clorm Documentation

David Rajaratnam

Feb 19, 2024

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Clorm is a Python library that provides an Object Relational Mapping (ORM) style interface to the Clingo Answer Set Programming (ASP) solver. It allows *facts* to be asserted to, and extracted from, the ASP solver in an intuitive and easy to use way. The goal of this library is to supplement the existing Clingo API to make it easier to build and maintain Python applications that integrate with Clingo.

When integrating an ASP program into a larger application a typical requirement is to model the problem domain as a statically written ASP program, but then to generate problem instances and process the results dynamically. Clorm makes this integration cleaner, both in terms of code readability but also by making it easier to refactor the python code as the ASP program evolves.

- Works with Python 3.7+ and Clingo 5.4+ (developed on Python 3.9 and Clingo 5.5)

Note: Clorm is now being hosted under the Potassco umbrella (which is the home of Clingo and the other ASP tools). The GitHub and Anaconda namespaces have been changed accordingly.

1.1 Installation

Clorm requires Python 3.7+ and Clingo 5.4+ (as of writing Clingo 5.7.1 is the latest version). There are many ways to install Clorm. In order of simplicity, it can be installed using *pip*, *conda*, from the Potassco PPA (for Ubuntu users), and finally from source.

1.1.1 Installing from a Python package manager

The *pip* package can be installed from PyPI:

```
$ pip install clorm
```

For *conda* installation, assuming you have already installed some variant of Anaconda, first you need to install Clingo:

```
$ conda install -c potassco clingo
```

Then install Clorm:

```
$ conda install -c potassco clorm
```

1.1.2 Installing from the Potassco PPA

Ubuntu users can install Clorm (and Clingo) from the Potassco PPA:

```
$ sudo add-apt-repository ppa:potassco/stable
$ sudo apt-get update
$ sudo apt install python3-clorm
```

Note: Unfortunately, the Clingo Ubuntu packages from the standard Ubuntu repository do not work correctly with Python. Even though the `clingo` executable has been compiled with Python support, the Clingo Python module itself is missing from these packages.

1.1.3 Installing from source

The project is hosted on github at https://github.com/potassco/clorm and can also be installed using git:

```
$ git clone https://github.com/potassco/clorm
$ cd clorm
$ pip install .        # or `python setup.py install` if you don't have pip
```

Note: The above instructions for installing from source assumes that you have already installed a version of Clingo that has been compiled with Python support, as well as the Python Clingo module.

For instructions on compiling and installing Clingo see: https://github.com/potassco/clingo/blob/master/INSTALL.md

1.2 Motivation

1.2.1 Object Relational Mapping (ORM)

ORM interfaces are a common way (especially in Python) of interacting with relational databases and there are some well-known Python ORMs (e.g., SQLAlchemy and Peewee). Fundamentally all ORMs provide a way of matching rows in a database table (or database view) to Python objects whose member variables correspond to the fields of the database table.

As well as mapping table rows to program objects, ORMs also provide facilities for building SQL queries using high-level primitives; rather than dealing with raw SQL strings.

1.2.2 An ORM Interface for Clingo

While the Clingo Python API is both extensive and flexible, however, it is also fairly low-level when it comes to getting data into, and out of, the solver. As a result there is typically a reasonable amount of boilerplate code that needs to be written in order to carry out even simple translations to and from Clingo. Furthermore without strong discipline such code can become interspersed throughout the Python code base.

This can be especially problematic as the ASP program evolves. Keeping the corresponding Python translation code up to date can be both cumbersome and error prone. For example, simply swapping the position of a parameter in a predicate and accidentally failing to update the corresponding Python code might not cause an immediate error in the program, but instead could cause subtle errors that are difficult to detect and debug.

An ORM interface can help to alleviate these problems. The ORM definitions that map ASP predicates to Python objects are defined in a single location and the ASP to Python translations are all generated automatically from the ORM class definitions.

In short, a Clingo ORM interface can make it easier to integrate Clingo and Python and to write Python code that is more readable and easier to maintain.

1.3 Quick Start

This section highlights the basic features of Clorm by way of a simple example. This example covers:

- Defining a simple data model,
- Combining a statically written ASP program with dynamically generated data,
- Running the Clingo solver,
- Querying and processing the solution returned by the solver.

This example is located in the project's `examples/quickstart` sub-directory; where the ASP program is `quickstart.lp` and the Python program is `quickstart.py`. There is also a version `embedded_quickstart.lp` that can be called directly from Clingo:

```
$ clingo embedded_quickstart.lp
```

This document is about the Clingo ORM API and it is therefore assumed that the reader is reasonably familiar with ASP syntax and how to write an ASP program. However, if this is not the case it is worth going to the Clingo docs for links to some good reference material.

While we assume that the reader is familiar with ASP, we do not assume that the reader is necessarily familiar with the official Clingo Python API. Since Clorm is designed to be used with the Clingo API we therefore provide some basic explanations of the relevant steps necessary to run the Clingo solver. However, for more detailed documentation see the Clingo API.

1.3.1 An Example Scenario

Imagine you are running a courier company and you have drivers that need to make daily deliveries. An item is delivered during one of four time slots, and you want to assign a driver to deliver each item, while also ensuring that all items are assigned and drivers aren't double-booked for a time slot.

You also want to apply some optimisation criteria. Firstly, you want to minimise the number of drivers that you use (for example, because bringing on a driver for a day has some fixed cost). Secondly, you want to deliver items as early in the day as possible.

1.3.2 The ASP Program

The above crieria can be encoded with the following simple ASP program:

```
time(1..4).

1 { assignment(I, D, T) : driver(D), time(T) } 1 :- item(I).
:- assignment(I1, D, T), assignment(I2, D, T), I1 != I2.

working_driver(D) :- assignment(_,D,_).
```

(continues on next page)

(continued from previous page)

```
#minimize { 1@2,D : working_driver(D) }.
#minimize { T@1,D : assignment(_,D,T) }.
```

You will notice that while the above ASP program encodes the *problem domain*, but it does not specify the *problem instance*, which in this case means specifying the individual delivery drivers and items to be delivered.

1.3.3 The Python Program

Unlike the ASP encoding of the *problem domain*, which is largely static and changes only if the requirements change, the *problem instance* changes daily. Hence it cannot be a simple static encoding but must instead be generated as part of the Python application that calls the ASP solver and processes the solution. Clorm is designed with this use-case in mind.

First the relevant libraries need to be imported.

```
from clorm import Predicate, ConstantStr
from clorm.clingo import Control
```

Note: Importing from `clorm.clingo` instead of `clingo`.

While it is possible to use Clorm with the raw clingo library, a wrapper library is provided to make the integration seemless. This wrapper (should) behave identically to the original module, except that it extends the functionality to offer integration with Clorm objects. It is also possible to monkey patch Clingo if this is your preferred approach (see *Integration with the Solver*).

1.3.4 Defining the Data Model

The most important step is to define a *data model* that maps the Clingo predicates to Python classes. Clorm provides the `Predicate` base class for this purpose; a `Predicate` sub-class defines a direct mapping to an underlying ASP logical predicate. The parameters of the predicate are specified using a number of *fields*, similar to a standard Python `dataclass`. In the Clorm context the fields can be thought of as *term definitions*, as they define how a logical *term* is converted to, and from, a Python object.

ASP's *logic programming* syntax allows for three primitive types: integer, string, and constant. From the Python side this corresponds to the standard types `int` and `str`, as well as a special Clorm defined type `ConstantStr`. Note: `ConstantStr` is sub-classed from `str` in order to disambiguate between ASP constants and strings, while still offering the same Python type checking behaviour of the `str` parent class.

```
class Driver(Predicate):
    name: ConstantStr

class Item(Predicate):
    name: ConstantStr

class Assignment(Predicate):
    item: ConstantStr
    driver: ConstantStr
    time: int
```

The above code defines three classes to match the ASP program's input and output predicates, where the name of the predicate to map to is derived from the declared class name.

`Driver` maps to the `driver/1` predicate, `Item` maps to `item/1`, and `Assignment` maps to `assignment/3` (note: the `/n` is a common logic programming notation for specifying the arity of a predicate or function). A predicate can contain zero or more fields.

The number of fields in the *Predicate* declaration must match the predicate arity and the order in which they are declared must also match the position of each term in the ASP predicate.

One thing to note here is that there is no *Predicate* sub-class that was defined corresponding to the `working_driver/1` predicate. Clorm does not require that *all* ASP predicates have a corresponding Python *Predicate* sub-class. In this case `working_driver/1` is only of interest within the ASP program itself and is not used for defining the relevant inputs and outputs of the solver, so there is no need to define any Python interface.

1.3.5 Using the Data Model to Generate Solutions

Once the data model has been defined it can be used to instantiate facts that are asserted to, or extract from, the ASP solver. In particular, it will be used to dynamically add the facts that make up a problem instance, and then to extract and print the *models* that correspond to solutions to the problem.

First, the *Control* object needs to be created and initialised, and the static problem domain encoding must be loaded.

```
ctrl = Control(unifier=[Driver,Item,Assignment])
ctrl.load("quickstart.lp")
```

The *Control* object controls the operations of the ASP solver. When the solver runs it generates *models*. These models constitute the solutions to the problem. Facts within a model are encoded as `clingo.Symbol` objects. The *unifier* argument is important as it defines which symbols are turned into *Predicate* instances.

For every symbol fact in the model, Clorm will successively attempt to *unify* (or match) the symbol against the predicates in the unifier list. When a match is found the symbol is used to define an instance of the matching predicate. Any symbol that does not unify against any of the predicates is ignored.

Once the control object is created and the unifier predicates specified the static ASP program is loaded.

Next we generate a problem instance by generating a lists of `Driver` and `Item` objects. These items are added to a *FactBase* object, which is a specialised set-like container for storing facts (i.e., predicate instances).

```
from clorm import FactBase

drivers = [ Driver(name=n) for n in ["dave", "morri", "michael" ] ]
items = [ Item(name="item{}".format(i)) for i in range(1,6) ]
instance = FactBase(drivers + items)
```

The `Driver` and `Item` constructors use named parameters that match the declared field names. While Clorm supports the use of positional arguments to initialise instances, doing so will potentially make the code harder to refactor. So in general you should avoid using positional arguments except for a few cases (eg., simple tuples where the order is unlikely to change).

Now, these input facts can be added to the control object and combined with the previously loaded ASP program to produce a *grounded* ASP program.

```
ctrl.add_facts(instance)
ctrl.ground([("base",[])])
```

At this point the control object is ready to be run and generate solutions. There are a number of ways in which the ASP solver can be run (see the Clingo API documentation). For this example we run it using a callback function, which is called each time a model is found.

```
solution=None
def on_model(model):
    global solution     # Note: use `nonlocal` keyword depending on scope
    solution = model.facts(atoms=True)

ctrl.solve(on_model=on_model)
if not solution:
    raise ValueError("No solution found")
```

The `on_model()` callback is triggered for every new model. Because of the ASP optimisation statements this callback can potentially be triggered multiple times before an optimal model is found. Also, note that if the problem is unsatisfiable then it will never be called and you should always check for this case.

The line `solution = model.facts(atoms=True)` extracts only instances of the predicates that were registered with the `unifier` parameter that was passed down through the *Control* object constructor. As mentioned earlier, any facts that fail to unify are ignored. In this case it ignores the `working_driver/1` instances. The unified facts are stored and returned in a *FactBase* object.

1.3.6 Querying

Note: As of Clorm 1.2.1 the new Query API should be the preferred query mechanism. It provides all the functionality of the old query interface and much more; including SQL-like `JOIN` clauses between predicates, and better control of how the query results are presented.

The final part of our Python program involves querying the solution to print out the relevant facts. In particular it would be useful to display all drivers and any jobs they have. To do this we call the factbase's *FactBase.query()* member function that returns a suitable *Query* object.

The query is defined in terms of a chaining over the member functions of a *Query* object. Each function call returns a modified copy of the *Query* object. This technique will be be familiar to users of Python ORM's such as SQLAlchemy or Peewee.

```
from clorm import ph1_

query=solution.query(Assignment)\
              .where(Assignment.driver == ph1_)\
              .order_by(Assignment.time)
```

The above query defines a search over the `Assignment` predicate to match the `driver` field to a special placeholder object `ph1_` and to return the assignments for that driver sorted by the delivery time. The value of `ph1_` will be provided when the query is executed. Here the *FactBase.query()* method mirrors a traditional SQL `FROM` clause.

We can now loop over the known drivers and execute the query for each driver. This is done by first *binding* the value of the placeholder `ph1_` to a specific value and calling the *Query.all()* method. This function returns a Python generator which is then used to execute and iterate over the results.

```
for d in drivers:
    assignments = list(query.bind(d.name).all())
    if not assignments:
```

(continues on next page)

(continued from previous page)

```
        print("Driver {} is not working today".format(d.name))
    else:
        print("Driver {} must deliver: ".format(d.name))
        for a in assignments:
            print("\t Item {} at time {}".format(a.item, a.time))
```

Calling `query.bind(d.name)` first creates a new query with the placeholder values assigned. Because `d.name` is the first parameter to the function call it matches against the placeholder `ph1_`. Clorm has four predefined placeholders `ph1_`,... , `ph4_`, but more can be created using the `ph_` function.

Running this example produces the following results:

```
$ cd examples
$ python quickstart.py
Driver dave must deliver:
         Item item5 at time 1
         Item item4 at time 2
Driver morri must deliver:
         Item item1 at time 1
         Item item2 at time 2
         Item item3 at time 3
Driver michael is not working today
```

Note, viewing the items for all drivers, including those drivers with no assignments, could be done simply with a single SQL `OUTER JOIN` query. Unfortunately, the Clorm Query API doesn't have an equivalent of an `OUTER JOIN`. While it can usually be simulated with a bit of extra Python code, in this case it was simplest to execute a query for each driver. Alternatively, if we were happy to only specify the drivers with assignments then the problem could be formulated in terms of a query with a grouping modifier.

```
query=solution.query(Assignment)\
              .group_by(Assignment.driver)\
              .order_by(Assignment.time)\
              .select(Assignment.item,Assignment.time)

for dname, grpit in query.all():
    print("Driver {} must deliver: ".format(dname))
    for item,time in grpit:
        print("\t Item {} at time {}".format(item, time))
```

Here the *Query.group_by()* method modifies the query generator output to return pairs of objects; where the first element of the pair consists of the elements specified by the grouping and the second element is an iterator over the matching elements for that group (here further ordered by delivery time). This is loosely analagous to how an SQL `GROUP BY` clause works. Similarly the *Query.order_by()* function operates like an SQL `ORDER BY` clause.

It is also worth noting that the *Query.select()* projection operator performs a similar function to an SQL `SELECT` clause to modify the output. Here, instead of returning the assignment item itself, it returns the two relevant parameter values.

1.4 Predicates and Fields

The heart of an ORM is defining the mapping between the predicates and Python objects. In Clorm this is acheived by sub-classing the `Predicate` class and specifying fields that map to the ASP predicate parameters.

1.4.1 The Basics

It is easiest to explain this mapping by way of a simple example. Consider the following ground atoms for the predicates `address/2` and `pets/2`. This specifies that the address of the entity `dave` is `"UNSW Sydney"` and `dave` has 1 pet.

```
address(dave, "UNSW Sydney").
pets(dave, 1).
```

Note: A note on ASP syntax. All predicates must start with a lower-case letter and consist of only alphanumeric characters (and underscore). ASP supports three basic types of *terms* (i.e., the parameters of a predicate); a *constant*, a *string*, and an *integer*. Like the predicate names, constants consist of only alphanumeric characters (and underscore) with a starting lower-case character. This is different to a string, which is quoted and can contain arbitrary characters including spaces.

ASP syntax also supports *complex terms* (also called *functions* but we will avoid this usage to prevent confusion with Python functions) which we will discuss later. Note, however that ASP does not support real number values.

To provide a mapping that satisfies the above predicate we need to sub-class the *Predicate* class and use the *ConstantStr* type specifier as well as the standard `int` and `str` to define the individual terms.

```
from clorm import Predicate, ConstantStr, IntegerField, field

class Address(Predicate):
   entity: ConstantStr
   details: str

class Pet(Predicate):
   entity: ConstantStr
   num: int = field(IntegerField, default=0)
```

The type annotations specify how the fields are to be translated to Clingo. The `entity` fields map Python strings to ASP constants, while the `Pet`'s `details` field maps Python strings to ASP strings. In constrast the `Pet`'s `num` field overrides its default `int` mapping by using the *field()* function to explicitly provide the field mapping as well as a default value. So, with the above class definitions we can instantiate some objects:

```
fact1 = Address(entity="bob", details="Sydney uni")
fact2 = Pet(entity="bob")
fact3 = Pet(entity="bill", num=2)
```

These object correspond to the following ASP *ground atoms* (i.e., facts):

```
address(bob,"Sydney uni").
pet(bob,0).
pet(bill,2).
```

There are some things to note here:

- *Predicate names*: ASP uses standard logic-programming syntax, which requires that the names of all predicate/complex-terms must begin with a lower-case letter and can contain only alphanumeric characters or underscore. Unless overriden, Clorm will automatically generate a predicate name for a *Predicate* sub-class by transforming the class name based on some simple rules:
  - If the first letter is a lower-case character then this is a valid predicate name so the name is left unchanged (e.g., `myPredicate => myPredicate`).
  - Otherwise, replace any sequence of upper-case only characters that occur at the beginning of the string or immediately after an underscore with lower-case equivalents. The sequence of upper-case characters can include non-alphabetic characters (eg., numbers) and this will still be treated as a single sequence of upper-case characters.
  - The above criteria covers a number of common naming conventions:
    * Snake-case: `My_Predicate => my_predicate`, `MY_Predicate => my_predicate`, `My_Predicate_1A => my_predicate_1a`,
    * Camel-case: `MyPredicate => myPredicate`, `MyPredicate1A => myPredicate1A`.
    * Acronym: `TCP1 => tcp1`.
- *Field order*: the order of declared term defintions in the predicate class is important.
- *Field names*: besides the Python keywords, Clorm also disallows the following reserved words: `raw`, `meta`, `clone`, `Field` as these are used as properties or functions of a *Predicate* object.
- *Constant vs string*: In the above example `"bob"` and `"Sydney uni"` are both Python strings but because of the `entity` field is declared as a *ConstantStr* (or the explicit *ConstantField* specifier) this ensures that the Python string `"bob"` is treated as an ASP constant. Note, currently it is the users' responsibility to ensure that the Python string passed to a constant term satisfies the syntactic restriction.
- The use of a *default value*: all term types support the specification of a default value.
- If the specified default is a function then this function will be called (with no arguments) when the predicate/complex-term object is instantiated. This can be used to generated unique ids or a date/time stamp.

1.4.2 Overriding the Predicate Name

As mentioned above, by default the predicate name is calculated from the corresponding class name by transforming the class name to match a number of common naming conventions. However, it is also possible to override the default predicate name with an explicit name.

