simplesat Documentation

Release 0.2.0

Enthought, Inc.

Jul 20, 2017

Contents

A library for SAT-based dependency handling. The `simplesat` library provides facilities for describing packages and their relationships, producing a set of CNF clauses, and producing a solution for the clauses, according to the notion of a "policy," which determines the order in which packages are tried.

Contents:

CHAPTER 1

Getting Started

Installation

To install the python package, do as follows:

```
git clone --recursive https://github.com/enthought/sat-solver
cd sat-solver
pip install -e .
```

Usage from the CLI

To try things out from the CLI, you need to write a scenario file (YAML format), see `simplesat/tests/simple_numpy.yaml` for a simple example.

To print the rules:

```
python scripts/print_rules.py simplesat/tests/simple_numpy.yaml
```

To print the operations:

```
python scripts/solve.py simplesat/tests/simple_numpy.yaml
```

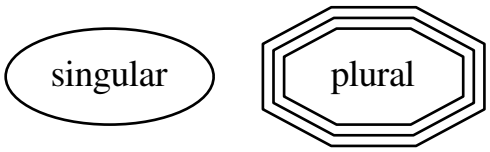
CHAPTER 2

Architecture

Simplesat's API is modeled after the Composer library from PHP.

For a good overview of the public API of the entire system, you should look at the *Scenario*, upon which all of our functional testing is based. The Scenario class shows how to build *PackageMetadata* instances from strings, use them to create a *Repository*, *Pool* and *Request* and pass them to a *DependencySolver* for resolution.

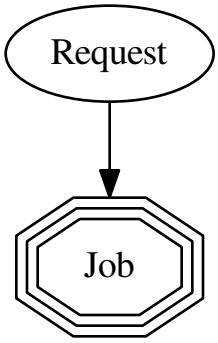
That said, pictures help. Let's look at how data flows through the object hierarchy. We'll use the following symbols to indicate singular objects and plural collections of objects.



Requests

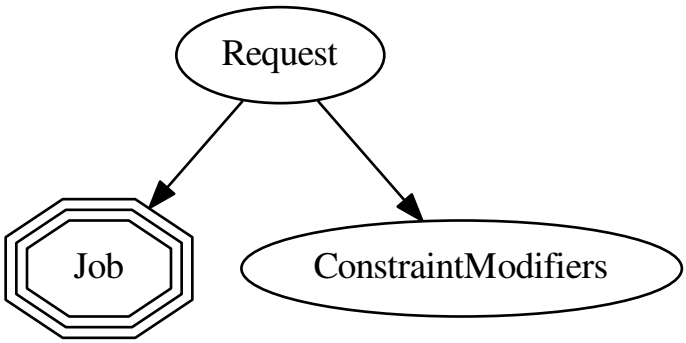
The purpose of the `simplesat` library as a whole is to produce a valid assignment of package states (installed or not) that satisfy some particular set of constraints. This is expressed as a `Transaction` that is to be applied to the "installed" repository. The *Request* object is our vehicle for communicating these constraints to the solver.

At its core, a *Request* is a collection of actions such as "install" and `Requirement` objects describing ranges, such as `numpy >= 1.8.1`, which together form `Job` rules. The *Request* can have any number of such jobs, all of which must be satisfiable simultaneously. If conflicting jobs are given, then the solver will fail with a `simplesat.errors.SatisifiabilityError`.



Constraint Modifiers

Additionally, one may attach `ConstraintModifiers` to the *`Request`*. These are used to modify the constraints of packages during the search for a solution.

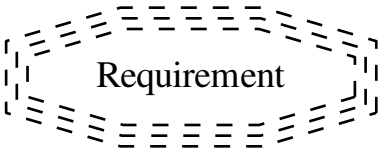


These constraints are not applied to the jobs themselves, only to their dependencies. For example, if one were to create an install job for `pandas < 0.17`, while at the same time specifying a constraint modifier that allows any version of pandas to satisfy any constraint, the modifier should *not* be applied. We assume that any constraint directly associated with a `Job` is explicit and intentional.

Note that *`Request`* objects do not carry any direct information about packages. They merely describes constraints that any solution of packages states must satisfy.

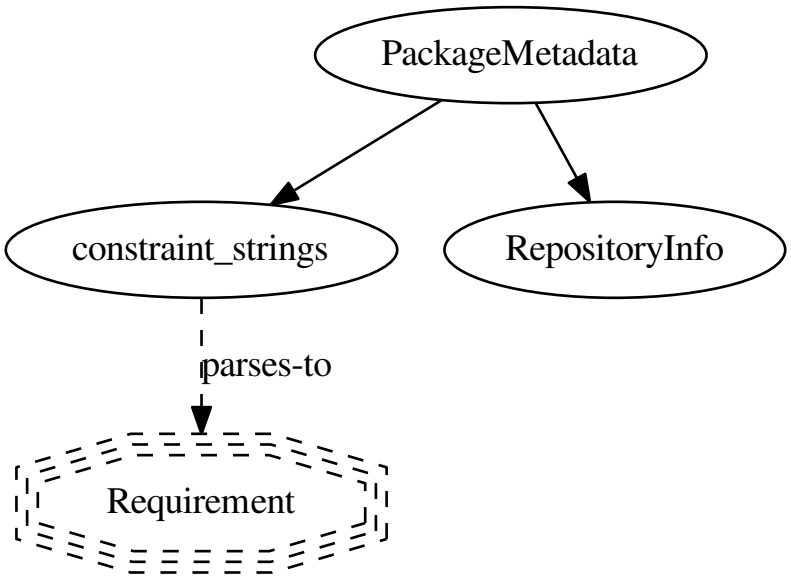
Package Hierarchy

A `RepositoryPackageMetadata` is the basic object describing a software package that we might want to install. It has attached to it a collection of strings describing the packages upon which it depends, referred to as `installed_requires`, as those with which it `conflicts`. To avoid paying the cost of parsing our entire universe of packages for every request, these attached constraints are not parsed into `Requirement` objects until they are passed to the *`Pool`* later on. We'll show them like this from now on to make it clear that they don't exist until needed.



