your 'tool glue' code.

The Anise engine executes arbitrary Python source code and provides some additional services like logging, parameter passing from command line, basic graphical user interface support, a plugin interface, a flexible event system, injecting code and data from other place, dependencies between code fragments, and more.

On top of this engine, Anise comes with a bunch of implementations that fulfill tasks (or parts of them) of software development. There is a testing module, a documentation- and homepage-generator, some package building methods and a lot more. The implementations use the event system in many places in order to allow customization in a

somewhat technical but very flexible way. Even so, those implementations are rather specific and it depends on the particular case, if, and how many of those implementations are useful.

CHAPTER

THREE

UP-TO-DATE?

Are you currently reading from another source than the homepage? Are you in doubt if that place is up-to-date? If yes, you should visit https://pseudopolis.eu/wiki/pino/projs/anise and check that. You are currently reading the manual for version 4.3.6278.

CHAPTER

FOUR

MATURITY

anise is in production-stable state.

CHAPTER

FIVE

DEPENDENCIES

There are external parts that are used by anise. Many thanks to the projects and all participants. Some installation methods might handle dependency installation automatically, while others leave that up to you.

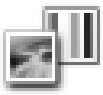
 *Python 3.7*, required

 *Typical GNU/Linux Desktop*, recommended

 *PyQt5 incl. WebEngine*, required (has alternatives) : for some user interface enhancements; you can comfortably work without it (inside the browser then)

 *subversion*, optional

 *git*, optional

 *font 'Symbola'*, included : for logo symbol; free for use; copied from here.

 *font 'Khula'*, included : for websites; by Erin McLaughlin; copied from here.

 *font 'Inconsolata'*, included : for websites; by Raph Levien; copied from here.

 *banner image*, included : _meta/background.png; license CC BY-SA 3.0; copied from here.

 *third-party project logos*, included : from some projects (see their websites for details).

all files in /_meta, included : if not mentioned otherwise, Copyright 2015 Josef Hahn under license CC BY-SA 3.0 license.

icon set 'oxygen', included : files 'anise/data/icons/userfeedback'; license see homepage.

CHAPTER

SIX

INTRODUCTION

Please read how to make Anise ready for the first steps in *Appendix: Installation*.

6.1 Typical usage scenarios

In very general words, the About section already described some scenarios, in which Anise could bring substantial advantages in terms of ergonomics, reliability and quality for your processes.

Basically, whenever you are going to automate the interaction of many different tools, Anise provides you with an engine for your glue code. This can either mean to orchestrate a large sequence of tool executions and to fulfill the dependencies they might have. But a large field of small tasks, each just calling one or two of tools, is also worth considering to consolidate in a list of Anise projects. Instead of zillions of different usage conventions, you run all of them in the same way on top of Anise with always the same user interface. This also includes access to the Anise user interface foundation, which is be the basis for many kinds of user interaction in task executions as well as lots of graphical helpers and configuration guides.

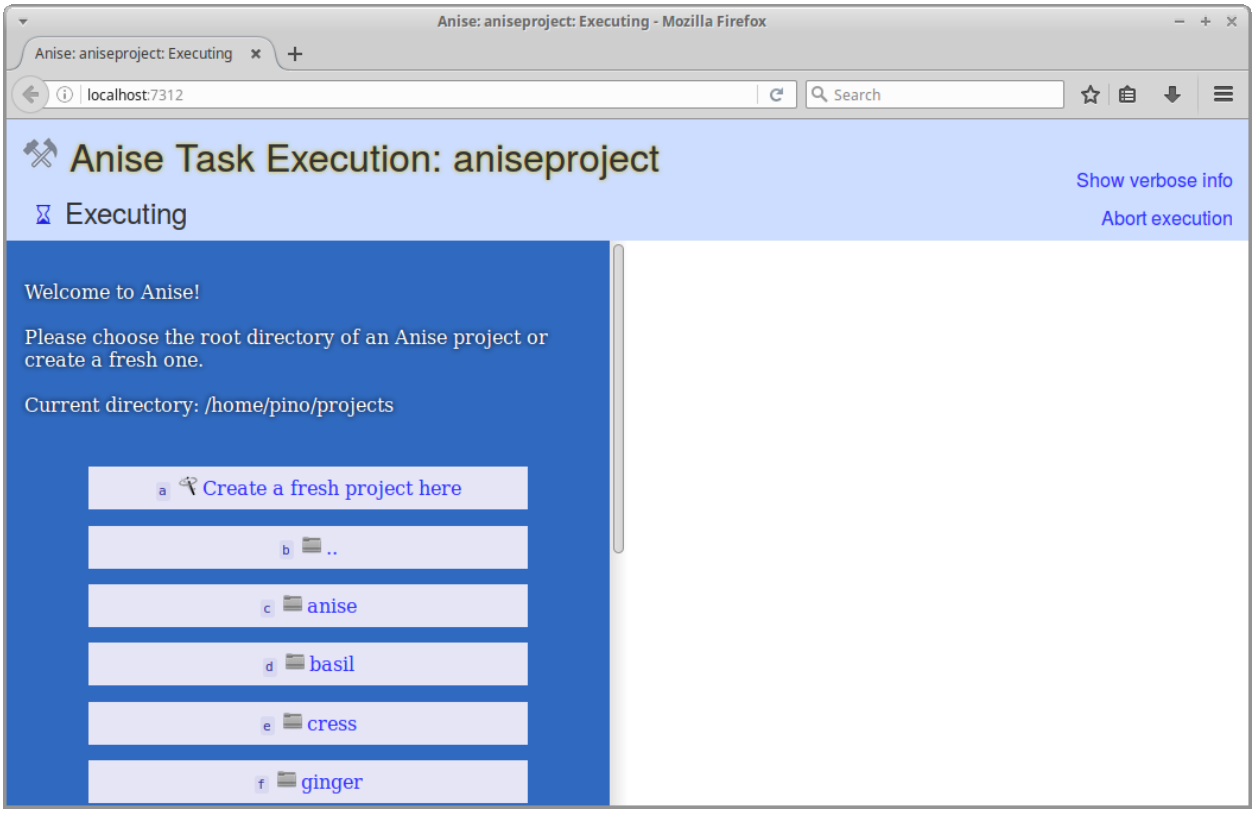
While the Anise engine is open for lots of very different automation tasks, the program comes with a large list of ready-to-use implementations for some specific tasks around software development and package deployment. Those tasks can run some compilers, create tarballs, Debian packages and some more, can run some test frameworks, documentation generation via Doxygen (development documentation with a programming interface reference as well as textual documentation) and can build and deploy a project website. Some of those included tasks may be useful for certain scenarios. But in most cases it is probably required to actually implement most of the tasks (i.e. the glue between your tools), while Anise provides a scaffold and a common facade for them.

You will learn later about the *The Anise execution model*, the *Programming Interface* and about ways to actually execute tasks. You will also read a bit more about the *specific task implementations* later.

6.2 First steps

The first steps are to create a project description file for your Anise automation project and to fill it with an actual configuration. This section shows a simple example that helps for a very trivial but functional Anise project.

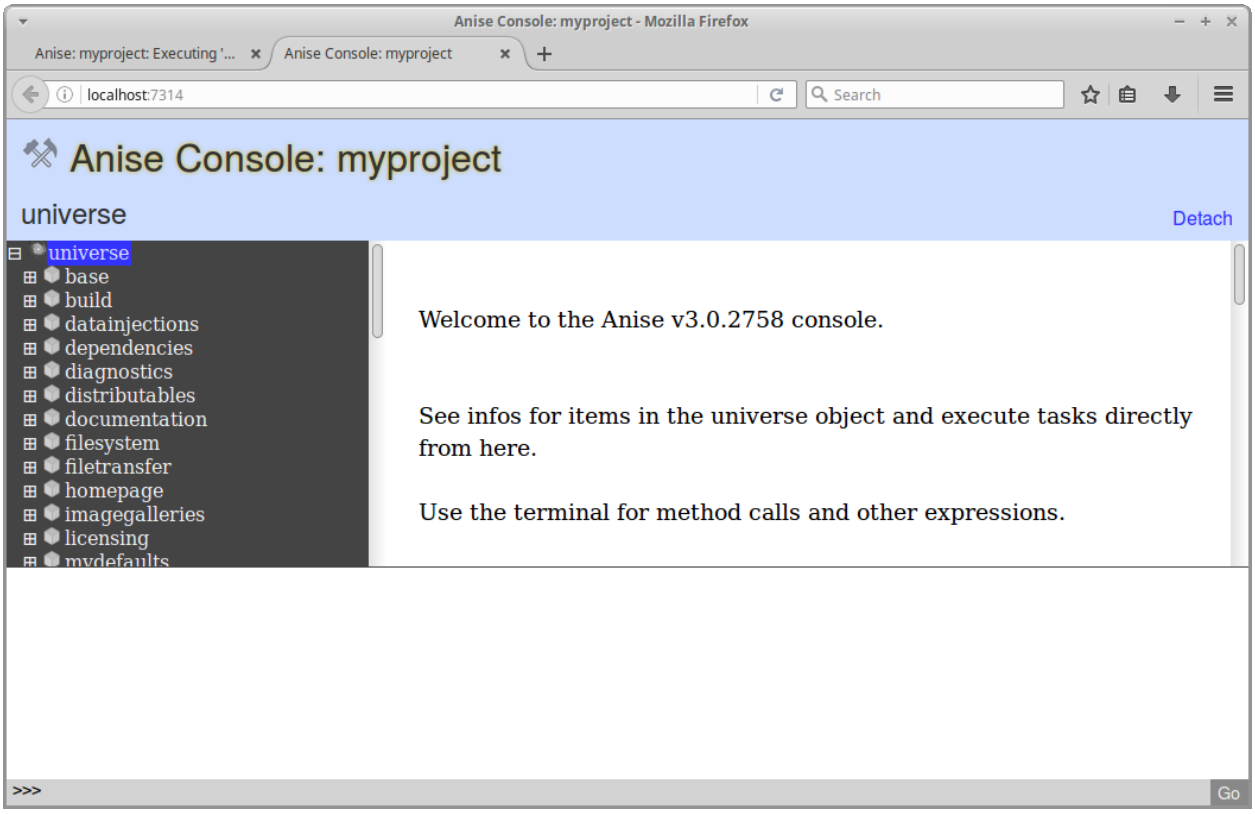
Start Anise from the start menu or execute `anise` on the command line. An Anise interface should come up, asking you for some action.



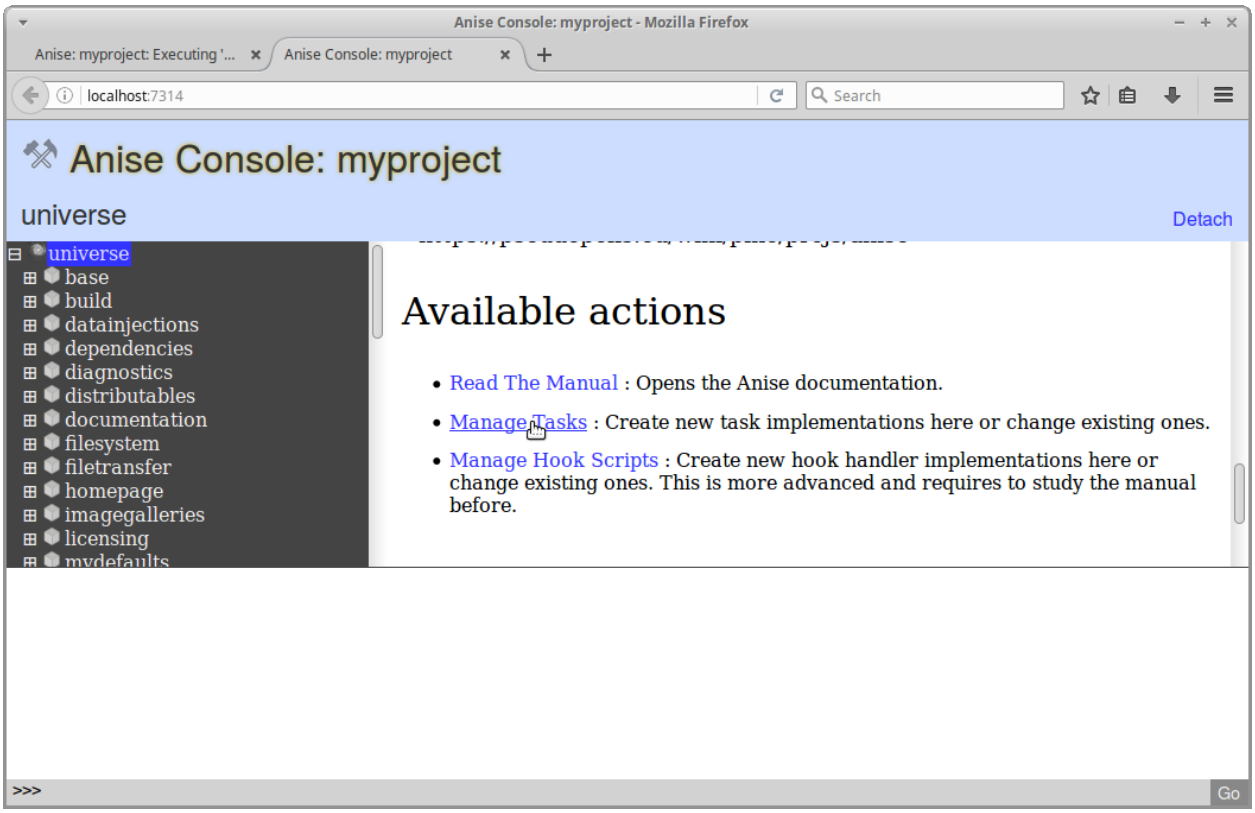
Choose the root directory of your project here. If you want to create a fresh automation project and you do not have any files yet, create a fresh directory for it somewhere beforehand.

Select 'Create a fresh project here' when you have reached the project root directory. This creates the file `_meta/_projectdesc` (Windows: `_meta_projectdesc`), which is called the *project description file*. This file can either be edited manually with an arbitrary text editor or in a guided way.

For a fresh project, the Anise Console will open (for an unfresh one, the console is still available as task `console` from the task chooser). It contains all kinds of configuration guides and is useful in other situations as well as we see later.

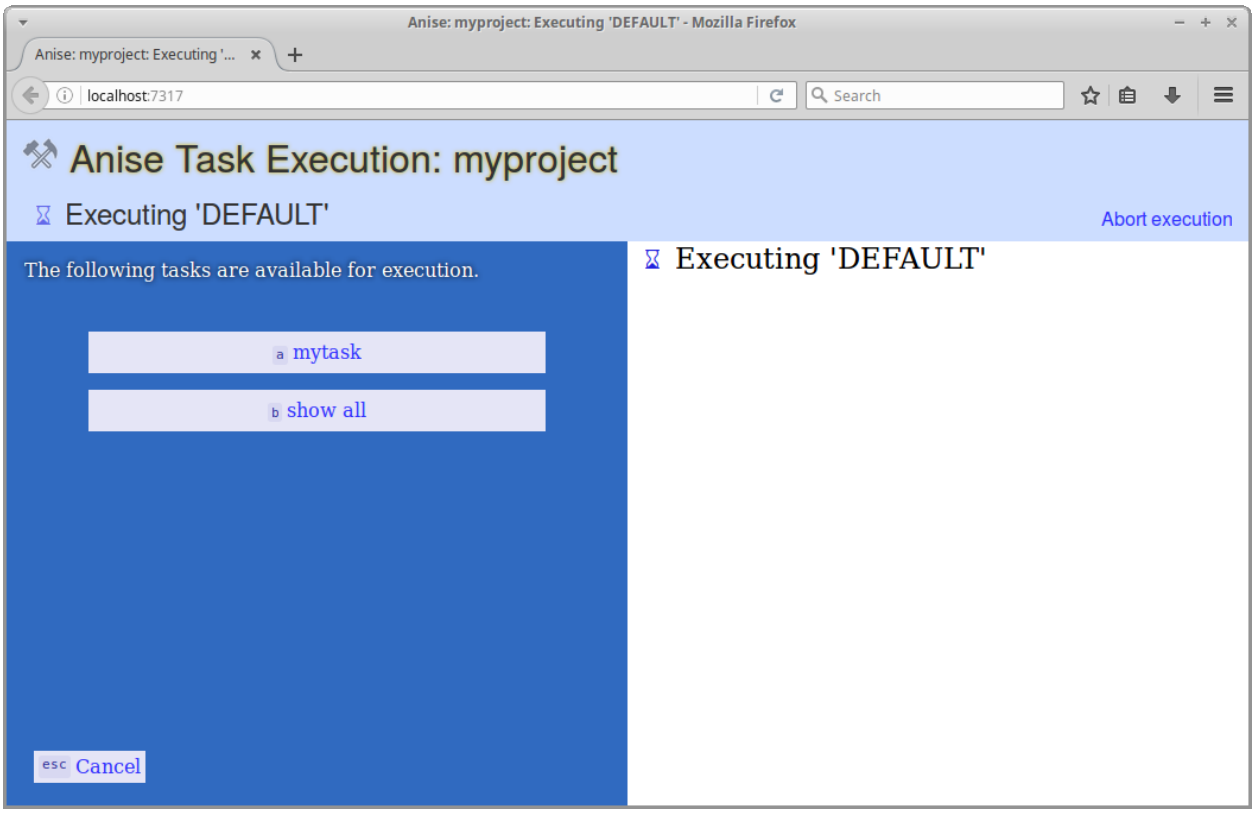


Find the “Manage Tasks” action in the main part, trigger it and follow the on-screen instructions.



In the end, you should have a new task with a more or less useful Python implementation added to your project.

After closing the console, execute **`anise`** again for a task chooser or **`anise mytask`** for directly executing the task `mytask`.



You will find more details about the console in the *Anise console* section.

CHAPTER

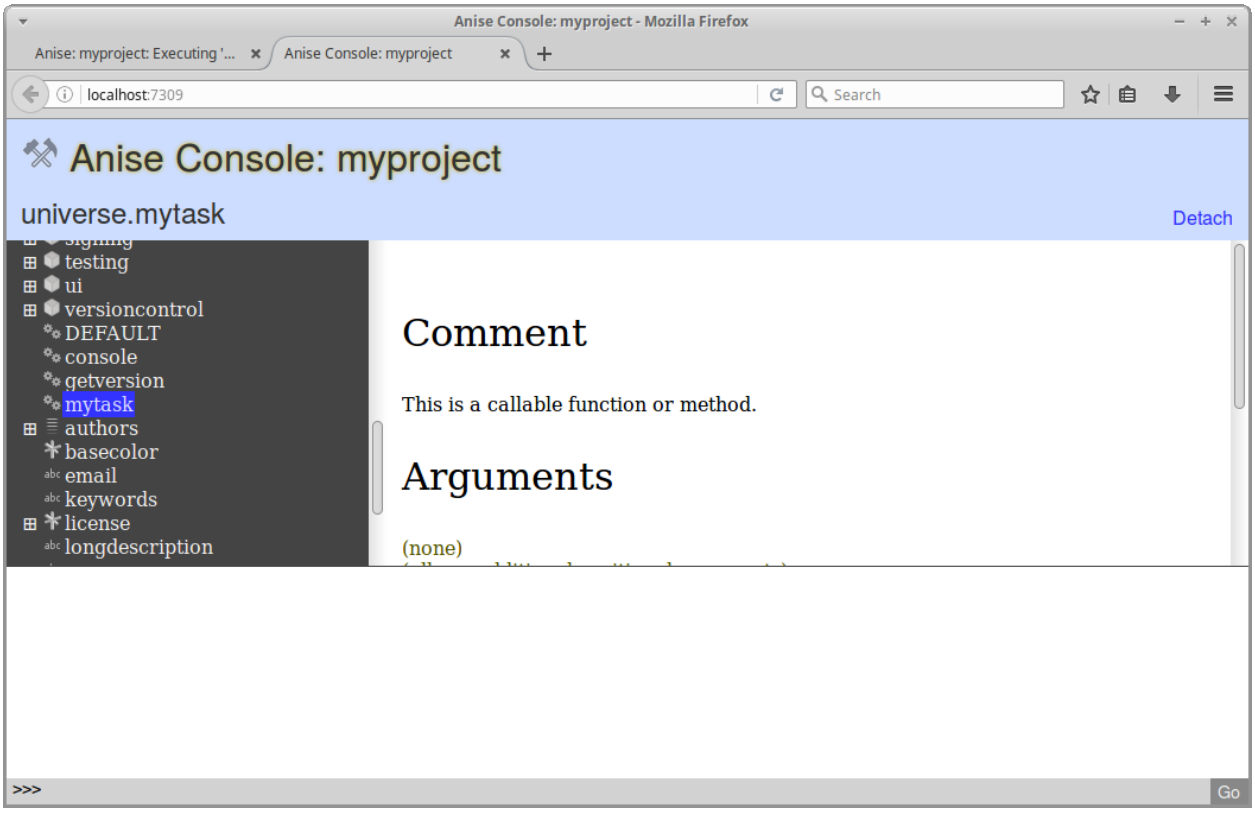
SEVEN

THE ANISE EXECUTION MODEL

The execution model describes what the Anise engine actually executes and which runtime environment and infrastructure it prepares for a task implementation.

7.1 The universe object

The universe object is the central part of the execution of tasks. It is one global data structure, which contains all the information for executing tasks. This includes all the metadata, like the project's name, maybe its license and so on, as well as all task implementations.



You will learn later how to get and manipulate this object in different situations.

7.2 Tasks

A task is the piece of program logic for execution on the Anise engine. Each automation action of your project is to be implemented in one task.

Technically, a task is just a simple Python function (for people who already took a look at `_meta/_projectdesc`: *`anise.features.base.pytask()`* is just a small wrapper around that fact) that is available as a member of the universe object. It must not require any arguments, otherwise the engine cannot execute it as a task. Instead, member variables from the universe objects or user interaction are used for getting information or parameters from outside.

In *First steps* you already created a task in a very direct way. Later parts will also show more indirect ways of creating tasks.

7.3 The project description file

The project description file must be either in the `_meta` subdirectory (recommended!) or in the root directory of your project and must be named `_projectdesc`. All configurations and every relevant piece of information about your project are specified in this file. The complete Anise operation will run just on top of it (this is not entirely correct, but correct enough for the moment).

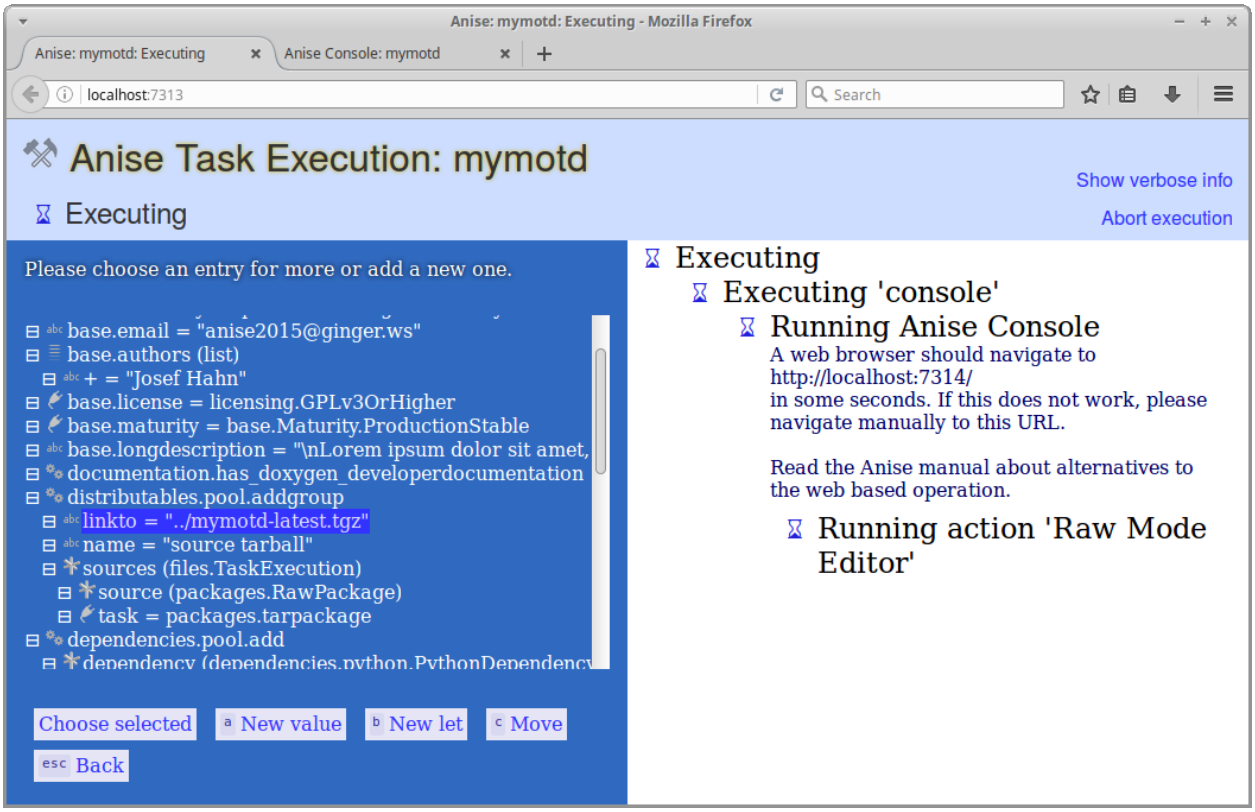
You might already have created the initial version of this file in the *First steps*. This file contains an xml document in an Anise-specific format. The general structure is the following:

```
<?xml version="1.0" ?>
<aniseprojectdesc>
    <val k="name" v="myprojectname"/>
    <val k="base.version" v="0.1"/>
    <let k="somefeature.someobject.somemethod">
        <val k="someparam" t="str" v="foo"/>
        <val k="anotherparam" t="bool" v="True"/>
    </let>
</aniseprojectdesc>
```

This file essentially populates the universe object with information before the actual execution begins. The root `aniseprojectdesc` contains a list of `val` and `let` nodes.