There are many reasons why you might not want to use the default predicate name mapping. For example, the Python class name that would produce the desired predicate name may already be taken. Alternatively, you might want to distinguish between predicates with the same name but different arities. Note: having predicates with the same name and different arities is a legitimate and common practice with ASP programming.

```
class Address2(Predicate, name="address"):
   entity: ConstantStr
   details: str

 class Address3(Predicate, name="address"):
   entity: ConstantStr
   details: str
   country: str
```

Instantiating these classes:

```
shortaddress = Address2(entity="dave", details="UNSW Sydney")
longaddress = Address3(entity="dave", details="UNSW Sydney", country="AUSTRALIA")
```

will produce the following matching ASP facts:

```
address(dave, "UNSW Sydney").
address(dave, "UNSW Sydney", "AUSTRALIA").
```

1.4.3 Nullary Predicates

A nullary predicate is a predicate with no parameters and is also a legitimate and reasonable thing to see in an ASP program. Defining a corresponding Python class is straightforward:

```
class ANullary(Predicate):
    pass

fact = ANullary()
```

The important thing to note here is that every instantiation of `ANullary` will correspond to the same ASP fact:

```
aNullary.
```

1.4.4 Complex Terms

So far we have shown how to create Python definitions that match predicates with simple terms. However, in ASP it is common to also use complex terms within a predicate, such as:

```
booking("2018-12-31", location("Sydney", "Australia")).
```

The Clorm *Predicate* class definition is able to support the flexiblity required to deal with complex terms.

```
from clorm import Predicate

class Location(Predicate):
   city: str
   country: str

class Booking(Predicate):
    date: str
    location: Location
```

Note: There is also a *ComplexTerm* class which is an alias for the *Predicate* class. For personal stylistic reasons you may prefer to use this alias to define classes that will only be used as complex terms. However there are cases where this separation breaks down. For example when dealing with the *reification* of facts there is nothing to be gained by providing two definitions for the predicate and complex term versions of the same non logical term:

```
p(q(1)).
q(1) :- p(q(1)).
```

In this example `q/1` is both a complex term and predicate and when providing the Python Clorm mapping it is simpler not to separate the two versions:

```
class Q(Predicate):
   a: int

class P(Predicate):
   a: Q
```

The predicate class containing complex terms can be instantiated in the obvious way:

```
bk=Booking(date="2018-12-31", location=Location(city="Sydney",country="Australia"))
```

As with the primitive terms it is possible to override the translation of complex terms, for example to provide defaults, by using the *field()* function. While the first parameter of the function must be a sub-class of *BaseField*, fortunately, every predicate sub-class has a corresponding, internally generated, *BaseField* sub-class which can be accessed though the *Field* property of that predicate class. So for example we can modify the `Booking` class definition to provide a default location.

```
class Booking(Predicate):
    date: str
    location: Location = field(Location.Field, default=Location("Potsdam", "Germany")

bk2=Booking(date="2019-12-14")
```

This second booking instance will correspond to the fact:

```
booking("2019-12-14", location("Potsdam", "Germany")).
```

1.4.5 Negative Facts

ASP follows standard logic programming syntax and treats the `not` keyword as **default negation** (also **negation as failure**). Using default negation is important to ASP programming as it can lead to more readable and compact modelling of a problem.

However, there may be times when having an explicit notion of negation is also useful, and ASP/Clingo does have support for **classical negation**; indicated syntactically using the `-` symbol:

```
{ a(1..2); b(1..2) }.
-b(N) :- a(N).
-a(N) :- b(N).
```

The above program chooses amongst the `a/1` and `b/1` predicates, then for every positive `a/1` fact, the corresponding `b/1` fact is negated and vice-versa. This will generate nine stable models. For example, if `a(2)` and `b(1)` are chosen, then the corresponding negative literals will be `-b(2)` and `-a(1)` respectively.

Note: Clingo supports negated literals as well as terms. However, tuples cannot be negated.

```
f(-g(a)).   % This is valid
f(-(a,b)).  % Error!!!
```

Clorm supports negation for any fact or term that can be negated by Clingo. Specifying a negative literal simply involves setting `sign=False` when instantiating the Predicate (or ComplexTerm). Note: unlike the field parameters, the `sign` parameter must be specified as a named parameter and cannot be specified using positional arguments.

```
class P(Predicate):
    a: int

neg_p1 = P(a=1,sign=False)
neg_p1_alt = P(1,sign=False)
assert neg_p1 == neg_p1_alt
```

Once instantiated, checking whether a fact (or a complex term) is negated can be determined using the `sign` attribute of Predicate instance.

```
assert neg_p1.sign == False
```

Finally, for finer control of the unification process, a Predicate/ComplexTerm can be specified to only unify with either positive or negative facts/terms by setting a `sign` meta attribute declaration.

```
class P_pos(Predicate, name="p", sign=True):
    a: int

class P_neg(Predicate, name="p", sign=False):
    a: int

% Instatiating facts
pos_p = P_pos(1)                    % Ok
neg_p_fail = P_pos(1,sign=False)    % throws a ValueError

neg_p = P_neg(1)                    % Ok
pos_p_fail = P_neg(1,sign=False)    % throws a ValueError

% Unifying against raw Clingo positive and negative facts
raws = [Function("p",Number(1)), Function("p",Number(1),positive=False)]
fb = unify([P_pos,P_neg], raw)
assert pos_p in fb
assert neg_p in fb
```

1.4.6 Field Definitions

Clorm provides a number of standard definitions that specify the mapping between Clingo's internal representation (some form of `Clingo.Symbol`) to more natural Python representations. ASP has three *simple terms*: *integer*, *string*, and *constant*, and Clorm provides three standard definition classes to provide a mapping to these fields: *IntegerField*, *StringField*, and *ConstantField*.

Clorm also provides a *SimpleField* class that can match to any simple term. This is useful when the parameter of a defined predicate can contain arbitrary simple term types. Clorm takes care of converting the ASP string, constant or integer to a Python string or integer object. Note that both ASP strings and constants are both converted to Python string objects.

In order to convert from a Python string object to an ASP string or constant, *SimpleField* uses a regular expression to determine if the string matches the pattern of a constant and treats it accordingly. For this reason *SimpleField* should be used with care in order to ensure expected behaviour, and using the distinct field types is often preferable.

Sub-classing Field Definitions

All field classes inherit from a base class *BaseField* and it's possible to define arbitrary data conversions by sub-classing *BaseField*. Clorm provides the standard sub-classes *StringField*, *ConstantField*, and *IntegerField*. Clorm also automatically generates an appropriate sub-class for every *Predicate* definition for use in a complex term.

However, it is sometimes also useful to explicitly sub-class the *BaseField* class, or sub-class one of its sub-classes. By sub-classing a sub-class it is possible to form a *data conversion chain*. To understand why this is useful we consider an example of specifying a date field.

Consider the example of an application that needs a date term for an event tracking application. From the Python code perspective it would be natural to use Python `datetime.date` objects. However, it then becomes a question of how to encode these Python date objects in ASP (noting that ASP only has three simple term types).

A useful encoding would be to encode a date as a string in **YYYYMMDD** format (or **YYYY-MM-DD** for greater readability). Dates encoded in this format satisfy some useful properties such as the comparison operators will produce the expected results (e.g., `"20180101" < "20180204"`). A string is also preferable to using a similiarly encoded integer value. For example, encoding the date in the same way as an integer would allow incrementing or subtracting a date encoded number, which could lead to unwanted values (e.g., `20180131 + 1 = 20180132` does not correspond to a valid date).

So, adopting a date encoded string we can consider a date based fact for the booking application that simply encodes that there is a New Year's eve party on the 31st December 2018.

```
booking("2018-12-31", "NYE party").
```

Using Clorm this fact can be captured by the following Python *Predicate* sub-class definition:

```
from clorm import *

class Booking(Predicate):
   date: str
   description: str
```

However, since we encoded the date as simply a `str` (which internally maps to *StringField*) it is now up to the user of the `Booking` class to perform the necessary translations to and from a Python `datetime.date` objects when necessary. For example:

```
import datetime
nye = datetime.date(2018, 12, 31)
nyeparty = Booking(date=int(nye.strftime("%Y-%m-%d")), description="NYE Party")
```

Here the Python `nyeparty` variable corresponds to the encoded ASP event, with the `date` term capturing the string encoding of the date. In the opposite direction to extract the date it is necessary to turn the date encoded string into an actual `datetime.date` object:

```
nyedate = datetime.datetime.strptime(str(nyepart.date), "%Y-%m-%d")
```

The problem with the above code is that the process of creating and using the date in the `Booking` object is cumbersome and error-prone. You have to remember to make the correct translation both in creating and reading the date. Furthermore the places in the code where these translations are made may be far apart, leading to potential problems when code needs to be refactored.

The solution to this problem is to create a sub-class of *BaseField* that performs the appropriate data conversion. However, sub-classing `Basefield` directly requires dealing with raw Clingo `Symbol` objects. A better alternative is to sub-class the *StringField* class so you only need to deal with the string to date conversion.

```
import datetime
from clorm import *

class DateField(StringField):
    pytocl = lambda dt: dt.strftime("%Y-%m-%d")
    cltopy = lambda s: datetime.datetime.strptime(s,"%Y-%m-%d").date()

class Booking(Predicate):
    date: datetime.date = field(DateField)
    description: StringField
```

The `pytocl` definition specifies the conversion that takes place in the direction of converting Python data to Clingo data, and `cltopy` handles the opposite direction. Because the `DateField` inherits from *`StringField`* therefore the `pytocl` function must output a Python string object. In the opposite direction, `cltopy` must be passed a Python string object and performs the desired conversion, in this case producing a `datetime.date` object.

With the newly defined `DateField` the conversion functions are all captured within the one class definition and interacting with the objects can be done in a more natural manner.

```
nye = datetime.date(2018,12,31)
nyeparty = Booking(date=nye, description="NYE Party")

print("Event {}: date {} type {}".format(nyeparty, nyeparty.date, type(nyeparty.date)))
```

will print the expected output:

```
Event booking(20181231,"NYE Party"): date "2018-12-31" type <class 'datetime.date'>
```

Note: The `pytocl` and `cltopy` functions can potentially be passed bad input. For example, when converting a clingo String symbol to a date object the passed string may not correspond to an actual date. In such cases these functions can legitimately throw either a `TypeError` or a `ValueError` exception. Internally, Clorm's framework will catch these two types of exceptions and will treat them as failures to unify when trying to unify clingo symbols to facts. Any other exception is passed through as a genuine error. This should be kept in mind if you are writing your own field class.

Restricted Sub-class of a Field Definition

Another reason to sub-class a field definition is to restrict the set of values that the field can hold. For example you could have an application where an argument of a predicate is restricted to a specific set of constants, such as the days of the week.

```
cooking(monday, "Jane"). cooking(tuesday, "Bill"). cooking(wednesday, "Bob").
cooking(thursday, "Anne"). cooking(friday, "Bill").
cooking(saturday, "Jane"). cooking(sunday, "Bob").
```

When defining a predicate corresponding to cooking/2 it is possible to simply use a `ConstantField` field for the days.

```
class Cooking1(Predicate):
   dow = ConstantField
   person = StringField
   class Meta: name = "cooking"
```

However, this would potentiallly allow for creating erroneous instances that don't correspond to actual days of the week (for example, with a spelling mistake):

```
ck = Cooking1(dow="mnday",person="Bob")
```

In order to avoid these errors it is necessary to subclass the *ConstantField* in order to restrict the set of values to the desired set. Clorm provides a helper function *refine_field()* for this use-case. It dynamically defines a new class that restricts the values of an existing field class.

```
DowField = refine_field(ConstantField,
   ["sunday","monday","tuesday","wednesday","thursday","friday","saturday"])

class Cooking2(Predicate, name="cooking"):
   dow: ConstantStr = field(DowField)
   person:str

ok=Cooking2(dow="monday",person="Bob")

try:
   bad = Cooking2(dow="mnday",person="Bob")  # raises a TypeError exception
except TypeError:
   print("Caught exception")
```

Note: The *refine_field()* function can also be called with only two arguments, rather than three, by ignoring the name for the generated class. In this case an anonymously generated name will be used.

As well as explictly specifying the set of refinement values, *refine_field()* also provides a more general approach where a function/functor/lambda can be provided. This function must take a single input and return `True` if that value is valid for the field. For example, to define a field that accepts only positive integers:

```
PosIntField = refine_field(NumberField, lambda x : x >= 0)
```

An alternative to using *refine_field()* to restrict the allowable values is to an explicitly specified set is to use *define_enum_field()*. This function allows Clorm to be used with standard Python Enum classes. So, the day-of-week example could be rewritten to use an Enum class:

```
import enum

class DOW(ConstantStr, enum.Enum):
    SUNDAY="sunday"
    MONDAY="monday"
    TUESDAY="tuesday"
    WEDNESDAY="wednesday"
    THURSDAY="thursday"
    FRIDAY="friday"
    SATURDAY="saturday"

class Cooking3(Predicate, name="cooking"):
    dow: DOW
    person: str

ok = Cooking3(dow=DOW.MONDAY,person="Bob")
```

One useful advantage of using an enumeration is Clorm has built in handling to allow it to be specified as a type annotation. This means that you do not have to explicitly call the *define_enum_field()* function to generate the appropriate field definition.

Finally, it should be highlighted that this mechanism for defining a field restriction works not just for validating the inputs into an ASP program. It can also be used to filter the outputs of the ASP solver as the invalid field values will not *unify* with the predicate.

For example, in the above program you can separate the cooks on the weekend from the weekday cooks.

```
WeekendField = refine_field(ConstantField, ["sunday","saturday"])
WeekdayField = refine_field(ConstantField, ["monday","tuesday","wednesday","thursday",
↪"friday"])

class WeekendCooking(Predicate, name="cooking"):
   dow: str = field(WeekendField)
   person: str

class WeekdayCooking(Predicate, name="cooking"):
   dow: str = field(WeekdayField)
   person: str
```

1.4.7 Using Positional Arguments

So far we have shown how to create Clorm predicate and complex term instances using keyword arguments that match their defined field names, as well as accessing the arguments via the fields as named properties. For example:

```
from clorm import *

class Contact(Predicate):
    cid: int
    name: str

c1 = Contact(cid=1, name="Bob")

assert c1.cid == 1
assert c1.name == "Bob"
```

However, Clorm also supports creating and accessing the field data using positional arguments:

```
c2 = Contact(2,"Bill")

assert c2[0] == 2
assert c2[1] == "Bill"
```

While Clorm does support the use of positional arguments for predicates, nevertheless it should be used sparingly because it can lead to brittle code that can be hard to debug, and can also be more difficult to refactor as the ASP program changes. However, there are genuine use-cases where it can be convenient to use positional arguments. In particular when defining very simple tuples, where the position of arguments is unlikely to change as the ASP program changes. We discuss Clorm's support for these cases in the following section.

1.4.8 Working with Tuples

Tuples are a special case of complex terms that often appear in ASP programs. For example:

```
booking("2018-12-31", ("Sydney", "Australia")).
```

For Clorm tuples are simply a *Predicate* sub-class where the name of the corresponding predicate is empty. While this can be set using an `is_tuple` property of the complex term's class, Clorm also provides specialised support using the more intuitive syntax of a Python tuple type annotations. For example, a predicate definition that unifies with the above fact can be defined simply (using the `DateField` defined earlier):

```
class Booking(Predicate):
    date: datetime.date = field(DateField)
    location: tuple[str, str]
```

Note: For Python versions earlier than 3.9 you need to specify the tuple type using the `Tuple` identifier from the `typing` module:

```
from typing import Tuple

class Booking(Predicate):
    date: datetime.date = field(DateField)
    location: Tuple[str, str]
```

Here the `location` field is defined as a pair of strings, without having to explictly define a separate *ComplexTerm* sub-class that corresponds to this pair. To instantiate the `Booking` class a Python tuple can also be used for the values of `location` field. For example, the following creates a `Boooking` instance corresponding to the `booking/2` fact above:

```
bk = Booking(date=datetime.date(2018,12,31), location=("Sydney","Australia"))
```

While it is unnecessary to define a seperate *Predicate* sub-class corresponding to the tuple, internally this is in fact exactly what Clorm does. Clorm will transform the above definition into something similar to the following:

```
class SomeAnonymousName(Predicate, name=""):
   city: str
   country: str

class Booking(Predicate):
    date: datetime.date = field(DateField)
    location: tuple[str, str] = field(SomeAnonymousName.Field)
```

Here the *Predicate* has an empty name, so it will be treated as a tuple rather than a complex term with a function name.

One important difference between the implicitly defined and explicitly defined versions of a tuple is that the explicit version allows for field names to be given, while the implicit version will have automatically generated names. However, for simple implicitly defined tuples it would be more common to use positional arguments anyway, so in many cases it can be the preferred alternative. For example:

```
bk = Booking(date=datetime.date(2018,12,31), location=("Sydney","Australia"))

assert bk.location[0] == "Sydney"
```

Note: As mentioned previously, using positional arguments is something that should be used sparingly as it can lead to brittle code that is more difficult to refactor. It should mainly be used for cases where the ordering of the fields in the tuple is unlikely to change when the ASP program is refactored.

1.4.9 Debugging Auxiliary Predicates

When integrating an ASP program into a Python based application there will be a set of predicates that are important for inputting a problem instance and outputting a solution. Clorm is intended to provide a clean way of interacting with these predicates.

However, there will typically be other auxiliary predicates that are used as part of the problem formalisation. While they may not be important from the Python application point of view they do become important during the process of developing and debugging the ASP program. During this process it can be cumbersome to build a detailed Clorm predicate definition for each one of these, especially when all you need to do is print the predicate instances to the screen, possibly sorted in some order.

Clorm solves this issue by providing a factory helper function *simple_predicate()* that returns a *Predicate* sub-class that will map to any predicate instance with that name and arity.

For example this function could be used for the above booking example if we wanted to extract the `booking/2` facts from the model but didn't care about mapping the data types for the individual parameters. For example to match the ASP fact:

```
booking("2018-12-31", ("Sydney", "Australia")).
```

instead of the explicit `Booking` definition above we could use the *simple_predicate()* function:

```
from clorm.clingo import Symbol, Function, String
from clorm import _simple_predicate

Booking_alt = simple_predicate("booking",2)
bk_alt = Booking_alt(String("2018-12-31"), Function("",[String("Sydney"),String(
↪"Australia")]))
```

Note, in this case in order to create these objects within Python it is necessary to use the Clingo functions to explictly create `clingo.Symbol` objects.

1.4.10 Dealing with Raw Clingo Symbols

As well as supporting simple and complex terms it is sometimes useful to deal with the raw `clingo.Symbol` objects created through the underlying Clingo Python API.

Raw Clingo Symbols

The Clingo API uses `clingo.Symbol` objects for dealing with facts; and there are a number of functions for creating the appropriate type of symbol objects (i.e., `clingo.Function()`, `clingo.Number()`, `clingo.String()`).

In essence the Clorm *Predicate* class simply provides a more convenient and intuitive way of constructing and dealing with these `clingo.Symbol` objects. In fact the underlying symbols can be accessed using the `raw` property of a *Predicate* instance.

```
from clorm import *    # Predicate, ConstantField, StringField
from clingo import *   # Function, String

class Address(Predicate):
   entity: ConstantStr
   details: str

address = Address(entity="dave", details="UNSW Sydney")

raw_address = Function("address", [Function("dave",[]), String("UNSW Sydney")])

assert address.raw == raw_address
```

Note: To construct clorm objects from raw clingo symbols involves *unifying* the clingo symbol with the *Predicate* or *ComplexTerm* sub-class. This typically happens when you have a list of symbols corresponding to a clingo model and you want to turn them into a set of clorm facts. See *Unification*, *Integration with the Solver*, and *unify()* for details about unification.