RepositoryInfo

A package object also has a `RepositoryInfo` attached to it, which is not currently used for solving, but provides information about the source of the package.

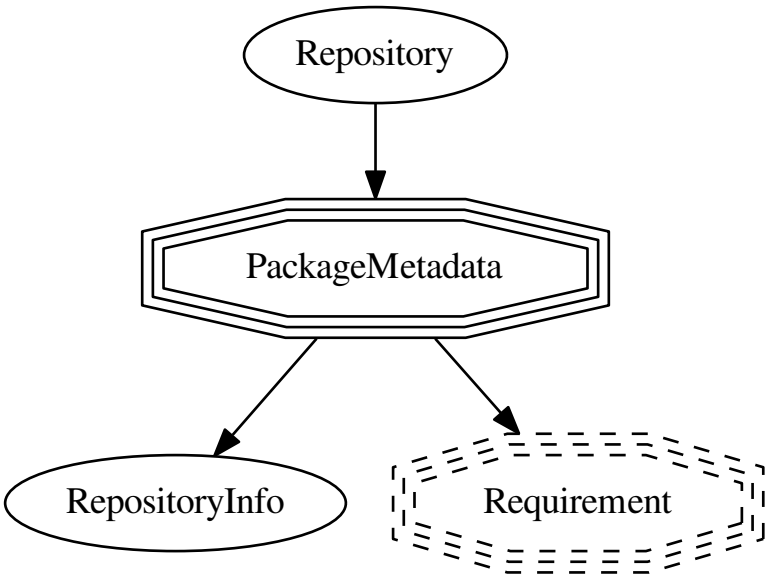


For testing or interactive exploration, these can be created via the `PrettyPackageStringParser`:

```
from okonomiyaki.versions import EnpkgVersion
ps = PrettyPackageStringParser(EnpkgVersion.from_string)
package = ps.parse_to_package(
    'foo 1.8.2; install_requires (bar ^= 3.0.0, baz == 1.2.3-4)
    '; conflicts (quux ^= 2.1.2)')
```

Repository

A `Repository` is made out of many of these such packages.

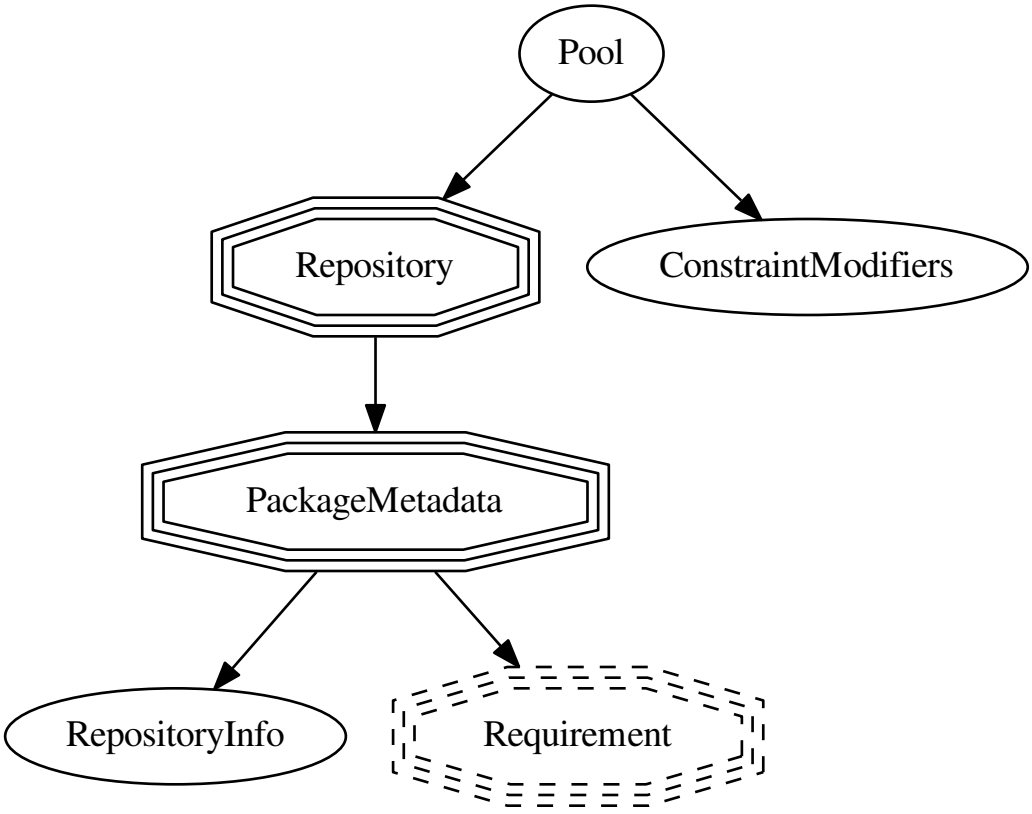


and can be created from them like so:

```
repo = Repository(iter_of_packages)
repo.add_package(additional_package)
```

Pool

The `Repository` class does not support any kind of complicated querying. When it is time to identify packages according to constraints such as `numpy >= 1.7.2`, we must create a *`Pool`*. A *`Pool`* contains many such `Repository` objects and exposes an API to query them for packages.