7.3.1 The raw mode editor

A graphical, yet not too abstract way to inspect and change the content of this file is the 'Raw Mode Editor'. You find it in the console, listed in the 'Available actions' for the root node.



7.3.2 The `val` node

A `val` node assigns values to the universe object (at least if it is a direct child of the root node).

The `k` attribute specifies the member name used for storing the value in the universe object.

The value either has a primitive type like a string or a boolean or something composed like an instance of a Python class. The type is specified in the `t` attribute. If a `val` node defines a non-primitive value, it contains nested `val` structures for each constructor parameter.

The types `list` and `dict` exist as typical containers. They also contain nested `val` structures holding the actual content. For the former one, the child nodes must not have `k` attributes (since elements have keys in dictionaries but not in list)!

Those values will potentially be read by the task at execution time in order to configure what it does in detail.

Reference type

There is one more type; it has a very special behavior: A value of type `objectref` contains a snippet of python code that will be evaluated at the time the xml file is read. It may be something as simple as a variable name like `<val k="base.license" oref="licensing.AGPLv3"/>` or something more complex like `<val k="sometext" oref="utils.basic.readfromfile('...')"/>`, which dynamically generates the content for a variable.

Note: This evaluation happens very early. It is not possible to refer to values that become assigned in later stages, like the *`anise.framework.engine.HOOK_AFTER_PREPARATION` hook*.

7.3.3 The `let` node

A `let` node is a different way to populate the universe object. It does not directly lead to a new member in it, but it triggers a management function (do not confuse those functions with tasks; tasks could become executed in later steps if the user requested it), which in turn modifies some data structures of the universe object. This encapsulates some rather complex data structures and give you an easier interface to them (as an example, think about a function for adding a bunch of file names together with descriptions to an mediagallery data structure).

The `k` attribute specifies the full qualified method name that includes the feature name (the *Features* section will explain this term), maybe a sub-structure within it, and, of course, the method name - like `documentation.pools.add`.

Parameter assignments to this function are specified with `val` subnodes. The `k` attributes of them specifiy the names of the parameters. As a shortcut, string parameters are also allowed to be written directly as attributes of the `let` node.

7.4 Executing tasks

Once a *project description file* is created and configured, the user primarily runs Anise for executing tasks. Which tasks are available is specified - although often in an indirect way - and/or parameterized in the project description file. Assuming to have a `dorelease` task available, simply call

```
anise dorelease
```

in a command line **from the project directory**. If this call returns a result value, it is written to the terminal. Find a description of *command-line options* below.

7.5 The hook system

The hook system is a simple and flexible event mechanism. The Anise engine itself as well as various parts of *feature* implementations use them for loosely working together. It is one of the most important and visible mechanisms in Anise.

Events are represented by a name string that typically is available as a constant somewhere. The name of this constant typically begins with `HOOK_`, like in `HOOK_MYEVENT`.

```
HOOK_MYEVENT = features.Hook()
```

A feature can register some handlers to it. Those will eventually be executed when the event is triggered.

Some piece of code may feel responsible for *providing* (resp. *triggering*) this event. It does so by asking the Anise infrastructure for a list of all registered handlers and executing them. The exact event execution model is up to this provider. It may execute all registered handlers, parts of them or cancel the execution when some conditions arrive.

You find more infos and sample code in *... use hooks* and *... define and trigger hooks*.

The exact meaning of the phrase 'hook' may differ from case to case: Sometimes it means an event itself, e.g. the notional `HOOK_MYEVENT`. In other situations it means a single handler for an event. It should be clear by the context which meaning applies.

CHAPTER

EIGHT

THE ANISE PROGRAMMING INTERFACE

The Anise Programming Interface provides ways for automation projects to interact with the Anise infrastructure. This section gives a coarse overview of all the building blocks and how they play together. It also touches some of the internal parts.

After reading the manual, the best way to get a less abstract knowledge is to take a look at *Appendix: Recipes, how to*.

There is also an API reference available, which gives detailed information about the existing classes, functions and parameters. Whenever the manual refers to those items (they all begin with `anise.`), it is a good idea to read the reference as well. It often contains lots of relevant additional details.

8.1 Framework

The package *anise.framework* contains the core foundation. The Anise engine resides here as well as its auxiliary data structures and object classes. Find those modules (and some more) in this namespace:

- *anise.framework.engine*: The Anise engine and some direct helpers.
- *anise.framework.projects*: This module read and writes *project description files* and implements *The universe object* (in *anise.framework.projects.Universe*) as well as the interaction between those parts.
- *anise.framework.features*: The Anise plug-in system (typically called 'features'). The entire high-level functionality of an Anise project (the custom one as well as included one) comes as a 'feature'. This module does not contain any actual features but just the framework logic for loading and running them. More details are described in the *Features* section.
- *anise.framework.exceptions*: Anise exception types.
- *anise.framework.files*: Some data structures for working with file hierarchies.
- *anise.framework.report*: Keeps progress information and output text of the task execution.

8.2 Features

Anise features provide the complete high-level automation functionality. Each available feature is plugged into the engine on startup. By adding handlers to some *hooks*, it can execute initialization steps for bringing some configuration data structures in place. Those are configured according to the actual project needs in the *project description file*. A feature can also provide task implementations - either for manually adding them or automatically added to the universe object and thereby directly ready for execution.

Manually adding a task is possible with *anise.framework.projects.Universe.addtask()*, as *some examples later on* will show.

The namespace *anise.features* contains all features that are included in Anise. Some of them fulfill rather special tasks, but others are more generic, maybe even have infrastructure characteristics, and often used by other features as well.

Referring to items of features from within a project description file requires a name in Python module notation relative to the feature loading locations! This means, for the features included in anise, the prefix `anise.features` is to be omitted.

```
<?xml version="1.0" ?>
<aniseprojectdesc>
    <let k="releasing.tasks.add" destination="/tmp/mymotdserver/">
        <!-- this is from anise.features.releasing --> ...
    </let>
    <let k="dependencies.pool.add">
        <!-- this is from anise.features.dependencies --> ...
    </let>
</aniseprojectdesc>
```

Note: Although the developer reference contains documentation about *anise.features* as well, the value of it is restricted due to its static nature. The relevant parts of most features appear at runtime by means of dynamic initialization routines and can differ depending on the configuration and situation. For a deeper understanding how an existing feature works, it is a much better approach to inspect it in the *Console*.

8.2.1 Sub-features

A feature name may be identical to a namespace that contains sub-items, similar as it is with Python modules. Due to that, there might be a feature `foo`, but also some sub-features `foo.abc` and `foo.xyz`.

8.3 Utils

The package *anise.utils* contains basic utility functions and low-level counterparts of many of the included features. They are often more flexible on the one hand, but more technical on the other hand. They also do not use the Anise infrastructure (there might be rare exceptions for some reasons), so they also do not interact with *The universe object*.

They are used by the Anise features, but are often flexible enough for being useful for custom features as well.

CHAPTER

NINE

MORE ABOUT EXECUTING ANISE

The **anise** executable can be called directly from a start menu link or from command line. It must be called from the root directory of your project (important also for start menu links). It is used for executing your tasks and for guided configuration (which can be more convenient than editing the xml file). The *Appendix: Command line* lists all available parameters.

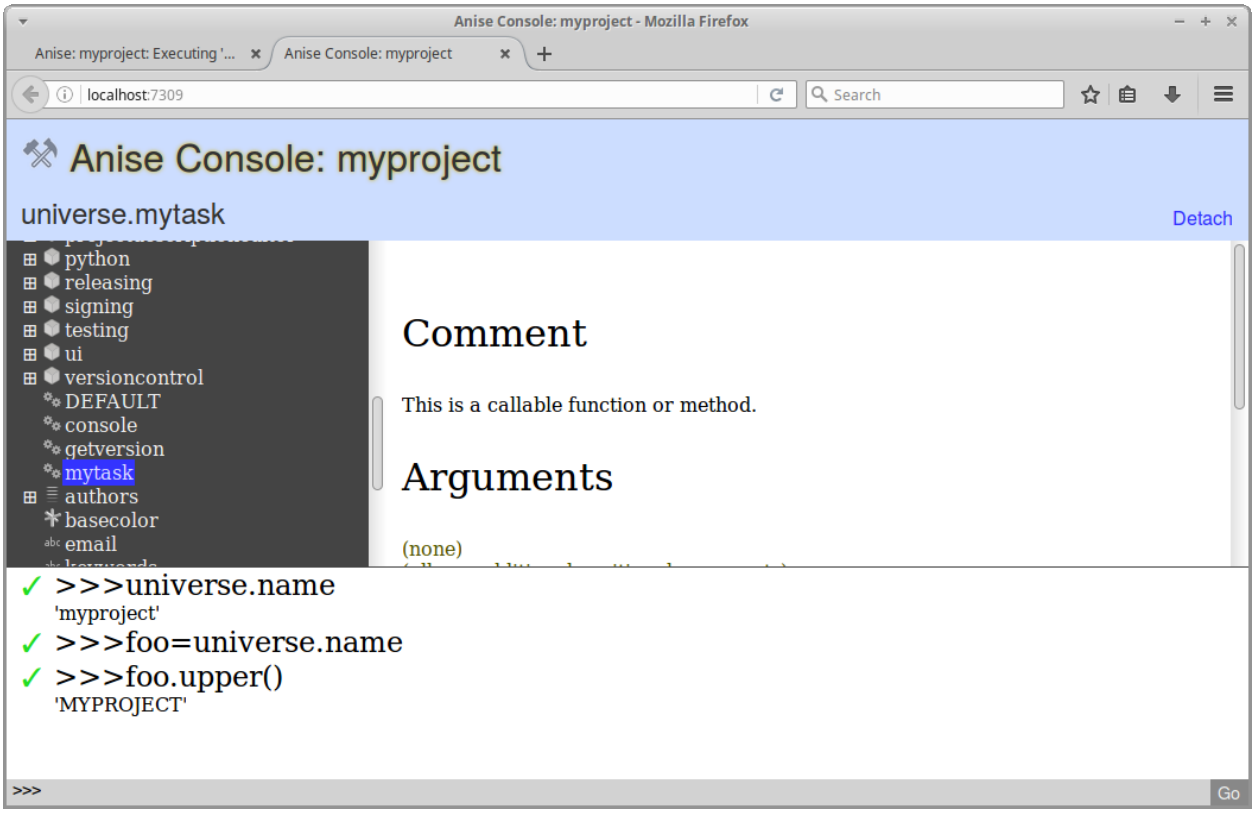
This tool runs either in web mode or in terminal mode. The web mode is much more comfortable and is the default. There is nothing special to know for the user and no special requirement. It just needs to start a web browser. If this is undesired, please add :samp:` –no-gui` to your **anise** command. The same functionality is available in terminal mode as well, but the usability of the web mode is much higher.

9.1 Anise console

The Anise Console is a tool for inspecting the universe object at runtime and to run various configuration tasks. More than that, it is a swiss army knife for issue diagnostics and feature development. The console is available as the `console` task from your task chooser, but is also started automatically when you create a fresh *project description file*.

9.1.1 Window parts

The console is made up of those parts:



In the top part, there is a header that shows some general infos and actions.

A tree on the left side shows the runtime content of *The universe object*.

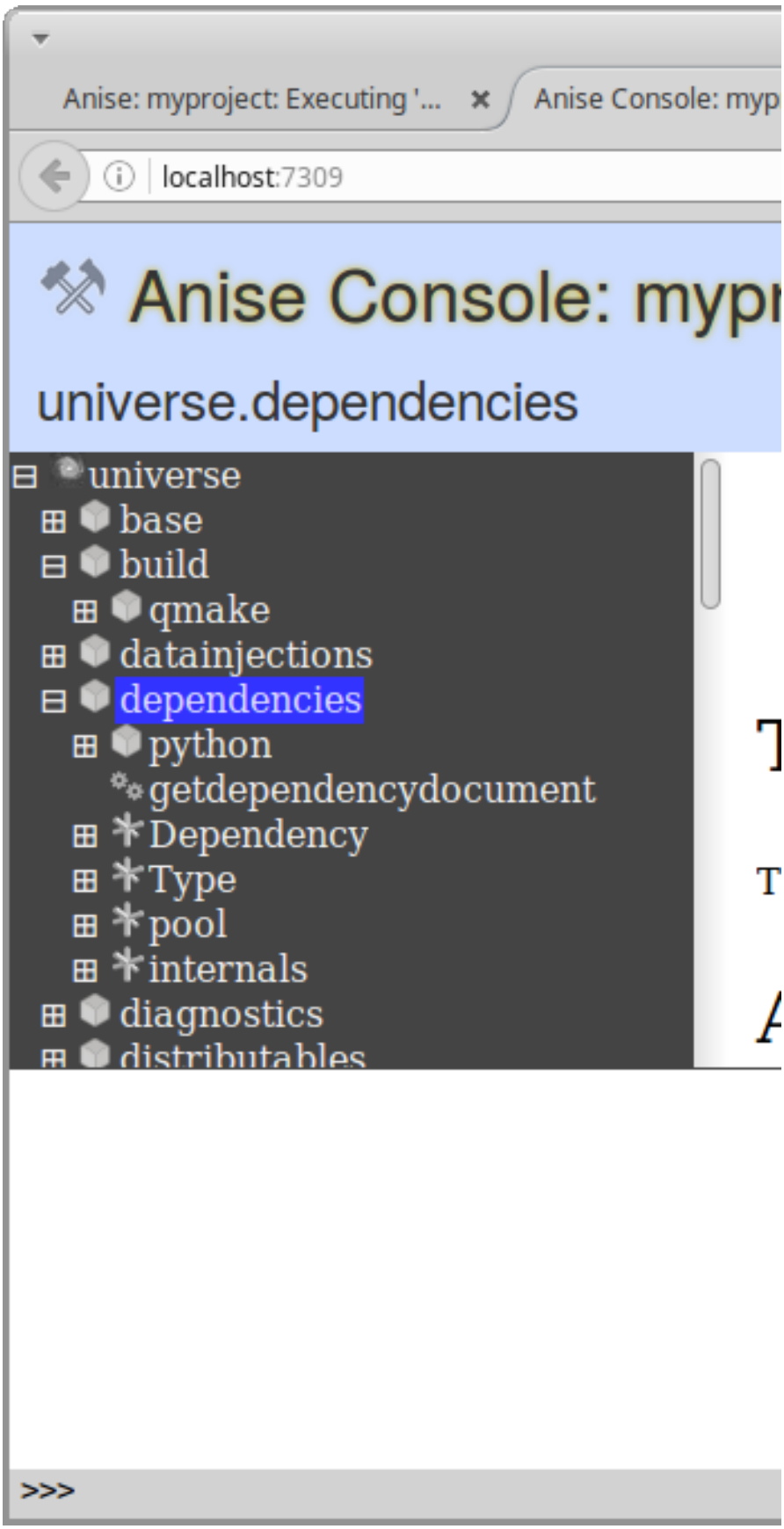
The info panel on the right side of the tree shows detail information about the node that is selected in the tree. It lists (amongst other things) actions that are available to execute directly in the console. Those actions contain graphical configuration guides as well as ways to interact with object in the console.

The terminal in the bottom displays processing output and allows to directly execute Python expressions.

Be aware that all object modifications you make in the terminal are volatile. They are able to modify the current state of *The universe object*, but they do not change the *project description file*. When you start Anise again, the old values will be back.

9.1.2 Configuration guides

As noted before, many Feature nodes provide configuration guides. Find those nodes by the 'package' symbol and select them in order to find those guides.


Anise: myproject: Executing '...
Anise Console: myp
localhost:7309
Anise Console: mypr
universe.dependencies
universe
base
build
qmake
datainjections
dependencies
python
getdependencydocument
Dependency
Type
pool
internals
diagnostics
distributables
>>>

Warning: The configuration of your project description and the runtime content of your universe object are two different things! At startup, the latter is created from the former one, but afterwards they are decoupled. Whenever you make any configuration changes in the console, you have to start a fresh Anise session in order to see the changes applied to your universe object.

CHAPTER

TEN

APPENDIX: COMMAND LINE

The Anise command-line tool understands this syntax:

```
anise [taskname] [options]*
```

If a taskname is given, Anise will execute this task. Otherwise it will execute the `DEFAULT` task, which typically is a task chooser.

Options can be some of the following:

- **`--projectfile [file]`** : Use this project file (instead of the typical location)
- **`--loadfeaturesfrom [path1:...:pathn]`** : Sets the paths for loading modules (separated by ":"; builtin features by "builtin")
- **`--debug`** : Enables debug log output
- **`--keeptemp`** : Do not remove the temporary stuff (for post-execution diagnostic)
- **`--no-autoopen`** : Never automatically open files after execution
- **`--no-gui`** : Do not try to use graphic mode for user interaction
- **`--createfresh`** : Create a fresh project description file
- **`--s:key=value`** : Sets a value to the universe object before execution; also works with `i`, `b` and `f` at the beginning for integers, bools and floats instead of strings
- **`--help`** : Shows this help text

The easiest way to begin working with an unknown Anise project is to open the graphical task chooser by just calling **`anise`**.

Note: Users of the Anise package for Windows need to open the 'Anise Shell'. Calling **`anise`** the above way is possible from within this environment only.

CHAPTER

ELEVEN

APPENDIX: RECIPES, HOW TO

11.1 ... use the included high-level features

The included high-level features of Anise are all visible in the *Console*. The easiest way is to check the interesting feature nodes there and use configuration guides provided there. Those often give only access to basic functionality and sometimes no guides are implemented at all. In such cases, use the documentation of the subnodes in order to understand how to use the feature.

11.2 ... implement custom functionality

There is a two different ways to add custom functionality:

11.2.1 The quick-and-dirty way

Use the 'Manage Tasks' action in the root node of the console in order to add an own task. You will then find this task in the task chooser and also execute it directly with **`anise mytask`**.

More flexibility brings the 'Manage Hook Scripts' action, which allows you to add handlers for *hooks* and so can add a much broader range of custom functionality. For more infos about a particular hook, find its node in the console and read what the info panel provides. Also read *... use hooks*. A hook script is executed in a very similar way as custom features (i.e. the clean way), so the following recipes typically can be used this way as well.

Both approaches will insert the customization code into your project description file.

11.2.2 The clean way

Implement your custom functionality as an Anise feature and add it to your Anise installation. The entire custom automation code of your project then has the same technical form as the included features. They all use the same infrastructure and are loaded and executed in the same way.

For adding custom logic this way, create a new file for your new feature and name it like `myfeature.py`. In order to actually add it to the Anise loading sequence, place this file into `$HOME/.anise/features` (for Windows users, this is `%USERPROFILE%.anisefeatures`). Placing it deeper into subdirectories give composed feature names, as explained *above*.

This file will be loaded by the Anise engine in a very similar way as Python would load plain modules. However, the feature loader makes the feature available as a member in *The universe object* (and also some bookkeeping stuff). The feature code typically defines some functions and classes for different reasons (see other recipes). It does not directly execute code, but may mark some of the defined objects as hook handlers, so adding all kinds of custom functionality to the Anise execution for all projects (since it is not bound to one project description file anymore).

For executing some code in the initialization of the universe object, the feature could like like this:

```
from anise.framework import engine
from anise.framework import features

@features.hookhandler(engine.HOOK_AFTER_PREPARATION)
def somefoolishinitialization():
    print("Hello World.")
```

This uses the *anise.framework.engine.HOOK_AFTER_PREPARATION* hook, which is triggered in the initialization of the universe object. Details about the existing *hooks* can be found in the *Console*. Find the hook nodes in the universe object tree and read what the info panel provides about them. Also read *... use hooks*.

11.3 ... implement tasks

A task is a Python function that implements a particular automation routine. Whenever the Anise engine executes a task, it gets the implementation (the Python function) by taking a particular member of *The universe object* and calls it.

A very easy way to add a new task (with some restrictions) is described in *... implement custom functionality*. The following shows how to add a task to the universe object in a feature implementation:

```
from anise.framework import engine
from anise.framework import features
from anise.framework.projects import universe

def somecrazyfunction():
    print("Hello World.")

@features.hookhandler(engine.HOOK_AFTER_PREPARATION)
def somefoolishinitialization():
    # of course we can also write arbitrarily complex code here,
    # which dynamically creates tasks depending on some conditions.
    universe.addtask(somecrazyfunction)
```

This feature allows you to directly execute `mytask` (e.g. from the task chooser or from command line) on all your Anise projects.

11.4 ... define and access variables

The *values* defined in the *project description file* become members of *The universe object* at runtime, as well as the defined *lets* indirectly assign further members.

At runtime, they can be read and modified in the very same way as you would do with any other Python object member. The *anise.framework.projects.universe* object is the current universe object. You will see this used in virtually every part of automation code, since it is 'the' global data and control hub for the entire automation logic.

```
from anise.framework.projects import universe

def sometask():
    universe.name2 = universe.name.upper()
    print("Your project name: " + universe.name2)
```

Last but not least, you can also set (primitive) values on the *command line*.

11.5 ... output or log messages

The Anise logging foundation can (and should) be used for all kinds of output. Read this example and *anise.utils.logging* for more details:

```
from anise import utils

...
def foo():
    ...
    utils.logging.loginfo("Hello World.")
    utils.logging.loginfo("Hello World 2.", printcaller=False)
```

Note: User dialogues, which *are explained later*, force the user attention, while logging is the right way for monologue-type communication.

11.6 ... load external features

For accessing stuff from an external feature, in most situations, it is enough to directly access the corresponding parts of the universe object, like this:

```
from anise.framework.projects import universe

...
def foo():
    ...
    universe.someotherfeature.someobject.foo = 42
    universe.someotherfeature.somemethod()
```

From time to time it is required to explicitely load an external feature. Early initialization phases are an example for such situations, since the universe object might be not yet populated.

You can load a feature directly by means of the Anise infrastructure and access objects in a direct way. The *anise.framework.features.loadfeature()* function loads another feature by name and returns an *anise.framework.features.Feature*:

```
from anise.framework import features
foo = features.loadfeature("foo").featuremodule
x = foo.some_object_in_foo
```

11.7 ... use hooks

The *hook system* is the Anise event mechanism. Customization of Anise behavior virtually always has to do with hooks in some way. An existing functionality (either something from the Anise engine, from included features or from custom features) can define and export a hook in order to provide an interaction point where new customization can plug in. It triggers the hook in some situations (i.e. it calls all registered handlers), so the external parts can act and influence the process, behavior or result of something.

In *former recipes* you already have seen how to use a hook by registering a handler to it. Before you can do so, you must find out which hook you want to use (e.g. from the documentation of the feature or part you want to customize

or by finding it in the console). You also have to find out the calling convention: Some hooks assume that classes are registered as handlers, while others work on function. For both of them, the argument signature must be known as well (for classes this is the constructor's signature). Find out those details in the documentation, so you are able to write a handler. For some hooks, the *Manage Hook Scripts* *console* action is able to create a valid skeleton.