Integrating Clingo Symbols into a Predicate Definition

There are some cases when it might be convenient to combine the simplicity and the structure of the Clorm predicate interface with the flexibility of the underlying Clingo symbol API. For this case it is possible to use the *RawField* class.

For example when modeling dynamic domains it is often useful to provide a predicate that defines what *fluents* hold (i.e., are true) at a given time point, but to allow the fluents themselves to have an arbitrary form.

```
time(1..5).

holds(X,T+1) :- fluent(X), not holds(X,T).

fluent(light(on)).
fluent(robotlocation(roby, kitchen)).

holds(light(on), 0).
holds(robotlocation(roby,kitchen), 0).
```

In this example instances of the `holds/2` predicate can have two distinctly different signatures for the first term (i.e., `light/1` and `robotlocation/2`). While the definition of the fluent is important at the ASP level, however, at the Python level we may not be interested in the structure of the fluent, only whether it holds or not. In such a case we can use a *RawField* to define the raw mapping from the fluent term to a Python object.

```
from clorm import Raw, Predicate

class Holds(Predicate):
   fluent: Raw
   time: int
```

RawField provides no data translation between ASP and Python and therefore has the useful property that it will unify with any `clingo.Symbol` object; in particular in this case it can be used to capture both the `light/1` and `robotlocation/2` complex terms.

When translating from Python to clingo, *RawField* expects objects of the type *Raw*, and returns objects of this type when translating from clingo to Python. *Raw* is simply a thin wrapper around the underlying `clingo.Symbol`.

For example, to create a create a Python fact that specifies that the light is on at time 0:

```
from clingo import Function
from clorm import Raw

sym_lighton = Function("light", [Function("on",[])])
lighton1 = Holds(fluent=Raw(sym_lighton), time=0)
```

1.4.11 Combining Field Definitions

The above example is useful for cases where you don't care about accessing the details of individual fluents and therefore it makes sense to simply treat them as a *RawField* complex term. However, the question naturally arises what to do if you do want more fine-grained access to these fluents.

There are a few possible solutions to this problem, but one obvious answer is to use a field that combines together multiple fields. Such a combined field could be specified manually by explicitly defining a *BaseField* sub-class. However, to simplify this process the *combine_fields()* factory function has been provided that will return such a combined sub-class. In fact Clorm uses standard Python union type annotation to implicitly generate such a mapping.

With reference to the ASP code of the previous example we could add the following Python integration:

```
from clorm import Predicate, ComplexTerm, IntegerField, ConstantField, combine_fields

class Light(Predicate):
   status: ConstantStr

class RobotLocation(Predicate, name="robotlocation"):
   robot: ConstantStr
   location: ConstantStr

class Holds(Predicate):
   fluent: Light | RobotLocation
   time: int
```

Note: For Python versions earlier than 3.11 you need to specify the union type using the `Union` identifier from the `typing` module:

```
from typing import Tuple

class Holds(Predicate):
   fluent: Union[Light, RobotLocation]
   time: int
```

When used explicitly, the *combine_fields()* function takes two arguments; the first is an optional field name argument and the second is a list of the sub-fields to combine. Note: when trying to unify a value with a combined field the raw symbol values will be unified with the underlying field definitions in the order that they are listed in the call to *combine_fields()*. This means that care needs to be taken if the raw symbol values could unify with multiple sub-fields; it will only unify with the first successful sub-field. In the above example this is not a problem as the two fluent field definitions do not overlap.

1.4.12 Dealing with Nested Lists

ASP does not have an explicit representation for lists. However a common convention for encoding lists is using a nesting of head-tail pairs; where the head of the pair is the element of the list and the tail is the remainder of the list, being another pair or an empty tuple to indicate the end of the list.

For example encoding a list of "nodes" [1,2,c] for some predicate p, might take the form:

```
p(nodes,(1,(2,(c,()))))).
```

While, such an encoding can be problematic and can lead to a grounding blowout, nevertheless when used with care can be very useful.

Unfortunately, getting facts containing these sorts of nested lists into and out of Clingo can be very cumbersome. To help support this type of encoding Clorm provides the *define_nested_list_field()* function. This factory function takes an element field class, as well as an optional new class name, and returns a newly created *BaseField* sub-class that can be used to convert to and from a list of elements of that field class. Clorm provides implicit support for this helper function with some extra type identifiers.

```
from clorm import Predicate, ConstantStr, HeadList

class P(Predicate):
   param: ConstantStr
   alist: HeadList[int]

p = P("nodes",[1,2,3])
assert str(p) == "p(nodes,(1,(2,(3,()))))"
```

1.4.13 Old Syntax

The preferred syntax for specifying predicates has changed with Clorm 1.5. The new syntax looks very similar to standard Python dataclasses or a modern Python library such as Pydantic. This new syntax integrates better with modern Python programming practices, for example using linters and type checkers.

The old syntax does not use Python type annotations and instead required the user to explicitly a *BaseField* sub-class for each term. It also required the use of a `Meta` sub-class to provide predicate meta-data, for example, to override the name of the predicate.

```
from clorm import Predicate, StringField, IntegerField

class Location(Predicate):
   city = StringField
   country = StringField

   class Meta:
      name = "mylocation"

class Booking(Predicate):
    date = StringField
    location = Location.Field
```

While the old syntax still works it should only be used as a fallback if it is not possible to specify some requirement using the new syntax. The old syntax will likely be deprecated at some point and eventually removed completely.

1.5 Fact Bases and Querying

As well as offering a high-level interface for mapping ASP facts to Python objects, Clorm also provides facilities for dealing with collections of facts. An ASP application typically does not simply deal with individual facts in isolation, but instead needs to deal in a collection of facts; whether they are the set of facts that make up the *problem instance* or, alternatively, the facts that constitute the *model* of a problem,

1.5.1 A Container for Facts

Clorm provides the *FactBase* class as a container for storing and querying facts. A *FactBase* behaves much like a normal Python `set` object with two caveats: firstly, it can only contain instances of *Predicate* sub-classes, and secondly, it provides an interface to a database-like query mechanism.

```
from clorm import Predicate, ConstantStr, FactBase

class Person(Predicate):
   id: ConstantStr
   address: str

class Pet(Predicate):
   owner: ConstantStr
   petname: str

dave = Person(id="dave", address="UNSW")
morri = Person(id="morri", address="UNSW")
dave_cat = Pet(owner="dave", petname="Frank")

fb = FactBase([dave,morri,dave_cat])

# The "in" and "len" operators work as expected
assert dave in fb
assert len(fb) == 3
```

A fact base can be populated at object construction time or later. It can also be manipulated using the standard Python set operators and member functions. Like a Python `set` object it has an *FactBase.add()* member function for adding

facts. However, because it can only store *Predicate* instances this function is able to be more flexible and has been overloaded to accept either a single fact or a collection of facts.

```
dave_dog = Pet(owner="dave", petname="Bob")
morri_cat = Pet(owner="morri", petname="Fido")
morri_cat2 = Pet(owner="morri", petname="Dusty")

fb.add(dave_dog)
fb.add([morri_cat, morri_cat2])

assert dave_dog in fb
assert morri_cat in fb
assert morri_cat2 in fb
```

1.5.2 Querying

An important motivation for providing a specialised *FactBase* container class for storing facts, as opposed to simply using a Python `list` or `set` object, is to support a rich mechanism for querying the contents of a collection.

When an ASP model is returned by the solver the application developer needs to process the model in order to extract the relevant information. The simplest mechanism to do this is to simply loop through the facts in the model. This loop will typically contain a number of conditional statements to determine what action to take for the given fact; and to store it if some sort of matching needs to take place.

However, this loop-and-test approach leads to unnecessary boilerplate code as well as making the purpose of the code more obscure. *FactBase* is intended to alleviate this problem by offering a database-like query mechanism for extracting information from a model.

Note: The following highlights the operations of the new Query API. As of Clorm 1.2.1 this new API should be the preferred search mechanism. It provides all the functionality of the old query interface and much more; including SQL-like joins between predicates and controlling how the query results are presented.

Simple Queries

Continuing the running example above the *FactBase.query()* method can be used to create *Query* objects.

```
query1=fb.query(Pet).where(Pet.owner == "dave")
query2=fb.query(Person).where(Person.id == "dave")
```

The queries are defined by chaining over the member functions of a *Query* object. Each function call returns a modified copy of the *Query* object. Here the member function *Query.where()* returns a modified copy of itself. This chaining technique will be be familiar to users of Python ORM's such as SQLAlchemy or Peewee, where it is used as a generator for SQL statements.

A query object needs to be executed in order to return the search results. There are number of end-points that can be used to execute the search. The *Query.all()* member function returns a generator to iterate over all matching search results:

```
assert set(query1.all()) == set([dave_cat,dave_dog])
```

The *Query.singleton()* member function returns the single matching item (and raises an exception if there is not exactly one match):

```
assert query2.singleton() == dave
```

The *Query.first()* member function returns the first matching item, and only raises an exception if there no matching items:

```
assert query2.first() == dave
```

The *Query.count()* member function returns the number of matching entries:

```
assert query1.count() == 2
```

Note: For comparison the following shows how these queries and results can be encoded using the legacy query API. The *FactBase.select()* method is used to create *clorm.Select* objects. Note: there is no matching member function for *Query.first()*.

```
query1_legacy=fb.select(Pet).where(Pet.owner == "dave")
query2_legacy=fb.select(Person).where(Person.id == "dave")

assert set(query1_legacy.get()) == set([dave_cat,dave_dog])
assert query2_legacy.get_unique() == dave
assert query1_legacy.count() == 2
```

An important difference between the old and new interfaces is that the call to *Select.get()* executes the query and returns the list of results. In contrast the call to *Query.all()* returns a generator and the query is executed by the generator during its iteration.

Queries with Joins

It is often useful to match instances of different predicates in the same way that you would join multiple database tables in an SQL query. To perform a search across multiple predicates it is first necessary to specify the predicates in the call to *FactBase.query()* and then specify to how these predicates are to be joined in the chained member function *Query.join()*

```
query3=fb.query(Person,Pet).join(Person.id == Pet.owner)
```

When a query contains multiple predicates the result will consist of tuples, where each tuple contains the facts matching the signature of predicates in the `query` clause. Mathematically, the tuples are a subset of the cross-product over instances of the predicates; where the subset is determined by the `join` clause.

```
assert set(query3.all()) == set([(dave,dave_cat),(dave,dave_dog),
                                 (morri,morri_cat),(morri,morri_cat2)])
```

Projections

Returning tuples of facts may not be convenient and a more usable output format may be desired. In such a case it is possible to specify a *Query.select()* specification to provide the *projection* of the results. This is much like the use of the SQL SELECT clause.

Note: Instead of formulating the query from scratch a new query can be defined as a refinement of an existing query.

```
query4=query3.select(Pet.petname, Person.address)

assert set(query4.all()) == set([("Bob","UNSW"),("Frank","UNSW"),
                                 ("Fido","UNSW"),("Dusty","UNSW")])
```

In the general case the query result is returned as a tuple consisting of the instances of the signature matching the *FactBase.query()* specification. However, if the result signature is for a single item, for example you only want to return the name of the pet, then returning a singleton tuple is not very intuitive. So, instead, when the result signature consists only of a single item then the API default behaviour is for the query result to return the items themselves rather than being wrapped in a singleton tuple.

```
query5=query3.select(Pet.petname)

assert set(query5.all()) == set(["Bob","Frank","Fido","Dusty"])
```

One important point to note when using projections is that the uniqueness of the output is no longer guaranteed. While the combinations of the cross-product of tuples being joined are guaranteed to be unique, once a *Query.select()* signature is specified this may no longer be the case. For example, if in the above query we only want to output the addresses of the owners of the different pets, the projection will lead to duplicate elements. These duplicates can be removed from the search by specifying the *Query.distinct()* modifier. In terms of SQL this is similar to specfying a SELECT DISTINCT query.

```
query6=query3.select(Person.address)
query7=query6.distinct()

assert query6.count() == 4
assert set(query6.all()) == set(["UNSW"])
assert list(query7.all()) == ["UNSW"]
```

Finally, for greatest flexibility the *Query.select()* member function can be passed a single Python *callable* object such as a function or lambda expression. The call signature of this object must match the signature specified in the *FactBase.query()* specification. The output of this callable are then presented as the results of the query.

```
query7=fb.query(Person,Pet).join(Person.id == Pet.owner)\
         .select(lambda pn,pt: f"{pt.petname} from {pn.address}")
```

Queries with Ordered Results

The *Query.order_by()* member function allows for the ordering of results similar to an SQL `ORDER BY` clause. Multiple fields can be listed as well as being able to specify ascending or descending sort order; with ascending order being the default and descending order specified by the `desc()` function.

```
from clorm import desc

query8=fb.query(Pet).order_by(Pet.owner, desc(Pet.petname))\
         .select(Pet.owner,Pet.petname)

assert list(query8.all()) == [("dave","Frank"),("dave","Bob"),
                              ("morri","Fido"),("morri","Dusty")]
```

Grouping the Query Results

Query results can be grouped in a similarly to an SQL `GROUP BY` clause using the *Query.group_by()* member function . An important distinction between SQL and Clorm's grouping mechanism is that Clorm does not support query aggregate functions, so any aggregating needs to be performed outside the query specification itself.

The *Query.group_by()* clause modifies the behaviour of the output of the generator returned *Query.all()*. Instead of simply iterating over the individual items, the iterator returns pairs where the first element of the pair is the group identifier (based on the `group_by` specification) and the second element is an iterator over the matching elements within the group.

```
query9=fb.query(Pet).group_by(Pet.owner)\
         .order_by(desc(Pet.petname)).select(Pet.petname)

result = [(oname, list(petnames)) for oname,petnames in query9.all()]
assert result == [("dave",["Frank","Bob"]),("morri",["Fido","Dusty"])]
```

Querying by Positional Arguments

As well as querying by field name (or sub-field name) it is also possible to query by the field (sub-field) position.

```
query10=fb.query(Pet).where(Pet[0] == "dave").order_by(Pet[1])
```

However, earlier warnings still hold; use positional arguments sparingly and only in cases where the order of elements will not change as the ASP code evolves.

Querying Predicates with Complex Terms

Querying Predicates with complex terms is no different to the simple case. A chain of "." notation expressions and positional arguments can be used to identify the appropriate field. For example we can replace the `Person` definition earlier to something containing a tuple:

```
from clorm import Predicate, ConstantStr, FactBase

class PersonAlt(Predicate):
   id: ConstantStr
   address: tuple[str, str]
```

(continues on next page)

(continued from previous page)

```
dave = PersonAlt(id="dave", address=("Newcastle","UNSW"))
morri = PersonAlt(id="morri", address=("Sydney","UNSW"))
torsten = PersonAlt(id="torsten", address=("Potsdam","UP"))

fb2 = FactBase([dave,morri,torsten])

query11=fb2.query(PersonAlt)\
           .where(PersonAlt.address[1] == "UNSW")\
           .select(PersonAlt.address[0])\
           .order_by(PersonAlt.address[1])

assert list(query11.all()) == ["Newcastle","Sydney"]
```

Complex Query Expressions

So far we have only seen Clorm's support for queiries with a single `where` clause, such as:

```
query12=fb.query(Pet).where(Pet.owner == "dave")
```

However, more complex queries can be specified. Firstly, a `where` clause can consist of a comma seperated list of clauses. These are treated as a conjunction:

```
# Search for pets named Bob that are owned by dave

query13=fb.query(Pet).where(Pet.petname == "Bob", Pet.owner == "dave")

assert query13.singleton() == dave_dog
```

It is also possible to specify more complex queries using the overloaded logical operators &, |, and ~.

```
# Find the Person with id "torsten" or whose university address is not "UP"
query14=fb2.query(PersonAlt)\
           .where((PersonAlt.id == "torsten") | ~(PersonAlt.address[1] == "UP"))

assert set(query14.all()) == set([dave,morri,torsten])

# Find the Person with id "dave" and with address "UNSW"
query15=fb2.query(PersonAlt)\
           .where((PersonAlt.id == "dave") & (PersonAlt.address[1] == "UNSW"))

assert query15.singleton() == dave
```

Clorm also provides the explicit functions *and_()*, *or_()*, and *not_()* for these logical operators, but the overloaded syntax is arguably more intuitive. With the explicit functions the above could also be written as:

```
query14alt=fb2.query(PersonAlt)\
              .where(or_(PersonAlt.id == "torsten", not_(PersonAlt.address[1] == "UP")))
query15alt=fb2.query(PersonAlt)\
              .where(and_(PersonAlt.id == "dave", PersonAlt.address[1] == "UNSW"))
```

Finally, it is also possible to test for membership of a collection using the *in_()* and *notin_()* functions.

```
query16=fb2.query(PersonAlt).where(in_(PersonAlt.id, ["dave","bob","sam"])

assert query16.singleton() == dave
```

Queries with Parameters

To support more flexible queries Clorm provides placeholders as a means of parameterising queries. Placeholders are named `ph1_` to `ph4_` and correspond to the positional parameters. These parameters are bounds to actual values by calling *`Query.bind()`* where the input parameter to the function call must match the declared placeholders.

```
from clorm import ph1_, ph2_

query12=fb.query(Pet).where((Pet.owner == ph1_) & (Pet.petname == ph2_))

assert query12.bind("dave","Bob").singleton() == dave_dog
assert query12.bind("dave","Fido").count() == 0
```

Additional placeholders can be defined using the `ph_()` function. For example, `ph_(5)` will create a placeholder for the 5th positional argument.

Clorm also supports **named placeholders**, which may be preferable if there are a larger number of parameters. A named placeholder is created by calling the `ph_()` function with a non-numeric first parameter, and are referenced in the call to *`Query.bind()`* using keyword function parameters. An advantange of named placeholders is that they allow for a default value to be set.

```
from clorm import ph_

query13=fb.query(Pet).where(Pet.owner == ph_("owner","dave"))

assert set(query13.all()) == set([dave_dog,dave_cat])
assert set(query13.bind(owner="morri").all()) == set([morri_cat,morri_cat2])
```

Querying Negative Facts/Complex-Terms

ASP problems can often by compactly modelled using only default negation instead of strong negation. Because of this the use of explicitly negated literals is not particularly common in ASP programs.

Nevertheless Clorm does support negated facts and the Clorm query mechanism support querying based on the sign of a fact or complex term.

```
from clorm import Predicate

class P(Predicate):
    a: int

p1 = P(1)
neg_p2 = P(2,sign=False)

fb3 = FactBase([p1,neg_p2])
assert fb3.query(P).where(P.sign == True).singleton() == p1
assert fb3.query(P).where(P.sign == False).singleton() == neg_p2
```

Querying the Predicate Itself

While it is possible to query fields (and sub-fields) of a predicate using the intutive "." syntax (eg., `Pet.owner == ph1_`), unfortunately, it is not possible to provide this intuitive syntax for querying the predicate itself (e.g., a query of `Pet < ph1_` will fail).