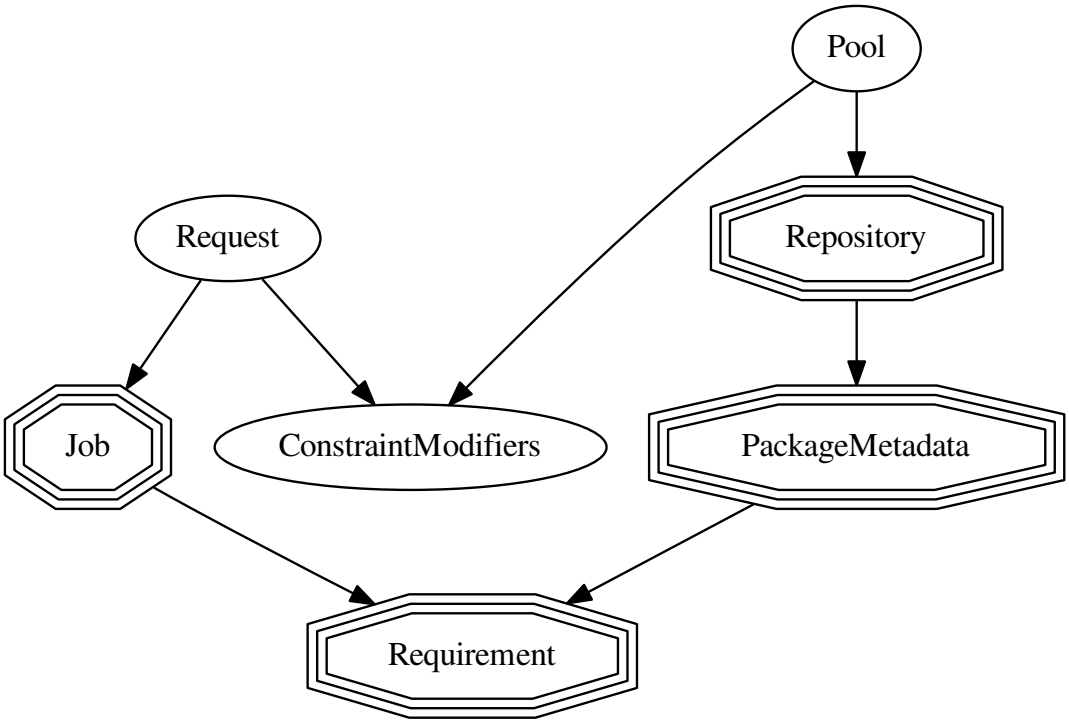


The `ConstraintModifiers<simplesat.constraints.ConstraintModifiers` object is also attached to the *`Pool`*. It is used to modify incoming `Requirement` objects before using them to query for matching packages. This happens implicitly in the *`Pool.what_provides()`* method. The result of such modification can be inspected directly by calling *`Pool.modify_requirement()`*, which is used internally. The *`Pool`* is used like so:

```
repository = Repository(packages)
requirement = InstallRequirement._from_string("numpy ^= 1.8.1")
pool = Pool([repository], modifiers=ConstraintModifiers())
package_metadata_instances = pool.what_provides(requirement)

# These are not modified. Used for handling e.g. jobs.
more_instances = pool.what_provides(requirement, modify=False)
```

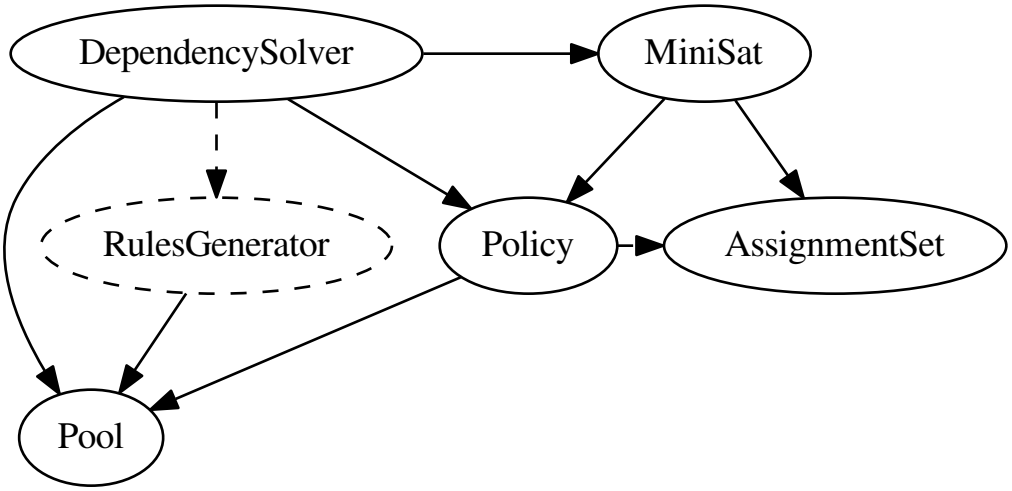
We now have a complete picture describing the organization of package data.