The *anise.framework.features.hookhandler()* function decorator is used for registering a hook handler in a feature:

```
from anise.framework import engine
from anise.framework import features
from anise.framework.projects import universe
from anise import utils

# yet another hook with trivial argument signature
@features.hookhandler(ui.HOOK_UIMODE_SELECTED)
def foo():
    utils.logging.loginfo('UI mode: ' + str(universe.ui.mode))
```

11.8 ... use hooks more dynamically

In some situations it is required to add hook handlers in a more dynamic way than in *... use hooks*, i.e. depending on some conditions.

The *anise.framework.projects.Universe.addhook()* function is used by program logic in order to add a hook handler:

```
from anise.framework.projects import universe

def myhandler():
    ...

def foo():
    ...
    if some_condition_fulfilled:
        universe.addhook(HOOK_SOMEHOOK, myhandler)
    ...
```

11.9 ... control the order of hook handlers

If the order of hook handlers matter, the `requires`, `provides` and `prepares` arguments of the handler insertions can be used for controlling them (see *anise.framework.features.hookhandler*). All those arguments expect lists of strings, which are used for describing dependencies between handlers. The engine will automatically compute a valid order on top of this.

Note: The `provides` value specifies the 'own symbols', i.e. which condition it participates to fulfill. Even if you leave it empty, it always contains the feature name and the full name of the handler function or class. This makes it easier to refer to particular elements from outside even if it does not explicitly provide any symbols.

11.10 ... define and trigger hooks

On the other side, some module can define a *hook* and trigger it in some situations in order to give external parties a way to customize the general behavior. The *anise.framework.projects.Universe.gethooks()* function is used by program logic in order to *provide* resp. *trigger* a hook. It returns all registered handlers for execution. Definition and execution is shown in the following example:

```
from anise.framework import engine
from anise.framework import features
from anise.framework.projects import universe
from anise import utils

HOOK_MYEVENT = features.Hook()

def mytask():
    result = 0
    # note how flexible you are with using the handlers
    for handler in universe.gethooks(HOOK_MYEVENT):
        result += handler()
    utils.logging.loginfo("Result: " + str(result))

@features.hookhandler(engine.HOOK_AFTER_PREPARATION)
def foo():
    universe.addtask(mytask)
```

The handlers are typically added from external features (so we need to use ... *load external features*) during initialization, like this:

```
from anise.framework import engine
from anise.framework import features

# we assume the feature from above available as 'foo'
foo = features.loadfeature("foo").featuremodule

@features.hookhandler(foo.HOOK_MYEVENT)
def foo():
    return 13.37
```

Note: *anise.framework.features.Hook* has some optional parameters, which can be used for providing additional information about your hook. None of them are required for using it, but they may help configuration guides or other auxiliary components for getting a better idea about what your hook does.

11.11 ... implement custom initialization steps

Use the *anise.framework.engine.HOOK_AFTER_PREPARATION* hook and ... *use hooks*:

```
from anise.framework.projects import universe
from anise.framework import engine
from anise.framework import features
from anise import utils

@features.hookhandler(engine.HOOK_AFTER_PREPARATION)
```

(continues on next page)

(continued from previous page)

```
def somefoolishinitialization():
    universe.somevalue = 123
    utils.logging.loginfo("Hello World.")
```

11.12 ... implement configuration classes and functions

The configuration of complex aspects in the *project description file* is often realized with custom classes carrying the specified configuration. Those can be constructed directly with nested `val` nodes in the project description file. This encapsulates complexity and allows for easier specifications in the project description file.

```
...
class Foo:
    def __init__(self, x, y):
        ...
```

can be used this way in the project description file (either as root value, as nester value or as a parameter somewhere):

```
<val t="myfeature.Foo">
    <val k="x" v="foobar"/>
    <val k="y" v="baz"/>
</val>
```

Configuration helper functions are small program routines. They are another way to provide a convenient configuration interface to some functionality for usage from within the *project description file*. They are used by means of an *The let node* in order to allow complex configuration specifications. It has somewhat similar use cases as the former point. However, since the former is a class and the latter is a function, there are all kinds of technical details that make either the one or the other match better to a particular situation.

```
...
def foo(x, y):
    ...
```

can be used this way in the project description file:

```
<let k="myfeature.foo">
    <val k="x" v="foobar"/>
    <val k="y" v="baz"/>
</let>
```

11.13 ... interact with the user

There are some methods that implement typical user interaction patterns. At runtime, they are available in *The universe object* as `ui.userfeedback`. Read the documentation of *anise.features.ui.Internals.UserFeedback* for all available methods and the following example:

```
from anise.framework.projects import universe

...
def foo():
    r = universe.ui.userfeedback.message("How are you?",
                                          btns=["Great.", "Not so good."])
```

(continues on next page)

(continued from previous page)

```
    if r == 0:
        universe.ui.userfeedback.messageok("I'm glad to hear that!")
    elif r == 1:
        universe.ui.userfeedback.messageok("That's bad news :-(")
```

11.14 ... read and modify the project description file

Some functionality aspects - typically configuration guides - need to read and write the *project description file* in a program routine. There is a programming interface available for those operations in *anise.framework.projects.IntermediateStructure*. The following example shows some usual operations:

```
from anise.framework.projects import universe
from anise.framework import projects

...
def foo():
    ais = projects.IntermediateStructure.fromfile(universe.specialpaths.projectfile)
    newletentry = projects.IntermediateStructure.Let("some.functionname")
    ais.addentry(newletentry)
    newvalentry = projects.IntermediateStructure.Value("bool", v=True)
    ais.addentry(newvalentry)
    newvalentry2 = projects.IntermediateStructure.Value("some.ClassName",
        foo=projects.IntermediateStructure.Value("objectref", oref="some.other.object
↪"),
        bar=projects.IntermediateStructure.Value.fromstring("Horray.")
    )
    ais.addentry(newvalentry2)
    ais.removeentry(newvalentry)
    newvalentry2.removearg("bar")
    ais.save()
```

11.15 ... implement configuration guides

The guided editors typically come somehow together with the corresponding features (although in Anise itself this relationship is realized differently). They include some graphical assistants for doing typical configuration handling.

A configuration assistant implements the `anise.features.projectdescriptioneditor.ProjectDescriptionEditorAssistant` interface and is registered as *hook* for `anise.features.projectdescriptioneditor.HOOK_GET_FEATURE_ACTIONS`.

They provide a filtered overview to the *The project description file* and can offer some custom actions in the overview. Those custom actions use the *user interaction methods* for communication with the user. They also need to *work with the project description*.

```
from anise.framework.projects import universe
from anise.framework import features

projectdescriptioneditor = features.loadfeature("projectdescriptioneditor").
↪featuremodule

@features.hookhandler(projectdescriptioneditor.HOOK_GET_FEATURE_ACTIONS)
```

(continues on next page)

(continued from previous page)

```
class MyAssistant(projectdescriptioneditor.ProjectDescriptionEditorAssistant):

    def __init__(self):
        super().__init__("Manage miracle", "Specify which miracles shall occur in␣
↪your project.",
                         "mymiraclefeature")

    def getcustomactions(self, ais):
        return [projectdescriptioneditor.CustomAction("Add new miracle",
                                                      self._add_miracle)]

    def _add_miracle(self, ais):
        name = universe.ui.userfeedback.input("Please enter a name for the miracle.",
                                              "Peace in the world")
        ...

    def getentries(self, ais):
        r = []
        for node in ais.entries():
            if node.islet and node.key == "mymiraclefeature.pool.add":
                r.append(e)
        return r
```

Additionally, you can register custom actions for particular kinds of nodes (instead of custom actions globally available in the overview). Those custom actions are easily accessible from the overviews and are also available in the *Raw Mode Editor*. Implement them this way:

```
from anise.framework.projects import universe
from anise.framework import features

projectdescriptioneditor = features.loadfeature("projectdescriptioneditor").
↪featuremodule

@features.hookhandler(projectdescriptioneditor.HOOK_GET_PROJECTDESCRIPTIONNODE_
↪CUSTOMACTION_IMPLEMENTATIONS)
class MyNodeCustomAction(projectdescriptioneditor.CustomAction):

    def __init__(self):
        super().__init__("Change miracle kind")

    def visible(self, ais, node):
        return node.islet and node.key == "mymiraclefeature.pool.add"

    def execute(self, ais, node):
        command = universe.ui.userfeedback.choice(...)
        ...
```

11.16 ... implement configuration guides (more flexible)

More flexible (but with more own responsibilities) than *the former recipe* is this interface:

```
from anise.framework.projects import universe
from anise.framework import features
from anise import utils

diagnostics = features.loadfeature("diagnostics").featuremodule
projectdescriptioneditor = features.loadfeature("projectdescriptioneditor").
↪featuremodule

@features.hookhandler(projectdescriptioneditor.HOOK_GET_FEATURE_ACTIONS)
class MyAction(diagnostics.FeatureAction):

    def __init__(self):
        super().__init__("Do Something", "Does something somehow.", forfeature=
↪"myfeature")

    def execute(self):
        utils.logging.loginfo("Hello World.")
```

11.17 ... hide some internal stuff of my feature in the console

The console shows the complete tree of the universe object at runtime, which can decrease the visibility of the relevant objects and can weaken the clearness by means of lots of internal bookkeeping structures. In order to soften this effect, you can define all internal pieces of your feature in the 'Internals' subclass, like this:

```
...

def something_important():
    ...
    Internals.something_for_internal_usage(...)
    ...

class Internals:

    def something_for_internal_usage(...):
        ...
```

11.18 ... provide some documentation for my custom stuff in the console

Use the Python docstring technique for providing documentation about functions and classes. You can use Doxygen notation in order to document function arguments.

```
def something_important(x):
    """
    This function does important things.
    :param x: Some important parameter.
    """
    ...
```

11.19 ... create execution scopes for reporting progress details

A program routine can often be seen as an assembly (nested or sequential) of substeps. Technically those are often encapsulated in functions, but from a high-level perspective, the degree of encapsulation can also be broader or finer.

Your program code can specify those blocks as execution scopes. The execution environment will show those blocks to the user together with progress information.

Specify execution scopes this way:

```
from anise.framework.projects import universe

def foo():
    ...
    with universe.executionscope.newsubscope("Doing something hilarious"):
        ...
    ...
```

11.20 ... deal with errors

The *anise.framework.exceptions* module contains special exception types for usage within Anise and the automation projects. It is a good idea for various reasons to consider using them in own `raise` statements. Sometimes it could also be required to use some of them in `except` clauses in order to react on a specific error situation.

11.21 ... work with file structures

Instances of *anise.framework.files.Filestructure* are typical for exchanging files or complete file structures, including subdirectories and many files. Many included features use them as arguments or return them. There are also some subclasses that are useful in some cases, like *anise.framework.files.TempDir*.

You can create such objects this way (alternatively use one of the subclasses):

```
from anise.framework import files

def foo():
    ...
    x = files.Filestructure("/path/to/something/in/filesystem")
    ...
```

And use them by calling some of its methods, like this:

```
def foo(x):  # takes a files.Filestructure
    ...
    # this copies the file structure to a destination.
    x.dl("/path/to/some/destination/in/filesystem")
    ...
```

11.22 ... use dynamic expressions in a project description file

In a *project description file*, a very generic tool are *objectref values*. They are defined by an arbitrary Python expression that becomes evaluated in initialization when the project description file is read.

```
<?xml version="1.0" ?>
<aniseprojectdesc>
    ...
    <val k="sometask" oref="some.feature.somemethod"/>
    ...
    <val k="name" oref="'MyPrOjEcTnAmE'.lower()"/>
    ...
</aniseprojectdesc>
```

11.23 ... embed Anise in other tools

Since Anise has a powerful *command-line interface*, it is very easy to call Anise as external process from your tool.

The Anise engine can also be imported in other Python applications and can be triggered in-process. For details about the latter option, please read the source code of the *anise.framework.engine* module.

CHAPTER

TWELVE

APPENDIX: SHALLOT PLUGIN

There is an Anise plugin for the Shallot file manager available under `_meta/shallot_plugin/`. It embeds some Anise functionality (like executing the tasks) directly into the file manager.

Find information about the Shallot project on the same website as Anise. The documentation of it explains how to install this plugin.

CHAPTER

THIRTEEN

APPENDIX: INSTALLATION

Install Anise via the installation package for your environment, if a suitable one exists for download. This also takes care of installing dependencies and doing preparation (unless mentioned otherwise in the installation procedure). After the installation, you can skip the rest of this section.

13.1 Source code archive

Use the source code archive as fallback. Extract it to a location that is convenient to you (Windows users need an external archive program; for example the great '7-Zip' tool). Also take a look at the *Dependencies* for external stuff you need to install as well.

It is highly recommended to also establish a command line link or alias for `anise/anise.py` so you just have to type **`anise`** (**`ln -s ...anise/anise.py /usr/local/bin/anise`** on Unix or any other operating system specific way). This is according to what the installation packages do and required for executing the exact same commands as used in this manual (otherwise you must substitute the full name for the short command names in this manual).

CHAPTER

FOURTEEN

API REFERENCE

14.1 anise package

14.1.1 Subpackages

anise.features package

Subpackages

anise.features.build_ package

Submodules

anise.features.build_.autotools module

anise.features.build_.cmake module

anise.features.build_.make module

anise.features.build_.qmake module

Module contents

Inner module for *anise.features.build*.

anise.features.dependencies_ package

Submodules

anise.features.dependencies_.python module

Module contents

Inner module for *anise.features.dependencies*.

anise.features.diagnostics_ package

Module contents

Inner module for *anise.features.diagnostics*.

anise.features.docrender_ package

Module contents

Inner module for *anise.features.docrender*.

anise.features.packages_ package

Submodules

anise.features.packages_.android module

Feature for creation of Android distribution packages.

class anise.features.packages_.android.**Internals**
Bases: object

static buildandroidpackage_with_ant(*, *projectroot*, *name*, *version*, *targetname*, *androidpath*)
Builds an Android package with Ant.

Parameters

- **projectroot** (*str*) – The path to the Android package.
- **name** (*str*) – The project name.
- **version** (*str*) – The project version.
- **targetname** (*str*) – The Ant target name that shall be built.
- **androidpath** (*str*) – Path to the Android sdk's android tool.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.android.**inject_projectdata_to_androidmanifest**(*outfile*)
Updates some project data (like version number) in the AndroidManifest.xml.

Parameters **outfile** (*str*) – The directory that holds the AndroidManifest.xml. Can be absolute or relative to the Anise project directory.

Return type None

anise.features.packages_.android.**package_with_ant**(*androidpath*, *source*, *relpath=''*, *targetname='debug'*)
Builds an Android package with Ant.

Parameters

- **androidpath** (*str*) – Full path to the android tool (something like .../android-sdk/tools/android).
- **source** (anise.framework.files.Filestructure) – The source file structure.

- **relpath** (*str*) – The root path of the android project, relative to `source`s root.
- **targetname** (*str*) – Name of the Ant target to build.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.appc module

Feature for creation of appc container images.

class anise.features.packages_.appc.**EnvironmentVariableDescription**(*name*, *value*)

Bases: object

Description for Appc environment variables to be added to the container image.

Parameters

- **name** (*str*) – The variable name.
- **value** (*str*) – The value.

class anise.features.packages_.appc.**Internals**

Bases: object

static _initproject()

static call(*cmd*, *raise_on_errors='unable to run acbuild'*)

Parameters

- **cmd** (*List[str]*) –
- **raise_on_errors** (*Optional[str]*) –

Return type None

class anise.features.packages_.appc.**PortDescription**(*name*, *protocol*, *port*)

Bases: object

Description for Appc network ports to be added to the container image.

Parameters

- **name** (*str*) – The port name.
- **protocol** (*str*) – The protocol (typically "tcp" or "udp").
- **port** (*int*) – The port number as available inside the container.

anise.features.packages_.appc.**appcdebianpackage**(*debianpackage*, *fullname=None*, *ports=None*, *environment_variables=None*, *exec-command=None*, *run=None*, *run_late=None*)

Builds an appc container image based on Debian.

You need an existing Debian package source.

Parameters

- **debianpackage** (anise.framework.files.Filestructure) – The source file structure containing the debian package.
- **fullname** (*Optional[str]*) – The full container name.

- **ports** (*Optional[List[anise.features.packages_.appc.PortDescription]]*) – A list of port descriptions to add to the container image.
- **environment_variables** (*Optional[List[anise.features.packages_.appc.EnvironmentVariableDescription]]*) – A list of environment variable descriptions to add to the container image.
- **execcommand** (*Optional[List[str]]*) – A list of strings to specify the command to execute inside the container after boot.
- **run** (*Optional[List[List[str]]]*) – A list of command-line lists to execute for preparation.
- **run_late** (*Optional[List[List[str]]]*) – A list of command-line lists to execute for preparation after installing the package.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.debian module

Feature for creation of Debian distribution packages.

class anise.features.packages_.debian.**Category**

Bases: object

Enumeration for Debian's categories for software packages.

ApplicationsAccessibility = ('Applications/Accessibility', 'System')

ApplicationsAmateurradio = ('Applications/Amateur Radio', 'Utility')

ApplicationsDatamanagement = ('Applications/Data Management', 'System')

ApplicationsEditors = ('Applications/Editors', 'Utility')

ApplicationsEducation = ('Applications/Education', 'Education')

ApplicationsEmulators = ('Applications/Emulators', 'System')

ApplicationsFilemanagement = ('Applications/File Management', 'System')

ApplicationsGraphics = ('Applications/Graphics', 'Graphics')

ApplicationsMobiledevices = ('Applications/Mobile Devices', 'Utility')

ApplicationsNetwork = ('Applications/Network', 'Network')

ApplicationsNetworkCommunication = ('Applications/Network/Communication', 'Network')

ApplicationsNetworkFiletransfer = ('Applications/Network/File Transfer', 'Network')

ApplicationsNetworkMonitoring = ('Applications/Network/Monitoring', 'Network')

ApplicationsNetworkWebbrowsing = ('Applications/Network/Web Browsing', 'Network')

ApplicationsNetworkWebnews = ('Applications/Network/Web News', 'Network')

ApplicationsOffice = ('Applications/Office', 'Office')

ApplicationsProgramming = ('Applications/Programming', 'Development')

ApplicationsProjectmanagement = ('Applications/Project Management', 'Development')

ApplicationsScience = ('Applications/Science', 'Education')

ApplicationsScienceAstronomy = ('Applications/Science/Astronomy', 'Education')

```
ApplicationsScienceBiology = ('Applications/Science/Biology', 'Education')
ApplicationsScienceChemistry = ('Applications/Science/Chemistry', 'Education')
ApplicationsScienceDataanalysis = ('Applications/Science/Data Analysis', 'Education')
ApplicationsScienceElectronics = ('Applications/Science/Electronics', 'Education')
ApplicationsScienceEngineering = ('Applications/Science/Engineering', 'Education')
ApplicationsScienceGeoscience = ('Applications/Science/Geoscience', 'Education')
ApplicationsScienceMathematics = ('Applications/Science/Mathematics', 'Education')
ApplicationsScienceMedicine = ('Applications/Science/Medicine', 'Education')
ApplicationsSciencePhysics = ('Applications/Science/Physics', 'Education')
ApplicationsScienceSocial = ('Applications/Science/Social', 'Education')
ApplicationsShells = ('Applications/Shells', 'System')
ApplicationsSounds = ('Applications/Sound', 'AudioVideo')
ApplicationsSystem = ('Applications/System', 'System')
ApplicationsSystemAdministration = ('Applications/System/Administration', 'System')
ApplicationsSystemHardware = ('Applications/System/Hardware', 'System')
ApplicationsSystemLanguageenvironment = ('Applications/System/Language Environment', '
ApplicationsSystemMonitoring = ('Applications/System/Monitoring', 'System')
ApplicationsSystemPackagemanagement = ('Applications/System/Package Management', 'Syst
ApplicationsSystemSecurity = ('Applications/System/Security', 'System')
ApplicationsTerminalemulators = ('Applications/Terminal Emulators', 'System')
ApplicationsText = ('Applications/Text', 'Utility')
ApplicationsTvandradio = ('Applications/TV and Radio', 'AudioVideo')
ApplicationsVideo = ('Applications/Video', 'AudioVideo')
ApplicationsViewers = ('Applications/Viewers', 'Utility')
ApplicationsWebdevelopment = ('Applications/Web Development', 'Development')
GamesAction = ('Games/Action', 'Game')
GamesAdventure = ('Games/Adventure', 'Game')
GamesBlocks = ('Games/Blocks', 'Game')
GamesBoard = ('Games/Board', 'Game')
GamesCard = ('Games/Card', 'Game')
GamesPuzzles = ('Games/Puzzles', 'Game')
GamesSimulation = ('Games/Simulation', 'Game')
GamesStrategy = ('Games/Strategy', 'Game')
GamesTools = ('Games/Tools', 'Game')
GamesToys = ('Games/Toys', 'Game')
Help = ('Help', 'System')
```

ScreenLocking = ('Screen/Locking', 'System')

ScreenSaving = ('Screen/Saving', 'System')

class anise.features.packages_.debian.**Internals**
Bases: object

static builddebpackage(*, *name*, *version*, *licensename*, *authors*, *email*, *summary*, *description*, *section*, *architecture*, *websiteurl*, *menuentries*, *executablelinks*, *requireddependencies*, *stronglyrecommendeddependencies*, *suggesteddependencies*, *services*, *rawpkgpath*, *projectfsroot*, *prerm=''*, *postinst=''*)
Builds a Debian package (.deb).

Parameters

- **name** (*str*) – The package name.
- **version** (*str*) – The package version string.
- **licensename** (*str*) – The name of the project license.
- **authors** (*List[str]*) – Project authors.
- **email** (*str*) – Project email address.
- **summary** (*str*) – Short project description.
- **description** (*str*) – Longer project description.
- **section** (*str*) – The Debian section name.
- **architecture** (*str*) – The Debian architecture name.
- **websiteurl** (*str*) – The project homepage url.
- **menuentries** (*List[MenuEntry]*) – Menu entries to register.
- **executablelinks** (*Dict[str, str]*) – Symlinks to executables to be created.
- **requireddependencies** (*List[str]*) – List of required dependencies.
- **stronglyrecommendeddependencies** (*List[str]*) – List of dependencies that are not strongly required but very typical.
- **suggesteddependencies** (*List[str]*) – List of optional dependencies.
- **services** (*List[ServiceDescription]*) – List of services to be registered.
- **rawpkgpath** (*str*) – The raw source directory; can be damaged.
- **projectfsroot** (*str*) – The project source directory path.
- **prerm** (*str*) – Bash code snippet executed before the package is removed on a target machine.
- **postinst** (*str*) – Bash code snippted executed after the package is installed on a target machine.

Return type *anise.framework.files.Filestructure*

class anise.features.packages_.debian.**MenuEntry**(*, *name*, *title*, *category*, *command*, *gui*, *icon*)
Bases: object

Specification for one menu entry added by a Debian installation package.

Parameters

- **name** (*str*) – Internal menu entry name.

- **title** (*str*) – Menu entry Title.
- **category** (*Tuple*) – Menu entry category. One of *Category*.
- **command** (*str*) – Menu entry command.
- **gui** (*bool*) – If the command opens a gui (instead of a terminal application).
- **icon** (*str*) – The path to a menu entry icon.

class anise.features.packages_.debian.**ServiceDescription**(*name*, *command*)

Bases: object

Description for Debian services to be included in a package.

Parameters

- **name** (*str*) – The display name.
- **command** (*str*) – The command to be executed.

anise.features.packages_.debian.**debpackage**(*source*, *dependencies=None*, *executablelinks=None*, *menuentries=None*, *services=None*, *prerm=''*, *postinst=''*, *architecture='all'*)

Builds a Debian package.

Parameters

- **source** (anise.framework.files.Filestructure) – A source file structure.
- **dependencies** (*Optional[List[*dependencies.Dependency*]]*) – A list of Debian dependencies for installation on target machines.
- **executablelinks** (*Optional[Dict[str, str]]*) – A dict for links to executables for deployment on target machines.
- **menuentries** (*Optional[List[*anise.features.packages_.debian.MenuEntry*]]*) – A list of menu entries for creation on target machines.
- **services** (*Optional[List[*anise.features.packages_.debian.ServiceDescription*]]*) – A list of services for deployment on target machines.
- **prerm** (*str*) – Piece of bash script executed before removal of the package on target machines.
- **postinst** (*str*) – Piece of bash script executed after installation of the package on the target machines.
- **architecture** (*str*) – The target architecture name.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.debian.**only_programfiles**(*source*)

Returns a filtered set of files for a Debian program directory (no documentation, license, tests, . . .).