Instead a helper function `path()` is provided for this special case.

```
from clorm import path

query14=fb.query(Pet).where(path(Pet) == dave_dog)
assert query14.count() == 1
```

Note, querying by the predicate itself is a boundary case. While testing for equality or inequality makes sense semantically, the semantics of a query based on an ordering operator doesn't always make sense (eg., `path(Pet) < dave_dog`).

Furthermore, when testing for equality or inequality it is usually simpler to not use the query mechanism and instead to use the basic Python set inclusion operation:

```
assert dave_dog in fb
```

Queries that modify the FactBase

Querying can be used to modify the underlying `FactBase` to acheive a similar effect to an SQL `DELETE` or `UPDATE` query. The *`Query.delete()`* end-point provides a mechanism to delete the matching facts of a query from the underlying `FactBase`.

For example, to delete the pets owned by people with the address "UNSW", we can identify the matching pets in the query.

```
fb.query(Person,Pet).join(Person.id == Pet.owner).where(Person.address == "UNSW")\
  .select(Pet).delete()
```

Clorm facts are immutable, so it is not possible to modify the facts themselves in the same way that one might want to perform an SQL `UPDATE` query. Nevertheless it is possible to provide query functions to make it easy to replace the selected facts within the `FactBase`. The *`Query.replace()`* and *`Query.modify()`* end-points provides the mechanism to modify the underlying `FactBase` based on the matches of a query.

For example, rather than deleting all pets owned by people living in "UNSW" we can instead use the *`Query.replace()`* end-point to assign these pets to a new owner, "Rob", replacing the existing Pet fact with a modified cloned fact.

```
def change_owner(pet):
    return pet.clone(owner="Rob")

fb.query(Person,Pet).join(Person.id == Pet.owner).where(Person.address == "UNSW") \
  .select(Pet).replace(change_owner)
```

The *`Query.replace()`* method takes a single function as an input. The input signature of the function must match the query selection criteria. The expected output of the function is a fact or set of facts that will be used to replace the matched facts. The matched facts are deleted and the replacements inserted in their place.

The *`Query.modify()`* method is a more general version of the *`Query.replace()`* method. This allows for greater control over which facts are deleted. In this case the parameter function must return a pair of fact sets. The first set contains the facts to be deleted and the second set the facts to be inserted.

Note, the behaviour of these modifying queries could also achieved by simply iterating over the results of a "normal" query and explictly buidling the delete and add lists. The advantage of using the special end-point methods is that it is more declarative and therefore more succint and less error prone. This can be especially convenient when chaining multiple modifications of a factbase.

FactBases with Indexes

A typical ASP program has models that contain relatively small numbers of facts (e.g., 10-100 facts). With such small numbers of facts, querying these facts from a *FactBase* can often be done without regard to performance considerations, since the solving of the combinatorial ASP problem will often dominate.

However, as the number of the number of facts increases so to does the cost of querying these facts from a *FactBase*. Eventually this can lead to a noticeable impact of performance.

In order to alleviate this problem a *FactBase* can be defined with indexes for one of more fields.

To highlight this the following example creates a simple test predicate that has two fields. Instances are created where the two fields have identical values, and these instances are added to a *FactBase* where one field is indexed and the other is not.

```
from clorm import Predicate

class Num(Predicate):
    to_idx: int
    not_to_idx: int

fb4 = FactBase([Num(to_idx=n,not_to_idx=n) for n in range(0,100000)], indexes=[Num.to_
↪idx])
```

We can now compare the timing differences between searching for a value where one query searches for a value based on the indexed field and the other query searches for the same value based on the non-indexed field.

```
import time

query15=fb4.query(Num).where(Num.to_idx == 50000)
query16=fb4.query(Num).where(Num.not_to_idx == 50000)


start_q15 = time.time()
assert query15.count() == 1
q15_time = time.time() - start_q15

start_q16 = time.time()
assert query16.count() == 1
q16_time = time.time() - start_q16

assert q15_time < q16_time
print("Indexed search {} vs non-indexed search {}".format(q15_time,q16_time))
```

To confirm that these two queries are indeed behaving differently we can examine the query plans for the respective queries by calling the *Query.query_plan()* methods.

```
print("Querying without indexing:\n{}\n".format(query15.query_plan()))
print("Query with indexing:\n{}\n".format(query16.query_plan()))
```

Note, currently, there is no official API for a query plan object so it is only possible to print the object for manual examination. The key aspect to notice here is that the search on the indexed field appears as a `keyed search` whereas the search on the non-indexed field appears as a `filter clause`. Essentially the non-indexed search has to examine every fact in the fact base while the indexed search doesn't.

```
Querying without indexing:
-------------------------------------------------------
QuerySubPlan:
        Input Signature: ()
        Root path: Num
        Indexes: (Num.to_idx,)
        Prejoin keyed search: [ Num.to_idx == 50000 ]
        Prejoin filter clauses: None
        Prejoin order_by: None
        Join key: None
        Post join clauses: None
        Post join order_by: None
-------------------------------------------------------

Query with indexing:
-------------------------------------------------------
QuerySubPlan:
        Input Signature: ()
        Root path: Num
        Indexes: (Num.to_idx,)
        Prejoin keyed search: None
        Prejoin filter clauses: ( [ Num.not_to_idx == 50000 ] )
        Prejoin order_by: None
        Join key: None
        Post join clauses: None
        Post join order_by: None
-------------------------------------------------------
```

A final note. As with indexing in databases, the use of indexes should be monitored carefully. The speed up in search must always be balanced the cost of constructing and maintaining the index.

1.6 Clingo Solver Integration

So for we have shown how to define predicates and fact bases, and perform queries over fact bases. However, we have not discussed in detail how these objects are integrated and interact with the Clingo ASP solver.

As detailed in *Raw Clingo Symbols*, at its most basic level a Clingo `Symbol` object can be extracted from a Clorm `Predicate` object using the `raw` property.

and conversely a `Symbol` object can be passed as the `raw` parameter in the `Predicate` constructor.

1.6.1 Wrapper classes

While extracting `Symbol` objects from `Predicates` is sufficient to use Clorm with the Clingo solver it is still not a particularly clean interface and can result in unnecessary boilerplate code. To avoid this the `clorm.clingo` module fully integrates Clorm objects into the Clingo API. It does this by providing wrapper classes around the key Clingo classes:

- `Control` is the heart of the interface to the reasoner and determines when and how the reasoner is called. The `Control.solve()` function starts the solving process. Depending on the parameters it can operate in different modes; for example it can use a callback that is called with `Model` ojects and can also return a `SolveHandle`.
- `Model` is the class that contains the facts (and other meta-data) associated with an ASP model. This class should not be instantiated explicitly and is instead passed within the `Control.solve` callback.
- `SolveHandle` provides a mechanism to iterate over the models. Similarly to the `Model` class it should not be instantiated explicitly and is instead returned from the `Control.solve()` function.

Control

`Control` is the main object for interacting with the ASP grounder and solver and provides a rich set of features. It allows an ASP program that is stored in a file to be loaded. It can then be grounded and solved and the generated models returned. It also allows for incremental solving, where a ground program can be extended and solved repeatedly, as well as both synchronous and asynchronous solving modes. These featuers are documented in the Clingo Control API so we only highlight the changes that Clorm introduces.

Clorm adds some new member functions as well as overloading some existing functions:

- `__init__()`. Clorm adds an optional `unifier` parameter for specifying a default list of `Predicate` subclasses (or a single `SymbolPredicateUnifier` object). This parameter is passed to any generated `Model` objects and is used during the unification process of converting raw symbols to Clorm facts. A second parameter `control_` is introduced. This parameter is mutually exclusive with the standard parameters for `clingo.Control`. The `control_` parameter allows an existing `clingo.Control` object to be passed to the wrapper and is a mechanism to allow Clorm to be used even when Python is embedded within Clingo. See the example `embedded_quickstart.lp` for more details, but the basics are that a Control object is passed to the embedded `main` function which is then wrapped in `clorm.clingo.Control`:
- `unifier`. A property to get and set the unifier even after the `clorm.clingo.Control` object has been instantiated.

```
#script(python).

import clorm.clingo

def main(ctrl_):
    ctrl = clorm.clingo.Control(control_=ctrl_)

    # ...
#end.
```

- `add_facts(facts)`. A new function that adds facts to an ASP program. The input can be any collection of facts (such as a `list`, `set`, or `clorm.FactBase`). A fact can be either a `clorm.Predicate` or `clingo.Symbol` object. Note however that a `clorm.FactBase` can only contain `clorm.Predicate` instances. **Warning**: because the initial facts in an ASP program will affect the grounding, new facts should only be added to the program **before** grounding.

```
from clorm.clingo import Control

ctrl = Control()
ctrl.load("quickstart.lp")
ctrl.add_facts(db)
ctrl.ground([("base",[])])
```

- `solve()`. This function provides a rich set of options for calling the solver and returning the results. These parameters are documented in the Clingo API. Clorm modifies this interface in three ways:
  - `assumptions` parameter. This parameter restricts the returned models to only those satisfying the given assumptions. This parameter must consist of a list of argument-boolean pairs. As well as `clingo.Symbol` objects the argument element is extended to allow `clorm.Predicate` instances or a collection of clingo symbols or clorm predicates. This makes it flexible so that, for example, a `FactBase` object can be specified as being either True or False in the model.
  - `on_model` callback parameter. Clorm modifies this interface so that a `clorm.clingo.Model` is pass to the callback function.
  - If the parameter `yield_=True` is specified then the return value of the function is a `clorm.clingo.SolveHandle` object. This object iterates over `clorm.clingo.Model` objects.
- `assign_external(external,truth)`. This function assigns a truth value to an external atom. This function has been overloaded so that the `external` parameter can also take a `clorm.Predicate` instance or a collection of external atoms (e.g., a `FactBase`), where the same truth value is assigned to all atoms in the collection.
- `release_external(external)`. This function releases an external atom so that it is permanently false. The function is overloaded so that the `external` parameter can also take a `clorm.Predicate` instance or a collection of extenal atoms.

Model

The Clingo Model object encapsulates an ASP model and the associated meta-data. It is passed to the `Clingo.solve(on_model=on_model)` callback. Clorm wraps the `Model` class to provide a mechanism to extract Clorm facts from the model. The additional and modified functions are:

- `facts(self, unifier=None, atoms=False, terms=False, shown=False, raise_on_empty=True)`. returns a fact base object constructed from unifying against the raw Clingo symbols within the model.

  The `unifier` parameter takes a list of `Predicate` sub-classes or a single `SymbolPredicateUnifier` which defines the predicates to unify against. If no `unifier` parameter is provided then a `unifier` must have been passed to the `clorm.clingo.Control` object.

  The `raise_on_empty` parameter specifies that a `ValueError` will be raised if the returned fact base is empty. This can happen for two reasons: there were no selected elements in the model or there were elements from the model but none of them was able to unify with the fact base. This parameter is potentially useful for debugging purposes. While returning an empty fact base can be legimate outcomes for some applications, however in many cases this would indicate a problem; either in the ASP program or in the declaration of the predicates to unify against.

  Apart from the `unifier` and `raise_on_empty` parameters the remaining parameters are the same as for the `Model.symbols()` function.
- `contains(self,fact)`. Extends `clingo.Model.contains()` to allow for a clorm facts as well as a clingo symbols.

SolveHandle

The Clingo SolveHandle object provides a mechanism for iterating over the models when the `yield_=True` option is specified in the `Control.solve()` function call. The various iterator functions are modified by Clorm, but its operations should be transparent to the user.

1.6.2 Monkey-patching

Clorm provides monkey patching of the `Control` class so that Clorm can be integrated into an existing code base with minimal effort.

```
from clorm import monkey; monkey.patch()
from clingo import Control
```

Note: In general monkey patching should be avoided where possible.

1.7 Embedding Python into ASP

As well as providing an API for calling the Clingo solver from Python, Clingo also supports embedding Python function calls within an ASP program. Clorm builds on these Clingo features by providing an extended interface for calling Python from an ASP program.

1.7.1 Specifying Type Cast Signatures

When calling (external) Python functions from within an ASP program the Python function is passed inputs encoded as objects of type `Clingo.Symbol`. The Clingo system then expects a return value of either a single `Clingo.Symbol` object or a list of `Clingo.Symbol` objects. It is the responsibility of the programmer to perform any necessary data type conversions.

For convenience Clingo does provide *some* exceptions to this basic procedure. These exceptions are to do with the output values where Clingo is able to make some assumptions. In particular:

- Python integer values will be automatically converted to a `clingo.Symbol` object using the `clingo.Number()` function. Similarly Python string values will be converted using the `clingo.String()` function.
- Python tuples will automatically be converted using the `clingo.Function()` function, with an empty string as the name parameter, which is how Clingo internally represents tuples.

While this automatic data conversion behaviour of the Clingo API can be convenient it is however a somewhat ad-hoc approach. In the first place while there is some automatic conversion of the outputs of the Python function, however, there is no automatic conversions for the inputs to the Python function. Secondly it cannot deal with arbitrary outputs. For example, it is not possible to specify that a Python string should be interpreted as a constant object rather than a string object. Also complex terms other than a tuples cannot be handled automatically.

To address these problems Clorm provides a more principled approach that allows for the specification of a *type cast signature* that defines how to automatically convert between the expected input and output types.

We explain Clorm's type casting features by way of a simple example. In particular, consider a Python `date_range` function that is given two dates (encoded in **YYYY-MM-DD** string format) and returns an enumerated list of dates within this range. This can be called from Clingo by prefixing the function with the `@` symbol:

```
date(@date_range("2019-01-01", 2019-01-05")).
```

The corresponding Python object needs to take the two input `clingo.String` symbols turn them into dates, compute the dates in the range, and return the enumerated list of outputs converted back into `clingo.String` symbols.

```
from clingo import String
from datetime import datetime, timedelta

def date_range(start, end):
    pystart = datetime.strptime(start.string, "%Y-%m-%d").date()
    pyend = datetime.strptime(end.string, "%Y-%m-%d").date()

    inc = timedelta(days=1)
    tmp = []
    while pystart < pyend:
        tmp.append(pystart.strftime("%Y%m%d"))
        pystart += inc
    return list(enumerate(tmp))
```

This function will be called by Clingo during the ASP program grounding process that will generate the list of ground facts:

```
date((0,"2019-01-01")). date((1,"2019-01-02")).
date((2,"2019-01-03")). date((3,"2019-01-04")).
```

Here the string objects that encode the two dates are first converted into Python date objects. Then when the appropriate dates are calculated they are translated back into strings and enumerated into list of pairs with the first element of each pair being the enumeration index and the second element being the date encoded string. Note: that the above Python code does takes advantage of the automatic clingo API type conversions.

Clorm provides a way to simplify this data translation through the use of a decorator that attaches a *type cast signature* to the function. Firstly the conversion to and from date objects can be removed from the function and instead declared as part of a `DateField`, thus simplifying the function but also making the code more re-usable.

```
from clorm import StringField
from datetime import datetime, timedelta

class DateField(StringField):
    pytocl = lambda dt: dt.strftime("%Y-%m-%d")
    cltopy = lambda s: datetime.datetime.strptime(s,"%Y-%m-%d").date()
```

The `DateField` sub-classes a `StringField` and provides the conversion between string objects and dates.

```
@make_function_asp_callable(DateField, DateField, [(IntegerField, DateField)])
def date_range(start, end):
    inc = timedelta(days=1)
    tmp = []
    while start < end:
        tmp.append(start)
        start += inc
    return list(enumerate(tmp))
```

The important point is that the type cast signature provides a mechanism to specify arbitrary data conversions both for the input and output data; including conversions generated from very complex terms specified as Clorm `Predicate`

sub-classes. Consequently, the programmer does not have to explicitly write the type conversion code and even existing functions can be decorated to be used as callable ASP functions.

Another point to note is that the Clorm specification is also able to use the simplified tuple syntax from the Clingo API to specify the enumerated pairs. In fact this code can be viewed as a short-hand for an explicit declaration of a `ComplexTerm` tuple and internally Clorm generates a signature equivalent to the following:

```
from clorm import Predicate, field

class EnumDate(Predicate, name=""):
    idx: int
    dt: datetime.date = field(DateField)

@make_function_asp_callable
def date_range(start : DateField, end : DateField) -> [EnumDate.Field]:
   ...
```

There are two decorator functions that Clorm provides:

- `make_function_asp_callable`: Wraps a normal function. Every function parameter is converted to and from the clingo equivalent.
- `make_method_asp_callable`: Wraps a member function. The first paramater is the object's `self` parameter so is passed through and only the remaining parameters are converted to their clingo equivalents.

In summary, the Clorm type cast signature provides a principled approach for specifying arbitrarily complex type conversions. Furthermore, by making this type conversion specification explicit it is clear what conversions will be performed and therefore makes for clearer and more re-usable code.

1.7.2 Specifying a Grounding Context

From Clingo 5.4 onwards, the Clingo grounding function allows a `context` parameter to be specified. This parameter defines a context object for the methods that are called by ASP using the @-syntax.

While this context feature can be used in a number of different ways, one way is simply as a convenient namespace for encapsulating the external Python functions that are callable from within an ASP program. Clorm provides support for this use-case through the use of a *context builder*.

`ContextBuilder` allows arbitrary functions to be captured within a context and assigned a data conversion signature (where the data conversion signature is specified in the same way as the `make_function_asp_callable` and `make_method_asp_callable` functions). It also allows the function to be given a different name when called from within the context.

Also like `make_function_asp_callable` and `make_method_asp_callable`, the context builder's `register` and `register_name` member functions can be called as decorators or as normal functions. However, unlike the standalone functions, a useful feature of the `ContextBuilder` member functions is that when called as a decorator they do not decorate the original functions but instead return the original function and only decorate the function when called from within the context.

Consider the decorated `date_range` function defined earlier. One issue with this function is that it can only be called from within an ASP program (unless you use `clingo.Symbol` inputs and outputs). However, a function that generates an enumerated date range is fairly useful in and of itself so it might be desireable to be called from other Python functions.

The `ContextBuilder` can be used to solve this problem.

```
from clorm import ContextBuilder

cb=ContextBuilder()

# decorator that registers the function with the context builder
@cb.register(DateField, DateField, [(IntegerField, DateField)])
def date_range(start, end):
    inc = timedelta(days=1)
    tmp = []
    while start < end:
        tmp.append(start)
        start += inc
    return list(enumerate(tmp))

# Use the function as normal to calculate a date range
sd=datetime.date(2010,1,5)
ed=datetime.date(2010,1,8)
dr=date_range(sd,ed)

ctx=cb.make_context()

# Use the decorated version from within the context
cl_dr = ctx.date_range(clingo.String("2010-01-05"),clingo.String("2010-01-08"))
```

The above example shows how the original `date_range` function is untouched but instead the context version is wrapped using the data conversion signature. The created context can then be passed as an argument during the grounding phase.

```
import clingo

ctrl=clingo.Control()

# Define an ASP program and import it into the control object
prgstr="""date(@date_range("2010-10-10", "2010-10-13")."""

with ctrl.builder() as b:
   clingo.parse_program(prgstr, lambda s: b.add(s))

# Ground using the context defined earlier
ctrl.ground([("base",[])],context=ctx)

# Solve
ctrl.solve()
```

The program defined in the string uses the `date_range` function defined by the earlier context and when solved will produce the expected answer set:

```
date((0,"2010-10-10")). date((1,"2010-10-11")). date((2,"2010-10-12")).
```

Of course multiple functions can be registered with a `ContextBuilder` and it can also be used as a form of code re-use to define multiple versions of a function with different signatures.

```
def add(a,b): a+b
```

(continues on next page)

(continued from previous page)

```
# Register two versions using the same function - one to add numbers and one
# to concat strings. Note: first argument is the new function name, last
# argument is the function; the middle arguments define the signature.
cb.register_name("addi", IntegerField, IntegerField, IntegerField, add)
cb.register_name("adds", StringField, StringField, StringField, add)

ctx=cb.make_context()

n1=clingo.Number(1); n2=clingo.Number(2); n3=clingo.Number(3)
s1=clingo.String("ab"); s2=clingo.String("cd"); s3=clingo.String("abcd")

assert ctx.addi(n1,n2) == n3
assert ctx.adds(s1,s2) == s3
```

1.8 Advanced Features

This chapter provides details of more advanced Clorm features. Clorm implements a number of functions and classes to provide its abstraction over the raw Clingo symbol objects. There may be more advanced use-cases where it is useful to have access to these features. Or at the very least it may help to provide a better understanding of the internal operations of Clorm.