MiniSAT Engine

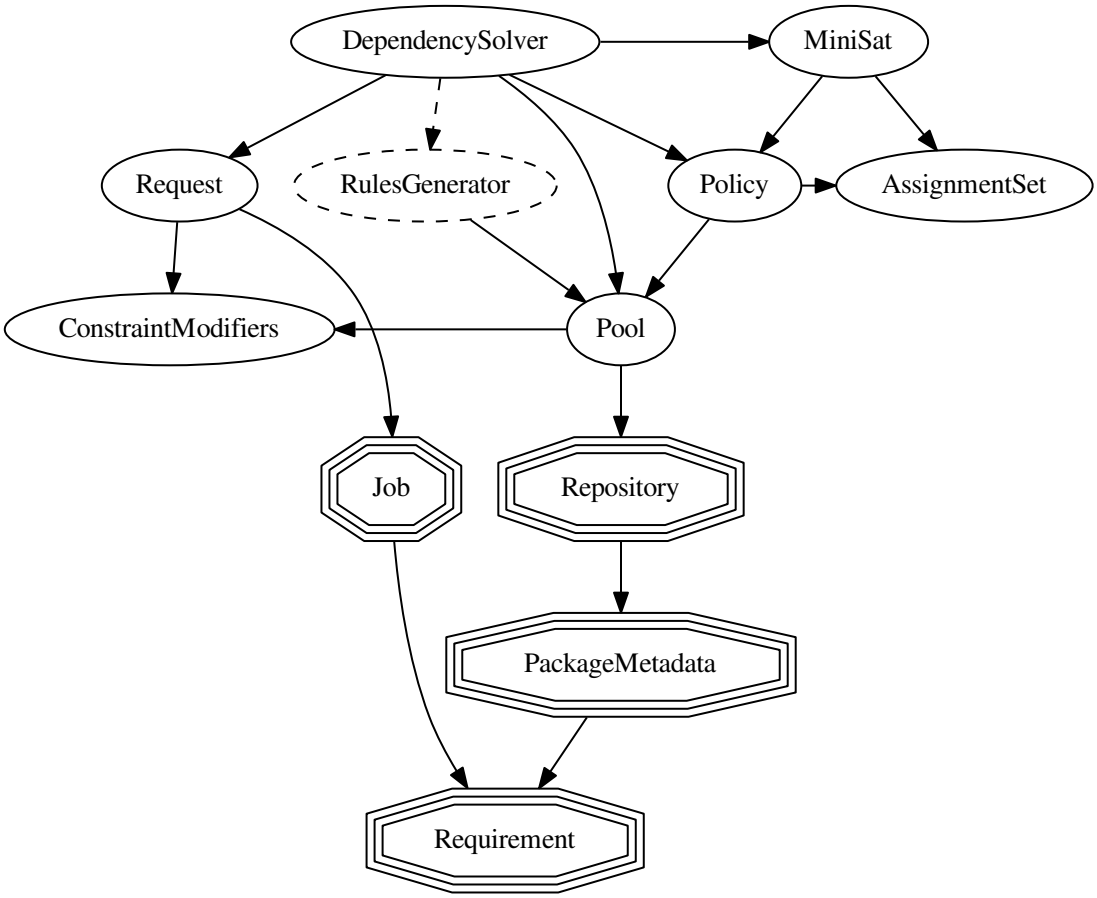
When it comes time to process a *Request* and find a suitable set of package assignments, we must create a `DependencySolver`. This in turn will initialize four pieces that together work to resolve the request.

- The first is the *Pool*, which we've already seen.
- The *Pool* is passed along with the *Request* to a `RulesGenerator`, which generates an appropriate set of conjunctive normal form (CNF) clauses describing the problem.
- Next is the `Policy`, which determines the order in which new package assignments are tried. The simplest possible `Policy` could suggest unassigned packages in arbitrary order, but typically we will want to do something more sophisticated.
- Lastly, we create a `MiniSat` object and feed it the rules from the `RulesGenerator` and the `Policy` to help make suggestions when it gets stuck. This is the core SAT solving engine. It is responsible for exploring the search space and returning an `AssignmentSet` that satisfies the clauses.



As the `MiniSat` explores the search space, it will update the `AssignmentSet`. When it reaches a point where it must make a guess to continue it will ask the `Policy` for a new package to try. The `Policy` looks at the `AssignmentSet` and *`Pool`* to choose a suitable candidate. This continues until either the `MiniSat` finds a solution or determines that the problem is unsatisifiable.

The entire system looks like this.


DependencySolver
MiniSat
Request
RulesGenerator
Policy
AssignmentSet
ConstraintModifiers
Pool
Job
Repository
PackageMetadata
Requirement

CHAPTER 3

SAT Solving

Encoding Relationships as Clauses

The `RulesGenerator` is responsible for rooting out all of the relevant packages for this problem and creating `PackageRule` objects describing their relationships. An example might be translating a requirement such as `numpy` into `(+numpy-1.8.1 | +numpy-1.8.2 | +numpy-1.8.3)`, where the `+` operator indicates that the package should be installed and `|` is logical OR. In prose one might read this as "Must install one of `numpy-1.8.1`, `numpy-1.8.2`, or `numpy-1.8.3`."

To build up a total set of rules, we start at each of our `Job` rules and cycle recursively though package metadata, adding new rules as we discover new packages. This is done by running each of our requirements through the `Pool` and asking it which packages match.

Constraint Modifiers

The key notion here is that `Pool.what_provides()` gives us a very flexible abstraction for package querying. When we want to manipulate the way package dependencies are handled, we don't need to modify the packages themselves, it is enough to modify the querying function such that it *responds* in the way that we want.

We attach the `ConstraintModifiers` to the `Pool` itself, and at query time, the `Pool` may *transform* the `Requirement` as necessary. The current implementation results in the transformations below. The original requirement is on the far left, with the result of each type of transformation to the right of it. `*` is a wild-card that matches any version.

Original	Allow newer	Allow older	Allow any
*	*	*	*
> 1.1.1-1	> 1.1.1-1	*	*
>= 1.1.1-1	>= 1.1.1-1	*	*
< 1.1.1-1	*	< 1.1.1-1	*
<= 1.1.1-1	*	<= 1.1.1-1	*
^= 1.1.1	>= 1.1.1	<= 1.1.1-*	*
== 1.1.1-1	>= 1.1.1-1	<= 1.1.1-1	*
!= 1.1.1-1	!= 1.1.1-1	!= 1.1.1-1	!= 1.1.1-1

Requirements

There are currently three different `Requirement` classes: *`Requirement`*, *`InstallRequirement`* and *`ConflictRequirement`*. They have no internal differences, but this split allows us to reliably track the origin of a requirement via its type and avoid using it in an inappropriate context.