Parameters **source** – An anise.framework.files.Filestructure.

Returns A filtered anise.framework.files.Filestructure.

anise.features.packages_.flatpak module

anise.features.packages_.oci module

Feature for creation of oci container images.

class anise.features.packages_.oci.**EnvironmentVariableDescription**(*name*, *value*)

Bases: object

Description for Oci environment variables to be added to the container image.

Parameters

- **name** (*str*) – The variable name.
- **value** (*str*) – The value.

class anise.features.packages_.oci.**Internals**

Bases: object

static _initproject()

static call(*cmd*, *raise_on_errors='unable to run buildah'*)

Parameters

- **cmd** (*List[str]*) –
- **raise_on_errors** (*Optional[str]*) –

Return type str

class anise.features.packages_.oci.**PortDescription**(*port*)

Bases: object

Description for Oci network ports to be added to the container image.

Parameters **port** (*int*) – The port number as available inside the container.

anise.features.packages_.oci.**ocidebianpackage**(*debianpackage*, *fullname=None*, *ports=None*, *environment_variables=None*, *execcommand=None*, *run=None*, *run_late=None*)

Builds an oci container image based on Debian.

You need an existing Debian package source.

Parameters

- **debianpackage** (anise.framework.files.Filestructure) – A file structure containing the debian package.
- **fullname** (*Optional[str]*) – The full container name.
- **ports** (*Optional[List[anise.features.packages_.oci.PortDescription]]*) – A list of PortDescription port descriptions to add to the container image.
- **environment_variables** (*Optional[List[anise.features.packages_.oci.EnvironmentVariableDescription]]*) – A list of EnvironmentVariableDescription descriptions to add to the container image.
- **execcommand** (*Optional[List[str]]*) – A list of strings to specify the command to execute inside the container after boot.

- **run** (*Optional[List[List[str]]]*) – A list of command-line lists to execute for preparation.
- **run_late** (*Optional[List[List[str]]]*) – A list of command-line lists to execute for preparation after installing the package.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.python module

anise.features.packages_.win32 module

Feature for creation of Win32 distribution packages.

class anise.features.packages_.win32.**Internals**

Bases: object

static buildwindowsinstallerpackage(*, *name*, *version*, *licensefile*, *authors*, *email*, *summary*, *architecture*, *websiteurl*, *menuentries*, *rawpkgpath*, *projectfsroot*, *projecticon=None*, *globalscript=''*, *oninitscript=''*, *oninstallscript=''*, *usemodernui2=False*)

Builds a Windows installer executable (.exe).

Parameters

- **name** (*str*) – The package name.
- **version** (*str*) – The package version.
- **licensefile** (*str*) – The path to the full text of the project's license.
- **authors** (*List[str]*) – Project authors.
- **email** (*str*) – Project email address.
- **summary** (*str*) – Short project description.
- **architecture** (*str*) – The Debian architecture name.
- **websiteurl** (*str*) – The project homepage url.
- **menuentries** (*List[MenuEntry]*) – Menu entries to register.
- **rawpkgpath** (*str*) – The raw source directory; can be damaged.
- **projectfsroot** (*str*) – The project source directory path.
- **projecticon** (*Optional[str]*) – Path to the project icon.
- **globalscript** (*str*) – NSIS script global part.
- **oninitscript** (*str*) – NSIS script that runs at initialization of the installer.
- **oninstallscript** (*str*) – NSIS script that runs after file copying of the installer.
- **usemodernui2** (*bool*) – If Modern UI 2 shall be used (brings a more fresh look).

Return type *anise.framework.files.Filestructure*

class anise.features.packages_.win32.**MenuEntry**(*, *title*, *command*, *icon=None*, *params=''*, *workdir='$INSTDIR'*)

Bases: object

Specification for one start menu entry added by a Windows package installer.

Parameters

- **title** (*str*) – Menu entry title.
- **command** (*str*) – Menu entry command.
- **icon** (*Optional[str]*) – The path to a menu entry icon.
- **params** (*str*) – Additional command parameters.
- **workdir** (*str*) – The working directory for this menu entry.

anise.features.packages_.win32.**deployqt5dlls**(*milieu*, *binsource=None*, *pluginssource=None*)

Returns the DLLs needed for Qt deployment on Windows.

Parameters

- **milieu** (milieus.Milieu) – The milieu to use. See milieus.Milieu for details.
- **binsource** (*Optional[anise.framework.files.Filestructure]*) – The directory where the Qt library binaries are stored.
- **pluginssource** (*Optional[anise.framework.files.Filestructure]*) – The directory where the Qt plugins are stored.

Return type *anise.framework.files.Filestructure*

anise.features.packages_.win32.**win32exepackage**(*source*, *architecture='all'*, *menuentries=None*, *projecticon=None*, *oninitscript=''*, *oninstallscript=''*, *usemodernui2=True*)

Builds a Win32 installation executable (as .exe file).

Parameters

- **source** (anise.framework.files.Filestructure) – An anise.framework.files.Filestructure.
- **architecture** (*str*) – The target architecture name.
- **menuentries** (*Optional[List[anise.features.packages_.win32.MenuEntry]]*) – A list of menu entries for creation on target machines.
- **projecticon** (*Optional[str]*) – Path to a project icon. Relative to the project root directory.
- **oninitscript** (*str*) – Additional NSIS on-init script.
- **oninstallscript** (*str*) – Additional NSIS on-install script.
- **usemodernui2** (*bool*) – If Modern UI 2 shall be used (brings a more fresh look).

Return type *anise.framework.files.Filestructure*

Module contents

Inner module for *anise.features.packages*.

anise.features.projectdescriptioneditor_ package

Submodules

anise.features.projectdescriptioneditor_.base module

anise.features.projectdescriptioneditor_.datainjections module

anise.features.projectdescriptioneditor_.dependencies module

anise.features.projectdescriptioneditor_.distributables module

anise.features.projectdescriptioneditor_.documentation module

anise.features.projectdescriptioneditor_.filetransfer module

class anise.features.projectdescriptioneditor_.filetransfer.**LocalFiletransfer**
Bases: object

ask()

class anise.features.projectdescriptioneditor_.filetransfer.**SshFiletransfer**
Bases: object

ask()

anise.features.projectdescriptioneditor_.filetransfer.**ask_new_filetransfer**(*msg*)

anise.features.projectdescriptioneditor_.homepage module

anise.features.projectdescriptioneditor_.imagegalleries module

anise.features.projectdescriptioneditor_.licensing module

anise.features.projectdescriptioneditor_.packages module

anise.features.projectdescriptioneditor_.testing module

anise.features.projectdescriptioneditor_.versioncontrol module

Module contents

Inner module for anise.features.projectdescriptioneditor.

anise.features.testing_ package

Submodules

anise.features.testing_.generic module

Feature for generic unit testing.

anise.features.testing_.generic.**genericTest**(*testrun*, *command*)
Runs an external command as test and interpret it as successful, if the exit code is `0`.

Parameters

- **testrun** (*testing.Internals.TestRun*) – A test run.
- **command** (*List[str]*) – The external command (as list of arguments).

Return type None

anise.features.testing_.pytest module

Feature for unit testing with pyunit.

anise.features.testing_.pytest.**pytestTests**(*testrun*, *testdir=None*)
Runs all pytest tests in the project. The discovery is done automatically.

Parameters

- **testrun** (*testing.Internals.TestRun*) – A test run.
- **testdir** (*Optional[str]*) – The root directory where to look for tests.

Return type None

anise.features.testing_.pyunit module

Feature for unit testing with pyunit.

anise.features.testing_.pyunit.**pyunitTest**(*testrun*, *testclass*, *testdir=None*)
Runs an pyunit test.

Parameters

- **testrun** (*testing.Internals.TestRun*) – A test run.
- **testdir** (*Optional[str]*) – The root directory where to look for tests.
- **testclass** (*str*) – The name of the test class.

Return type None

anise.features.testing_.pyunit.**pyunitTests**(*testrun*, *testdir=None*, *filepattern='*.py'*)
Runs all pyunit tests in the project (or all that fulfill specified patterns). The discovery is done automatically.

Parameters

- **testrun** (*testing.Internals.TestRun*) – A test run.
- **testdir** (*Optional[str]*) – The root directory where to look for tests.
- **filepattern** (*str*) – A file pattern describing which files to inspect.

Return type None

Module contents

Inner module for `anise.features.testing`.

anise.features.ui_ package

Subpackages

anise.features.ui_.terminal package

Submodules

anise.features.ui_.terminal.helpers module

Feature for helpers related to terminal-based user interfaces.

`anise.features.ui_.terminal.helpers.`**`prompt`**(*head*, *msg*)

Parameters

- **head** (*str*) –
- **msg** (*str*) –

Return type str

`anise.features.ui_.terminal.helpers.`**`showhead`**(*head*, *msg*, *withterm=True*)

Parameters

- **head** (*str*) –
- **msg** (*str*) –
- **withterm** (*bool*) –

Return type None

Module contents

anise.features.ui_.web package

Submodules

anise.features.ui_.web.helpers module

Feature for helpers related to web-based user interfaces.

class `anise.features.ui_.web.helpers.`**`AniseWebApplication`**(***b*)

Bases: `anise.data.Yaji.yaji.app.Application`

Parameters

- **parentid** – Optional id of a parent Application, which e.g. can be redirected back to after exit.
- **returntoparent** – If to return back to the *parentid* application after exit in the user's browser.
- **show_browser_closed_notification** – If to show a useful notification (including a way to close the application) after the browser was closed by the user and restarted.
- **stop_implicitly_when_browser_closed** – If to consider the application as intendedly stopped when the user has closed the browser instead of opening a new one.
- **skip_shutdown_dialog** – If not to show a ui-blocking 'application stopped' dialog on ui shutdown.
- **browser_hook_heartbeat_threshold** – The time window within the backend excepts a heartbeat from the browser side before it tries to open a new browser window.
- **head1** – The 1st level header text.
- **head2** – The 2nd level header text.
- **icon** – The window icon.
- **mainview_icon** – The icon of the mainview header.

_do_answeruserfeedback(*index*, *answer*)

Parameters **index** (*int*) –

_do_getprogresses(*since*)

Parameters **since** (*int*) –

_execution_scopepart_added(_)

_get_userfeedback_web_answers(*requestindex*)

_send_userfeedback_web_request(*request*)

_shutdown()

onbrowserreopened()
Reacts on the fact that the browser was opened again. *Override this method in custom subclasses or leave the default implementation.*

oninitialize()
Initializes the application. *Override this method in custom subclasses or leave the default implementation.*

This is called during application startup, so later than __init__ (or never if the app does not start).

Note: You should just override __init__ instead if possible!

---

onopenbrowsererror()
Reacts on errors when opening the browser. *Override this method in custom subclasses or leave the default implementation.*

onopenbrowserinformationoutput(*kind*, *message*)
Reacts on information output (e.g. by printing it to stdout) while opening the browser. *Override this method in custom subclasses or leave the default implementation.*

Parameters

- **kind** – The internal code of the message.

- **message** – The message text.

onprocessrequesterror(*request*, *error*)
Reacts on process request errors, e.g. by logging. *Override this method in custom subclasses or leave the default implementation.*

Parameters

- **request** – The path part of the request url.
- **error** – The Exception.

start()
Starts the application.

Note: If your instance was already started and stopped, you have to create a new instance for another run.

---

stop()
Stops the application.

- It requests the parent application's returntakeover procedure in some situations (see later),
- then invokes stopping the user interface (triggers *_yj_stopui*),
  - which on browser side leads to 'uiShutdown' and back redirection to parent application in some situations
- then shuts down the http server.
  - so the _yj_stopui event might never actually arrive, depending on timings (not an actual problem).

A uiShutdown disables the user interface by triggering the *OnUiShutdown* event (but only one time; further calls are ignored). Application code can register custom handlers to this event, but it does at least this:

- visually disables the user interface and shows a 'shut down' text if *skip_shutdown_dialog==False* (the default)
- if it happened due to the user tried to close a PyHtmlView and *show_browser_closed_notification==True* (not the default): makes a '_yj_application_tryclosebrowser' request, so PyHtmlView support can close the window or parent applications can take control back

The browser side also runs *uiShutdown* e.g. when

- *_yj_pullevent* requests fail (assumption: the application code side has stopped), or
- after a *yaji.stop()* was called (see later).

A uiShutdown does not stop stuff like event polling, and will leave *yaji.isrunning==True*! The browser side *yaji.stop()* does the following:

- requests '_yj_application_stop', which calls *Application.stop()* on application code side
  - when backend answered: running uiShutdown
  - with disabled user interface during the request if
    - *skip_shutdown_dialog==False* (the default) or
    - it was explicitly called this way

Browser side calls *yaji.stop()* e.g. in those situations:

- custom application code on browser side triggered it for application exit

- a '_yj_unhandled_client_error' occurred and the backend decided for shutdown (which is not the default)
- by PyHtmlView support when the user tries to close the window

There is a flag *yaji.isrunning* that is *True* if the backend application is not known to be stopped so far, but might even be *True* after a uiShutdown. It will become *False* when

- *_yj_pullevent* requests fail (assumption: the application code side has stopped), or
- the *_yj_stopui* (see above) event was received.

anise.features.ui_.web.webexec module

Module contents

Submodules

anise.features.ui_.terminal module

anise.features.ui_.web module

Module contents

Inner module for *anise.features.ui*.

anise.features.versioncontrol_ package

Submodules

anise.features.versioncontrol_.git module

anise.features.versioncontrol_.piget module

anise.features.versioncontrol_.svn module

Module contents

Inner module for *anise.features.versioncontrol*.

Submodules

anise.features.base module

Feature that contains basic versatile tools. They are not associated to any particular kind of functionality.

class anise.features.base.**Internals**
Bases: object

static _beautify_longdescription()

Return type None

static _initproject()

Return type None

static extractfilesfromretval(*r*, *autoopen*)
Can be used after executing a task in the anise engine for making a result file available in the target directory. Before you do so, all files created as result are in a temporary area, which will be flushed when the process terminates.

Parameters

- **r** (*Any*) – The raw result (arbitrary object, list, whatever).
- **autoopen** (*bool*) – If anise shall automatically open a file in the result if there is a hint set this way.

Return type Any

static getbackgroundimage()
Determines the location of the background image.

Returns File path relative to the project root directory.

Return type Optional[str]

static getlogoimage()
Determines the location of the background image.

Returns File path relative to the project root directory.

Return type Optional[str]

static getprojectstatusdescription()
Returns a short plaintext describing the specified project maturity (`universe.base.maturity`).

Return type str

static getregisteredtasks()
Determines the list of all available tasks.

Return type *anise.framework.files.TextFileByContent*

class anise.features.base.**Maturity**
Bases: object

Enumeration describing possible project states.

Alpha = 'Development Status :: 3 - Alpha'

Beta = 'Development Status :: 4 - Beta'

Inactive = 'Development Status :: 7 - Inactive'

Mature = 'Development Status :: 6 - Mature'

Planning = 'Development Status :: 1 - Planning'

PreAlpha = 'Development Status :: 2 - Pre-Alpha'

ProductionStable = 'Development Status :: 5 - Production/Stable'

anise.features.base.**pyscript**(*code*, *hook*, *provides=None*, *requires=None*, *prepares=None*)
Adds a Python function given as source code text to a hook (see Anise documentation for details about hooks).

Note: This call only has an effect, if this hook will be triggered afterwards. So, usage in a project description file has no effect for hooks that are triggered in very early stages!

Parameters

- **code** (*str*) – The Python code block to register as a hook handler.
- **hook** (*str*) – The hook name (i.e. the event). This is typically a variable that begins with `HOOK_`.
- **provides** (*Optional[List[str]]*) – Used for dependency ordering of hooks. A list of string symbols that this method provides.
- **requires** (*Optional[List[str]]*) – Used for dependency ordering of hooks. A list of string symbols that this method requires. Each method that provides one of those symbols is guaranteed to be executed before.
- **prepares** (*Optional[List[str]]*) – Used for dependency ordering of hooks. A list of string symbols that this method prepares. Each method that provides one of those symbols is guaranteed to be executed afterwards.

Return type None

anise.features.base.**pytask**(*code*, *taskname*, *decorators=None*, *label=None*)

Adds a task implementation to the universe object, so it is available for execution, e.g. directly from command line or from the graphical task chooser.

Parameters

- **code** (*str*) – The Python code block to register as task implementation.
- **taskname** (*str*) – The name for this new task.
- **decorators** (*Optional[List[str]]*) – Additional Python function decorators to apply.
- **label** (*Optional[str]*) – An optional label text.

Return type None

anise.features.base.**set_basecolor**(*r*, *g*, *b*)

Sets the project base color. This color is used for all kinds of styling.

Parameters

- **r** (*float*) – Red value (`0.0` to `1.0`).
- **g** (*float*) – Green value (`0.0` to `1.0`).
- **b** (*float*) – Blue value (`0.0` to `1.0`).

Return type None

anise.features.build module

Features for building binaries from source files.

anise.features.datainjections module

Feature for injecting some kinds of data at anise runtime into some file structures, in order to make it available to other build processes (Makefiles, ...) or to the project's runtime.

class anise.features.datainjections.**Internals**
Bases: object

class Pool
Bases: object

Storage of all data injections.

add(*method*, *outfile*, *propnames=None*, *do_in_source=False*, ***params*)
Add a data injection.

Parameters
- **method** (*Callable*) – The implementation for the injection. This is the name of a function with the parameters outfile and data, writing the data (a dict with key/value pairs) to a given file in some format (as absolute path or relative to the project root directory).
- **outfile** (*str*) – Path to the target file for the injections (relative to anise project root path).
- **propnames** (*Optional[List[str]]*) – List of entry names you want to inject (beyond the few default ones).
- **do_in_source** (*bool*) – If the source itself should be updated as well (instead of only the on-the-fly copies for packages).
- **params** – Additional parameters for calling the injection implementation.

Return type None

static _initproject()

Return type None

static _sync()
Sync if not marked as omitsyncvcs.

Return type bool

anise.features.datainjections.**getessentials**(*propnames=None*)
Returns essential project data for injecting it into some file.

Parameters **propnames** (*Optional[List[str]]*) – Additional properties to inject on top of the very fundamental ones.

Returns a list of name/value tuples containing the data for injections.

Return type List[Tuple[str, Any]]

anise.features.datainjections.**inject_c**(*outfile=None*, *data=None*)
Injects project data into a C++ header file, so it is available at program's runtime.

Parameters
- **outfile** (*Optional[str]*) – The target file for injection.
- **data** (*Optional[List[Tuple[str, Any]]]*) – The data to inject.

Return type *anise.framework.files.Filestructure*

anise.features.datainjections.**inject_python**(*outfile=None*, *data=None*, *asclass=None*)
Injects project data into a Python file, so it is available at program's runtime.

Parameters

- **outfile** (*Optional[str]*) – The target file for injection.
- **data** (*Optional[List[Tuple[str, Any]]]*) – The data to inject.
- **asclass** (*Optional[str]*) – Optional class name. If set, the data is written as members into a class with this name.

Return type *anise.framework.files.Filestructure*

anise.features.dependencies module

Features for specifying and processing third-party packages that are required by the project in some way.

class anise.features.dependencies.**Dependency**(*type*, *objectname=None*, *comment=''*, *icon=None*, *visible=True*, *displayname=None*, ***kwargs*)
Bases: object

A dependency.

Parameters

- **type** (anise.features.dependencies.Type) – The dependency type.
- **objectname** (*Optional[str]*) – A descriptive name for this dependency (e.g. a library name).
- **comment** (*str*) – A comment.
- **icon** (*Optional[str]*) – The name of a dependency icon.
- **visible** (*bool*) – If it is visible in the dependency list.
- **displayname** (*Optional[str]*) – A display variant of the object name.
- **kwargs** – Additional arguments.

class anise.features.dependencies.**Internals**
Bases: object

class Pool
Bases: object

Storage of all dependencies.

add(*dependency*)
Adds a dependency.
Parameters **dependency** (anise.features.dependencies.Dependency) – The dependency to add.
Return type None

static _add_homepage_section()

Return type None

static _initproject()

Return type None

static getdependencydocument()
Creates a documentation source that describes the project dependencies in a human-readable form.

It is mostly used for including it into Doxygen documentation source.

Returns The documentation source.

Return type str

class anise.features.dependencies.**Type**
Bases: object

Enumeration of different types of dependencies. It describes how strongly required a dependency is.

Included = 'included'

Optional = 'optional'

Recommended_HasAlternatives = 'recommended'

Required = 'required'

Required_HasAlternatives = 'required (has alternatives)'

anise.features.diagnostics module

Feature for universe object exploration and problem analysis.

class anise.features.diagnostics.**FeatureAction**(*label*, *description*, *forfeature*)
Bases: object

Abstract base class for an action that can be triggered from the feature overview in the console.

Parameters

- **label** (*str*) – The label text.
- **description** (*str*) – The description text (may be a bit longer than label).
- **forfeature** (*str*) – Full name of the feature to associate this action with (like 'foo.bar').

execute()
Executes this feature action. *Override this method in custom subclasses.*

Return type None

visible()
Checks if this feature action shall be visible. *Override this method in custom subclasses or leave the default implementation.*

Return type bool

anise.features.diagnostics.**HOOK_GET_FEATURE_ACTIONS = <anise.framework.features.Hook object**
Hook triggered for collecting all implementations for feature actions, as offered in the console.

class anise.features.diagnostics.**Internals**
Bases: object

class ChildrenInformation(*p*)
Bases: object

class Arguments(*args*, *varargs*, *varkw*, *doc*)
Bases: object

class ConsoleWebModule

Bases: *anise.features.ui_.web.helpers.AniseWebApplication*

Parameters

- **parentid** – Optional id of a parent Application, which e.g. can be redirected back to after exit.
- **returntoparent** – If to return back to the *parentid* application after exit in the user's browser.
- **show_browser_closed_notification** – If to show a useful notification (including a way to close the application) after the browser was closed by the user and restarted.
- **stop_implicitly_when_browser_closed** – If to consider the application as intendedly stopped when the user has closed the browser instead of opening a new one.
- **skip_shutdown_dialog** – If not to show a ui-blocking 'application stopped' dialog on ui shutdown.
- **browser_hook_heartbeat_threshold** – The time window within the backend excepts a heartbeat from the browser side before it tries to open a new browser window.
- **head1** – The 1st level header text.
- **head2** – The 2nd level header text.
- **icon** – The window icon.
- **mainview_icon** – The icon of the mainview header.

_do_exec(*p*)

_do_execaction(*p*, *idx*)

Parameters **idx** (*int*) –

_do_info(*p*)

_do_listfeatures(*p*)

_do_listkeys(*p*)

_do_listmembers(*p*)

_send_userfeedback_web_request(*request*)

oninitialize()

Initializes the application. *Override this method in custom subclasses or leave the default implementation.*

This is called during application startup, so later than __init__ (or never if the app does not start).

Note: You should just override __init__ instead if possible!

class ReadManualApplication

Bases: *anise.features.diagnostics.FeatureAction*

Parameters

- **label** – The label text.
- **description** – The description text (may be a bit longer than label).
- **forfeature** – Full name of the feature to associate this action with (like 'foo.bar').

_manualpath()
Return type Optional[str]

execute()
Executes this feature action. *Override this method in custom subclasses.*

_featureactions = None

static _initproject()

static _initproject2()

static action(*i*)

Parameters **i** (*int*) –

Return type None

static checkmembervisible(*parent*, *membername*, *hidemodules=True*)

Parameters

- **parent** (*Any*) –
- **membername** (*str*) –
- **hidemodules** (*bool*) –

Return type bool

static correctname(*s*)

Parameters **s** (*str*) –

Return type str

static detach()

Return type None

static featureactions()

Return type List[*anise.features.diagnostics.FeatureAction*]

static featureactionsforfeature(*s*)

Parameters **s** (*str*) –

Return type [<class 'anise.features.diagnostics.FeatureAction'>]

static getdocumentation(*obj*)

static getfunctionparamspec(*func*)

Parameters **func** (*Callable*) –

static geticon(*parent*, *membername*)

Parameters

- **parent** (*Any*) –
- **membername** (*str*) –

Return type str

static getobjects(*obj*)

static getpriority(*parent*, *membername*)

Parameters

- **parent** (*Any*) –
- **membername** (*str*) –

Return type int

static getuniverseattr(*membername*)

Parameters **membername** (*str*) –

Return type Optional[Any]

static getuniversedocumentation()

Return type str

static info(*v='universe'*)

Parameters **v** (*str*) –

Return type None

infostring = "\n Information about '{valname}'\n{desc}\n"

static issimplevalue(*v*)

Parameters **v** (*type*) –

Return type bool

static isunexpandable(*obj*)

Parameters **obj** (*Any*) –

Return type bool

listfeaturesstring = "\n Information about 'universe'\n\nLoaded features:\n{features}

anise.features.diagnostics.**console**()

Executes the diagnostics console. Either call this task directly from command line or from within another task implementation in order to inspect the items of the universe object.