1.8.1 Introspection of Predicate Definitions

A number of properties of a `Predicate` definition can be accessed through the `meta` property of the class. To highlight these features we assume the following definitions:

```
from clorm import Predicate

class Address(Predicate):
    street: str
    city: str

class Person(Predicate):
    name: str
    address: Address
```

Firstly the name and arities of the complex term and predicate can be examined:

```
assert Address.meta.name == "address"
assert Address.meta.arity == 2
assert Person.meta.name == "person"
assert Person.meta.arity == 2
```

The fields, and sub-fields, of a predicate that are specified as being indexed are also available:

```
assert set(Person.meta.indexes) == set([Person.name, Address.city])
```

It is possible to introspect the field names of a predicate:

```
assert set(Person.meta.keys()) == set(["name","city"])
```

1.8.2 Raw Clingo and Clorm Facts

The Clingo reasoner deals in `clingo.Symbol` objects while Clorm facts provide an intuitive abstraction on top of the underlying raw symbols. Clorm and the Clingo integration library `clorm.clingo` minimises the need to deal with explicitly coverting between the two.

However, there may still be use-cases where it is useful to deal explicitly with the underlying raw clingo symbol objects. For example, if the user choses not to use the `clorm.clingo` integration module but instead to use the main `clingo` module.

Unification

In logical terms, unification involves transforming one expression into another through term substitution. We co-op this terminology for the process of transforming `Clingo.Symbol` objects into Clorm facts. This unification process is integral to using Clorm since it is the main process by which the symbols within a Clingo model are transformed into Clorm facts.

A `unify` function is provided that takes at least two parameters; a *unifier* and a list of raw clingo symbols. It then tries to unify the list of raw symbols with the predicates in the unifier. It returns a `FactBase` containing the facts, or a list of facts if the parameter `ordered=True`, that resulted from the unification of the symbols with the first matching predicate. If a symbol was not able to unify with any predicate it is ignored.

```
from clingo import Function, String
from clorm import Predicate, unify

class Person(Predicate):
   name: str
   address: str

good_raw = Function("person", [String("Dave"),String("UNSW")])
bad_raw = Function("nonperson", [])
fb = unify([Person], [bad_raw, good_raw])
assert list(fb) == [Person(name="Dave", address="UNSW")]
assert len(fb.indexes) == 1
```

Note: In general it is a good idea to avoid defining multiple predicate definitions that can unify to the same symbol. However, if a symbol can unify with multiple predicate definitions then the `unify` function will match only the first predicate definition in the list of predicates.

To get more fined grained behaviour, such as controlling which fields are indexed, the user can also use a `SymbolPredicateUnfier` helper function. The symbol predicate unifer can then be passed to the unify function instead of a list of predicates.

```
from clingo import Function, String
from clorm import Predicate, ConstantStr, unify

spu = SymbolPredicateUnifier()
```

(continues on next page)

(continued from previous page)

```
class Person(Predicate):
    name: str
    address: str

class Person(Predicate):
    id: ConstantStr
    address: str

good_raw = Function("person", [String("Dave"),String("UNSW")])
bad_raw = Function("nonperson", [])
fb = unify(spu, [bad_raw, good_raw])
assert list(fb) == [Person(name="Dave", address="UNSW")]
assert len(fb.indexes) == 0
```

This function has two other useful features. Firtly, the option `raise_on_empty=True` will throw an error if no clingo symbols unify with the registered predicates, which can be useful for debugging purposes.

The final option is the `delayed_init=True` option that allow for a delayed initialisation of the `FactBase`. What this means is that the symbols are only processed (i.e., they are not unified agaist the predicates to generate facts) when the `FactBase` object is actually used.

This is also useful because there are cases where a fact base object is never actually used and is simply discarded. In particular this can happen when the ASP solver generates models as part of the `on_model()` callback function. If applications only cares about an optimal model or there is a timeout being applied then only the last model generated will actually be processed and all the earlier models may be discarded (see *Integration with the Solver*).

1.9 Experimental Features

> **Warning:** The following are experimental features that shouldn't be considered part of the official Clorm API. It is not meant to be used and mostly for exploring and developing ideas.

1.9.1 Reusable Components

This is a work in progress

The goal is to slowly build up a comprehensive set of Clorm components (definitions and functions) that could be used to build readable ASP programs that can be easily integrated into Python applications.

Library components that are currently in the works:

- **date**: Contains definitions and functions for dealing with dates and enumerated dates.
- **timeslot**: This library will help to deal with blocks of time; for example dividing a day into 15 minute blocks.

1.9.2 JSON Encoding and Decoding

Clorm provides functions for encoding and decoding facts as JSON. The motivation is to be able to pass around facts between different processes. For example, you may want to generate a problem instance as part of a main web application but then pass off the problem instance to a worker processes that actually calls the solver.

Note: The JSON encoding of the clingo.Symbol objects generated here is not the same as running clingo with the `--outf=2` argument. The output here is intended as an format for passing around facts between processes and not to be particularly human readable. In contrast the clingo output in more human readable but would require parsing to regenerate the original symbol objects.

The `FactBaseCoder` class is a helper class to be able to encode/decode JSON for particular Predicate sub-classes. The predicates can be supplied at construction.

```
from clorm import Predicate
from clorm.json import FactBaseCoder

class Afact(Predicate):
    aint: int
    astr: str

fb_coder = FactBaseCoder([Afact])
```

Alternatively, the `FactBaseCoder` can also be used as a decorator to register predicates.

```
from clorm import Predicate
from clorm.json import FactBaseCoder

fb_coder = FactBaseCoder()

class Fun(Predicate):
    aint: int
    astr: str

class Tup(Predicate, name=""):
    aint: int
    astr: str

@fb_coder.register
class Afact(Predicate):
    aint: int
    afun: Fun

@fb_coder.register
class Bfact(Predicate):
    astr: str
    atup: Tup
```

Once the fact coder has been created and the appropriate predicates registered, it then provides `encoder` and `decoder` member functions that can be used with the standard Python JSON functions.

Assuming the above code:

```
import json

afact1 = Afact(aint=10, afun=Fun(aint=1, astr="a"))
afact2 = Afact(aint=20, afun=Fun(aint=2, astr="b"))
bfact1 = Bfact(astr="aa", atup=Tup(aint=1, astr="a"))
bfact2 = Bfact(astr="bb", atup=Tup(aint=2, astr="b"))

json_str = json.dumps([afact1,afact2,bfact1,bfact2], default=fb_coder.encoder)

facts = json.loads(json_str, object_hook=fb_coder.decoder)
```

As a convenience the fact coder provides member functions: `dump`, `dumps`, `load`, `loads` that call the respective json functions with the appropriate encoder and decoder functions. So the above calls to the json functions can be simplified to:

```
json_str = fb_coder.dumps([afact1,afact2,bfact1,bfact2])

facts = fb_coder.loads(json_str)
```

1.10 API Documentation

The heart of the Clorm ORM involves defining the mapping from *ground predicates* to member variables of a Python object. There are two aspects to this: *fields*, that define how *logical terms* are mapped to Python objects, and *predicate* definitions, that define predicate and function names, their arities and the appropriate field for each of the parameters.

1.10.1 Fields

Fields provide a specification for how to convert `clingo.Symbol` objects into the appropriate/intuitive Python object. As of Clorm version 1.5, the preferred mechanism to specify fields is to use standard Python type annotations. The primitive logical terms *integer* and *string* are specified using the standard Python type `int` and `str` respectively, while logical *constant* terms are specified using a specially defined type:

class clorm.**ConstantStr**

A special `str` sub-class for specifying constant logical terms.

Internally within Clorm the type specifiers are mapped to a set of special classes that contain the functions for converting between Python and the `clingo.Symbol` objects. These special classes are referred to as *field definitions* and are subclassed from a common base class *BaseField*.

class clorm.**BaseField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

Define a mapping from ASP logical terms to Python objects.

A field is used as part of a `ComplexTerm` or `Predicate` definition. It specifies the translation between ASP terms and Python objects.

It contains two class functions `cltopy` and `pytocl` that implement the translation from Clingo to Python and Python to Clingo respectively. For `BaseField` these functions are abstract. `StringField`, `IntegerField`, and `ConstantField` are standard sub-classes that provide translations for the ASP simple terms; *string*, *integer* and *constant*.

These `BaseField` subclasses can be further sub-classes to build a chain of translations.

To sub-class BaseField (or one of its sub-classes) simply specify `cltopy` and `pytocl` functions that take an input and perform some translation to an output format.

`RawField` is a special `BaseField` sub-class that provides direct pass-through for raw clingo symbol objects. It cannot be sub-classed further.

Note: the `cltopy` and `pytocl` functions are legitmately allowed to throw either a `TypeError` or `ValueError` exception when provided with bad input. These exceptions will be treated as a failure to unify when trying to unify clingo symbols to facts. However, any other exception is passed through as a genuine error. This should be kept in mind if you are writing your own field class.

Example

```
import datetime

class DateField(StringField):
          pytocl = lambda dt: dt.strftime("%Y%m%d")
          cltopy = lambda s: datetime.datetime.strptime(s,"%Y%m%d").date()
```

Because `DateField` sub-classes `StringField`, rather than sub-classing `BaseField` directly, it forms a longer data translation chain:

clingo symbol object – BaseField – StringField – DateField – python date object

Here the `DateField.cltopy` is called at the end of the chain of translations, so it expects a Python string object as input and outputs a date object. `DateField.pytocl` does the opposite and inputs a date object and is expected to output a Python string object.

Parameters

- **default** – A default value (or function) to be used when instantiating a `Predicate` or `ComplexTerm` object. If a Python `callable` object is specified (i.e., a function or functor) then it will be called (with no arguments) when the predicate/complex-term object is instantiated.
- **index** (*bool*) – Determine if this field should be indexed by default in a `FactBase`. Defaults to `False`.

abstract static cltopy(*v*)

Called when translating data from Clingo to Python

abstract static pytocl(*v*)

Called when translating data from Python to Clingo

property default

Returns the default value for the field (or `None` if no default was set).

Note: 1) if a function was specified as the default then testing `default` will call this function and return the value, 2) if your BaseField sub-class allows a default value of `None` then you need to check the `has_default` property to distinguish between no default value and a `None` default value.

property has_default

Returns whether a default value has been set

property has_default_factory

Returns whether a default value has been set

property index

Returns whether this field should be indexed by default in a *FactBase*

For sub-classes of *BaseField* the abstract member functions *cltopy()* and *pytocl()* must be implemented to provide the conversion from Clingo to Python and Python to Clingo (respectively).

Clorm provides three standard sub-classes corresponding to the string, constant, and integer primitive logical terms: *StringField*, *ConstantField*, and *IntegerField*.

class clorm.**StringField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

A field to convert between a Clingo.String object and a Python string.

class clorm.**ConstantField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

A field to convert between a simple `Clingo.Function` object and a Python string.

Note: currently `ConstantField` treats a string with a starting "-" as a negated constant. In hindsight this was a mistake and is now *deprecated*. While I don't think anyone actually used this functionality (since it was never documented) nevertheless I will keep it there until the Clorm version 2.0 release.

class clorm.**IntegerField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

A field to convert between a Clingo.Number object and a Python integer.

clorm.**field**(*basefield: Type[BaseField] | Tuple[_FieldDefinition, ...]*) → Any

clorm.**field**(*basefield: Type[BaseField] | Tuple[_FieldDefinition, ...]*, ***, *default: _T*) → _T

clorm.**field**(*basefield: Type[BaseField] | Tuple[_FieldDefinition, ...]*, ***, *default: _T*, *kw_only: bool*) → _T

clorm.**field**(*basefield: Type[BaseField] | Tuple[_FieldDefinition, ...]*, ***, *default_factory: Callable[[], _T]*) → _T

Return a field definition.

This function is used to override the type annotation field specification. A different field can be specificied as well as default values.

This function operates in a similar way to `dataclass.field()` in that it allows for more field specification options.

Special Fields

Clorm provides a number of fields for some special cases. The *SimpleField* can be used to define a field that matches to any primitive type. Note, however that special care should be taken when using *SimpleField*. While it is easy to disambiguate the Clingo to Python direction of the translation, it is harder to disambiguate between a string and constant when converting from Python to Clingo, since strings and constants are both represented using `str`. For this direction a regular expression is used to perform the disambiguation, and the user should therefore be careful not to pass strings that look like constants if the intention is to treat it like a logical string.

class clorm.**SimpleField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

A class that represents a field corresponding to any simple term: *string*, *constant*, or *integer*.

Converting from an ASP string, constant, or integer will produce the expected Python string or integer object. However, since ASP strings and constants both map to Python strings therefore converting from Python to ASP is less straightforward. In this case it uses a regular expression to determine if the string matches an ASP constant or if it should be treated as a quoted string.

Because of this potential for ambiguity it is often better to use the distinct `IntegerField`, `ConstantField`, and `StringField` classes rather than the `SimpleField` class.

A *RawField* is useful when it is necessary to match any term, whether it is a primitive term or a complex term. This encoding provides no traslation of the underlying Clingo Symbol object and simply wraps it in a special *Raw* class.

class clorm.**Raw**(*symbol: Symbol | NoSymbol*)

A wrapper around a raw `Symbol` object.

The `Raw` object is for use with a `RawField` field definition. It provides a thin wrapper around a `clingo.Symbol` (or `noclingo.Symbol`) objects. `RawField` unifies with all symbols as it doesn't try to access any of the underlying structure of the symbol object. Instead the symbol object is wrapped in a `Raw` object. To access the underlying `clingo.Symbol` object simply access the read-only `symbol` property.

property symbol

Access the underlying clingo.Symbol object``.

class clorm.**RawField**(*default: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*, *index: ~typing.Any = <clorm.orm.core._MISSING_TYPE object>*)

A field that unifies with any raw symbol.

`RawField` unifies with all symbols as it doesn't try to access any of the underlying structure of the symbol object. Instead the symbol object is wrapped in a `Raw` object. To access the underlying `clingo.Symbol` object simply access the read-only `symbol` property of the `Raw` object.

Finally, there are a number of function generators that can be used to define some special cases; refining or combining existing fields or allowing for a complicated pattern of fields. While Clorm doesn't explicitly allow recursive terms to be defined it does provide a number of encodings of list like terms. Note, it is sometimes possible to avoid the explicit use of these functions and instead to rely on some predefined type annotation mappings.

clorm.**refine_field**(*field_class: Type[_BF]*, *values: Iterable[_T] | Callable[[_T], bool]*, ***, *name: str | None = None*) → Type[_BF]

Factory function that returns a field sub-class with restricted values.

A helper factory function to define a sub-class of a BaseField (or sub-class) that restricts the allowable values. For example, if you have a constant in a predicate that is restricted to the days of the week ("monday", ..., "sunday"), you then want the Python code to respect that restriction and throw an error if the user enters the wrong value (e.g. a spelling error such as "wednsday"). Restrictions are also useful for unification if you want to unify based on some specific value.

Example

```
WorkDayField = refine_field(ConstantField,
   ["monday", "tuesday", "wednesday", "thursday", "friday"])

class WorksOn(Predicate):
    employee = ConstantField()
    workday = WorkdDayField()
```

Instead of a passing a list of values the second parameter can also be a function/functor. If is parameter is callable then it is treated as a function that takes a field value and returns true if it is a valid value.

Example

```
PosIntField = refine_field(NumberField,lambda x : x >= 0)
```

The function must be called using positional arguments with either 2 or 3 arguments. For the 3 argument case a class name is specified for the name of the new field. For the 2 argument case an anonymous field class name is automatically generated.

Example

```
WorkDayField = refine_field(ConstantField,
   ["monday", "tuesday", "wednesday", "thursday", "friday"])
```

Positional argument:

field_class: the field that is being sub-classed

values|functor: a list of values or a functor to determine validity

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

clorm.**define_enum_field**(*parent_field: Type[*BaseField*]*, *enum_class: Type[Enum]*, *, *name: str = ''*) → Type[*BaseField*]

Factory function that returns a BaseField sub-class for an Enum

Enums are part of the standard library since Python 3.4. This method provides an alternative to using refine_field() to provide a restricted set of allowable values.

Example

```
class IO(ConstantStr,Enum):
    IN="in"
    OUT="out"

# A field that unifies against ASP constants "in" and "out"
IOField = define_enum_field(ConstantField,IO)

# Note, when specified within a Predicate it is possible to avoid calling this
# function directly and the predicate class should simply reference the ``IO`` type
# annotation.

class X(Predicate):
   x: IO
```

Positional argument:

field_class: the field that is being sub-classed

enum_class: the Enum class

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

clorm.**combine_fields**(*fields: Sequence[Type[BaseField]]*, *, *name: str = ''*) → Type[*BaseField*]

Factory function that returns a field sub-class that combines other fields

A helper factory function to define a sub-class of BaseField that combines other BaseField subclasses. The subclass is defined such that it's `pytocl()` (respectively `cltopy()`) function tries to return the value returned by the underlying sub-field's `pytocl()` ((respectively `cltopy()`) function. If the first sub-field fails then the second is called, and so on until there are no matching sub-fields. If there is no match then a TypeError is raised.

Example

```
MixedField = combine_fields([ConstantField,IntegerField])
```

Positional args:

field_subclasses: the fields to combine

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

clorm.**define_nested_list_field**(*element_field: Type[BaseField]*, *, *headlist: bool = True*, *reverse: bool = False*, *name: str = ''*) → Type[*BaseField*]

Factory function that returns a BaseField sub-class for nested lists

This function is a helper factory function to define a sub-class of BaseField that can covert a list of elements to/from ASP.

ASP doesn't have an explicit notion of a sequence or list, but sometimes it is useful to encode a list as a series of nested pairs. There are two basic ways to do this, with each way having two sub-forms:

(head,list) - the list is encoded recursively with a first head element and a second element representing the remainder of the list. The end of the list is indicated by an empty tuple.

Example:

```
(1,(2,(3,())))          % Encodes a sequence (1,2,3)
```

The sub-form is for the list to be treated as reversed in the ASP encoding.