We care about the difference between a requirement created from `package.install_requires` vs one created from `package.conflicts` vs one created from parsing a pretty string into a Job. It only makes sense for `modifiers` to apply to constraints created from `install_requires`; we don't want to modify a constraint that the user explicitly gave us and we don't know what it means to `allow_newer` for a conflicts constraint at all. By creating an `InstallRequirement` only when reading `package.install_requires` and then explicitly checking for that class at the only point where we might modify it, we can prevent ourselves from modifying the wrong kind of requirement. The same goes for `ConflictRequirement`, although there is currently no use case differentiating it from a plain `Requirement`.

Top-level ("Job") requirements are created by external code because the only way to communicate a requirement to the system is via a `Requirement` object attached to a `Request`. All others are created as needed by the `RulesGenerator` while it puts together rules based on package metadata.

So user-given requirements like `install foo^=1.0` or `update bar` are turned into normal `Requirement` objects because they should *not* be modified. **Getting this wrong can lead to "install inconsistent sets of packages" bugs.**

When to use each requirement class

`InstallRequirement` Requirements derived from `package.install_requires` metadata. For example:

```
for constraints in package.install_requires:
  req = InstallRequirement.from_constraints(constraints)
```

> **Note:** Currently, this is the only type of requirement that can be passed to `modify_requirement`.

`ConflictRequirement` Requirements derived from `package.conflicts` metadata. For example:

```
for constraints in package.conflicts:
  req = ConflictRequirement.from_constraints(constraints)
```

`Requirement`, All other requirements, including those coming directly from a user via a *`simplesat.request.Request`*.

CHAPTER 4

Comparing with PHP's Composer library

First, clone composer's somewhere on your machine:

```
git clone https://github.com/composer/composer
```

Then, use the `scripts/scenario_to_php.py` script to write a PHP file that will print the composer's solution for a given scenario:

```
python scripts/scenario_to_php.py \
    --composer-root <path to composer github checkout> \
    simplesat/tests/simple_numpy.yaml \
    scripts/print_operations.php.in

python scripts/scenario_to_php.py \
    --composer-root <path to composer github checkout> \
    simplesat/tests/simple_numpy.yaml \
    scripts/print_rules.php.in
```

This will create `scripts/print_operations.php` and `scripts/print_rules.php` scripts you can simply execute with `php`:

```
php scripts/print_rules.php
php scripts/print_operations.php
```

CHAPTER 5

API Reference

This covers all of the interfaces in Simplesat. For an overview of how these pieces fit together, take a look at *Architecture*.

Main Interface

For testing of the validity of a set of requirements, typical usage might be the following:

```
installed_repository = Repository([package1, package2])
remote_repository = Repository([package1, package3, package4])

R = Requirment.from_string
requirements = [R('package1 > 1.2.3'), R('package4 < 2.8')]
repositories = [installed_repository, remote_repository]

if packages_are_consistent(installed_repository):
    print("Installed packages are OK!")

if requirements_are_satisfiable(repositories, requirements):
    print("The requirements are mutually compatible.")
else:
    print("The requirements conflict.")

if requirements_are_complete(repositories, requirements):
    print("These requirements include all necessary dependencies.")
else:
    print("The requirements are incomplete. Dependencies are missing.")
```

simplesat.dependency_solver.**packages_are_consistent**(*packages*, *modifiers=None*)
Return True if all packages can be installed together.

> **Note:** This will return *False* if more than one version of a package is present because we only permit one at a

time.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages to check for consistency.
- **modifiers** (*ConstraintModifiers, optional*) – If not None, modify requirements before resolving packages.

Returns True if every package in *packages* can be installed simultaneously, otherwise False.

Return type bool

simplesat.dependency_solver.**requirements_are_complete**(*packages*, *requirements*, *modifiers=None*)

Return True if *requirements* includes all required transitive dependencies. I.e. it will report whether all the packages that are needed are explicitly required.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages available to draw from when satisfying requirements.
- **requirements** (*iterable of Requirement*) – The requirements used to identify relevent packages.
- **modifiers** (*ConstraintModifiers, optional*) – If not None, modify requirements before resolving packages.

Returns True if the requirements specify all necessary packages.

Return type bool

simplesat.dependency_solver.**requirements_are_satisfiable**(*packages*, *requirements*, *modifiers=None*)

Determine if the *requirements* can be satisfied together.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages available to draw from when satisfying requirements.
- **requirements** (*list of Requirement*) – The requirements used to identify relevent packages.
- **modifiers** (*ConstraintModifiers, optional*) – If not None, modify requirements before resolving packages.

Returns Return True if the *requirements* can be satisfied by the *packages*.

Return type bool

simplesat.dependency_solver.**satisfy_requirements**(*packages*, *requirements*, *modifiers=None*)

Find a collection of packages that satisfy the requirements.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages available to draw from when satisfying requirements.
- **requirements** (*list of Requirement*) – The requirements used to identify relevent packages.

- **modifiers** (*ConstraintModifiers, optional*) – If not None, modify requirements before resolving packages.

Returns Return a tuple of packages that together satisfy all of the *requirements*. The packages are in topological order.

Return type tuple of PackageMetadata

Raises SatisfiabilityError – If the *requirements* cannot be satisfied using the *packages*.

simplesat.dependency_solver.**simplify_requirements**(*packages*, *requirements*)
Return a reduced, but equivalent set of requirements.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages available to draw from when satisfying requirements.
- **requirements** (*list of Requirement*) – The requirements used to identify relevent packages.

Returns The reduced requirements.