Return type None

anise.features.distributables module

Feature for defining and processing distributables (installation packages for target systems, . . .).

class anise.features.distributables.**FileGroup**(*sources*, *name*, *description=None*, *linkto=None*)

Bases: object

A file group.

Parameters

- **sources** (*Union[anise.framework.files.Filestructure, List[anise.framework.files.Filestructure]]*) – The file structures that return the distributables. For a single entry, you may also pass it directly without a list around it.
- **name** (*str*) – The file group name.
- **description** (*Optional[str]*) – The file group description text.
- **linkto** (*Optional[str]*) – An optional link path that is always kept up to date (pointing to the latest version). This is only applied to the first file in the group.

class anise.features.distributables.**Internals**
Bases: object

class Pool
Bases: object

Storage of all dependencies.

addgroup(*sources*, ***kwargs*)
Adds a group of distributables.

Each group is stored as *FileGroup* instance.
Parameters
- **sources** (*Union[anise.framework.files.Filestructure, List[anise.framework.files.Filestructure]]*) – The file structures that return the distributables. For a single entry, you may also pass it directly without a list around it.
- **kwargs** – Additional parameters for the generator tasks.

Return type None

find_by_name(*name*)
Searches a file group by name and returns it.
Parameters **name** (*str*) – The name of the group to find.
Return type *anise.features.distributables.FileGroup*

get()
Prepares the pool for further processing (i.e. firing HOOK_PREPARE_DISTRIBUTABLES_POOL) and returns the pool.
Return type List[*anise.features.distributables.FileGroup*]

static _add_homepage_section()

static _initproject()

static _initproject2()

anise.features.docrender module

Feature for generating all kinds of documentation like output.

class anise.features.docrender.**AbstractApiReferenceLanguage**
Bases: object

Base class for a programming language in api references. See *ApiReferencePiece*.

sphinx_config(*rootpath*, *outpath*)
Returns additional Sphinx conf.py configuration.

Parameters
- **rootpath** (*str*) – Path to the original source directory.
- **outpath** (*str*) – Path to the root directory of the Sphinx input.

Return type str

sphinx_transfer(*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory outpath.

Parameters
- **srcpath** (*str*) – The root path of the sources directory.

- **outpath** (*str*) – The root path of the intermediate directory given to Sphinx.
- **heading** (*str*) – The api reference heading.

Return type None

class anise.features.docrender.**AbstractDoxygenSupportedApiReferenceLanguage**(*, *_doxygenopts=None*, *extract_all=True*, *extract_private=True*, *extract_package=True*, *extract_static=True*, *extract_local_classes=T* *extract_local_methods=* *extract_anon_nspaces=* *extract_priv_virtual=Tr* *file_patterns='*'*, *inline_inherited_membe* *inherit_docs=True*, *hide_undoc_members* *hide_undoc_classes=* *exclude_patterns=''*, *predefined=None*)

Bases: *anise.features.docrender.AbstractApiReferenceLanguage*

Language support for api references powered by Doxygen.

Parameters

- **_doxygenopts** (*Optional[Dict[str, str]]*) –
- **extract_all** (*bool*) –
- **extract_private** (*bool*) –
- **extract_package** (*bool*) –
- **extract_static** (*bool*) –
- **extract_local_classes** (*bool*) –
- **extract_local_methods** (*bool*) –
- **extract_anon_nspaces** (*bool*) –
- **extract_priv_virtual** (*bool*) –
- **file_patterns** (*str*) –

- **inline_inherited_memb** (*bool*) –
- **inherit_docs** (*bool*) –
- **hide_undoc_members** (*bool*) –
- **hide_undoc_classes** (*bool*) –
- **exclude_patterns** (*str*) –
- **predefined** (*Optional[List[str]]*) –

__info (*outpath*)

sphinx_config (*rootpath*, *outpath*)
Returns additional Sphinx `conf.py` configuration.

Parameters

- **rootpath** – Path to the original source directory.
- **outpath** – Path to the root directory of the Sphinx input.

sphinx_transfer (*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory `outpath`.

Parameters

- **srcpath** – The root path of the sources directory.
- **outpath** – The root path of the intermediate directory given to Sphinx.
- **heading** – The api reference heading.

class anise.features.docrender.**AbstractOutputSpec** (*path*)
Bases: `object`

Base class for documentation output specifications. See *render()*.

Parameters **path** (*str*) – The output path.

resultpath (*projectname*, *buildpath*)
Returns the path to the actual result file/directory, relative to `buildpath`.

Parameters

- **projectname** (*str*) – The project name (also document header).
- **buildpath** (*str*) – The Sphinx build output path.

Return type str

sphinx_config (*path*)
Returns additional Sphinx `conf.py` configuration.

Parameters **path** (*str*) – Path to the root directory of the Sphinx input.

Return type str

sphinx_formatname ()
Returns the Sphinx format name.

Return type str

class anise.features.docrender.**AbstractPiece** (*name*, *rootpath*, *, *sortidx*, *source=None*)
Bases: `object`

A subdocument. See *render()*.

Parameters

- **name** (*str*) – The piece name.
- **rootpath** (*str*) – The root directory or file that contains the piece content.
- **sortidx** (*int*) – Sort order index.
- **source** (*Optional[anise.framework.files.Filestructure]*) –

copyto(*path*)
Copies required data files for this piece to a location.

Parameters **path** (*str*) – The target location (usually inside the intermediate structure given to Sphinx).

Return type None

property name
The piece name.

Return type str

property rootpath
The root directory or file that contains the piece content.

Return type str

property sortidx
The sort order index.

Return type int

property source
The source file structure.

Return type *anise.framework.files.Filestructure*

sphinx_config(*path*)
Returns additional Sphinx `conf.py` configuration.

Parameters **path** (*str*) – Path to the root directory of the Sphinx input.

Return type str

class anise.features.docrender.**ApiReferencePiece**(*name*, *rootpath*, *language*, *, *heading=None*, *sortidx=100000*, *source=None*)
Bases: *anise.features.docrender.AbstractPiece*

An api reference.

Parameters

- **name** (*str*) – The piece name.
- **rootpath** (*str*) – The root directory or file that contains the piece content.
- **language** (anise.features.docrender.AbstractApiReferenceLanguage) – The programming language related to this api reference.
- **heading** (*Optional[str]*) – The api reference heading.
- **sortidx** (*int*) – Sort order index.

copyto(*path*)
Copies required data files for this piece to a location.

Parameters **path** – The target location (usually inside the intermediate structure given to Sphinx).

sphinx_config(*path*)
Returns additional Sphinx `conf.py` configuration.

Parameters **path** – Path to the root directory of the Sphinx input.

class anise.features.docrender.**CppApiReferenceLanguage**(***kwargs*)
Bases: *anise.features.docrender.AbstractDoxygenSupportedApiReferenceLanguage*

C++ language support for api references.

class anise.features.docrender.**HtmlOutputSpec**(*path*, ***, *theme=None*, *homepagemode=False*, *title=None*, *short_title=None*, *short_desc=None*, *masterlink=None*)
Bases: *anise.features.docrender.AbstractOutputSpec*

HTML documentation output.

Parameters

- **path** (*str*) – The output path.
- **theme** (*Optional[str]*) – The sphinx html theme name.
- **homepagemode** (*bool*) – If to render output for a homepage with slight different stylings and behaviors.
- **title** (*Optional[str]*) – The html title.
- **short_title** (*Optional[str]*) – The short html title.
- **short_desc** (*Optional[str]*) – The short description. Ignored by most themes.
- **masterlink** (*Optional[str]*) – Url that overrides the target of the main heading (which is also a link).

resultpath(*projectname*, *buildpath*)
Returns the path to the actual result file/directory, relative to `buildpath`.

Parameters

- **projectname** – The project name (also document header).
- **buildpath** – The Sphinx build output path.

sphinx_config(*path*)
Returns additional Sphinx `conf.py` configuration.

Parameters **path** – Path to the root directory of the Sphinx input.

sphinx_formatname()
Returns the Sphinx format name.

class anise.features.docrender.**Internals**
Bases: `object`

class MediaDirs
Bases: `object`

Storage of media directories that are used by documentation sources.

add(*path*)
Adds a media directory.

Parameters **path** (*str*) – Path to the media directory, absolute or relative to the project directory.
Return type None

get_mediadirs()
Returns a list of all media directories.
Return type List[str]

class ShortSnippets
Bases: object

Storage of text snippets for use in the documentation texts like |a_foo|.

add(*name*, *text*)
Adds a text snippet to the storage, so it can be used.
Parameters
- **name** (*str*) – The snippet alias name. This is the foo in |a_foo|.
- **text** (*str*) – The snippet substitution text.

Return type None

getall()
Returns all aliases and their snippets.
Return type Dict[str, str]

static _initproject()

static _initproject2()

static scalecolor(*c*, *brightness*, *saturation=None*)
Internally used for scaling a color tuple to a new one with the same hue but a given brightness.

Parameters
- **c** (*Tuple[float, float, float]*) –
- **brightness** (*float*) –
- **saturation** (*Optional[float]*) –

Return type Tuple[float, float, float]

class anise.features.docrender.**JavaScriptApiReferenceLanguage**
Bases: *anise.features.docrender.AbstractApiReferenceLanguage*

JavaScript language support for api references.

__jsfiles(*df*)

Parameters **df** (*str*) –

Return type List[str]

__scanjsfile(*f*)

Parameters **f** (*str*) –

Return type str

sphinx_config(*rootpath*, *outpath*)
Returns additional Sphinx conf.py configuration.

Parameters
- **rootpath** – Path to the original source directory.
- **outpath** – Path to the root directory of the Sphinx input.

sphinx_transfer(*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory `outpath`.

Parameters

- **srcpath** – The root path of the sources directory.
- **outpath** – The root path of the intermediate directory given to Sphinx.
- **heading** – The api reference heading.

class anise.features.docrender.**PdfOutputSpec**(*path*)
Bases: *anise.features.docrender.AbstractOutputSpec*

PDF documentation output.

Parameters **path** (*str*) – The output path.

resultpath(*projectname*, *buildpath*)
Returns the path to the actual result file/directory, relative to `buildpath`.

Parameters

- **projectname** – The project name (also document header).
- **buildpath** – The Sphinx build output path.

sphinx_formatname()
Returns the Sphinx format name.

class anise.features.docrender.**PlaintextOutputSpec**(*path*)
Bases: *anise.features.docrender.AbstractOutputSpec*

Plaintext documentation output.

Parameters **path** (*str*) – The output path.

resultpath(*projectname*, *buildpath*)
Returns the path to the actual result file/directory, relative to `buildpath`.

Parameters

- **projectname** – The project name (also document header).
- **buildpath** – The Sphinx build output path.

sphinx_formatname()
Returns the Sphinx format name.

class anise.features.docrender.**Python3ApiReferenceLanguage**(*show_undoc_members=True*, *show_protected_members=True*, *exclude=None*, *preparecode=''*)
Bases: *anise.features.docrender.AbstractApiReferenceLanguage*

Python 3 language support for api references.

Parameters

- **show_undoc_members** (*bool*) –
- **show_protected_members** (*bool*) –
- **exclude** (*Optional[List[str]]*) –
- **preparecode** (*str*) –

patch_property_types_in_docstrings(*pd*)
Deprecated since version all.

Parameters **pd** (*str*) –

Return type None

sphinx_config(*rootpath*, *outpath*)
Returns additional Sphinx `conf.py` configuration.

Parameters

- **rootpath** – Path to the original source directory.
- **outpath** – Path to the root directory of the Sphinx input.

sphinx_transfer(*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory `outpath`.

Parameters

- **srcpath** – The root path of the sources directory.
- **outpath** – The root path of the intermediate directory given to Sphinx.
- **heading** – The api reference heading.

class anise.features.docrender.**RstGenerator**
Bases: `object`

Generator for some documentation source parts.

static heading(*s*, *, *variant='='*, *sub=False*, *anchor=None*)
Generates documentation source for a section heading.

Parameters

- **s** (*str*) – The heading text.
- **variant** (*str*) – What heading variant to use. See rst syntax for details.
- **sub** (*bool*) – If to use the sub-variant of `variant` headings, i.e. with just the symbol line *below* the heading text. See rst syntax for details.
- **anchor** (*Optional[str]*) – Anchor name for usage by rst :samp:`ref`s.

Return type str

class anise.features.docrender.**RstTextsPiece**(*name*, *rootpath*, *, *sortidx=0*, *source=None*)
Bases: *anise.features.docrender.AbstractPiece*

A reStructureText formatted file or a directory of such files.

Parameters

- **name** (*str*) – The piece name.
- **rootpath** (*str*) – The root directory or file that contains the piece content.
- **sortidx** (*int*) – Sort order index.

copyto(*path*)
Copies required data files for this piece to a location.

Parameters **path** – The target location (usually inside the intermediate structure given to Sphinx).

anise.features.docrender.**docgen**
alias of *anise.features.docrender.RstGenerator*

anise.features.docrender.**render**(*pieces*, *outputspec*, *projectname*, *, *author*, *heading=None*, *mainbody=None*, *basecolor=None*, *mediadirs=None*, *exclude_in_toc=None*, *include_in_toc=None*, *shortsnippets=None*, *backgroundimage=None*, *logoimage=None*)
Generate documentation output for some documentation pieces and an output specification.

Parameters

- **pieces** (*List[anise.features.docrender.AbstractPiece]*) – Documentation pieces to include.
- **outputspec** (anise.features.docrender.AbstractOutputSpec) – The output specification.
- **projectname** (*str*) – The project name, used as (part of) top-level title.
- **author** (*str*) – The project author(s).
- **heading** (*Optional[str]*) – The main heading text.
- **mainbody** (*Optional[str]*) – reStructuredText source for the main body.
- **basecolor** (*Optional[Tuple[float, float, float]]*) – The base color (used for styling by some kinds of outputs).
- **mediadirs** (*Optional[List[str]]*) – Additional media source paths.
- **exclude_in_toc** (*Optional[List[str]]*) – List of piece names to exclude for the table of contents. Default: Exclude none.
- **include_in_toc** (*Optional[List[str]]*) – List of piece names to include for the table of contents. Default: Include all.
- **shortsnippets** (*Optional[Dict[str, str]]*) – Substitution snippets, used by `|a_foo|` inside document sources.
- **backgroundimage** (*Optional[str]*) – Absolute path to the background image (will be copied).
- **logoimage** (*Optional[str]*) – Absolute path to the logo image (will be copied).

Return type None

anise.features.documentation module

Feature for specifying and processing project documentation.

class anise.features.documentation.**CppApiReferenceLanguage**(***kwargs*)
Bases: *anise.features.docrender.AbstractDoxygenSupportedApiReferenceLanguage*

C++ language support for api references.

anise.features.documentation.**DoxygenDescribedApiReferenceLanguage**
alias of *anise.features.docrender.AbstractDoxygenSupportedApiReferenceLanguage*

class anise.features.documentation.**ExportFormat**
Bases: object

Enumeration of output formats.

HTML = 'HTML'
HTML documentation export.

PDF = 'PDF'
PDF documentation export.

Plaintext = 'Plaintext'
Plaintext documentation export.

class anise.features.documentation.**HomepageExport**(*key*, *index=100000*, ***kwargs*)
Bases: *anise.features.documentation.Internals.Export*

Describes an export of an Anise documentation directly into the homepage.

Parameters

- **key** (*str*) – One of the keys stored in pool. Defines which documentation to export.
- **index** (*int*) – A position number. Used for ordering.

class anise.features.documentation.**Internals**
Bases: object

class AbstractFileExport(*outputfile*, ***kwargs*)
Bases: *anise.features.documentation.Internals.Export*

Abstract base class for exports, which write documentation content to files.

Parameters **outputfile** (*str*) – The content destination path.

class Export(*format*, *name=None*, *heading=None*, *mediadirs=None*, *mainbody=None*, *exclude_in_toc=None*, *include_in_toc=None*)
Bases: object

Defines a documentation export.

Parameters

- **format** (*str*) – The output format (one of *ExportFormat*).
- **name** (*Optional[str]*) – The export definition name. Finding definitions by name is possible with *exports.getbyname*.
- **heading** (*Optional[str]*) – The title.
- **mediadirs** (*Optional[List[str]]*) – Directories for additional media files like images used in the documentation.
- **mainbody** (*Optional[Union[str, Callable[[], str]]]*) – Additional text to append to the selected documentation.
- **exclude_in_toc** (*Optional[List[str]]*) – List of piece names to exclude for the table of contents. Default: Exclude none.
- **include_in_toc** (*Optional[List[str]]*) – List of piece names to include for the table of contents. Default: Include all.

generate(*targetpath*)
Generates the content.
Parameters **targetpath** (*str*) – The content destination path.
Return type None

class Exports
Bases: object

Storage of all documentation export definitions (i.e. where and how documentation products are to be created).

add(*export*)
Exports a documentation to some targets.
Parameters **export** (anise.features.documentation.Internals.Export) – A export definition that specifies a documentation export target.
Return type None

getbyname(*name*)
Returns all export definitions with a given name.
Parameters **name** (*str*) – A export definition name.
Returns List of export definitions.
Return type List[*anise.features.documentation.Internals.Export*]

class Pieces
Bases: object

Storage for documentation pieces.

add_api_reference(*, *language*, *name=None*, *rootpath=None*, ***b*)
Adds an api reference piece to the storage of documentation pieces.
Parameters
- **language** (anise.features.docrender.AbstractApiReferenceLanguage) – The programming language that is associated with this api.
- **name** (*Optional[str]*) – The internal piece name.
- **rootpath** (*Optional[str]*) – The root path of the source code.
- **b** – Additional piece arguments. See docrender.ApiReferencePiece.

Return type None

get_pieces()
Returns a list of all documentation pieces.
Return type List[*anise.features.docrender.AbstractPiece*]

class Pool
Bases: object

Storage of all documentation text generators.

add(*key*, *src*)
Adds a documentation.
Parameters
- **key** (*str*) – The documentation key for storage (later used for referencing this text).
- **src** (*Union[str, Callable[[], str]]*) – The documentation source text (or a function that returns one).

Return type None

gettexts()
Returns all texts.
Returns A list of key/text tuples.
Return type List[Tuple[str, str]]

static _add_homepage_section()

static _add_release_task_for_export_to_source()

static _compose_readmes()

static _initproject()

static generate_homepage(*, *head1*, *short_desc*, *targetdir*, *sections*, *basecolor=None*, *aliases=None*)

Writes the project homepage. You find it afterwards in `index.html` (and tons of other files).

Parameters

- **head1** (*str*) – The first heading (typically the project name).
- **short_desc** (*str*) – The additional heading description (typically a one-liner).
- **targetdir** (*str*) – The directory for the html output.
- **basecolor** (*Optional[Tuple[float, float, float]]*) – The basecolor for colouring the output.
- **sections** (*List[Tuple[str, str, str]]*) – A list of content tuples.
- **aliases** (*Optional[Dict[str, str]]*) – Custom command aliases.

Return type str

static generate_raw(*, *targetpath*, *outputformat*, *mainbody=None*, *head1=None*, *basecolor=None*, *documentations=None*, *short_desc=None*, *homepagemode=False*, *mediadirs=None*, *aliases=None*, *exclude_in_toc=None*, *include_in_toc=None*)

Generates some documentation or output. Mostly used by some higher-level functions in this module.

Parameters

- **targetpath** (*str*) – The directory for the output.
- **outputformat** (*str*) – The desired output format. See *ExportFormat*.
- **mainbody** (*Optional[str]*) – Auxiliary rst source code for additional content.
- **head1** (*Optional[str]*) – The first heading.
- **basecolor** (*Optional[Tuple[float, float, float]]*) – The basecolor for colouring the output.
- **documentations** (*Optional[List[Tuple[str, str]]]*) – An Anise documentations tuple for further pages that do not exist in the sourcecode (e.g. the readme sources) as added by `anise.features.documentation.pool.add`.
- **short_desc** (*Optional[str]*) – The additional heading description (typically a one-liner) (html output only).
- **homepagemode** (*bool*) – If to generate html output for a homepage; styling differs a bit, sidebar behavior, ... (html output only).
- **mediadirs** (*Optional[List[str]]*) – Directories for additional media files like images used in the documentation.
- **aliases** (*Optional[Dict[str, str]]*) – Custom command aliases.
- **exclude_in_toc** (*Optional[List[str]]*) – List of piece names to exclude for the table of contents. Default: Exclude none.
- **include_in_toc** (*Optional[List[str]]*) – List of piece names to include for the table of contents. Default: Include all.

Return type None

static writedocumentations(*destdir*, *throw_on_existing*, *fortype*)

Immediately exports the documentation as the export definitions currently specify.

Parameters

- **destdir** (*str*) – The destination directory path.
- **throw_on_existing** (*bool*) – If to raise an exception for an already existing destination.
- **fortype** (Type[Internals.AbstractFileExport]) – Which export type to process.

Return type None

class anise.features.documentation.**JavaScriptApiReferenceLanguage**
Bases: *anise.features.docrender.AbstractApiReferenceLanguage*

JavaScript language support for api references.

__jsfiles(*df*)

Parameters **df** (*str*) –

Return type List[str]

__scanjsfile(*f*)

Parameters **f** (*str*) –

Return type str

sphinx_config(*rootpath*, *outpath*)
Returns additional Sphinx `conf.py` configuration.

Parameters

- **rootpath** – Path to the original source directory.
- **outpath** – Path to the root directory of the Sphinx input.

sphinx_transfer(*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory `outpath`.

Parameters

- **srcpath** – The root path of the sources directory.
- **outpath** – The root path of the intermediate directory given to Sphinx.
- **heading** – The api reference heading.

class anise.features.documentation.**PackageExport**(***kwargs*)
Bases: *anise.features.documentation.Internals.AbstractFileExport*

Describes an export of an Anise documentation into the distribution package(s).

Parameters **outputfile** – The content destination path.

class anise.features.documentation.**Python3ApiReferenceLanguage**(*show_undoc_members=True*, *show_protected_members=True*, *exclude=None*, *prepare-code=''*)
Bases: *anise.features.docrender.AbstractApiReferenceLanguage*

Python 3 language support for api references.

Parameters

- **show_undoc_members** (*bool*) –
- **show_protected_members** (*bool*) –

- **exclude** (*Optional[List[str]]*) –
- **preparecode** (*str*) –

patch_property_types_in_docstrings (*pd*)
Deprecated since version all.

Parameters **pd** (*str*) –

Return type None

sphinx_config (*rootpath*, *outpath*)
Returns additional Sphinx `conf.py` configuration.

Parameters

- **rootpath** – Path to the original source directory.
- **outpath** – Path to the root directory of the Sphinx input.

sphinx_transfer (*srcpath*, *outpath*, *heading*)
Transfers auxiliary files and prepares the intermediate directory `outpath`.

Parameters

- **srcpath** – The root path of the sources directory.
- **outpath** – The root path of the intermediate directory given to Sphinx.
- **heading** – The api reference heading.

class anise.features.documentation.**RstTextsPiece** (*name*, *rootpath*, *, *sortidx=0*, *source=None*)
Bases: *anise.features.docrender.AbstractPiece*

A reStructureText formatted file or a directory of such files.

Parameters

- **name** (*str*) – The piece name.
- **rootpath** (*str*) – The root directory or file that contains the piece content.
- **sortidx** (*int*) – Sort order index.

copyto (*path*)
Copies required data files for this piece to a location.

Parameters **path** – The target location (usually inside the intermediate structure given to Sphinx).

class anise.features.documentation.**SourceExport** (**kwargs*)
Bases: *anise.features.documentation.Internals.AbstractFileExport*

Describes an export of an Anise documentation back into the source tree.