Example:

```
(3,(2,(1,())))          % Encodes a sequence (1,2,3)
```

(list,tail) - the list is encoded recursively as a sub-list first element and a second tail element. The empty sub-list is indicated by an empty tuple.

Example:

```
((((),1),2),3)          % Encodes a sequence (1,2,3)
```

Again the sub-form version is to reverse the list.

```
((((),3),2),1)          % Encodes a sequence (1,2,3)
```

The choice of nested list encodings will depend on how it is used within the ASP code. However, the head-list-pair approach in non-reverse order is also used in *lisp* and *prolog* so for this reason it is set as the default approach here.

Note: the fields of facts should be immutable. This means that you must use a tuple and not a list object when creating the sequence.

Example

```
# Unifies against a nested sequence of constants
NestedListField = define_nested_list_field(ConstantField,name="NLField")
```

Positional args:

element_field: the field type for each sequence element

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

headlist: use head-list encoding (default: True)

reverse: use the reverse order for the list (default: False)

clorm.**define_flat_list_field**(*element_field: Type[*BaseField*]*, *, *name: str = ''*) → Type[*BaseField*]

Factory function that returns a BaseField sub-class for flat lists

This function is a helper factory function to define a sub-class of BaseField to can covert a list/tuple of elements to and from arbitrary length ASP tuples.

Note: this is different to defining a fixed-length tuple in Clorm as a sub-class of *clorm.Predicate*. The elements of a predicate can be can be part of a query search. In contrast the variable length tuple defined by this function provide a more straightforward mapping but doesn't allow for individual elements of the tuple to be referenced in a query.

Example

```
# Unifies against a flat sequence of constants
FlatListField = define_flat_list_field(ConstantField)
```

Positional args:

element_field: the field type for each sequence element

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

1.10.2 Predicates and complex terms

In logical terminology predicates and terms are both considered *non logical symbols*; where the logical symbols are the operator symbols for *conjunction*, *negation*, *implication*, etc. While simple terms (constants, strings, and integers) are handled by Clorm as special cases, complex terms and predicates are both encapsulated in the `Predicate` class, with `ComplexTerm` simply being an alias to this class.

class clorm.**Predicate**(**args*, ***kwargs*)

Encapsulates an ASP predicate or complex term in an easy to access object.

This is the heart of the ORM model for defining the mapping of a complex term or predicate to a Python object. `ComplexTerm` is simply an alias for `Predicate`.

Example

```
class Booking(Predicate):
    date: str
    time: str
    name: str = field(StringField, default="relax")

b1 = Booking("20190101", "10:00")
b2 = Booking("20190101", "11:00", "Dinner")
```

Field names can be any valid Python variable name subject to the following restrictions:

- it cannot start with a "_", or
- it cannot be be one of the following reserved words: "meta", "raw", "clone", or "Field".

The constructor creates a predicate instance (i.e., a *fact*) or complex term.

Note: Using the `raw` parameter is no longer supported. You should use the *unify()* function or *Unifier* class instead.

Parameters

****kwargs** –

- if a single named parameter `raw` is specified then it will try to unify the parameter with the specification, or
- named parameters corresponding to the field names.

Field

A BaseField sub-class corresponding to a field definition for this class.

meta

The meta data (definitional information) for the Predicate. It contains:

name

The name of the ASP predicate/complex-term. Empty if it is a tuple.

is_tuple

Is the ASP predicate/complex-term a tuple.

arity

Arity of the predicate/complex-term.

clorm.**ComplexTerm**

alias of *Predicate*

clorm.**simple_predicate**(*predicate_name: str*, *arity: int*, *, *name: str = ''*, *module: str = ''*) → Type[*Predicate*]

Factory function to define a predicate with only RawField arguments.

A helper factory function that takes a name and an arity and returns a predicate class that is suitable for unifying with predicate instances of that name and arity. It's parameters are all specified as RawFields.

This function is useful for debugging ASP programs. There may be some auxillary predicates that you aren't interested in extracting their values but instead you simply want to print them to the screen in some order.

The function must be called using positional arguments with either 2 or 3 arguments. For the 3 argument case a class name is specified for the name of the new predicate. For the 2 argument case an anonymous predicate class name is automatically generated.

Positional argument:

predicate_name: the name of the ASP predicate to match against

arity: the arity for the ASP predicate

Optional keyword-only arguments:

name: name for new class (default: anonymously generated).

1.10.3 Fact Bases and Queries

`Predicate` instances correspond to facts. A `FactBase` provides a container for storing facts. It allows predicate fields to be indexed and provides a basic query mechanism for accessing elements.

class clorm.**FactBase**(*facts: Iterable[Predicate] | Callable[[], Iterable[Predicate]] | None = None*, *indexes: Iterable[PredicatePath] | None = None*)

A fact base is a container for facts (i.e., Predicate sub-class instances)

`FactBase` can be behave like a specialised `set` object, but can also behave like a minimalist database. It stores facts for `Predicate` types (where a predicate type loosely corresponds to a *table* in a database) and allows for certain fields to be indexed in order to perform more efficient queries.

The initaliser can be given a collection of predicates. If it is passed another FactBase then it simply makes a copy (including the indexed fields).

FactBase also has a special mode when it is passed a functor instead of a collection. In this case it performs a delayed initialisation. This means that the internal data structures are only populated when the FactBase is actually used. This mode is particularly useful when extracting facts from models. Often a program will only want to keep the data from the final model (for example, with optimisation we often want the best model before a timeout). Delayed initialisation is useful will save computation as only the last model will be properly initialised.

Parameters

- **facts** (*[Predicate]*|*FactBase*|*callable*) – a list of facts (predicate instances), a fact base, or a functor that generates a list of facts. If a functor is passed then the fact base performs a delayed initialisation. If a fact base is passed and no index is specified then an index will be created matching in input fact base.
- **indexes** (*Field*) – a list of fields that are to be indexed.

add(*arg:* Predicate | *Iterable[Predicate]*) → None

Add a single fact or a collection of facts.

Because a `FactBase` can only hold *Predicate* sub-class instances this member function has been overloaded to take either a single *Predicate* sub-class instance or a collection of *Predicate* sub-class instances.

Parameters

arg – a single fact or a collection of facts.

asp_str(*, *width: int = 0*, *commented: bool = False*, *sorted: bool = False*) → str

Return a ASP string representation of the fact base.

The generated ASP string representation is syntactically correct ASP code so is suitable for adding as the input to to an ASP program (or writing to a file for later use in an ASP program).

By default the order of the facts in the string is arbitrary. Because *FactBase* is built on a *OrderedDict* (which preserves insertion order) the order of the facts will be deterministic between runs of the same program. However two FactBases containing the same facts but constructed in different ways will not produce the same output string. In order to guarantee the same output the *sorted* flag can be specified.

Parameters

- **width** – tries to fill to a given width by putting more than one fact on a line if necessary (default: 0).
- **commented** – produces commented ASP code by adding a predicate signature and turning the Predicate sub-class docstring into a ASP comments (default: False).
- **sorted** – sort the output facts, first by predicates (name,arity) and then by the natural order of the instances for that predicate (default :False).

clear()

Clear the fact base of all facts.

copy() → *FactBase*

Implements the set copy() function

difference(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → *FactBase*

Implements the set difference() function

difference_update(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → None

Implements the set difference_update() function

discard(*arg:* Predicate) → None

Remove a fact from the fact base.

facts() → List[*Predicate*]

Return all facts.

intersection(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → *FactBase*

Implements the set intersection() function

intersection_update(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → None

Implements the set intersection_update() function

pop() → *Predicate*

Pop an element from the FactBase.

query(*__ent0: Type[_T0]*) → UnGroupedQuery[_T0]

query(*__ent0: Type[_T0]*, *__ent1: Type[_T1]*) → UnGroupedQuery[Tuple[_T0, _T1]]

query(*__ent0: Type[_T0]*, *__ent1: Type[_T1]*, *__ent2: Type[_T2]*) → UnGroupedQuery[Tuple[_T0, _T1, _T2]]

query(*__ent0: Type[_T0]*, *__ent1: Type[_T1]*, *__ent2: Type[_T2]*, *__ent3: Type[_T3]*) → UnGroupedQuery[Tuple[_T0, _T1, _T2, _T3]]

query(*__ent0: Type[_T0]*, *__ent1: Type[_T1]*, *__ent2: Type[_T2]*, *__ent3: Type[_T3]*, *__ent4: Type[_T4]*) → UnGroupedQuery[Tuple[_T0, _T1, _T2, _T3, _T4]]

query(**roots: Any*) → UnGroupedQuery[Any]

Define a query using the new Query API *Query*.

The parameters consist of a predicates (or aliases) to query (like an SQL FROM clause).

Parameters
***predicates** – predicate or predicate aliases

Returns
Returns a Query object for specifying a query.

remove(*arg:* Predicate) → None

Remove a fact from the fact base (raises an exception if no fact).

select(*root*)

Define a select query using the old Query API.

Note: This interface will eventually be deprecated when the new *Query API* is finalised. The entry point to this Query API is through the *FactBase.query()* method.

Parameters
predicate – The predicate to query.

Returns
Returns a Select query object for specifying a query.

symmetric_difference(*other: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → *FactBase*

Implements the set symmetric_difference() function

symmetric_difference_update(*other: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → None

Implements the set symmetric_difference_update() function

union(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → *FactBase*

Implements the set union() function

update(**others: Iterable[*Predicate*] | Callable[[], Iterable[*Predicate*]]*) → None

Implements the set update() function

property predicates: Tuple[Type[*Predicate*], ...]

Return the list of predicate types that this fact base contains.

A FactBase can generate formatted ASP facts using the function *FactBase.add()*. This string of facts can be passed to the solver or written to a file to be read. Mirroring this functionality an ASP string or file containing facts can also be read directly into a FactBase (without the indirect process of having to create a clingo.Control object).

clorm.**parse_fact_string**(*aspstr: str*, *unifier: Iterable[Type[*Predicate*]]*, ***, *factbase:* FactBase *| None = None*, *raise_nomatch: bool = False*, *raise_nonfact: bool = False*) → *FactBase*

Parse a string of ASP facts into a FactBase

Facts must be of a simple form that can correspond to clorm predicate instances. Rules that are NOT simple facts include: any rule with a body, a disjunctive fact, a choice rule, a theory atom, a literal with an external @-function reference, a literal that requires some mathematical calculation (eg., "p(1+1).")

NOTE: Currently, this function is not safe when running in NOCLINGO mode and will raise a NotImplementedError if called.

Parameters

- **aspstr** – an ASP string containing the facts
- **factbase** – if no factbase is specified then create a new one
- **unifier** – a list of clorm.Predicate classes to unify against
- **raise_nomatch** – raise UnifierNoMatchError on a fact that cannot unify
- **raise_nonfact** – raise FactParserError on any non simple fact (eg. complex rules)

clorm.**parse_fact_files**(*files: Sequence[str]*, *unifier: Iterable[Type[*Predicate*]]*, ***, *factbase:* FactBase *| None = None*, *raise_nomatch: bool = False*, *raise_nonfact: bool = False*) → *FactBase*

Parse the facts from a list of files into a FactBase

Facts must be of a simple form that can correspond to clorm predicate instances. Rules that are NOT simple facts include: any rule with a body, a disjunctive fact, a choice rule, a theory atom, a literal with an external @-function reference, a literal that requires some mathematical calculation (eg., "p(1+1).")

NOTE: Currently, this function is not safe when running in NOCLINGO mode and will raise a NotImplementedError if called.

Parameters

- **files** – a list of ASP files containing the facts
- **factbase** – if no factbase is specified then create a new one
- **unifier** – a list of clorm.Predicate classes to unify against
- **raise_nomatch** – raise UnifierNoMatchError on a fact that cannot unify
- **raise_nonfact** – raise FactParserError on any non simple fact (eg. complex rules)

One of the more important features of a `FactBase` is its ability to be queried. There are a number of classes and functions to support the specification of fact base queries.

class clorm.**Placeholder**

An abstract class for defining parameterised queries.

Currently, Clorm supports 4 placeholders: ph1_, ph2_, ph3_, ph4_. These correspond to the positional arguments of the query execute function call.

class clorm.**Select**

An abstract class that defines the interface to original Query API.

Note: This interface will eventually be deprecated when the new *Query API* is finalised.

`Select` query objects cannot be constructed directly. Instead a `Select` object is returned by the *FactBase.select()* function. Given a `FactBase` object `fb`, a specification is of the form:

```
query = fb.select(<predicate>).where(<expression>).order_by(<ordering>)
```

where <predicate> specifies the predicate type to search for, <expression> specifies the search criteria and <ordering> specifies a sort order when returning the results. The `where()` and `order_by()` clauses are omitted when not required.

abstract where(**expressions*)

Set the select statement's where clause.

The where clause consists of a set of boolean and comparison expressions. This expression specifies a search criteria for matching facts within the corresponding `FactBase`.

Boolean expression are built from other boolean expression or a comparison expression. Comparison expressions are of the form:

```
<PredicatePath> <compop> <value>
```

where <compop> is a comparison operator such as ==, !=, or <= and <value> is either a Python value or another predicate path object refering to a field of the same predicate or a placeholder.

A placeholder is a special value that issubstituted when the query is actually executed. These placeholders are named `ph1_`, `ph2_`, `ph3_`, and `ph4_` and correspond to the 1st to 4th arguments of the `get`, `get_unique` or `count` function call.

Args:
expressions: one or more comparison expressions.

Returns:
Returns a reference to itself.

abstract order_by(**fieldorder*)

Provide an ordering over the results.

Parameters
fieldorder – an ordering over fields

Returns
Returns a reference to itself.

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

Return all matching entries.

get_unique(**args*, ***kwargs*)

Return the unique matching entry (or raise an exception)

count(**args*, ***kwargs*)

Return the number of matches.

class clorm.**Delete**

An abstract class that defines the interface to a original delete query API.

Note: This interface will eventually be deprecated when the new *Query API* is finalised.

`Delete` query objects cannot be constructed directly. Instead a `Delete` object is returned by the `FactBase.delete()` function. Given a `FactBase` object `fb`, a specification is of the form:

```
query = fb.delete(<predicate>).where(<expression>)
```

where <predicate> specifies the predicate type to search for, <expression> specifies the search criteria. The `where()` clause can be omitted in which case all predicates of that type will be deleted.

abstract where(**expressions*)

Set the select statement's where clause.

See the documentation for `Select.where()` for further details.

abstract execute(**args*, ***kwargs*)

Function to execute the delete query

class clorm.**Query**

An abstract class that defines the interface to the Clorm Query API v2.

Note: This new Query API replaces the old Select/Delete mechanism and offers many more features than the old API, especially allowing joins similar to an SQL join between tables.

This interface is complete and unlikely to change - however it is being left open for the moment in case there is strong user feedback.

`Query` objects cannot be constructed directly.

Instead a `Query` object is returned by the *FactBase.query()* function. Queries can take a number of different forms but contain many of the components of a traditional SQL query. A predicate definition (as opposed to a predicate instance or fact) can be viewed as an SQL table and the parameters of a predicate can be viewed as the fields of the table.

The simplest query must at least specify the predicate(s) to search for. This is specified as parameters to the *FactBase.query()* function. Relating this to a traditional SQL query, the `query` clause can be viewed as an SQL `FROM` specification.

The query is typicaly executed by iterating over the generator returned by the *Query.all()* end-point.

```
from clorm import FactBase, Predicate, IntegerField, StringField

class Option(Predicate):
    oid = IntegerField
    name = StringField
    cost = IntegerField
    cat = StringField

class Chosen(Predicate):
    oid = IntegerField

fb=FactBase([Option(1,"Do A",200,"foo"),Option(2,"Do B",300,"bar"),
             Option(3,"Do C",400,"foo"),Option(4,"Do D",300,"bar"),
             Option(5,"Do E",200,"foo"),Option(6,"Do F",500,"bar"),
             Chosen(1),Chosen(3),Chosen(4),Chosen(6)])

q1 = fb.query(Chosen)     # Select all Chosen instances
result = set(q1.all())
assert result == set([Chosen(1),Chosen(3),Chosen(4),Chosen(6)])
```

If there are multiple predicates involved in the search then the query must also contain a *Query.join()* clause to specify the predicates parameters/fields to join.

```
q2 = fb.query(Option,Chosen).join(Option.oid == Chosen.oid)
```

Note: As an aside, while a `query` clause typically consists of predicates, it can also contain predicate *aliases* created through the *alias()* function. This allows for queries with self joins to be specified.

When a query contains multiple predicates the result will consist of tuples, where each tuple contains the facts matching the signature of predicates in the `query` clause. Mathematically the tuples are a subset of the cross-product over instances of the predicates; where the subset is determined by the `join` clause.

```
result = set(q2.all())

assert result == set([(Option(1,"Do A",200,"foo"),Chosen(1)),
                      (Option(3,"Do C",400,"foo"),Chosen(3)),
                      (Option(4,"Do D",300,"bar"),Chosen(4)),
                      (Option(6,"Do F",500,"bar"),Chosen(6))])
```

A query can also contain a `where` clause as well as an `order_by` clause. When the `order_by` clause contains a predicate path then by default it is ordered in ascending order. However, this can be changed to descending order with the `desc()` function modifier.

```
from clorm import desc

q3 = q2.where(Option.cost > 200).order_by(desc(Option.cost))
```

(continues on next page)

(continued from previous page)

```
result = list(q3.all())
assert result == [(Option(6,"Do F",500,"bar"),Chosen(6)),
                  (Option(3,"Do C",400,"foo"),Chosen(3)),
                  (Option(4,"Do D",300,"bar"),Chosen(4))]
```

The above code snippet highlights a feature of the query construction process. Namely, that these query construction functions can be chained and can also be used as the starting point for another query. Each construction function returns a modified copy of its parent. So in this example query `q3` is a modified version of query `q2`.

Returning tuples of facts is often not the most convenient output format and instead you may only be interested in specific predicates or parameters within each fact tuple. For example, in this running example it is unnecessary to return the `Chosen` facts. To provide the output in a more useful format a query can also contain a `select` clause that specifies the items to return. Essentially, this specifies a _projection_ over the elements of the result tuple.

```
q4 = q3.select(Option)

result = list(q4.all())
assert result == [Option(6,"Do F",500,"bar"),
                  Option(3,"Do C",400,"foo"),
                  Option(4,"Do D",300,"bar")]
```

A second mechanism for accessing the data in a more convenient format is to use a `group_by` clause. In this example, we may want to aggregate all the chosen options, for example to sum the costs, based on their membership of the `"foo"` and `"bar"` categories. The Clorm query API doesn't directly support aggregation functions, as you could do in SQL, so some additional Python code is required.

```
q5 = q2.group_by(Option.cat).select(Option.cost)

result = [(cat, sum(list(it))) for cat, it in q5.all()]
assert result == [("bar",800), ("foo", 600)]
```

The above are not the only options for a query. Some other query modifies include: *Query.distinct()* to return distinct elements, and *Query.bind()* to bind the value of any placeholders in the `where` clause to specific values.

A query is executed using a number of end-point functions. As already shown the main end-point is *Query.all()* to return a generator for iterating over the results.

Alternatively, if there is at least one element then to return the first result only (throwing an exception only if there are no elements) use the *Query.first()* method.