Return type tuple of Requirement

class simplesat.constraints.requirement.**ConflictRequirement**(*name*, *constraints=None*)
A Requirement that describes packages which must not be installed.

class simplesat.constraints.requirement.**InstallRequirement**(*name*, *constraints=None*)
A Requirement that describes packages to be installed.

class simplesat.constraints.requirement.**Requirement**(*name*, *constraints=None*)
Requirements instances represent a 'package requirement', that is a package + version constraints.

Parameters

- **name** (*str*) – PackageInfo name
- **specs** (*seq*) – Sequence of constraints

classmethod from_constraints(*constraint_tuple*)
Return a Requirement object from a PackageMetadata constraint tuple.

Parameters **constraints** – A 2-tuple of constraints where the first element is the distribution name, and the second is a tuple of tuple of string, representing a disjuntion of conjunctions of version ranges, e.g. `('nose', (('< 1.4', '>= 1.3'),))`.

Returns A Requirement that matches the given constraints.

Return type *Requirement*

Raises

- InvalidConstraint – If there is more than one conjunction. In less formal terms, we do not currently support the OR operator.
- InvalidConstraint – If the constraint tuple has the wrong shape.

classmethod from_package_string(*package_string*, *version_factory=<bound method type.from_string of <class 'okonomiyaki.versions.enpkg.EnpkgVersion'>>*)
Creates a requirement from a package full version.

Parameters

- **package_string** (*str*) – The package string, e.g. 'numpy-1.8.1-1'
- **version_factory** (*callable, optional*) – A function from version strings to version objects.

Returns A requirement matching the exact package and version in *package_string*.

Return type *Requirement*

has_any_version_constraint
True if there is any version constraint.

matches(*version_candidate*)
Returns True if the given version matches this set of requirements, False otherwise.

Parameters **version_candidate** (*obj*) – A valid version object (must match the version factory of the requirement instance).

to_constraints()
Return a constraint tuple as described by *from_constraints()*.

simplesat.constraints.requirement.**parse_package_full_name**(*full_name*)
Parse a package full name (e.g. 'numpy-1.6.0-1') into a (name, version_string) pair.

Functional classes

Internally, these make use of the DependencySolver. To use it yourself, you'll need to create some Packages, populate at least one Repository with them, add *that* to a Pool and give all of that to the constructor. Then you can make some *Requirements* that describe what you'd like to do, add them to a Request and pass it to solve.

class simplesat.dependency_solver.**DependencySolver**(*pool*, *remote_repositories*, *installed_repository*, *use_pruning=True*, *strict=False*)
Top-level class for resolving a package management scenario.

The solver is configured at construction time with packages repositories and a Policy and exposes an API for computing a Transaction that describes what to do.

Parameters

- **pool** (Pool) – Pool against which to resolve Requirements.
- **remote_repositories** (*list of Repository*) – Repositories containing package available for installation.
- **installed_repository** (Repository) – Repository containing the packages which are currently installed.
- **use_pruning** (*bool, optional*) – When True, attempt to prune package operations that are not strictly necessary for meeting the requirements. Without this, packages whose assignments have changed as an artefact of the search process, but which are not needed for the solution will be modified.

  A typical example might be the installation of a dependency for a package that was proposed but later backtracked away.
- **strict** (*bool, optional*) – When true, behave more harshly when dealing with broken packages. INFO level log messages become WARNINGs and missing dependencies become errors rather than causing the package to be ignored.

```
>>> from simplesat.constraints.package_parser import \
...     pretty_string_to_package as P
>>> numpy1921 = P('numpy 1.9.2-1; depends (MKL 10.2-1)')
>>> mkl = P('MKL 10.3-1')
>>> installed_repository = Repository([mkl])
>>> remote_repository = Repository([mkl, numpy1921])
>>> request = Request()
>>> request.install(Requirement.from_string('numpy >= 1.9'))
>>> request.allow_newer('MKL')
>>> pool = Pool([installed_repo] + remote_repos)
>>> pool.modifiers = request.modifiers
>>> solver = DependencySolver(pool, remote_repos, installed_repo)
>>> transaction = solver.solve(request)
```

solve(*request*)

Given a request return a Transaction that would satisfy it.

Parameters **request** (Request) – The request that should be satisifed.

Returns The operations to apply to resolve the *request*.

Return type Transaction

Raises SatisfiabilityError – If no resolution is found.

solve_with_hint(*request*)

Given a request return a Transaction that would satisfy it.

If the solver cannot find a solution, it will raise a SatisfiabilityErrorWithHint exception, that contains enough information to give a human-readable error message.

Parameters **request** (Request) – The request that should be satisifed.

Returns The operations to apply to resolve the *request*.

Return type Transaction

Raises SatisfiabilityErrorWithHint – If no resolution is found.

class simplesat.request.**Request**(*modifiers=NOTHING*, *jobs=NOTHING*)

A proposed change to the state of the installed repository.

The Request is built up from Requirement objects and ad-hoc package constraint modifiers.

Parameters **modifiers** (*ConstraintModifiers, optional*) – The contraint modifiers are used to relax constraints when deciding on which packages meet a requirement.

```
>>> from simplesat.request import Request
>>> from simplesat.constraints import Requirement
>>> request = Request()
>>> recent_mkl = Requirement.from_string('MKL >= 11.0')
>>> request.install(recent_mkl)
>>> request.jobs
[_Job(requirement=Requirement('MKL >= 11.0-0'), kind=<JobType.install: 1>)]
>>> request.modifiers
ConstraintModifiers(allow_newer=set(), allow_any=set(), allow_older=set())
>>> request.allow_newer('MKL')
>>> request.modifiers.asdict()
{'allow_older': [], 'allow_any': ['MKL'], 'allow_newer': []}
```

Package Hierarchy

class simplesat.package.**PackageMetadata**(*name*, *version*, *install_requires=None*, *conflicts=None*, *provides=None*)

PackageMetadata represents an immutable, versioned Python distribution and its relationship with other packages.

class simplesat.repository.**Repository**(*packages=None*)

A Repository is a set of packages that knows about which package it contains.