Parameters **outputfile** – The content destination path.

anise.features.filesystem module

Feature for some basic filesystem operations.

anise.features.filesystem.**filterbyextension**(*source*, *exclude=None*, *include=None*, *casesensitive=False*)

Filters a filestructure filename extension.

Parameters

- **source** (anise.framework.files.Filestructure) – The source file structure.
- **exclude** (*Optional[List[str]]*) – A list of extensions to filer away.
- **include** (*Optional[List[str]]*) – A list of extensions to not filter away.
- **casesensitive** (*bool*) – If the check is to be applied in a case sensitive manner.

Return type *anise.framework.files.Filestructure*

anise.features.filesystem.**filterbyfunction**(*acceptorfunction*, *source*)

Filters a filestructure by a path acceptor function.

Parameters

- **acceptorfunction** (*Callable[[str], bool]*) – A function returning a bool for a relative file path.
- **source** (anise.framework.files.Filestructure) – The source file structure.

Return type *anise.framework.files.Filestructure*

anise.features.filesystem.**filterbyname**(*source*, *exclude=None*, *include=None*, *casesensitive=False*)

Filters a filestructure by the file name.

Parameters

- **source** (anise.framework.files.Filestructure) – The source file structure.
- **exclude** (*Optional[List[str]]*) – A list of file names to filer away.
- **include** (*Optional[List[str]]*) – A list of file names to not filter away.
- **casesensitive** (*bool*) – If the check is to be applied in a case sensitive manner.

Return type *anise.framework.files.Filestructure*

anise.features.filesystem.**filterbynamepattern**(*source*, *exclude=None*, *include=None*, *casesensitive=False*)

Filters a filestructure by a file name pattern.

Parameters

- **source** (anise.framework.files.Filestructure) – The source file structure.
- **exclude** (*Optional[List[str]]*) – A list of file name patterns to filer away.
- **include** (*Optional[List[str]]*) – A list of file name patterns to not filter away.
- **casesensitive** (*bool*) – If the check is to be applied in a case sensitive manner.

Return type *anise.framework.files.Filestructure*

anise.features.filesystem.**subtree**(*source*, *root=''*, *copy=False*)

Gets a subtree of a filestructure.

Parameters

- **source** (anise.framework.files.Filestructure) – The source file structure.
- **root** (*str*) – The relative path to the new root directory.
- **copy** (*bool*) – If the file structure is to be copied (instead of using the original tree).

Return type *anise.framework.files.Filestructure*

anise.features.filetransfer module

Feature for transferring file structures to a destination place, e.g. on a web or file server.

class anise.features.filetransfer.**Internals**
Bases: object

static createprogresscallback(*execscope*)

static generatenewindex(*appidx*, *indexpattern*, *projectname*)

static updateindexfile(*appidxpath*, *indexpattern*, *projectname*)

anise.features.filetransfer.**localupload**(*, *destination*, *source*, *indexpattern='{name}\n'*, *contentdescription='content'*, *collectlinkshook=None*)
Deploys content to a local place (e.g. for testing).

Parameters

- **destination** (*str*) – The destination file path.
- **source** (anise.framework.files.Filestructure) – Source file structure.
- **indexpattern** (*str*) – Pattern for index entries.
- **contentdescription** (*str*) – Textual description of the kind of content.
- **collectlinkshook** (*Optional[anise.framework.features.Hook]*) – Which hook to trigger for collecting a list of symlinks to create/refresh after upload.

anise.features.filetransfer.**sshupload**(*, *host*, *username*, *destination*, *source*, *identityfile=None*, *indexpattern='{name}\n'*, *port=22*, *transferstrategy=None*, *contentdescription='content'*, *collectlinkshook=None*)
Deploys content to a server (e.g. a web server) via ssh.

Parameters

- **host** (*str*) – The server host name or address.
- **username** (*str*) – The login username.
- **destination** (*str*) – The destination file path.
- **source** (anise.framework.files.Filestructure) – Source file structure.
- **identityfile** (*Optional[str]*) – Path to a ssh identity file for login via public-key.
- **indexpattern** (*str*) – Pattern for index entries.
- **port** (*int*) – Port number.
- **transferstrategy** (*Optional[str]*) – File transfer strategy.
- **contentdescription** (*str*) – Textual description of the kind of content.

- **collectlinkshook** (*Optional[anise.framework.features.Hook]*) – Which hook to trigger for collecting a list of symlinks to create/refresh after upload.

Return type None

anise.features.gpg module

Feature for gpg encryption.

anise.features.gpg.**call_pgp**(**args*, *raise_on_errors='unable to call gpg'*, ***kwargs*)
Calls gpg with some arguments.

Parameters

- **args** (*str*) –
- **raise_on_errors** (*str*) –

Return type Tuple[int, str]

anise.features.gpg.**get_key**(*keyid*, *homedir=None*, *witharmor=True*, *withbase64=True*)
Returns a pgp key.

Parameters

- **keyid** (*str*) – The key id to look for.
- **homedir** (*Optional[str]*) – The pgp home directory.
- **witharmor** (*bool*) – If to apply an armor to the output.
- **withbase64** (*bool*) – If to apply additional base64 encoding to the output.

Return type bytes

anise.features.homepage module

Feature for generating a project homepage.

class anise.features.homepage.**Internals**
Bases: object

class Sections
Bases: object

Storage of all homepage sections.

add(*index*, *key*, *header*, *textfct*)
Adds a homepage section.

Parameters

- **index** (*int*) – A position index number. This controls the ordering of sections.
- **key** (*str*) – The section name. Also used as heading.
- **header** (*str*) – The section header text.
- **textfct** (*Callable[[], str]*) – A function returning the section documentation source.

Return type None

getsectionbykey(*key*)
Gets the section record for a key. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **key** (*str*) – The section name.

Return type Tuple[int, str, Callable[[], str], str]

gettextfct(*key*)
Returns the generator function for a key, that returns the section documentation source.
Parameters **key** (*str*) – The section name.
Return type Callable[[], str]

replacetextfct(*key*, *textfct*)
Replaces a generator function for a key.
Parameters
- **key** (*str*) – The section name.
- **textfct** (*Callable[[], str]*) – The new generator function.

Return type None

static _basic_homepage_sections()

static _initproject()

anise.features.homepage.**makehomepage**()
Creates the project homepage from the registered sections.

Return type *anise.framework.files.Filestructure*

anise.features.licensing module

anise.features.mediagalleries module

Feature for specifying, generating and processing image galleries. It is typically used as part of the project homepage.

class anise.features.mediagalleries.**Internals**
Bases: object

class Pool
Bases: object

Storage of all image galleries.

add(*gallery*, *name=''*)
Adds an image gallery.
Parameters
- **gallery** (anise.features.mediagalleries.MediaGallery) – A media gallery specification.
- **name** (*str*) – Gallery name.

Return type None

static _add_homepage_section()

static _initproject()

class anise.features.mediagalleries.**Media**(*path*, *description*, *mediatype*)
Bases: object

One image in a gallery.

Parameters
- **path** (*str*) – The image file path.
- **description** (*str*) – Description text.
- **mediatype** (*str*) – The media type. Something like 'video' or 'image'.

class anise.features.mediagalleries.**MediaGallery**(*path=None*, *paths=None*)
Bases: object

An image gallery.

Parameters

- **path** (*Optional[str]*) – The directory path where the image gallery is stored.
- **paths** (*Optional[str]*) – The directory paths where the image gallery is stored.

_matchfile(*name*)

Parameters **name** (*str*) –

Return type Optional[str]

cloneto(*destpath*)
Clones the media gallery directory to a different place.

Parameters **destpath** (*str*) – The destination where to copy the image gallery data. Absolute path.

Returns The new MediaGallery instance pointing to the cloned directory.

Return type *anise.features.mediagalleries.MediaGallery*

getmedialist()
Returns the list of media in the gallery.

Return type List[*anise.features.mediagalleries.Media*]

anise.features.milieus module

Features that provide tools to run a procedure in different environments in a transparent way.

class anise.features.milieus.**DefaultMilieu**
Bases: *anise.features.milieus.Milieu*

The default milieu.

class anise.features.milieus.**Internals**
Bases: object

static _initproject()

static get(*x=None*, *defaultmilieu=None*)
Returns a *Milieu* (or *MilieuComposite*).

Typically this milieu is created by the given specification. The specification is a list of tuples. Each tuple has at least a `'class'` key pointing to a Milieu(Composite) subclass and optional parameters. The milieux will be created stacked together.

Parameters

- **x** (*Any*) – The milieu specification (or a direct milieu instance).
- **defaultmilieu** (*Optional[anise.features.milieus.Milieu]*) – The default milieu to create if no other one is specified.

Return type Union[*anise.features.milieus.Milieu*, *anise.features.milieus.MilieuComposite*]

static registermilieu(*name*, *createcallable*)
Registers a milieu by a name.

This allows shorter names for large class names, and also to register easier factories.

Parameters

- **name** (*str*) – The name to register the new milieu with.
- **createcallable** (*Type[Milieu]*) – The class or callable that creates the instances.

class anise.features.milieus.**Milieu**

Bases: object

Milieux are a functionality that allows a build task to work in different environments.

This is advanced functionality and requires the build task to be coded in a special way. Instead of directly executing actions, e.g. executing external commands, it must use the *call()* method of a Milieu instance for it.

Tasks can get a Milieu by calling milieus.get (*Internals.get()*). Making milieux available on the other hand is possible with defaultmilieu and milieus.register (*Internals.registermilieu()*).

See subclasses or implement an own one. See also *MilieuComposite*.

Subclasses often not only implement methods from this class, but also add new tools to it.

_initialize()

Called from outside for initialization. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

call(**cmdline*, *shell=False*, *decode=True*, *raise_on_errors=None*, *split_output_channels=False*)

Executes a command line.

Parameters

- **cmdline** (*List[str]*) – The command line as list of strings.
- **shell** (*bool*) – If the command shall be interpreted by a shell (only the first part of cmdline will apply!).
- **decode** (*bool*) – If the result is to be decoded to a string.
- **raise_on_errors** (*Optional[str]*) – If errors shall raise exceptions (also specifies the exception text in this case).
- **split_output_channels** (*bool*) – If to separately record output and error stream.

Returns Tuple of returncode, program output.

Return type Tuple[int, Any]

createtempdir(*dirname=None*)

Returns a fresh temporary directory.

Parameters **dirname** (*Optional[str]*) – An optional name that the root directory should get; if not given, it gets an random name.

Return type *anise.framework.files.TempDir*

initialize()

Initializes the milieu (composite). *Override this method in custom subclasses or leave the default implementation.*

Return type None

class anise.features.milieus.**MilieuComposite**
Bases: object

A milieu composite brings parts of functionality that can be stacked on top of a Milieu.

They behave largely equally to `Milieu`s.

_initialize()
Called from outside for initialization. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

initialize()
Initializes the milieu (composite). *Override this method in custom subclasses or leave the default implementation.*

Return type None

setinner(*inner*)
Sets the inner milieu. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **inner** (*Union[Milieu, MilieuComposite]*) –

Return type None

anise.features.packages module

Features for creating distribution packages.

Those are typically packages for download by end users.

class anise.features.packages.**Internals**
Bases: object

static _initproject()

_makerawpackage_cache = None

static enrichrawpackage(*packagedir*)
Adds files to a raw packages (as created by makerawpackage), like the license file.

Parameters **packagedir** – The raw package root directory path.

static getversion()
Determines the version string of the project.

Return type str

static makerawpackage()
Builds a raw package in some flavor. This is used by other components (e.g. packaging) for creating distributable binary packages.

Return type *anise.framework.files.Filestructure*

class anise.features.packages.**RawPackage**
Bases: *anise.framework.files.TaskExecution*

A file structure that contains the raw package, which is the typical starting point for any further packaging steps or for fetching parts of the project file structure for some processing.

This is a convenience shortcut, which is equivalent to a *anise.framework.files.TaskExecution* with *Internals.makerawpackage()* as content generator.

Parameters **task** – The task implementation for execution.

anise.features.packages.**tarpackage**(*source*, *namepostfix=None*, *pkgdescription=''*)
Builds a tarball that can be distributed. This is mostly a source package and is possibly not the most convenient package type for an end-user.

Parameters

- **source** (anise.framework.files.Filestructure) – The source file structure.
- **namepostfix** (*Optional[str]*) – Postfix for the package root directory name.
- **pkgdescription** (*str*) – The package description text.

Returns A list containing new tarball.

Return type *anise.framework.files.Filestructure*

anise.features.packagestore module

Feature for storing packages on disk, for the ability to publish not only the latest release but also older ones.

class anise.features.packagestore.**Internals**
Bases: object

static _initproject()

static quotename(*s*)
Quotes a name for using as filename.

Parameters **s** (*str*) – A string to quote.

Returns A quoted string that can be used as a filename.

Return type str

static sliceversion(*versionstring*)
Slices a version string to a tuple.

Parameters **versionstring** – Version as string.

Returns Version tuple.

Return type Tuple[Any, str]

static unquotename(*s*)
Unquotes a name quoted by *quotename()* back.

Parameters **s** (*str*) – The quoted string.

Returns The original name.

Return type str

anise.features.packagestore.**enable**()
Enables using the package store.

Return type None

anise.features.projectdescriptioneditor module

anise.features.python module

Feature for some stuff around Python.

class anise.features.python.**Internals**
Bases: object

static django_quickstart(*relpath*)
Starts the application in a Django development server for fast testing. It updates the database schema before, if required.

Parameters **relpath** (*str*) – Path to the Django application. Relative to the project root directory.

anise.features.python.**is_django_project**(*relpath=''*)
Configures a project to be a Django project.

Parameters **relpath** (*str*) – Path to the Django application. Relative to the project root directory.

Return type None

anise.features.releasing module

Feature for a particular `release` task. It collects actions to execute for releasing, e.g. generating and deploying a project homepage, by triggering a hook. The `release` task is provided, which bundles all those actions.

anise.features.releasing.**HOOK_RELEASE = <anise.framework.features.Hook object>**
Hook triggered for collecting all sub-tasks for `release`.

class anise.features.releasing.**Internals**
Bases: object

class Releasetasks
Bases: object

Storage of all actions to execute for releasing.

add(*task*, *, *provides=None*, *requires=None*, *prepares=None*, ***params*)
Specifies a new task and adds it into the releasing chain.

Parameters

- **task** (*Callable*) – The task implementation to add.
- **provides** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook participates to provide (dependency provider).
- **requires** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook requires to be executed before.
- **prepares** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook must prepend.
- **params** – Additional parameters for calling that task.

static _initproject1()

static _initproject2()

static release()
Releases the project. Mostly this is executing all registered hookhandlers for *HOOK_RELEASE*.

Return type None

anise.features.signing module

Feature for signing binaries with cryptographic certificates.

class anise.features.signing.**Internals**
Bases: object

static p12cert2pem(*, *openssslpath*, *p12path*, *pemoutpath*, *password*)
Converts a .p12 certificate to a `.pem` file.

Parameters

- **opensslpath** (*str*) – Path to the OpenSSL binary.
- **p12path** (*str*) – Path to the p12 certificate file.
- **pemoutpath** (*str*) – Destination path for the pem file.
- **password** (*str*) – Certificate password of the p12 file.

Return type None

static win32_osslsigncode(*, *binpath*, *pempath*, *outpath*, *projectname*, *projecthomepage*, *osslsigncodepath*, *password*)
Signs a win32 binary with osslsigncode.

Parameters

- **binpath** (*str*) – Path to the binary to sign.
- **pempath** (*str*) – Path to the pem certificate file to use.
- **outpath** (*str*) – Destination path for the signed binary.
- **projectname** (*str*) – The project name.
- **projecthomepage** (*Optional[str]*) – The project homepage url.
- **osslsigncodepath** (*str*) – Path to the osslsigncode tool.
- **password** (*str*) – Certificate password.

Return type None

class anise.features.signing.**OsslsigncodeConfiguration**(*p12path*, *intermediatepath=None*, *osslsigncodepath='osslsigncode'*, *opensslpath='openssl'*, *password=''*)
Bases: object

Signing configuration for signing a win32 binary with the 'osslsigncode' tool.

Parameters

- **p12path** (*str*) – Path to the .p12 file.
- **intermediatepath** (*Optional[str]*) – Path to intermediate certificates.
- **osslsigncodepath** (*str*) – Path to the osslsigncode tool.
- **opensslpath** (*str*) – Path to the openssl tool.
- **password** (*str*) – The certificate passphrase.

anise.features.signing.**signwin32_osslsigncode**(*signconfig*, *source*, *required=True*)
Signs a win32 binary with a cryptographic code-signing certificate.

Parameters

- **signconfig** (anise.features.signing.OsslsigncodeConfiguration) – A configuration object.
- **source** (anise.framework.files.Filestructure) – The source file structure.
- **required** (*bool*) – If a failure is critical or can be ignored.

Return type *anise.framework.files.Filestructure*

anise.features.testing module

anise.features.ui module

Feature for user interfaces and interaction with the user.

class anise.features.ui.**ChoiceTree**
Bases: object

A tree of possible choices. Typically used in `UserFeedback.treechoice()`.

For adding nodes to the tree, use the *ChoiceTree.Node.createchild()* method, which is available on the root node of this tree as well as on all deeper nodes.

class Node(*key*, *val*, *parent*, *childs*, *image*)
Bases: object

One node in a tree of possible choices.

createchild(*key*, *val*, *image=None*)
Creates a new node in tree as as child of this node.
Parameters
- **key** (*str*) – The key string. This is displayed in tree.
- **val** (*str*) – The value description string. It is displayed in the tree as well.
- **image** (*Optional[str]*) – Icon name.

Returns The new node.
Return type *anise.features.ui.ChoiceTree.Node*

class anise.features.ui.**Internals**
Bases: object

class SetImplicitStopAllowed(*val*)
Bases: object

Controls if implicit application stop is enabled (e.g. not asking user for confirmation). *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

class UserFeedback
Bases: object

This interface provides mechanisms for asking the user for some kinds of interaction.

choice(*msg*, *choices*, *canceltext*, *image*, *choiceimages*)
Asks the user to choose from a list of alternatives.
Parameters
- **msg** (*str*) – The message to show.
- **choices** (*List[str]*) – List of choices.
- **canceltext** (*Optional[str]*) – Label of the Cancel button.
- **image** (*Optional[str]*) – Icon name.

- **choiceimages** (*Optional[List[Optional[str]]]*) – Icon names for the choices.

Returns The index of the choice the user made, or `None` if user cancelled.

Return type int

input(*msg*, *default*, *image*)

Asks the user to enter text (single line).

Parameters

- **msg** (*str*) – The message to show.
- **default** (*str*) – The default text.
- **image** (*Optional[str]*) – Icon name.

Returns The entered text, or `None` if user cancelled.

Return type Optional[str]

message(*msg*, *btns*, *image*)

Shows a message and asks the user to trigger one of the buttons.

Parameters

- **msg** (*str*) – Message text.
- **btns** (*List[str]*) – Buttons to show.
- **image** (*Optional[str]*) – Icon name.

Returns The index of the selected button.

Return type int

messageok(*msg*, *image=None*)

Shows a message and asks the user to confirm it.

Parameters

- **msg** (*str*) – The message to show.
- **image** (*Optional[str]*) – Icon name.

Return type None

messageyesno(*msg*, *image=None*)

Shows a message and asks the user to trigger either Yes or No.

Parameters

- **msg** (*str*) – The message to show.
- **image** (*Optional[str]*) – Icon name.

Returns `True`, if the user selected Yes.

Return type bool

multilineinput(*msg*, *default*, *image*)

Asks the user to enter text (multi line).

Parameters

- **msg** (*str*) – The message to show.
- **default** (*str*) – The default text.
- **image** (*Optional[str]*) – Icon name.

Returns The entered text, or `None` if user cancelled.

Return type Optional[str]

treechoice(*msg*, *choicetree*, *canceltext*, *actions*, *image*)

Asks the user to choose from a tree of alternatives.

Parameters

- **msg** (*str*) – The message to show.
- **choicetree** (anise.features.ui.ChoiceTree) – The tree of available choices.
- **canceltext** (*Optional[str]*) – Label of the Cancel button.
- **actions** (*Optional[List[str]]*) – Additional actions.
- **image** (*Optional[str]*) – Icon name.

Returns The chosen tree node, or `None` if user cancelled.

Return type Optional[*anise.features.ui.ChoiceTree.Node*]

class UserFeedbackProxy

Bases: *anise.features.ui.Internals.UserFeedback*

This user feedback implementation forwards requests to another internal `UserFeedback`, depending on which mode (web, terminal) is selected.

_getimpl()

choice(*msg*, *choices*, *canceltext='Cancel'*, *image=None*, *choiceimages=None*)

Asks the user to choose from a list of alternatives.

Parameters

- **msg** – The message to show.
- **choices** – List of choices.
- **canceltext** – Label of the Cancel button.
- **image** – Icon name.
- **choiceimages** – Icon names for the choices.

Returns The index of the choice the user made, or `None` if user cancelled.

input(*msg*, *default=''*, *image=None*)

Asks the user to enter text (single line).

Parameters

- **msg** – The message to show.
- **default** – The default text.
- **image** – Icon name.

Returns The entered text, or `None` if user cancelled.

message(*msg*, *btns*, *image=None*)

Shows a message and asks the user to trigger one of the buttons.

Parameters

- **msg** – Message text.
- **btns** – Buttons to show.
- **image** – Icon name.

Returns The index of the selected button.

multilineinput(*msg*, *default=''*, *image=None*)

Asks the user to enter text (multi line).

Parameters

- **msg** – The message to show.
- **default** – The default text.
- **image** – Icon name.

Returns The entered text, or `None` if user cancelled.

treechoice(*msg*, *choicetree*, *canceltext='Cancel'*, *actions=None*, *image=None*)

Asks the user to choose from a tree of alternatives.

Parameters

- **msg** – The message to show.
- **choicetree** – The tree of available choices.
- **canceltext** – Label of the Cancel button.
- **actions** – Additional actions.
- **image** – Icon name.

Returns The chosen tree node, or `None` if user cancelled.

static _default()

Opens the graphical task chooser and executes the selected task afterwards.

static _default_baddir()

Opens the graphical guide for starts in non-Anise directories.

static initproject1()

static initproject2()

class anise.features.ui.**Mode**
Bases: object

Enumeration of user interface modes.

TERMINAL = 'TERMINAL'
Terminal user interface.

WEB = 'WEB'
Web user interface.

anise.features.ui.**chooser_hide_showall**(*hide=True*)
Sets the availability of the *show all* button in the task chooser, that also shows the unimportant tasks.

Parameters **hide** (*bool*) – If the *show all* button is to be hidden.

Return type None

anise.features.ui.**enforce_terminal_mode**()
Force usage of terminal mode.

Return type None

anise.features.versioncontrol module

Features for interaction with version control systems.

class anise.features.versioncontrol.**ChangeLogChange**(*changetext*, *commitmessage*)
Bases: object

A change entry in a change log.

Parameters

- **changetext** (*str*) – The change text.
- **commitmessage** (anise.features.versioncontrol.CommitMessage) – The commit message that this version label is defined by.

class anise.features.versioncontrol.**ChangeLogRevision**(*versionlabel*, *commitmessage*)
Bases: object

A revision section in a change log.

Parameters

- **versionlabel** (*str*) – The version label.
- **commitmessage** (anise.features.versioncontrol.CommitMessage) – The commit message that this version label is defined by.

class anise.features.versioncontrol.**CommitMessage**(*message*, *revision*, *committime=None*, *username=None*, ***kwargs*)
Bases: object

A commit message.

Parameters

- **message** (*str*) – The message text.
- **revision** (*int*) – The revision.
- **committime** (*Optional[datetime.datetime]*) – The time of committing.
- **username** (*Optional[str]*) – The user who committed.
- **kwargs** – Additional commit data.

class anise.features.versioncontrol.**Internals**

Bases: object

class PseudoVersionControlSystem

Bases: *anise.features.versioncontrol.Internals.VersionControlSystem*

A pseudo version control system implementation, which does nothing. This is the default as long as no real version control system is configured.

syncversioncontrolsystem(*commitmessage=''*, *forceskipped=True*, *forceunchanged=False*)

Synchronizes the local working copy with the version control system. *Override this method in custom subclasses.*

Parameters

- **commitmessage** – Commit message text.
- **forceskipped** – Force synchronizing even if the user decided to skip that.
- **forceunchanged** – Force synchronizing even if there are no changes to commit.

class VersionControlSystem

Bases: object

A version control system.