Or if there must be exactly one element (and to throw an exception otherwise) use *Query.singleton()*.

To count the elements of the result there is *Query.count()*.

Finally to delete all matching facts from the underlying FactBase use *Query.delete()*.

abstract join(**expressions*)

Specifying how the predicate/tables in the query are to be joined.

Joins are expressions that connect the predicates/tables of the query. They range from a pure SQL-like inner-join through to an unrestricted cross-product. The standard form is:

```
<PredicatePath> <compop> <PredicatePath>
```

with the full cross-product expressed using a function:

```
cross(<PredicatePath>,<PredicatePath>)
```

Every predicate/table in the query must be reachable to every other predicate/table through some form of join. For example, given predicate definitions F, G, H, each with a field anum:

```
query = fb.query(F,G,H).join(F.anum == G.anum,cross(F,H))
```

generates an inner join between F and G, but a full cross-product between F and H.

Finally, it is possible to perform self joins using the function alias that generates an alias for the predicate/table. For .. rubric:: Example

```
from clorm import alias

FA=alias(F) query = fb.query(F,G,FA).join(F.anum == FA.anum,cross(F,G))
```

generates an inner join between F and itself, and a full cross-product between F and G.

Parameters
expressions – one or more join expressions.

Returns
Returns the modified copy of the query.

abstract where(**expressions*)

Sets a list of query conditions.

The where clause consists of a single (or list) of simple/complex boolean and comparison expressions. This expression specifies a search criteria for matching facts within the corresponding FactBase.

Boolean expression are built from other boolean expression or a comparison expression. Comparison expressions are of the form:

```
<PredicatePath> <compop> <value>
```

where <compop> is a comparison operator such as ==, !=, or <= and <value> is either a Python value or another predicate path object refering to a field of the same predicate or a placeholder.

A placeholder is a special value that allows queries to be parameterised. A value can be bound to each placeholder. These placeholders are named ph1_, ph2_, ph3_, and ph4_ and correspond to the 1st to 4th arguments when the bind() function is called. Placeholders also allow for named arguments using the "ph_("<name>") function.

Args:
expressions: one or more comparison expressions.

Returns:
Returns the modified copy of the query.

abstract order_by(**expressions*)

Specify an ordering over the results.

Parameters
field_order – an ordering over fields

Returns
Returns the modified copy of the query.

abstract group_by(**expressions*)

Specify a grouping over the results.

The grouping specification is similar to an ordering specification but it modifies the behaviour of the query to return a pair of elements, where the first element of the pair is the group identifier (based on the specification) and the second element is an iterator over the matching elements.

When both a `group_by` and `order_by` clause is provided the `order_by` clause is used to sort the elements within each matching group.

Parameters
field_order – an ordering over fields to group by

Returns
Returns the modified copy of the query.

abstract select(**outsig*)

Provides a projection over the query result tuples.

Mathematically the result tuples of a query are the cross-product over instances of the predicates. However, returning tuples of facts is often not the most convenient output format and instead you may only be interested in specific parameters/fields within each tuple of facts. The `select` clause specifies a _projection_ over each query result tuple.

Each query result tuple is guaranteed to be distinct, since the query is a filtering over the cross-product of the predicate instances. However, the specification of a projection can result in information being discarded and can therefore cause the projected query results to no longer be distinct. To enforce uniqueness the *Query.distinct()* flag can be specified. Essentially this is the same as an `SQL SELECT DISTINCT ...` statement.

Note: when *Query.select()* is used with the *Query.delete()* end-point the *select* signature must specify predicates and not parameter/fields within the predicates.

Finally, instead of a projection specification, a single Python *callable* object can be specified with input parameters matching the query signature. This callable object will then be called on each query tuple and the results will make up the modified query result. This provides the greatest flexibility in controlling the output of the query.

Parameters
output_signature – the signature that defines the projection or a callable object.

Returns
Returns the modified copy of the query.

abstract distinct()

Return only distinct elements in the query.

This flag is only meaningful when combined with a *Query.select()* clause that removes distinguishing elements from the tuples.

Returns
Returns the modified copy of the query.

abstract bind(**args*, ***kwargs*)

Bind placeholders to specific values.

If the `where` clause has placeholders then these placeholders must be bound to actual values before the query can be executed.

Parameters

- ***args** – positional arguments corresponding to positional placeholders
- ****kwargs** – named arguments corresponding to named placeholders

Returns
Returns the modified copy of the query.

abstract tuple()

Force returning a tuple even for singleton elements.

In the general case the output signature of a query is a tuple; consisting either of a tuple of facts or a tuple of parameters/fields if a `select` projection has been specified.

However, if the output signature is a singleton tuple then by default the API changes its behaviour and removes the tuple, returning only the element itself. This typically provides a much more useful and intutive interface. For example, if you want to perform a sum aggregation over the results of a query, aggregating over the value of a specific parameter/field, then specifying just that parameter in the `select` clause allows you to simply pass the query generator to the standard Python `sum()` function without needing to perform a list comprehension to extract the value to be aggregated.

If there is a case where this default behaviour is not wanted then specifying the `tuple` flag forces the query to always return a tuple of elements even if the output signature is a singleton tuple.

Returns
Returns the modified copy of the query.

abstract heuristic(*join_order*)

Allows the query engine's query plan to be modified.

This is an advanced option that can be used if the query is not performing as expected. For multi-predicate queries the order in which the joins are performed can affect performance. By default the Query API will try to optimise this join order based on the `join` expressions; with predicates with more restricted joins being higher up in the join order.

This join order can be controlled explicitly by the `fixed_join_order` heuristic function. Assuming predicate definitions `F` and `G` the query:

```
from clorm import fixed_join_order

query=fb.query(F,G).heuristic(fixed_join_order(G,F)).join(...)
```

forces the join order to first be the `G` predicate followed by the `F` predicate.

Parameters
join_order – the join order heuristic

Returns
Returns the modified copy of the query.

abstract all()

Returns a generator that iteratively executes the query.

Note. This call doesn't execute the query itself. The query is executed when iterating over the elements from the generator.

Returns
Returns a generator that executes the query.

abstract singleton()

Return the single matching element.

An exception is thrown if there is not exactly one matching element or a `group_by()` clause has been specified.

Returns
Returns the single matching element (or throws an exception)

abstract count()

Return the number of matching element.

Typically the number of elements consist of the number of tuples produced by the cross-product of the predicates that match the criteria of the `join()` and `where()` clauses.

However, if a `select()` projection and `unique()` flag is specified then the `count()` will reflect the modified the number of unique elements based on the projection of the query.

Furthermore, if a `group_by()` clause is specified then `count()` returns a generator that iterates over pairs where the first element of the pair is the group identifier and the second element is the number of matching elements within that group.

Returns
Returns the number of matching elements

abstract first()

Return the first matching element.

An exception is thrown if there are no one matching element or a `group_by()` clause has been specified.

Returns
Returns the first matching element (or throws an exception)

abstract delete()

Delete matching facts from the `FactBase()`.

In the simple case of a query with no joins then `delete()` simply deletes the matching facts. If there is a join then the matches consist of tuples of facts. In this case `delete()` will remove all facts from the tuple. This behaviour can be modified by using a `select()` projection clause that selects only specific predicates. Note: when combined with `select()` the output signature must specify predicates and not parameter/fields within predicates.

An exception is thrown if a `group_by()` clause has been specified.

Returns
Returns the number of facts deleted.

abstract modify(*fn*)

Modify matching facts in a `FactBase()`.

Clorm facts are immutable, so in order to simulate something like an SQL `UPDATE` operation, the `modify()` query end-point provides a convenient way to replace or modify matching facts in a query with some (related) facts.

This call operates in the same way as the `delete()` end-point but the matching facts are first passed to the parameter function `fn`. This function must return a pair consisting of 1) the facts to delete from the factbase, 2) the facts to add to the fact base. These two sets of facts are maintained until the end of the query and the delete set of facts are removed followed by the add set being added.

Note, this behaviour could also achieved by iterating over the results of the query normally and maintaining separate "delete" and "add" lists which are then acted upon at the end of the query. The advantage of the `modify()` call is to provide a simple and declarative way to do it which can be convenient especially when chaining multiple modifications of a factbase.

The function `fn` must have the same input signature as the query output. If the output of `fn` must be a pair consisting of the elements to delete and element to add. For flexibility, each set can be `None` or a single clorm fact or an iterator over clorm facts.

An exception is thrown if a `group_by()` or ``uclause has been specified.

Example assuming a FactBase `fb` consisting of `F` and `G` predicate facts:

def mod_fn(f: F, g: G):
return ({f, g}, {f.clone(a=10), g.clone(a=10)})

fb.query(F,G).join(F.a == G.a).where(F.a == 2).modify(mod_fn)

fb.query(F,G).join(F.a == G.a).where(F.a == 10).select(F).modify(lambda f: (None, {f.clone(a=20)}))

Returns
Returns a (delete_count, add_count) pair indicating the total number of facts deleted and added to the factbase.

abstract replace(*fn*)

Replace matching facts in a `FactBase()`.

This function is a convenient special case of the more general `modify()` function.

While the `modify()` parameter function must return a del-add pair, the `replace()` parameter function returns only an add entry, since all matching facts will be deleted.

Returns
Returns a (delete_count, add_count) pair indicating the total number of facts deleted and added to the factbase.

abstract query_plan(**args*, ***kwargs*)

Return a query plan object outlining the query execution.

A query plan outlines the query will be executed; the order of table joins, the searches based on indexing, and how the sorting is performed. This is useful for debugging if the query is not behaving as expected.

Currently, there is no fixed specification for the query plan object. All the user can do is display it for reading and debugging purposes.

Returns
Returns a query plan object that can be stringified.

class clorm.**PredicatePath**(*pathseq: List[PathIdentity | str]*)

PredicatePath implements the intuitive query syntax.

Every defined *Predicate* sub-class has a corresponding *PredicatePath* sub-class that mirrors its field definitions. This allows it to be used when specifying the components of a query; such as the sign and the fields or sub-fields of a predicate (eg., `Pred.sign`, `Pred.a.b` or `Pred.a[0]`).

When the API user refers to a field (or sign) of a Predicate sub-class they are redirected to the corresponding *PredicatePath* object of that predicate sub-class.

While instances of this class (and sub-classes) are externally exposed through the API, users should not explicitly instantiate instances themselves.

Predicate path subclasses provide attributes and indexed items for refering to sub-paths. When a user specifies `Pred.a.b.c` the *Predicate* sub-class `Pred` seemlessly passes off to an associated *PredicatePath* object, which then returns a path corresponding to the specifications.

Fields can be specified either by name through a chain of attributes or using the array indexes. This is implemented in the overloaded `__getitem__` function which allows for name or positional argument specifications.

The most important aspect of a predicate path object is that it overloads the boolean operators to return a comparison condition. This is what allows for query specifications such as `Pred.a.b == 2` or `Pred.a.b == ph1_`.

Finally, because the name `meta` is a Clorm keyword and can't be used as a field name it is used as a property referring to an internal class with functions for use by the internals of the library. API users should not use this property.

1.10.4 Query Support Functions

The following functions support the query specification.

`clorm.path`(*arg*, *exception=True*)

Returns the *PredicatePath* corresponding to some component.

This function is useful for users for the special case of referring to the *PredicatePath* that corresponding to a *Predicate* object. For example to specify a comparison in a query to match a specific instance to some placeholder you need to reference the predicate using a path.

Example:

```
from clorm import FactBase, Predicate, ConstantField, path

class F(Predicate):
   a = ConstantField

fb = FactBase([F("foo"),F("bar")])

qBad=fb.query(F).where(F == F("bar"))    # This won't do what you expect

qGood=fb.query(F).where(path(F) == F("bar"))
```

Note: The technical reason for not supporting the more intuitive syntax above is that it would require overloading the comparison operators of the predicate class itself; which would break the behaviour of Python in many other contexts.

Returns

Returns a *PredicatePath* object corresponding to the input specification.

`clorm.alias`(*predicate: Type[_P]*, *name: str = ''*) → Type[_P]

`clorm.alias`(*predicate:* PredicatePath | *Hashable*, *name: str = ''*) → Type[*Predicate*]

Return an alias *PredicatePath* instance for a *Predicate* sub-class.

A predicate alias can be used to support self joins in queries. The alias has all the same fields (and sub-fields) as the "normal" path associated with the predicate.

For example, consider a simple (and not properly normalised) friend fact base with a predicate that uniquely identifies people and friends, by a id number, and you want to output the friend connections in an intuitive manner.

Example

```
from clorm import FactBase, Predicate, IntegerField, StringField, alias

class F(Predicate):
    pid = IntegerField
    name = StringField
    fid = IntegerField

fb=FactBase([F(1,"Adam",3),F(2,"Betty",4),F(3,"Carol",1),F(4,"Dan",2)])

FA = alias(F)
q=fb.query(F,FA).join(F.pid == FA.fid).select(F.name,FA.name)

for p,f in q.all():
    print("Person {} => Friend {}".format(p,f))
```

Returns
Returns an alias *PredicatePath* for the predicate.

clorm.**cross**(**args*)

Return a cross-product join condition

clorm.**ph_**(*value*, **args*, ***kwargs*)

A function for building new placeholders, either named or positional.

clorm.**not_**(**conditions*)

Return a boolean condition that is the negation of the input condition

clorm.**and_**(**conditions*)

Return a the conjunction of two of more conditions

clorm.**or_**(**conditions*)

Return the disjunction of two of more conditions

clorm.**in_**(*path*, *seq*)

Return a query operator to test membership of an item in a collection

clorm.**notin_**(*path*, *seq*)

Return a query operator to test non-membership of an item in a collection

1.10.5 Calling Python From an ASP Program

Clorm provides a number of decorators that can make it easier to call Python from within an ASP program. The basic idea is that Clorm provides all the information required to convert data between native Python types and clingo.Symbol objects. Therefore functions can be written by only dealing with Python data and the Clorm decorators will wrap these functions with the appropriate data conversions based on a given signature.

clorm.**make_function_asp_callable**(**args: Any*) → Callable[[...], Any]

A decorator for making a function callable from within an ASP program.

Can be called in a number of ways. Can be called as a decorator with or without arguments. If called with arguments then the arguments must correspond to a *type cast signature*.

A *type cast signature* specifies the type conversions required between a python function that is called from within an ASP program and a set of corresponding Python types.

A type cast signature is specified in terms of the fields that are used to define a predicate. It is a list of elements where the first n-1 elements correspond to type conversions for a functions inputs and the last element corresponds to the type conversion for a functions output.

Parameters

- **sigs** (**sigs*) – A list of function signature elements.
- **[** (*- Inputs. Match the sub-elements*) – -1] define the input signature while the last element defines the output signature. Each input must be a a BaseField (or sub-class).
- **Output** (-) – Must be BaseField (or sub-class) or a singleton list containing a BaseField (or sub-class).

If no arguments are provided then the function signature is derived from the function annotations. The function annotations must conform to the signature above.

If called as a normal function with arguments then the last element must be the function to be wrapped and the previous elements conform to the signature profile.

clorm.**make_method_asp_callable**(**args: Any*) → Callable[[...], Any]

A decorator for making a member function callable from within an ASP program.

See `make_function_asp_callable` for details. The only difference is that the first element of the function is ignore as it is assumed to be the `self` or `cls` parameter.

It may also be useful to deal with a predeclared type cast signature.

class clorm.**TypeCastSignature**(**sigs: Any*, *module: str | None = None*)

Defines a signature for converting to/from Clingo data types.

Args:

module: Name of the module where the signature is defined sigs(*sigs): A list of signature elements.

- Inputs. Match the sub-elements [:-1] define the input signature while the last element defines the output signature. Each input must be a BaseField (or sub-class).
- Output: Must be BaseField (or sub-class) or a singleton list containing a BaseField (or sub-class).

Example

```
import datetime

class DateField(StringField):
          pytocl = lambda dt: dt.strftime("%Y%m%d")
          cltopy = lambda s: datetime.datetime.strptime(s,"%Y%m%d").date()

drsig = TypeCastSignature(DateField, DateField, [DateField], module = "__main__")

@drsig.wrap_function
def date_range(start, end):
    return [ start + timedelta(days=x) for x in range(0,end-start) ]
```

The function `date_range` that takes a start and end date and returns the list of dates within that range.

When *decorated* with the signature it provides the conversion code so that the decorated function expects a start and end date encoded as Clingo.String objects (matching YYYYMMDD format) and returns a list of Clingo.String objects corresponding to the dates in that range.

static is_return_element(*se: Any*) → bool

An output element must be an output field or a singleton iterable containing an output fields; where an output field is a BaseField sub-class or tuple that recursively reduces to a BaseField sub-class.

wrap_function(*fn*)

Function wrapper that adds data type conversions for wrapped function.

Parameters
fn – A function satisfing the inputs and output defined by the TypeCastSignature.

wrap_method(*fn*)

Member function wrapper that adds data type conversions for wrapped member functions.

Parameters
fn – A function satisfing the inputs and output defined by the TypeCastSignature.

From Clingo 5.4 onwards, the Clingo grounding function allows a *context* parameter to be specified. This parameter defines a context object for the methods that are called by ASP using the @-syntax.

class clorm.**ContextBuilder**

Context builder simplifies the task of building grounding context for clingo. This is a new clingo feature for Clingo 5.4 where a context can be provided to the grounding function. The context encapsulates the external Python functions that can be called from within an ASP program.

`ContextBuilder` allows arbitrary functions to be captured within a context and assigned a conversion signature. It also allows the function to be given a different name when called from within the context.

The context builder's `register` and `register_name` member functions can be called as decorators or as normal functions. A useful feature of these functions is that when called as decorators they do not wrap the original function but instead return the original function and only wrap the function when called from within the context. This is unlike the `make_function_asp_callable` and `make_method_asp_callable` functions which when called as decorators will replace the original function with the wrapped version.

Example:

The following nonsense ASP program contains embedded python with functions registered with the context builder (highlighting different ways the register functions can be called). A context object is then created by the context builder and used during grounding. It will produce the answer set:

```
f(5), g(6), h("abcd").
```

```
f(@addi(1,4)).
g(@addi_alt(2,4)).
h(@adds("ab","cd")).

#script(python).

from clorm import IntegerField,StringField,ContextBuilder

IF=IntegerField
SF=StringField
cb=ContextBuilder()
```

(continues on next page)

(continued from previous page)

```
# Uses the function annotation to define the conversion signature
@cb.register
def addi(a : IF, b : IF) -> IF : return a+b

# Register with a different name
@cb.register_name("addi_alt")
def add2(a : IF, b : IF) -> IF : return a+b

# Register with a different name and override the signature in the
# function annotation
cb.register_name("adds", SF, SF, SF, addi)

ctx=cb.make_context()

def main(prg):
    prg.ground([("base",[])],context=ctx)
    prg.solve()

#end.
```

make_context(*cls_name='Context'*)

Return a context object that encapsulates the registered functions

register(**args*)

Register a function with the context builder.

Parameters

***args** – the last argument must be the function to be registered. If there is more than one argument then the earlier arguments define the data conversion signature. If there are no earlier arguments then the signature is extracted from the function annotations.

register_name(*func_name*, **args*)

Register a function with assigning it a new name witin the context.