It also supports the iterator protocol. Iteration is guaranteed to be deterministic and independent of the order in which packages have been added.

Parameters **packages** (*list of PackageMetadata*) – The packages available in this repository.

```
>>> from simplesat.constraints.package_parser import \
...     pretty_string_to_package as P
>>> mkl = P('MKL 10.3-1')
>>> numpy1921 = P('numpy 1.9.2-1; depends (MKL)')
>>> numpy1922 = P('numpy 1.9.2-2; depends (MKL, libgfortran)')
>>> repository = Repository([mkl, numpy1922])
>>> repository.add_package(numpy1921)
>>> assert list(repository) == some_pkgs + [another_one]
>>> numpies = repository.find_packages['numpy']
>>> assert numpies == [numpy1921, numpy1922]
```

add_package(*package_metadata*)

Add the given package to this repository.

Parameters **package** (PackageMetadata) – The package metadata to add. May be a subclass of PackageMetadata.

Note: If the same package is added multiple times to a repository, every copy will be available when calling find_package or when iterating.

---

find_package(*name*, *version*)

Search for the first match of a package with the given name and version.

Parameters

- **name** (*str*) – The package name to look for.
- **version** (*EnpkgVersion*) – The version to look for.

Returns **package** – The corresponding metadata.

Return type *PackageMetadata*

find_packages(*name*)

Returns an iterable of package metadata with the given name, sorted from lowest to highest version.

Parameters **name** (*str*) – The package's name

Returns **packages** – Iterable of PackageMetadata instances (order is from lower to higher version)

Return type iterable

update(*iterable*)
Add the packages from the given iterable into this repository.

class simplesat.pool.**Pool**(*repositories=None*, *modifiers=None*)
A pool of repositories.

The main feature of a pool is to search for every package matching a given requirement.

Parameters

- **repositories** (*list of Repository, optional*) – The repositories to query for packages.
- **modifiers** (*ConstraintModifiers, optional*) – If given, modify the requirements prior to querying.

add_repository(*repository*)
Add the repository to this pool.

Parameters **repository** (Repository) – The repository to add

id_to_package(*package_id*)
Returns the package of the given 'package id'.

id_to_string(*package_id*)
Convert a package id to a nice string representation.

iter_package_ids()
Iterate over all package ids.

iter_packages()
Iterate over all PackageMetadata objects.

modify_requirement(*requirement*)
Return requirement modified by the pool's ConstraintModifiers.

package_id(*package*)
Returns the 'package id' of the given package.

what_provides(*requirement*, *use_modifiers=True*)
Computes the list of packages fulfilling the given requirement.

Parameters

- **requirement** (Requirement) – The requirement to match candidates against.
- **use_modifiers** (*bool*) – If True, modify the requirement according to self.modifiers.

Returns The packages satisfying *requirement*.

Return type list of PackageMetadata

Conveniences

simplesat.constraints.package_parser.**constraints_to_pretty_strings**(*constraint_tuples*)
Convert a sequence of constraint tuples as used in PackageMetadata to a list of pretty constraint strings.

Parameters **constraint_tuples** (*tuple of constraint*) – Sequence of constraint tuples, e.g. (("MKL", (("< 11", ">= 10.1"),)),)

simplesat.constraints.package_parser.**package_to_pretty_string**(*package*)
Given a PackageMetadata instance, returns a pretty string.

class simplesat.test_utils.**Scenario**(*packages*, *remote_repositories*, *installed_repository*, *request*, *decisions*, *operations*, *pretty_operations*, *failure=None*)

A high level description of a scenario that should be solved.

The Scenario class bundles together several important related pieces of data that together characterize a package management scenario. This includes a *Request*, a singular *Repository* representing packages that are currently installed and a list of Repository representing available packages.

The key feature is the ability to create one from a human-readable yaml description:

```
>>> Scenario.from_yaml(io.StringIO(u'''
...     packages:
...         - MKL 10.2-1
...         - MKL 10.3-1
...         - numpy 1.7.1-1; depends (MKL == 10.3-1)
...         - numpy 1.8.1-1; depends (MKL == 10.3-1)
...
...     request:
...         - operation: "install"
...           requirement: "numpy"
... ''')
```

simplesat.test_utils.**generate_rules_for_requirement**(*pool*, *requirement*, *installed_map=None*)

Generate CNF rules for a requirement.

Parameters

- **pool** (Pool) – A Pool of Repositories to use when fulfilling the requirement.
- **requirement** (Requirement) – The description of the package to be installed.

Returns **rules** – Package rules describing the given scenario.

Return type list

simplesat.test_utils.**parse_package_list**(*packages*)

Yield PackageMetadata instances given an sequence of pretty package strings.

Parameters **packages** (*iterator*) – An iterator of package strings (e.g. 'numpy 1.8.1-1; depends (MKL ^= 10.3)').

simplesat.dependency_solver.**requirements_from_packages**(*packages*)

Return a list of requirements, one to match each package in *packages*.

Parameters **packages** (*iterable of PackageMetadata*) – The packages for which to generate requirements.

Returns The matching requirements.

Return type tuple of Requirement

simplesat.dependency_solver.**packages_from_requirements**(*packages*, *requirements*, *modifiers=None*)

Return a new tuple that only contains packages explicitly mentioned in the requirements.

Parameters

- **packages** (*iterable of PackageMetadata*) – The packages available for inclusion in the result.
- **requirements** (*list of Requirement*) – The requirements used to identify relevant packages. All packages that satisfy any of the requirements will be included.

- **modifiers** (*ConstraintModifiers, optional*) – If not None, modify requirements before resolving packages.