Override this for providing an own version control system.

addchange(*changetext=None*)

Adds a change to the changelog.

Parameters **changetext** (*Optional[str]*) – The changelog entry text (None: input dialog).

Return type None

addlabel(*label=None*)

Adds a version label for the current state.

Parameters **label** (*Optional[str]*) – The label string (None: input dialog).

Return type None

bindvcs(*l*)

Binds the current project to this version control system. Uses user interaction for getting details like urls. *Override this method in custom subclasses or leave the default implementation.*

Parameters **l** (anise.framework.projects.IntermediateStructure.Let) – The let configuration node for storing additional parameters.

Returns True if the project was bound successfully.

Return type bool

branchfromhere(*branchname=None*)

Creates a new branch from here and switches to it. Unpushed changes will become part of that new branch. *Override this method in custom subclasses.*

Parameters **branchname** (*Optional[str]*) – The branch name (None: input dialog).

Return type None

createrawpackage()
Generates a raw package. This package is the basis for most (or all) of the other package types. *Override this method in custom subclasses.*
Return type *anise.framework.files.Filestructure*

fetchrawcommitmessage(*revision*)
Fetches a raw commit message. Do not use it directly, but getcommitmessage instead. *Override this method in custom subclasses or leave the default implementation.*
Parameters **revision** (*int*) – The revision.
Returns The raw commit message as string in a custom format.
Return type str

findvcs()
Checks if the current project is bound to a version control system of this kind. *Override this method in custom subclasses or leave the default implementation.*
Return type bool

getchangelog(*asstring=True*)
Computes the project change log from information in specially formatted vcs commit messages.
Parameters **asstring** (*bool*) – If to output the changelog as string (instead of an internal data structure).
Return type Union[str, List[*anise.features.versioncontrol.ChangeLogRevision*]]

getcommitmessage(*revision*)
Returns the raw commit message (as formatted by the vcs tool) for a revision.
Parameters **revision** (*int*) – The revision.
Return type *anise.features.versioncontrol.CommitMessage*

getcommitmessagefromrawcommitmessage(*s*)
Gets the actual commit message from the raw *fetchrawcommitmessage()* commit message output. Do not use it directly, but *getcommitmessage()* instead. *Override this method in custom subclasses or leave the default implementation.*
Parameters **s** (*str*) – The raw commit message string.
Return type *anise.features.versioncontrol.CommitMessage*

getfullversion()
Returns a complete version string. *Override this method in custom subclasses.*
Return type str

getheadrevision()
Gets the head revision. *Override this method in custom subclasses or leave the default implementation.*
Return type Optional[int]

getpreviousrevision(*revision*)
Gets the predecessor revision for a given revision. *Override this method in custom subclasses or leave the default implementation.*
Parameters **revision** (*int*) – A revision.
Returns The predecessing revision.
Return type Optional[int]

storechange(*changetext*)
Stores a new entry to the changelog.

Do not use this directly, but *addchange()* instead.
Parameters **changetext** (*str*) – The changelog entry text.
Return type None

storelabel(*label*)
Stores a new version label.

Do not use this directly, but *addlabel()* instead.
Parameters **label** (*str*) – The label string.
Return type None

syncversioncontrolsystem(*commitmessage=''*, *forceskipped=True*, *forceunchanged=False*)
Synchronizes the local working copy with the version control system. *Override this method in custom subclasses.*
Parameters
- **commitmessage** (*str*) – Commit message text.
- **forceskipped** (*bool*) – Force synchronizing even if the user decided to skip that.
- **forceunchanged** (*bool*) – Force synchronizing even if there are no changes to commit.

Return type None

static _add_homepage_section()

static _prepare_fallbacks()

static _prepare_vcstasks()

anise.features.versioncontrol.**setvcs**(*vcs*)
Sets the version control system for this project.

Parameters **vcs** (anise.features.versioncontrol.Internals.VersionControlSystem) – A version control system.

Return type None

Module contents

High level functionality provided by Anise (read documentation for more!).

Anise features are a plugin system. All high-level functionality of anise is implemented in one of the features that are bundled with Anise. They are loaded by the feature loader *anise.framework.features* at runtime.

anise.framework package

Submodules

anise.framework.engine module

The anise engine initializes all the anise plattform stuff and executes a task from a project file.

anise.framework.engine.**HOOK_AFTER_PREPARATION = <anise.framework.features.Hook object>**
Hook triggered after the project description file is applied to the universe object.

anise.framework.engine.**HOOK_SHUTDOWN = <anise.framework.features.Hook object>**
Hook triggered in the end of the execution.

anise.framework.engine.**HOOK_TASK_RESULT_AVAILABLE = <anise.framework.features.Hook object>**
Hook triggered when the task result is available.

class anise.framework.engine.**SpecialPaths**(*projectfile*, *projectdir*)
Bases: object

A data structure of some particular special paths.

Parameters

- **projectfile** (*str*) – The path to the anise project description file. Their name is usually _projectdesc.
- **projectdir** (*str*) – The path to the project root directory.

anise.framework.engine.**_cmdline**(*args*)
Parses the *command line*.

Parameters **args** (*List[str]*) –

Return type Any

anise.framework.engine.**_parsevalue**(*sl*)
Parses a *command line* universe object value assignment like s:name=foo.

Returns A tuple of type,name,value

Parameters **sl** (*str*) –

Return type Tuple[Optional[str], Optional[str], Optional[str]]

anise.framework.engine.**execute**(*taskname*, *projectfile*, *projectdir*, *values*, *loadfeaturesfrom*)
Executes a task.

Parameters

- **taskname** (*str*) – The name of the task to be executed.
- **projectfile** (*str*) – The path to the anise project description file. Their name is usually _projectdesc.
- **projectdir** (*str*) – The path to the project root directory.
- **values** (*Dict[str, Any]*) – Additional values to be set to the universe object.
- **loadfeaturesfrom** (*List[str]*) – List of paths where anise features shall be loaded from.

Return type Any

anise.framework.engine.**main**()
The main method when called from command line. Parses command line arguments and executes the chosen task.

Return type None

anise.framework.exceptions module

Exception subclasses.

exception anise.framework.exceptions.**AniseError**
Bases: Exception

An error in the execution of Anise.

exception anise.framework.exceptions.**BadCommunicationError**
Bases: *anise.framework.exceptions.AniseError*

The communication with some other component failed.

exception anise.framework.exceptions.**BadFormatError**
Bases: *anise.framework.exceptions.AniseError*

Some input is wrongly formatted.

exception anise.framework.exceptions.**BadInputError**
Bases: *anise.framework.exceptions.AniseError*

A bad value was given as some input.

exception anise.framework.exceptions.**ImplementationError**
Bases: *anise.framework.exceptions.AniseError*

A software error in an implementation occurred.

exception anise.framework.exceptions.**InternalError**
Bases: *anise.framework.exceptions.ImplementationError*

An internal error in the anise code occurred.

exception anise.framework.exceptions.**NotImplementedError**
Bases: *anise.framework.exceptions.InternalError*

Some unimplemented functionality was called.

exception anise.framework.exceptions.**ProcessExecutionFailedError**(*text*, *output=None*)
Bases: *anise.framework.exceptions.AniseError*

The execution of an external process failed.

Parameters

- **text** (*str*) –
- **output** (*str*) –

exception anise.framework.exceptions.**RequirementsMissingError**
Bases: *anise.framework.exceptions.AniseError*

Something required is not in place.

exception anise.framework.exceptions.**UnexpectedSituationError**
Bases: *anise.framework.exceptions.AniseError*

An unexpected situation led to an error.

anise.framework.features module

The Anise plug-in system.

This package does not include the Anise high-level features itself. You would find them in *anise.features*. This package is the part of the Anise infrastructure that loads and manages those features.

For each Anise run, it automatically loads the built-in *anise.features* as well as custom features from `~/.anise/features` and `/var/lib/anise/features`. Due to their locations, the latter ones are useful for per-machine and per-user customization.

In an arbitrary Anise execution context, you may call *anise.framework.features.loadfeature()* in order to access functionality from other features, like here:

```
from anise.framework import features
_somefeature = features.loadfeature("somefeature").featuremodule
_somefeature.some_method()
x = _somefeature.some_variable
```

class anise.framework.features.**Feature**(*name*, *hookhandlers*, *featuremodule*, *hooks*)

Bases: object

A feature is some encapsuled part of anise high-level functionality. This class represents a loaded feature at runtime. The framework as well as other features access this object for interacting with that feature. Direct usage of high-level features from within _projectdesc is intended to not use this Feature objects but to use the embedded properties of the universe object.

Parameters

- **name** (*str*) – The optional name of the feature.
- **hookhandlers** (*List[Any]*) – All hook handlers this feature registers globally (it may do that later on in a function as well).
- **featuremodule** (*Any*) – The feature module.
- **hooks** (*Dict[str, anise.framework.features.Hook]*) – List of Hook instances.

anise.framework.features.**HOOK_BEFORE_DEFINITION = <anise.framework.features.Hook object>**

This hook gets executed by the Anise engine right before the _projectdesc file is loaded.

anise.framework.features.**HOOK_BEFORE_EXECUTION = <anise.framework.features.Hook object>**

This hook gets executed by the Anise engine after the _projectdesc file is loaded and right before the chosen task itself is called. At this time all the assignments and global statements in the _projectdesc are already applied.

class anise.framework.features.**Hook**(*hookkind='FUNCTION'*, ***, *hide=False*, *interesting=5*, *doc=None*, *wizard_functionparams=None*, *wizard_functionadditionallog=None*, *wizard_functionbody=None*, *wizard_classbase=None*, *wizard_classbody=None*, *wizard_loadfeatures=None*)

Bases: object

A hook. See documentation for details.

Parameters

- **hookkind** (*str*) – A hook kind (one of *HookKind*).
- **hide** (*bool*) – If this hook is to be hidden in hook choosers.
- **interesting** (*int*) – A number that controls how interesting this hook is in a chooser.
- **doc** (*Optional[str]*) – Hook documentation text.
- **wizard_functionparams** (*Optional[List[str]]*) – Info for hook handler wizards. List of function parameters.
- **wizard_functionadditionallog** (*Optional[List[str]]*) – Info for hook handler wizards. List of additional variable to log out in a function.
- **wizard_functionbody** (*Optional[str]*) – Info for hook handler wizards. Additional function body part.
- **wizard_classbase** (*Optional[str]*) – Info for hook handler wizards. Base class.

- **wizard_classbody** (*Optional[str]*) – Info for hook handler wizards. Additional class body part.
- **wizard_loadfeatures** (*Optional[List[str]]*) – Info for hook handler wizards. Additional features to load.

class anise.framework.features.**HookHandler**(*hookobj*, *function*, *provides*, *requires*, *prepares*)

Bases: object

A hook handler bound to some hook. See documentation for details.

Parameters

- **hookobj** (anise.framework.features.Hook) – The hook.
- **function** (*Callable*) – The handler function/class.
- **provides** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook participates to provide (dependency provider).
- **requires** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook requires to be executed before.
- **prepares** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook must prepend.

property function

The handler function/class.

Return type Callable

property hookobj

The hook.

Return type *Hook*

property prepares

The symbol names this hook must prepend.

Return type List[str]

property provides

The symbol names this hook participates to provide (dependency provider).

Return type List[str]

property requires

The symbol names this hook requires to be executed before.

Return type List[str]

class anise.framework.features.**HookKind**

Bases: object

Enumeration of hook kinds.

CLASS = 'CLASS'

A class kind hook.

FUNCTION = 'FUNCTION'

A function kind hook.

class anise.framework.features.**Internals**

Bases: object

Internal stuff.

static inject_defaults_to_universe(*u*)
Adds feature data structures to the universe object.

Parameters **u** (projects.Universe) –

Return type None

static inject_hooks_to_universe(*u*)
Adds hooks to the universe object, which are loaded in feature loading.

Parameters **u** (projects.Universe) –

Return type None

static loadfeatures(*featurespath*)
Loads features from a path.

Parameters **featurespath** (*str*) –

Return type None

static resetfeatures()
Resets the features.

Return type None

class anise.framework.features.**PseudoFeature**
Bases: object

This is an auxiliary class, which fixes the gap that appears, e.g. if a feature foo.bar exists, but no foo one.

anise.framework.features.**getfeatures**()
Returns a list of all features.

Return type List[*anise.framework.features.Feature*]

anise.framework.features.**hookhandler**(*hookobj*, *, *provides=None*, *requires=None*, *prepares=None*)
Used by a feature module to register a handler for a hook globally. The result should be used as Python function decorator.

Parameters

- **hookobj** (anise.framework.features.Hook) – The hook.
- **provides** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook participates to provide (dependency provider).
- **requires** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook requires to be executed before.
- **prepares** (*Optional[Union[str, List[str]]]*) – A list of symbol names this hook must prepend.

Return type Callable

anise.framework.features.**loadfeature**(*name*)
Load a feature by name from the predefined sources (as optional command-line argument). If if is already loaded, it returns the existing one.

Parameters **name** (*str*) –

Return type *anise.framework.features.Feature*

anise.framework.files module

File structures, used for data exchange between methods or as result.

class anise.framework.files.**AbstractRebuildDirectoriesFilestructure**
Bases: *anise.framework.files.TaskExecution*

Abstract base class for merging directory structures together.

See the subclasses.

Parameters **task** – The task implementation for execution.

__getmerged()

Return type *anise.framework.files.Filestructure*

getdests()
Returns a list of transfer destinations.

Returns A list of tuples, each beginning with the destination path (and other custom ones).

Return type List[Tuple[str, *anise.framework.files.Filestructure*]]

getsource(*desttuple*)
Returns the transfer source for a destination tuple (as it was part of *getdests()*' result).

Parameters **desttuple** (*Tuple[str, anise.framework.files.Filestructure]*) –

Return type *anise.framework.files.Filestructure*

class anise.framework.files.**ExcludeByName**(*source*, *names*)
Bases: *anise.framework.files.TempDir*

Excludes some files/directories in a directory by name.

Parameters

- **source** (*anise.framework.files.Filestructure*) – The source file structure.
- **names** (*List[str]*) – Names to exclude.

_initialize()
The internal initialization routine, which fetches dynamic content (if any). Called from outside before actual accesses occur. *Override this method in custom subclasses or leave the default implementation.*

class anise.framework.files.**Filestructure**(*path*, *relative=False*)
Bases: object

This class represents a file or a directory structure with many subdirectories and files. There are methods for writing those items to disk and/or to modify the original structure in the filesystem.

It is more powerful for file exchange than working just with file paths, because a file structure can also carry arbitrary metadata information and allows dynamic content creation (see e.g. *TaskExecution*).

Instances of this class (and its subclasses) are used by many features for file exchange.

Parameters

- **path** (*Optional[str]*) – Path to the file or directory.
- **relative** (*bool*) – If path is relative to the project directory.

_initialize()
The internal initialization routine, which fetches dynamic content (if any). Called from outside before actual accesses occur. *Override this method in custom subclasses or leave the default implementation.*

Return type None

property data
The file structure metadata.

Return type Dict[str, Any]

datakeys()
Returns a list of keys for all stored metadata properties.

Return type List[str]

dl(*to=None*, ***, *subpath=''*, *progresscallback=None*)
Copies the complete filestructure or a subdirectory to a new destination.

Parameters

- **to** (*Optional[str]*) – If given, it specifies the destination path; if not given, a new directory in the current working directory will be used.
- **subpath** (*str*) – If given, only this subdirectory will be copied.
- **progresscallback** (*Optional[Callable[[int, int], None]]*) – Callback function for retrieving progress information.

Returns The destination path.

Return type str

getdata(*k*, *deflt*)
Gets a metadata property.

Parameters

- **k** (*str*) – The key name.
- **deflt** (*Optional[Any]*) – The default value (returned if the key does not exist).

Return type Optional[Any]

initialize()
Initializes this Filestructure. When this is called, the dynamic content (if any) is created and written to disc. It is automatically called for typical usage and is not required to be called manually.

Return type None

mv(*newname*)
Moves the file structure to a new path. Returns a new file structure resulting in a movement from the old one, but also updates this one internally.

Parameters **newname** (*str*) – The new name (absolute or relative).

Return type *anise.framework.files.Filestructure*

property path
The path to this file structure in the filesystem.

Return type str

setautoopen(*path*, *interminal=False*)
Places a hint that requests the Anise engine to automatically open a file within this structure when it is returned to the user.

Parameters

- **path** (*str*) – The relative path to the file that should automatically be opened.

- **interminal** (*bool*) – If this hint is for terminal (non-graphical) usage.

Return type None

setdata(*k*, *v*)
Sets a metadata property.

Parameters

- **k** (*str*) – The key name.
- **v** (*Optional[Any]*) – The value.

Return type None

with_modified_rootname(*newname*)
Returns a (temporary) file structure clone with a new directory name.

Parameters **newname** (*str*) – The name that the new root directory should get.

Return type *anise.framework.files.Filestructure*

class anise.framework.files.**MergedFilestructure**(*fromlist=None*, *frommap=None*)
Bases: *anise.framework.files.AbstractRebuildDirectoriesFilestructure*

Merges many file structures together in one tree.

Parameters

- **fromlist** (*Optional[List[anise.framework.files.Filestructure]]*) – A list of file structures. Become root directory entries with their original file names.
- **frommap** (*Optional[Union[Dict[str, anise.framework.files.Filestructure], List[Tuple[str, anise.framework.files.Filestructure]]]]*) – A dict of string/filestructure. Values become part of the result tree with the dict key specifying a subpath.

getdests()
Returns a list of transfer destinations.

Returns A list of tuples, each beginning with the destination path (and other custom ones).

getsource(*desttuple*)
Returns the transfer source for a destination tuple (as it was part of *getdests()*' result).

class anise.framework.files.**RemappedFilestructure**(*source*, *frommap=None*)
Bases: *anise.framework.files.AbstractRebuildDirectoriesFilestructure*

Remaps a file structure's inner directory hierarchy into a new tree.

Parameters

- **source** (*anise.framework.files.Filestructure*) – The source file structure.
- **frommap** (*Optional[Union[Dict[str, str], List[Tuple[str, str]]]]*) – A dict of string/filestructure. Values are relative source paths with the dict key specifying a target subpath.

getdests()
Returns a list of transfer destinations.

Returns A list of tuples, each beginning with the destination path (and other custom ones).

getsource(*desttuple*)
Returns the transfer source for a destination tuple (as it was part of *getdests()*' result).

class anise.framework.files.**TaskExecution**(*task*, ***kwargs*)
Bases: *anise.framework.files.Filestructure*

Holds a function and some parameters for calling later on (like a delegate; or a more explicit, specialized and 'xml-compatible' version of a lambda) in order to get a file structure as function result. It acts as a proxy on top of this file structure.

Parameters **task** (*Callable*) – The task implementation for execution.

_initialize()
The internal initialization routine, which fetches dynamic content (if any). Called from outside before actual accesses occur. *Override this method in custom subclasses or leave the default implementation.*

class anise.framework.files.**TempDir**(*dirname=None*, *temprootpath=None*, *namepattern='anise.{pid}.{i}'*)
Bases: *anise.framework.files.Filestructure*

A special file structure that automatically creates a temporary directory in background.

Example:

```
def sometask():
    mytmp = files.TempDir()
    with open(mytmp.path + "/somefile", "w") as f:
        f.write("some content for our new file")
    return mytmp
```

Parameters

- **dirname** (*Optional[str]*) – An optional name that the root directory should get; if not given, it gets an random name.
- **temprootpath** (*Optional[str]*) – An optional alternate temp root directory path.
- **namepattern** (*str*) – The directory name pattern.

__cleanup()
Deletes the original temp dir from the hard drive.

Return type None

mv(**a*, ***b*)
Moves the file structure to a new path. Returns a new file structure resulting in a movement from the old one, but also updates this one internally.

Parameters **newname** – The new name (absolute or relative).

class anise.framework.files.**TextFileByContent**(*content*)
Bases: *anise.framework.files.Filestructure*

Holds a function and some parameters for calling later on (like a delegate; or a more explicit, specialized and 'xml-compatible' version of a lambda) in order to get a file structure as function result. It acts as a proxy on top of this file structure.

Parameters **content** (*str*) – The text content for the file.

anise.framework.globalvars module

Global variables that control some global aspects of anise behavior. They are potentially influenced by command line.

anise.framework.imports module

Imports some modules for availability within `_projectdesc`.

Used internally.

anise.framework.projectinformations module

anise.framework.projects module

Implementation for the Anise universe object and some services around it.

class anise.framework.projects.**IntermediateStructure**(*entries=None*)
Bases: object

A data structure that holds an in-memory representation of the project description. It is not the in-memory representation of the live universe object state. It is like the file structure of the project description file, but not exactly the same. There is a parser and a generator for translating between a project description file and an IntermediateStructure.

Parameters **entries** (*Optional[List[IntermediateStructure.Node]]*) – The new list of direct subitems.

Return type None

class Let(*k*, ***args*)
Bases: *anise.framework.projects.IntermediateStructure.Node*

A Let entry in the project description structure. This corresponds to a `let` node in the xml format. It represents the execution of a parameterized part of anise feature program logic.

:param k The full qualified method name.

Parameters **k** (*str*) –

class Node(*k*, ***args*)
Bases: object

A node in the intermediate structure.

Parameters **k** (*str*) – The node key.

args()
Returns the list of stored argument keys.
Return type List[str]

clearargs()
Clears the list of arguments.
Return type None

getarg(*key*)
Returns an argument.
Parameters **key** (*str*) – The argument key.
Return type *anise.framework.projects.IntermediateStructure.Value*

property isvalue
If this is a value node. See also *IntermediateStructure.Value*.
Return type bool

property key
The key name.
Return type str

property parent
The parent node.

The root node returns its *IntermediateStructure*.
Return type Union['IntermediateStructure.Node', 'IntermediateStructure']

remove()
Removes this node.
Return type None

removearg(*key*)
Removes an argument.
Parameters **key** (*str*) – The argument key.
Return type *anise.framework.projects.IntermediateStructure.Value*

setarg(*key*, *node*)
Sets an argument.
Parameters
- **key** (*str*) – The argument key.
- **node** (anise.framework.projects.IntermediateStructure.Value) – The argument value.

Return type None

exception ParseProjectDescriptionError
Bases: *anise.framework.exceptions.BadFormatError*

Parsing error like bad input xml.

exception ParseProjectDescriptionInternalError
Bases: *anise.framework.exceptions.InternalError*

Parsing error that seems like a bug.

class Value(*t*, *, *k=None*, *v=None*, *oref=None*, ***args*)
Bases: *anise.framework.projects.IntermediateStructure.Node*

A Value entry in the project description structure. This corresponds to a `val` node in the xml format. It represents an arbitrary value (primitive or composed type).

:param t The value type. :param k The key (i.e. property name in parent object). :param v (optional) The value string for a primitive value. :param oref (optional) The oref string for a objectref value.

Parameters
- **t** (*str*) –
- **k** (*Optional[str]*) –
- **v** (*Optional[str]*) –
- **oref** (*Optional[str]*) –

_realign_listvalues()
Makes internal list realignment tasks.

This is required to call after the list structure changes.

Return type None

appendarg(*node*)
Appends a new value node to a list node.
Parameters **node** (anise.framework.projects.IntermediateStructure.Value) –
Return type None

static frombool(*val*, *key=None*)
Convenience method that creates a boolean Value.
Parameters
- **val** (*Any*) –
- **key** (*Optional[str]*) –

Return type *anise.framework.projects.IntermediateStructure.Value*

static fromstring(*val*, *key=None*)
Convenience method that creates a string Value.
Parameters
- **val** (*Any*) –
- **key** (*Optional[str]*) –

Return type *anise.framework.projects.IntermediateStructure.Value*

getprimitiverepr()
Returns a string representation of the value, if it is a primitive one (None otherwise).
Return type Optional[str]

static istrue(*s*)
Returns True iff the given string can represent a boolean True value.
Parameters **s** (*str*) –
Return type bool

property isvalue
If this is a value node. See also *IntermediateStructure.Value*.
Return type bool

property oref
The objectref (for oref values).
Return type str
This property is also settable.

property value
The value (for primitive values).
Return type str
This property is also settable.

property vtype
The value type name.
Return type str
This property is also settable.

static _fromxml_let(*xentry*)
Creates a let node from an xml node.

Parameters **xentry** (*xml.etree.ElementTree.Element*) –

Return type *anise.framework.projects.IntermediateStructure.Let*

static _fromxml_val(*xentry*)
Creates a value node from an xml node.