Parameters

- **func_name** – the new name for the function within the context.
- ***args** – the last argument must be the function to be registered. If there is more than one argument then the earlier arguments define the data conversion signature. If there are no earlier arguments then the signature is extracted from the function annotations.

1.10.6 Integration with the Solver

Clorm provides some helper functions to get clorm facts into and out of the solver (ie a *clingo.Control* object).

clorm.**control_add_facts**(*ctrl: Control*, *facts: Iterable[*Predicate | *Symbol]*) → None

Assert a collection of facts to the solver

Provides a flexible approach to asserting facts to the solver. The facts can be either *clingo.Symbol* objects or clorm facts and can be in any type of be in any collection (including a *clorm.FactBase*).

Parameters

- **ctrl** – a *clingo.Control* object

- **facts** – the collection of facts to be asserted into the solver

clorm.**symbolic_atoms_to_facts**(*symbolic_atoms: SymbolicAtoms*, *unifier: Iterable[Type[Predicate]]*, ***, *facts_only: bool = False*, *factbase:* FactBase | *None = None*) → *FactBase*

Extract *clorm.FactBase* from *clingo.SymbolicAtoms*

A *clingo.SymbolicAtoms* object is returned from the *clingo.Control.symbolic_atoms* property. This property is a view into the internal atoms within the solver and can be examined at anytime (ie. before/after the grounding/solving). Some of the atoms are trivially true (as determined by the grounder) while others may only be true in some models.

Parameters

- **symbolic_atoms** – a *clingo.SymbolicAtoms* object
- **unifier** – a list of Clorm Predicate sub-classes to unify against
- **facts_only** (*default False*) – return facts only or include contingent literals
- **factbase** (*default None*) – add to existing FactBase or return a new FactBase

clorm.**unify**(*unifier: Iterable[Type[Predicate]] | SymbolPredicateUnifier*, *symbols: Iterable[Symbol | NoSymbol]*, *ordered: Literal[True]*) → List[*Predicate*]

clorm.**unify**(*unifier: Iterable[Type[Predicate]] | SymbolPredicateUnifier*, *symbols: Iterable[Symbol | NoSymbol]*) → *FactBase*

Unify raw symbols against a list of predicates or a SymbolPredicateUnifier.

Symbols are tested against each predicate unifier until a match is found. Since it is possible to define multiple predicate types that can unify with the same symbol, the order of the predicates in the unifier matters. With the *ordered* option set to *True* a list is returned that preserves the order of the input symbols.

Parameters

- **unifier** – a list of predicate classes or a SymbolPredicateUnifier object.
- **symbols** – the symbols to unify.
- **(default** (*ordered*) – False): optional to return a list rather than a FactBase.

Returns

a FactBase containing the unified facts, indexed by any specified indexes,
or a list if the ordered option is specified

To further simplify the interaction with the solver, Clorm provides a `clingo` replacement module that offers better integration with Clorm facts and fact bases. This module simply wraps and extends a few key Clingo classes.

Instead of:

```
import clingo
```

use:

```
import clorm.clingo
```

For convenience the `clingo.Control` class can also be monkey patched so that it can used seemlessly with existing code bases.

```
from clorm import monkey; monkey.patch()
import clingo
```

Here we document only the extended classes and the user is referred to the Clingo API documentation for more details.

class clorm.clingo.**Control**(**args: Any*, ***kwargs: Any*)

Control object for the grounding/solving process.

Behaves like `clingo.Control` but with modifications to deal with Clorm facts and fact bases.

Adds an additional parameter `unifier` to specify how any generated clingo models will be unified with the clorm Predicate definitions. The unifier can be specified as a list of predicates or as a SymbolPredicateUnifier object.

An existing `clingo.Control` object can be passed using the `control_` parameter.

Control object for the grounding/solving process.

Parameters

- **arguments** – Arguments to the grounder and solver.
- **logger** – Function to intercept messages normally printed to standard error.
- **message_limit** – The maximum number of messages passed to the logger.

Notes

Note that only gringo options (without *–text*) and clasp's search options are supported. Furthermore, you must not call any functions of a *Control* object while a solve call is active.

add(**args*, ***kwargs*) → None

Extend the logic program with the given non-ground logic program in string form.

This function provides two overloads:

```python def add(self, name: str, parameters: Sequence[str], program: str) -> None:

    ...

def add(self, program: str) -> None:
    return self.add("base", [], program)

```

Parameters

- **name** – The name of program block to add.
- **parameters** – The parameters of the program block to add.
- **program** – The non-ground program in string form.

See also:

Control.ground

add_facts(*facts: Iterable[*Predicate | *Symbol]*) → None

Add facts to the control object. Note: facts must be added before grounding.

This function can take an arbitrary collection containing a mixture of `clorm.Predicate` and `clingo.Symbol` objects. A `clorm.FactBase` is also a valid collection but it can only contain `clorm.Predicate` instances.

Parameters
facts – a collection of `clorm.Predicate` or `clingo.Symbol` objects

assign_external(*external: Iterable[*Predicate | *Symbol | int]*, *truth: bool | None*) → None

Assign a truth value to an external fact (or collection of facts)

A fact can be a raw clingo.Symbol object, a clorm.Predicate instance, or a program literal (an int). If the external is a collection then the truth value is assigned to all elements in the collection.

This function extends `clingo.Control.release_external`.

Assign a truth value to an external atom.

Parameters

- **external** – A symbol or program literal representing the external atom.
- **truth** – A Boolean fixes the external to the respective truth value; and None leaves its truth value open.

See also:

Control.release_external, `clingo.solving.SolveControl.symbolic_atoms`, `clingo.symbolic_atoms.SymbolicAtom.is_external`

Notes

The truth value of an external atom can be changed before each solve call. An atom is treated as external if it has been declared using an *#external* directive, and has not been released by calling *Control.release_external* or defined in a logic program with some rule. If the given atom is not external, then the function has no effect.

For convenience, the truth assigned to atoms over negative program literals is inverted.

backend() → Backend

Returns a *Backend* object providing a low level interface to extend a logic program.

See also:

`clingo.backend`

cleanup() → None

Cleanup the domain used for grounding by incorporating information from the solver.

This function cleans up the domain used for grounding. This is done by first simplifying the current program representation (falsifying released external atoms). Afterwards, the top-level implications are used to either remove atoms from the domain or mark them as facts.

See also:

Control.enable_cleanup

Notes

Any atoms falsified are completely removed from the logic program. Hence, a definition for such an atom in a successive step introduces a fresh atom.

With the current implementation, the function only has an effect if called after solving and before any function is called that starts a new step.

Typically, it is not necessary to call this function manually because automatic cleanups are enabled by default.

get_const(*name: str*) → Symbol | None

Return the symbol for a constant definition of form:

#const name = symbol.

Parameters
name – The name of the constant to retrieve.

Return type
The function returns *None* if no matching constant definition exists.

ground(*parts: Sequence[Tuple[str, Sequence[Symbol]]] = (('base', ()),)*, *context: Any = None*) → None

Ground the given list of program parts specified by tuples of names and arguments.

Parameters

- **parts** – List of tuples of program names and program arguments to ground.
- **context** – A context object whose methods are called during grounding using the @-syntax (if omitted, those from the main module are used).

Notes

Note that parts of a logic program without an explicit *#program* specification are by default put into a program called *base* without arguments.

interrupt() → None

Interrupt the active solve call.

Notes

This function is thread-safe and can be called from a signal handler. If no search is active, the subsequent call to *Control.solve* is interrupted. The result of the *Control.solve* method can be used to query if the search was interrupted.

load(*path: str*) → None

Extend the logic program with a (non-ground) logic program in a file.

Parameters
path – The path of the file to load.

register_observer(*observer: Observer*, *replace: bool = False*) → None

Registers the given observer to inspect the produced grounding.

Parameters

- **observer** – The observer to register. See below for a description of the requirede interface.

- **replace** – If set to true, the output is just passed to the observer and nolonger to the underlying solver (or any previously registered observers).

See also:

`clingo.backend`

register_propagator(*propagator: Propagator*) → None

Registers the given propagator with all solvers.

Parameters
propagator – The propagator to register.

See also:

`clingo.propagator`

release_external(*external: Iterable[*Predicate | *Symbol* | *int]*) → None

Release an external fact (or collection of facts)

A fact can be a raw clingo.Symbol object, a clorm.Predicate instance, or a program literal (an int). If the external is a collection then the truth value is assigned to all elements in the collection.

This function extends `clingo.Control.release_external`.

Release an external atom represented by the given symbol or program literal.

This function causes the corresponding atom to become permanently false if there is no definition for the atom in the program. Otherwise, the function has no effect.

Parameters
external – The symbolic atom or program atom to release.

Notes

If the program literal is negative, the corresponding atom is released.

Examples

The following example shows the effect of assigning and releasing and external atom.

```
>>> from clingo.symbol import Function
>>> from clingo.control import Control
>>>
>>> ctl = Control()
>>> ctl.add("base", [], "a. #external b.")
>>> ctl.ground([("base", [])])
>>> ctl.assign_external(Function("b"), True)
>>> print(ctl.solve(on_model=print))
b a
SAT
>>> ctl.release_external(Function("b"))
>>> print(ctl.solve(on_model=print))
a
SAT
```

solve(**args: Any*, ***kwargs: Any*) → *SolveHandle* | SolveResult

Run the clingo solver.

This function extends `clingo.Control.solve()` in two ways:

1) The `assumptions` argument is generalised so that in the list of argument-boolean pairs the argument can be be a clingo symbol, or clorm predicate instance, or a collection of clingo symbols or clorm predicates.

2) It produces either a `clorm.clingo.SolveHandle` wrapper object or a `clorm.clingo.Model` wrapper objects as appropriate (depending on the `yield_`, `async_`, and `on_model` parameters).

Starts a search.

Parameters

- **assumptions** – List of (atom, boolean) tuples or program literals (see *clingo.symbolic_atoms.SymbolicAtom.literal*) that serve as assumptions for the solve call, e.g., solving under assumptions *[(Function("a"), True)]* only admits answer sets that contain atom *a*.
- **on_model** – Optional callback for intercepting models. A *clingo.solving.Model* object is passed to the callback. The search can be interruped from the model callback by returning False.
- **on_unsat** – Optional callback to intercept lower bounds during optimization.
- **on_statistics** – Optional callback to update statistics. The step and accumulated statistics are passed as arguments.
- **on_finish** – Optional callback called once search has finished. A *clingo.solving.SolveResult* also indicating whether the solve call has been intrrupted is passed to the callback.
- **on_core** – Optional callback called with the assumptions that made a problem unsatisfiable.
- **yield** – The resulting *clingo.solving.SolveHandle* is iterable yielding *clingo.solving.Model* objects.
- **async** – The solve call and the method *clingo.solving.SolveHandle.resume* of the returned handle are non-blocking.

Returns

- The return value depends on the parameters. If either *yield_* or
- *async_* is true, then a handle is returned. Otherwise, a
- *clingo.solving.SolveResult* is returned.

See also:

`clingo.solving`

Notes

If neither *yield_* nor *async_* is set, the function returns a *clingo.solving.SolveResult* right away.

In gringo or in clingo with lparse or text output enabled, this function just grounds and returns a *clingo.solving.SolveResult* where *clingo.solving.SolveResult.unknown* is true.

If this function is used in embedded Python code, you might want to start clingo using the *–outf=3* option to disable all output from clingo.

Asynchronous solving is only available in clingo with thread support enabled. Furthermore, the on_model and on_finish callbacks are called from another thread. To ensure that the methods can be called, make sure to not use any functions that block Python's GIL indefinitely.

This function as well as blocking functions on the *clingo.solving.SolveHandle* release the GIL but are not thread-safe.

property configuration

Object to change the configuration.

property control_: Control

Returns the underlying clingo.Control object.

property enable_cleanup

Whether to enable automatic calls to *Control.cleanup*.

property enable_enumeration_assumption

Whether do discard or keep learnt information from enumeration modes.

If the enumeration assumption is enabled, then all information learnt from clasp's various enumeration modes is removed after a solve call. This includes enumeration of cautious or brave consequences, enumeration of answer sets with or without projection, or finding optimal models; as well as clauses added with *clingo.solving.SolveControl.add_clause*.

Notes

Initially the enumeration assumption is enabled.

In general, the enumeration assumption should be enabled whenever there are multiple calls to solve. Otherwise, the behavior of the solver will be unpredictable because there are no guarantees which information exactly is kept. There might be small speed benefits when disabling the enumeration assumption for single shot solving.

property is_conflicting

Whether the internal program representation is conflicting.

If this (read-only) property is true, solve calls return immediately with an unsatisfiable solve result.

Notes

Conflicts first have to be detected, e.g., initial unit propagation results in an empty clause, or later if an empty clause is resolved during solving. Hence, the property might be false even if the problem is unsatisfiable.

property statistics

A *dict* containing solve statistics of the last solve call.

See also:

`clingo.statistics`

Notes

The statistics correspond to the *–stats* output of clingo. The detail of the statistics depends on what level is requested on the command line. Furthermore, there are some functions like *Control.release_external* that start a new solving step resetting the current step statistics. It is best to access the statistics right after solving.

This property is only available in clingo.

property symbolic_atoms

An object to inspect the symbolic atoms.

See also:

`clingo.symbolic_atoms`

property theory_atoms

An iterator over the theory atoms in a program.

See also:

`clingo.theory_atoms`

property unifier: SymbolPredicateUnifier | None

Get/set the unifier.

Unifier can be specified as a SymbolPredicateUnifier or a collection of Predicates. Always returns a SymbolPredicateUnifier (or None).

class clorm.clingo.**Model**(*model: Model*, *unifier: List[Type[*Predicate*]] | SymbolPredicateUnifier | None = None*)

Provides access to a model during a solve call.

Objects mustn't be created manually. Instead they are returned by `clorm.clingo.Control.solve` callbacks.

Behaves like `clingo.Model` but offers better integration with clorm facts and fact bases.

Provides access to a model during a solve call and provides a *SolveContext* object to influence the running search.

Notes

The string representation of a model object is similar to the output of models by clingo using the default output.

Model objects cannot be constructed from Python. Instead they are obained during solving (see *Control.solve*). Furthermore, the lifetime of a model object is limited to the scope of the callback it was passed to or until the search for the next model is started. They must not be stored for later use.

contains(*fact:* Predicate | *Symbol*) → bool

Return whether the fact or symbol is contained in the model. Extends `clingo.Model.contains` to allow for clorm facts as well as a clingo symbols.

Efficiently check if an atom is contained in the model.

Parameters
atom – The atom to lookup.

Return type
Whether the given atom is contained in the model.

Notes

The atom must be represented using a function symbol.

extend(*symbols: Sequence[Symbol]*) → None

Extend a model with the given symbols.

Parameters
symbols – The symbols to add to the model.

Notes

This only has an effect if there is an underlying clingo application, which will print the added symbols.

facts(**args: Any*, ***kwargs: Any*) → *FactBase*

Returns a FactBase containing the facts in the model that unify with the SymbolPredicateUnifier.

This function provides a wrapper around the `clingo.Model.symbols` functions, but instead of returning a list of symbols it returns a FactBase containing the facts represented as `clorm.Predicate` sub-class instances.

Parameters

- **unifier** (*list | SymbolPredicateUnifier*) – used to unify and instantiate FactBase (Default: passed via the constructor if specified in the *clorm.clingo.Control* object)
- **atoms** – select all atoms in the model (Default: False)
- **terms** – select all terms displayed with #show statements (Default: False)
- **shown** – select all atoms and terms (Default: False)
- **theory** – select atoms added with Model.extend() (Default: False)
- **complement** – return the complement of the answer set w.r.t. to the atoms known to the grounder. (Default: False)
- **raise_on_empty** – raise a ValueError if the resulting FactBase is empty (Default: False)

is_consequence(*literal: int*) → bool | None

Check if the given program literal is a consequence.

The function returns *True*, *False*, or *None* if the literal is a consequence, not a consequence, or it is not yet known whether it is a consequence, respectively.

While enumerating cautious or brave consequences, there is partial information about which literals are consequences. The current state of a literal can be requested using this function. If this function is used during normal model enumeration, the function just returns whether a literal is true of false in the current model.

Parameters
literal – The given program literal.

Return type
Whether the given program literal is a consequence.

is_true(*literal: int*) → bool

Check if the given program literal is true.

Parameters
literal – The given program literal.

Return type
Whether the given program literal is true.

symbols(*atoms: bool = False*, *terms: bool = False*, *shown: bool = False*, *theory: bool = False*, *complement: bool = False*) → Sequence[Symbol]

Return the list of atoms, terms, or CSP assignments in the model.

Parameters

- **atoms** – Select all atoms in the model (independent of *#show* statements).
- **terms** – Select all terms displayed with *#show* statements in the model.
- **shown** – Select all atoms and terms as outputted by clingo.
- **theory** – Select atoms added with *Model.extend*.
- **complement** – Return the complement of the answer set w.r.t. to the atoms known to the grounder.

Return type
The selected symbols.

Notes

Atoms are represented using functions (*Symbol* objects), and CSP assignments are represented using functions with name *"$"* where the first argument is the name of the CSP variable and the second its value.

property context

Object that allows for controlling the running search.

property cost

Return the list of integer cost values of the model.

The return values correspond to clasp's cost output.

property model_: Model

Returns the underlying clingo.Model object.

property number

The running number of the model.

property optimality_proven

Whether the optimality of the model has been proven.

property priority

Return the priorities of the model's cost values.

property thread_id

The id of the thread which found the model.

property type

The type of the model.

class clorm.clingo.**SolveHandle**(*handle: SolveHandle*, *unifier: List[Type[*Predicate*]] | SymbolPredicateUnifier | None = None*)

Handle for solve calls.

Objects mustn't be created manually. Instead they are returned by `clorm.clingo.Control.solve`.

Behaves like `clingo.SolveHandle` but iterates over `clorm.clingo.Model` objects.

Handle for solve calls.

They can be used to control solving, like, retrieving models or cancelling a search.

See also:

Control.solve

Notes

A *SolveHandle* is a context manager and must be used with Python's *with* statement.

Blocking functions in this object release the GIL. They are not thread-safe though.

cancel() → None

Cancel the running search.

See also:

`clingo.control.Control.interrupt`

core() → List[int]

The subset of assumptions that made the problem unsatisfiable.

get() → SolveResult

Get the result of a solve call.

If the search is not completed yet, the function blocks until the result is ready.

model() → Model | None

Get the current model if there is any.

resume() → None

Discards the last model and starts searching for the next one.

Notes

If the search has been started asynchronously, this function starts the search in the background.

wait(*timeout: float | None = None*) → bool

Wait for solve call to finish or the next result with an optional timeout.

If a timeout is given, the behavior of the function changes depending on the sign of the timeout. If a postive timeout is given, the function blocks for the given amount time or until a result is ready. If the timeout is negative, the function will block until a result is ready, which also corresponds to the behavior of the function if no timeout is given. A timeout of zero can be used to poll if a result is ready.

Parameters

timeout – If a timeout is given, the function blocks for at most timeout seconds.

Return type

Indicates whether the solve call has finished or the next result is ready.

property solvehandle_: SolveHandle

Access the underlying clingo.SolveHandle object.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

J

L

M

N

O

P

Q

R

S

T

U

W