Returns A tuple containing the relevant packages.

Return type Tuple of PackageMetadata

Exceptions

exception simplesat.errors.**SatisfiabilityErrorWithHint**(*unsat*, *conflicting_jobs*)
A satistibiality error class with information about minimally unsatisfiable problem.

This is used when one wants to give more human-readable error messages about conflicts and other satistiability issues.

Lower level utilities

These are used internally.

simplesat.utils.graph.**backtrack**(*end*, *start*, *visited*)
Return a tuple of nodes from *start* to *end* by recursively looking up the current node in *visited*. *visited* is a dictionary of one-way edges between nodes.

simplesat.utils.graph.**breadth_first_search**(*start*, *neighbor_func*, *targets*, *target_func=None*, *visited=None*)
Return an iterable of paths from *start* to each reachable terminal node *end*.

Parameters

- **start** (*node*) – The starting point of the search
- **neighbor_func** (*callable*) – Returns the neighbors of a node
- **targets** (*set*) – The nodes we're searching for. The search terminates when each member of *targets* has been encountered at least once, but only path is returned per target.
- **target_func** (*callable, optional*) – If given, then *target_func* is applied to node and the result is used to determine if *node* is a target via `target_func(node) in targets`.
- **visited** (*dict, optional*) – If given, it will be used to track the current path. You can use it to directly inspect the search path after calling breadth_first_search().

Yields path (*tuple of nodes*) – A path from node *start* to some node *end* such that *terminate_func(end)* is in *targets*, by following neighbors as given by *neighborfunc(node)*:

```
>>> start = 0
>>> targets = {10, 4}
>>> def target_func(node):
...     return node*2
>>> def neighbor_func(node):
...     return [node + 1]
>>> tuple(breadth_first_search(start, neighbor_func, targets, target_
↪func))
((0, 1, 2), (0, 1, 2, 3, 4, 5))
```

simplesat.utils.graph.**connected_nodes**(*node*, *neighbor_func*, *visited=None*)

Recursively build up a set of nodes connected to *node* by following neighbors as given by *neighbor_func(node)*, i.e. "flood fill."

```
>>> def neighbor_func(node):
...     return {-node, min(node+1, 5)}
>>> connected_nodes(0, neighbor_func)
{-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}
```

simplesat.utils.graph.**package_lit_dependency_graph**(*pool*, *package_lits*, *closed=True*)

Return a dict of nodes to edges, suitable for use with *toposort()*.

Parameters

- **pool** (Pool) – The pool to use when resolving package literals to packages.
- **package_lits** (*iterable of int*) – The package literals to build the dependency graph for. These can be positive or negative. The sign will be maintained.
- **closed** (*bool, optional*) – If True, only include edges to packages dependencies that are themselves in *package_lits*. No package literals that are not in *package_lits* will appear in the graph.

Returns **nodes_to_edges** – A dict of package_literals to sets of package_literals, as described in *toposort()*.

Return type dict

simplesat.utils.graph.**toposort**(*nodes_to_edges*)

Return an iterator over topologically sorted groups of nodes.

Output is a list of sets in topological order. The first set consists of items with no dependences, each subsequent set consists of items that depend upon items in the preceeding sets.

Parameters **nodes_to_edges** (*dict from node to set(node)*) – A directed graph expressed as a dictionary of edges whose keys are nodes and values are all of the nodes on which the key depends.

For example, if node 1 depends on 2, we have `{1: {2}, 2: set()}`.

Yields *set of nodes* – Each yielded set contains nodes which depend only on nodes that have already been yielded in a previous set. The first set contains the nodes with no outgoing edges.

Raises ValueError – If the graph contains cyclic dependencies.

```
>>> print '\n'.join(repr(sorted(x)) for x in toposort2({
...     2: set([11]),
...     9: set([11,8]),
...     10: set([11,3]),
...     11: set([7,5]),
...     8: set([7,3]),
...     }))
{3, 5, 7}
{8, 11}
{2, 9, 10}
```

simplesat.utils.graph.**transitive_neighbors**(*nodes_to_edges*)

Return the set of all reachable nodes for each node in the nodes_to_edges adjacency dict.

CHAPTER 6

Glossary

Repository A collection of packages from a single source. An example of repository might be the packages already installed on the system, or a set of available packages from a package server.

Pool A collection of multiple Repositories. The pool provides an interface for querying repositories for packages that satisfy Requirements.

Policy A strategy for proposing the next package to try when the solver must make an assumption.

Package In the object hierarchy, a "package" refers to a `PackageMetadata` instance. This describes a package, its dependencies "`install_requires`" and the packages with which it conflicts.

Colloquially, this refers to any kind of software distribution we might be trying to manage.

Request The operations that we wish to apply to the collection of packages. This might include installing a new package, removing a package, or upgrading all installed packages.

Requirement An object representation of a package range string, such as `numpy > 1.8.2-2` or `pip ^= 8.0.1`. These are created from dependency information attached to `PackageMetadata` and passed to the `Pool` to query the available packages.

CHAPTER 7

Bibliography

- Niklas Eén, Niklas Sörensson: An Extensible SAT-solver. SAT 2003
- Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, Sharad Malik: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. Proc. ICCAD 2001, pp. 279-285.
- Donald Knuth: The art of computer programming. Vol. 4, Pre-fascicle 6A, Par. 7.2.2.2. (Satisfiability).

On the use of SAT solvers for managing packages:

- Fosdem 2008 presentation: Using SAT for solving package dependencies. More details on the SUSE wiki.
- The 0install project.
- Chris Tucker, David Shuffelton, Ranjit Jhala, Sorin Lerner: OPIUM: Optimal Package Install/Uninstall Manager. Proc. ICSE 2007, pp. 178-188

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

S

T

U

W