Parameters **xentry** (*xml.etree.ElementTree.Element*) –

Return type *anise.framework.projects.IntermediateStructure.Value*

static _fromxml_val_isdict(*xentry*, *withoutkeys=False*)
Checks if a particular xml node represents a value dict.

Parameters

- **xentry** (*xml.etree.ElementTree.Element*) –
- **withoutkeys** (*bool*) –

Return type bool

static _fromxml_val_islist(*xentry*)
Checks if a particular xml node represents a value list.

Parameters **xentry** (*xml.etree.ElementTree.Element*) –

Return type bool

static _fromxml_val_isstring(*xentry*)
Checks if a particular xml node represents a string.

Parameters **xentry** (*xml.etree.ElementTree.Element*) –

Return type bool

static _inject_to_universe_let(*entry*, *g*, *l*)
Executes the let action from a let node.

Parameters

- **entry** (anise.framework.projects.IntermediateStructure.Let) –
- **g** (*Dict[str, Optional[Any]]*) –
- **l** (*Dict[str, Optional[Any]]*) –

Return type None

static _inject_to_universe_resolve(*oref*, *g*, *l*)
Returns the resolved object for an objectref string.

Parameters

- **oref** (*str*) –
- **g** (*Dict[str, Optional[Any]]*) –
- **l** (*Dict[str, Optional[Any]]*) –

Return type Optional[Any]

static _inject_to_universe_value(*entry*, *g*, *l*)
Returns a plain Python value from a value node.

Parameters

- **entry** (anise.framework.projects.IntermediateStructure.Value) –
- **g** (*Dict[str, Optional[Any]]*) –
- **l** (*Dict[str, Optional[Any]]*) –

Return type Optional[Any]

static _toxml_let(*entry*)
Converts a let node to an xml node.

Parameters **entry** (anise.framework.projects.IntermediateStructure.Let) –

Return type xml.etree.ElementTree.Element

static _toxml_val(*entry*, *xcurr*)
Converts a value node to an xml node.

Parameters **entry** (anise.framework.projects.IntermediateStructure.Value) –

Return type xml.etree.ElementTree.Element

addentry(*node*)
Adds a new entry to the list of direct subentries.

Parameters **node** (anise.framework.projects.IntermediateStructure.Node) – The new node.

Return type None

entries()
Returns the list of direct subitems.

Return type List[*anise.framework.projects.IntermediateStructure.Node*]

find(*key*, *lets=False*, *vals=False*)
Find all :samp:`let`s and/or :samp:`val`s with a certain key.

Parameters

- **key** (*str*) –
- **lets** (*bool*) –
- **vals** (*bool*) –

Return type List[*anise.framework.projects.IntermediateStructure.Node*]

static fromfile(*filename*)
Generates an IntermediateStructure for an xml file.

Parameters **filename** (*str*) – Path to the project description file.

Return type *anise.framework.projects.IntermediateStructure*

static fromxml(*xmlstring*, *sourcefilepath=None*)
Generates an IntermediateStructure for an xml document.

Parameters

- **xmlstring** (*str*) – XML content.
- **sourcefilepath** (*Optional[str]*) – Path to the source file, e.g. used for *save()*.

Return type *anise.framework.projects.IntermediateStructure*

inject_to_universe(*u*, *g*, *l*)
Injects the (always static) information from this project description into a living universe object (typically for executing a task on top of it).

Parameters

- **u** (anise.framework.projects.Universe) – The universe object.
- **g** (*Dict[str, Optional[Any]]*) – The globals dict.
- **l** (*Dict[str, Optional[Any]]*) – The locals dict.

Return type None

removeentry(*node*)
Removes an entry from the list of direct subentries.

Parameters **node** (anise.framework.projects.IntermediateStructure.Node) – The child node to remove.

Return type None

removeentryat(*i*)
Removes an entry from the list of direct subentries.

Parameters **i** (*int*) – The entry position to remove.

Returns The deleted node.

Return type *anise.framework.projects.IntermediateStructure.Node*

static resolveobjectref(*oref*)
Returns the resolved object for an objectref string.

Parameters **oref** (*str*) – The objectref string (e.g. `'homepage.makehomepage'` or `'some.foo.Class().meth()+1'`).

Return type Optional[Any]

save()
Saves the current state of this intermediate structure back to the disk.

This typically works only for instances that were loaded from disk.

Return type None

setentries(*entries*)
Sets the list of direct subentries.

Parameters **entries** (*List[IntermediateStructure.Node]*) – The new list of nodes.

Return type None

tofile(*filename*)
Saves the current state of this intermediate structure to a file.

Parameters **filename** (*str*) – The destination path.

Return type None

toxml()
Generates an xml representation for this intermediate structure.

Return type str

class anise.framework.projects.**Universe**(*d*, *executionscope=None*)
Bases: object

Implementation for universe objects.

Parameters

- **d** (*Dict[str, Optional[Any]]*) –
- **executionscope** (*Optional[report.ExecutionScope]*) –

class RegisteredTask(*parentname*, *taskname*, *label*, *important*, *invisible*, *sortindex*, *function*)
Bases: object

A task that is registered to the universe object. You typically do not need to work with this structure directly. Use *Universe.addtask()* to add new tasks.

Parameters

- **parentname** (*str*) –
- **taskname** (*str*) –
- **label** (*Optional[str]*) –
- **important** (*bool*) –
- **invisible** (*bool*) –
- **sortindex** (*int*) –
- **function** (*Callable*) –

addhook(*hookobj*, *fct*, *params=None*, *provides=None*, *requires=None*, *prepares=None*)
Adds a handler for a hook to the universe object.

Parameters

- **hookobj** (anise.framework.features.Hook) –
- **fct** (*Callable*) –
- **params** (*Optional[Dict[str, Optional[Any]]]*) –
- **provides** (*Optional[Union[str, List[str]]]*) –
- **requires** (*Optional[Union[str, List[str]]]*) –
- **prepares** (*Optional[Union[str, List[str]]]*) –

Return type *anise.framework.features.HookHandler*

addtask(*function*, *name=''*, ***, *label=None*, *important=True*, *invisible=False*, *sortindex=0*)
Registers a task and makes it available for execution.

Parameters

- **function** (*Callable*) – The function that implements the task.
- **name** (*str*) – The task name.
- **label** (*Optional[str]*) – The task label text.
- **important** (*bool*) – If this is an important/interesting task (presented differently in choosers).
- **invisible** (*bool*) – If this task should be hidden in choosers.
- **sortindex** (*int*) – A number that controls the display order of tasks in choosers.

Return type None

append(*d*)
Stores a dictionary of values as properties of this universe object.

Parameters **d** (*Dict[str, Optional[Any]]*) –

Return type None

gethooks(*hookobj*)
Gets the handlers for a hook in the correct order.

Parameters **hookobj** (anise.framework.features.Hook) – The hook you want to get handlers for.

Return type List[Callable]

getvalue(*key*, *defaultvalue=None*)
Returns the value of a certain property for the current universe object.

If this property does not exist, it returns the `defaultval`. Beyond that, `getvalue("foo.bar")` is similar to `universe.foo.bar`.

Parameters

- **key** (*str*) – The key name of the property to get.
- **defaultvalue** (*Optional[Any]*) – This value is returned if the specified property does not exist.

Return type Optional[Any]

registeredtasks()
Returns a list of all registered tasks.

Return type List[*anise.framework.projects.Universe.RegisteredTask*]

removehook(*hookhandler*)
Removes a hook that was added before.

Parameters **hookhandler** (anise.framework.features.HookHandler) – The HookHandler you got in *addhook()*.

Return type None

setvalue(*key*, *value*)
Sets the value of a certain property for the current universe object.

If a parent property does not exist, it does nothing. `setvalue("foo.bar", "baz")` is similar to `universe.foo.bar = "baz"`.

Parameters

- **key** (*str*) – The key name of the property to set.
- **value** (*Optional[Any]*) – The new value for the specified property.

Returns If successful.

Return type bool

class anise.framework.projects.**_UniverseProxy**
Bases: object

Internally used for the universe object for allowing just to write `universe` instead of `universe()`.

anise.framework.projects.**currentuniverse**()
Returns the current universe object.

Return type Optional[*anise.framework.projects.Universe*]

anise.framework.projects.**getvalue**(*key*, *defaultvalue=None*)
A convenience method returning the value of a certain property for the current universe object.

Parameters

- **key** (*str*) – The key name of the property to get.
- **defaultvalue** (*Optional[Any]*) – This value is returned if the specified property does not exist.

Return type Optional[Any]

anise.framework.projects.**setcurrentuniverse**(*u*)
Sets the current universe object. If another one is currently set, it becomes active again when u is unset. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Parameters **u** (anise.framework.projects.Universe) –

Return type None

anise.framework.projects.**setvalue**(*key*, *value*)
A convenience method setting the value of a certain property for the current universe object.

Parameters

- **key** (*str*) – The key name of the property to set.
- **value** (*Optional[Any]*) – The new value for the specified property.

Returns If successful.

Return type bool

anise.framework.projects.**universe = <anise.framework.projects._UniverseProxy object>**
The universe object. See *Universe*.

anise.framework.projects.**unsetcurrentuniverse**()
Unsets the current universe object. *This is part of a particular piece of internal infrastructure and is typically not required to be used directly.*

Return type None

anise.framework.report module

The Anise report facility keeps track of the steps of the execution process, the progresses and all output messages.

class anise.framework.report.**AniseOutputStream**(*originalstream*)
Bases: object

Proxy for terminal output/error channel, so we can record output messages.

Parameters **originalstream** (*TextIO*) –

write(*b*)

class anise.framework.report.**ExecutionScope**(*title*)
Bases: *anise.framework.report.ExecutionScopePart*

An execution scope is one piece of work to do in a complete task executions.

It is the parent of another execution scope (just the root scope does not have a parent) and can have child scopes. This leads to a tree structure of scope nodes. Each node can have metadata infos like a title or a progress. Those are displayed during the execution of a task (together with other *ExecutionScopePart* subclasses' instances).

Parameters **title** (*str*) – The scope title text.

class Progress
Bases: object

Progress info for an execution scope.

tojson()
Returns a simple object structure for json transfers.
Return type Dict[str, Optional[Any]]

class ProgressDeterminate(*value*)
Bases: *anise.framework.report.ExecutionScope.Progress*

Progress info for a determinate progress.

Parameters **value** (*float*) – The progress between 0.0 and 1.0.

class ProgressDoneWithUnknownSuccess
Bases: *anise.framework.report.ExecutionScope.Progress*

Progress info for an execution that is terminated but without explicit success information.

class ProgressFailed
Bases: *anise.framework.report.ExecutionScope.Progress*

Progress info for a failed execution.

class ProgressFinished
Bases: *anise.framework.report.ExecutionScope.Progress*

Progress info for a finished execution.

class ProgressIndeterminate
Bases: *anise.framework.report.ExecutionScope.Progress*

Progress info for an indeterminate progress.

appendchild(*child*)
Appends a child scope to this scope.

Parameters **child** (anise.framework.report.ExecutionScopePart) –

Return type None

newsubscope(*title*)
Adds a new sub scope to the current top scope.

Parameters **title** (*str*) – The scope title text.

Return type *anise.framework.report.ExecutionScopePart*

property progress
The current progress state of this scope.

Return type *Progress*

This property is also settable.

property rawsubparts
List of all direct subparts of this scope.

Return type List[‘ExecutionScopePart’]

property title
The title string of this scope.

Return type str

class anise.framework.report.**ExecutionScopePart**(*value*, *hints=None*)
Bases: object

Base class of *ExecutionScope* (read there for more). It also includes logging output or output from external programs, which are displayed together with real execution scopes during task execution.

Parameters

- **value** (*str*) – The content value.
- **hints** (*Optional[List[str]]*) – A list of hint symbols (typically used for log message severity).

property address
The address of this part.

Return type Tuple

getscopepartsbyidrange(*minid=None*, *maxid=None*)
Gets a list of execution scope parts by a range of IDs.

Parameters

- **minid** (*Optional[int]*) –
- **maxid** (*Optional[int]*) –

Return type List[*anise.framework.report.ExecutionScopePart*]

property hints
The content hint.

This is typically used for storing the severity of log messages, so the presentation can differ.

Return type List[str]

property id
The ID of this part.

Return type int

property value
The content value of this execution scope.

Return type str

class anise.framework.report.**RootExecutionScope**
Bases: *anise.framework.report.ExecutionScope*

The root execution scope.

Parameters **title** – The scope title text.

anise.framework.report.**_logmessage**(*text*, *severity*, *terminalonly*, *tofile*, *textescaper*)
A message logger implementation that also records the logging output to the report.

Parameters

- **text** (*str*) – The text to log.
- **severity** (*int*) – The severity. One of *anise.utils.logging.Severity*.
- **terminalonly** (*bool*) – If to log only to terminal/file.
- **tofile** (*TextIO*) – Log to this file instead of terminal.
- **textescaper** (*Callable[[str], str]*) – A function that escapes text in a custom way before logging.

Return type None

anise.framework.report.**install_proxies**()
Installs stdout/stderr proxies, so we can record all output made.

Return type None

anise.framework.report.**topscope**(*i=0*)
Returns a scope from the top of the stack (without removing it).

Parameters **i** (*int*) – Which scope to return (counting from top).

Return type *anise.framework.report.ExecutionScope*

Module contents

The infrastructure parts of Anise.

The infrastructure executes Python code in a `_projectdesc` file and makes some syntactical tricks leading to shorter (and better readable) files. This tricks include the direct access to the universe object and the automatic imports it makes before executing `_projectdesc`.

In most places, the Anise infrastructure tries to be as thin a layer above Python as possible instead of reinventing the wheel again and again. Although on some places Anise actually has to do some more own stuff in order to bring some functionality.

Read about the individual infrastructure parts in the subpackages here.

The starting point is *anise.framework.engine*, which is the active part responsible for executing tasks.

anise.utils package

Submodules

anise.utils.basic module

Basic helpers for filesystem and other operating system related stuff.

class anise.utils.basic.**ChDir**(*path*)
Bases: object

Temporarily changes the current working directory in a scope of the `with` keyword.

Parameters **path** (*str*) – The path to the new working directory.

class anise.utils.basic.**Mount**(*src*, *dst*, *options=None*, *mounter=None*, *unmounter=None*, *needs-root=True*)
Bases: object

Mounts a volume. Can either be used automatically with the `with` keyword or manually.

Parameters

- **src** (*str*) – Mount source.
- **dst** (*str*) – Mount destination.
- **options** (*Optional[List[str]]*) – Optional additional mount options.
- **mounter** (*Optional[List[str]]*) – Optional string list overriding the default mount call.

- **unmounter** (*Optional[List[str]]*) – Optional string list overriding the default unmount call.
- **needsroot** (*bool*) – If this mount action will need root permissions.

mount()
Manually mount the volume.

Return type None

unmount()
Manually unmount the volume.

Return type None

anise.utils.basic.**call**(**cmdline*, *shell=False*, *decode=True*, *raise_on_errors=None*, *split_output_channels=False*)
Executes a command line.

Parameters

- **cmdline** (*Union[str, List[str]]*) – The command line as list of arguments.
- **shell** (*bool*) – If the command shall be interpreted by a shell (only the first part of cmdline will apply!).
- **decode** (*bool*) – If the result is to be decoded to a string.
- **raise_on_errors** (*Optional[str]*) – If errors shall raise Exceptions (also specifies the exception text in this case).
- **split_output_channels** (*bool*) – If to separately record output and error stream.

Returns Tuple of returncode, program output.

Return type Tuple[int, Union[AnyStr, Tuple]]

anise.utils.basic.**callanise**(**args*)
Calls the Anise process (in a platform independent way).

Parameters **args** (*str*) – The command line arguments to pass.

Return type None

anise.utils.basic.**execwithretry**(*fct*, **args*, *maxtries=10*, *delaysecs=0.2*, ***kwargs*)
Executes a function with a retry mechanism (executing it again when an exception was raised).

Parameters

- **fct** (*Callable*) – The function to execute.
- **args** (*Optional[Any]*) – Arguments for the function call.
- **maxtries** (*int*) – Maximum try count.
- **delaysecs** (*float*) – Delay (in seconds) betweens tries.
- **kwargs** (*Optional[Any]*) – Keyword arguments for the function call.

Return type Optional[Any]

anise.utils.basic.**getdriveletters**()
Returns a list of drive letters that exist on this system (only win32).

Return type List[str]

anise.utils.basic.**maketarball**(*sourcepath*, *tarpath*)
Makes a tar archive from a given directory structure.

Parameters

- **sourcepath** – The root source directory of the new tarball (becomes a directory in the archive as typical for tarballs).
- **tarpath** (*str*) – The destination path for the tar archive.

Return type None

anise.utils.basic.**mkdirs**(*dname*)
Creates a directory path recursively if it does not yet exist.

Parameters **dname** (*str*) – The directory path.

Return type None

anise.utils.basic.**openfile**(*filepath*)
Opens a file path in the operating system's default way.

Parameters **filepath** (*str*) – A file path.

Return type None

anise.utils.basic.**openurl**(*url*)
Opens a url (or file path) in the operating system's default way.

Parameters **url** (*str*) – A url or file path.

Return type None

anise.utils.basic.**readfromfile**(*filename*, *onestring=True*, *notexist_default=<object object>*, *binary=False*)
Reads the content from a file.

Parameters

- **filename** (*str*) – The file path.
- **onestring** (*bool*) – If the content is to be read in one single string (instead of a list of lines).
- **notexist_default** (*Any*) – The default content when the file does not exist.
- **binary** (*bool*) – If this is binary content instead of text.

Return type AnyStr

anise.utils.basic.**verify_tool_installed**(*toolname*)
Checks if a tool is installed and raises an exception if not.

Parameters **toolname** (*str*) – The tool name as to execute on command line.

Return type None

anise.utils.basic.**writetofile**(*filename*, *content*)
Writes some content to a file.

Parameters

- **filename** (*str*) – The file path.
- **content** (*AnyStr*) – The content to write.

Return type None

anise.utils.data module

Helpers for creation and handling of some data structures.

anise.utils.data.**constructobject**(***d*)
Creates an object with properties from the given keyword arguments.

Parameters **d** (*Optional[Any]*) –

Return type object

anise.utils.data.**createobjectstructure**(*root*, *newnames*, *objclass=None*)
Creates a chain of new objects, so you can access property chains like `foo.bar.baz.bam` on the root object.

Parameters

- **root** (*Any*) – The object to populate with the new structure.
- **newnames** (*str*) – The complete property chain.
- **objclass** (*Optional[Type]*) – Which object class to use for the cascaded new members.

Returns Tuple of the deepest object in the cascade (corresponding to the last part of the chain) and its parent.

Return type Tuple[object, object]

anise.utils.logging module

Logging.

class anise.utils.logging.**Severity**
Bases: `object`

Enumeration of log message severities.

Debug = 0
Debug logging severity.

Error = 300
Error logging severity.

Info = 100
Info logging severity.

Warning = 200
Warning logging severity.

anise.utils.logging.**_logmessage_default**(*text*, *severity*, *terminalonly*, *tofile*, *textescaper*)
This is the fallback lowlevel logging implementation. Typically it gets overridden by a different one in early initialization steps.

Return type None

anise.utils.logging.**logdebug**(*text*, *printcaller=True*, *skiplevels=1*, *terminalonly=False*)
Logs a text message with debug severity.

Parameters

- **text** (*str*) – The log message.
- **printcaller** (*bool*) – Also logs the caller name.

- **skiplevels** (*int*) – If given, it controls the position on the call stack that is used for logging the caller name.
- **terminalonly** (*bool*) – If this log message shall just go to the terminal and not to the report, the web interface, etc. Only used by the infrastructure.

Return type None

anise.utils.logging.**logerror**(*text*, *printcaller=True*, *skiplevels=1*, *terminalonly=False*)
Logs a text message with error severity.

Parameters

- **text** (*str*) – The log message.
- **printcaller** (*bool*) – Also logs the caller name.
- **skiplevels** (*int*) – If given, it controls the position on the call stack that is used for logging the caller name.
- **terminalonly** (*bool*) – If this log message shall just go to the terminal and not to the report, the web interface, etc. Only used by the infrastructure.

Return type None

anise.utils.logging.**logexception**(*exception*, *prolog='Error: '*)
Logs an exception.

Parameters

- **exception** (*BaseException*) – The occurred exception. This only works from within the `except`-handler of this exception.
- **prolog** (*str*) – This is prepended to the exception text in the output.

Return type None

anise.utils.logging.**loginfo**(*text*, *printcaller=True*, *skiplevels=1*, *terminalonly=False*)
Logs a text message with info severity.

Parameters

- **text** (*str*) – The log message.
- **printcaller** (*bool*) – Also logs the caller name.
- **skiplevels** (*int*) – If given, it controls the position on the call stack that is used for logging the caller name.
- **terminalonly** (*bool*) – If this log message shall just go to the terminal and not to the report, the web interface, etc. Only used by the infrastructure.

Return type None

anise.utils.logging.**logmessage**(*text*, *severity*, *printcaller=True*, *skiplevels=1*, *terminalonly=False*)
Logs a text message.

Parameters

- **text** (*str*) – The log message.
- **severity** (*int*) – The message severity. One of the values from *Severity*.
- **printcaller** (*bool*) – Also logs the caller name.
- **skiplevels** (*int*) – If given, it controls the position on the call stack that is used for logging the caller name.

- **terminalonly** (*bool*) – If this log message shall just go to the terminal and not to the report, the web interface, etc. Only used by the infrastructure.

Return type None

anise.utils.logging.**logwarning**(*text*, *printcaller=True*, *skiplevels=1*, *terminalonly=False*)
Logs a text message with warning severity.

Parameters

- **text** (*str*) – The log message.
- **printcaller** (*bool*) – Also logs the caller name.
- **skiplevels** (*int*) – If given, it controls the position on the call stack that is used for logging the caller name.
- **terminalonly** (*bool*) – If this log message shall just go to the terminal and not to the report, the web interface, etc. Only used by the infrastructure.

Return type None

anise.utils.logging.**setlowlevellogger**(*fct*)
Sets the lowlevel logging implementation.

Parameters **fct** (*Callable[[str, int, bool, TextIO, Callable[[str], str]], None]*) – The logger function.

Return type None

anise.utils.ssh module

Utilities for ssh usage.

class anise.utils.ssh.**Mount**(*src*, *dst*, *options=None*, *port=22*, *identityfile=None*)
Bases: *anise.utils.basic.Mount*

Mounts a volume via ssh.

Parameters

- **src** (*str*) – The remote mount source (an ssh address like `'user@machine'`).
- **dst** (*str*) – The local mount destination.
- **options** (*Optional[List[str]]*) – Optional additional mount options.
- **port** (*int*) – Optional tcp port.
- **identityfile** (*Optional[str]*) – Optional identity file for authentication.

anise.utils.ssh.**call**(*server*, *command*, *options=None*, *port=22*, *identityfile=None*)
Executes a command via ssh.

Parameters

- **server** (*str*) – The ssh server hostname.
- **command** (*str*) – The command to execute.
- **options** (*Optional[List[str]]*) – Additional ssh options.
- **port** (*int*) – Port number.
- **identityfile** (*Optional[str]*) – Optional identity file for authentication.

Return type str

anise.utils.ssh.**copy**(*src*, *dst*, *options=None*, *port=22*, *identityfile=None*)

Copies a local directory to a remote place via ssh.

Parameters

- **src** (*str*) – The source.
- **dst** (*str*) – The destination.
- **options** (*Optional[List[str]]*) – Optional additional mount options.
- **port** (*int*) – Optional tcp port.
- **identityfile** (*Optional[str]*) – Optional identity file for authentication.

Return type None

anise.utils.threading module

Threading.

class anise.utils.threading.**Thread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs=None*, *, *daemon=None*)

Bases: threading.Thread

A thread implementation that is perfectly compatible with the internal Anise bookkeeping.

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

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

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

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

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

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

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

start()

Start the thread's activity.

It must be called at most once per thread object. It arranges for the object's run() method to be invoked in a separate thread of control.

This method will raise a RuntimeError if called more than once on the same thread object.

Module contents

Utility functions from several categories which make life easier. All stuff here is not tightly bound to Anise infrastructure but can easily be understood separately. Most stuff here is used by deeper parts of Anise.

14.1.2 Module contents

The Anise program.

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

D

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W