SWI-Prolog Python interface

Jan Wielemaker
SWI-Prolog Solutions b.v.
E-mail: jan@swi-prolog.org

November 23, 2024

Abstract

This package implements a bi-directional interface between Prolog and Python using portable low-level primitives. The aim is to make Python available to Prolog and visa versa with minimal installation effort while providing a high level bi-directional interface with good performance.

The API is being developed in close cooperation with the XSB and Ciao teams as a pilot for the PIP (*Prolog Improvement Proposal*) initiative. Janus should become the de-facto standard interface between Python and Prolog.

Contents

1 Introduction

Python has a huge developer community that maintains a large set of resources, notably interfaces to just about anything one can imagine. Making such interfaces directly available to Prolog can surely be done. However, developing an interface typically requires programming in C or C++, a skill that is not widely available everywhere. Being able to access Python effortlessly from Prolog puts us in a much better position because Python experience is widely available in our target audience. This solution was proposed in [Andersen & Swift, 2023, Swift & Andersen, 2023], initially developed for XSB.

Janus provides a bi-directional interface between Prolog and Python using the low-level C API of both languages. This makes using Python from Prolog as simple as taking the standard SWI-Prolog distribution and loading library `janus`. Using Prolog from Python is as simple as `import janus_swi as janus` and start making calls. Both Prolog and Python being dynamically typed languages, we can define an easy to use interface that provides a *latency* of about one μS.

The Python interface is modeled after the recent JavaScript interface developed for the WASM (Web Assembly) version. That is

- A di-directional data conversion is defined. See section 2.
- A Prolog predicate `py_call/2` to call Python functions and methods, as well as access and set object attributes.
- A non-deterministic Prolog predicate `py_iter/2` to enumerate a Python *iterator*.
- A Python function `janus.query_once()` to evaluate a Prolog query as `once/1`, providing input to Prolog variables using a Python dict and return a Python dict with bindings for each Prolog output variable.
- A python function `janus.apply_once()` to call a Prolog predicate with N *input arguments* followed by exactly one *output argument*. This provides a faster and easier to use interface to compliant predicates.
- Python iterators `janus.query()` and `janus.apply()` that provide access to non-deterministic Prolog predicates using the calling conventions of `janus.query_once()` and `janus.apply_once()`.

The API of Janus is the result of discussions between the SWI-Prolog, XSB and Ciao lang teams. It will be reflected in a PIP (*Prolog Improvement Proposal*). Considering the large differences in designs and opinions in Prolog implementation, the PIP does not cover all aspects of the API. Many of the predicates and functions have a *Compatibility* note that explains the relation of the SWI-Prolog API and the PIP. We summarize the differences in section 14.

2 Data conversion

The bi-directional conversion between Prolog and Python terms is summarized in the table below. For compatibility with Prolog implementations without native dicts we support converting the `{k1:v1, k2:v2, ...}` to dicts. Note that `{k1:v1, k2:v2}` is syntactic sugar for `{}(',' (:(k1,v1), :(k2,v2)))`. We allow for embedding this in a `py`(*Term*) such that, with

`py` defined as *prefix operator*, `py{k1:v1, k2:v2}` is both valid syntax as SWI-Prolog dict as as ISO Prolog compliant term and both are translated into the same Python dict. Note that `{}` translates to a Python string, while `py({})` translates into an empty Python dict.

By default we translate Python strings into Prolog atoms. Given we support strings, this is somewhat dubious. There are two reasons for this choice. One is the pragmatic reason that Python uses strings both for *identifiers* and arbitrary text. Ideally we'd have the first translated to Prolog atoms and the latter to Prolog strings, but, because we do not know which strings act as identifier and which as just text, this is not possible. The second is to improve compatibility with Prolog systems that do not support strings. Note that `py_call/3` and `py_iter/3` provide the option `py_string_as`(*Type*) to obtain strings in an alternative format, where *Type* is one of `atom`, `string`, `codes` or `chars`.

Prolog		**Python**	**Notes**
Variable	$\longrightarrow$	-	(instantiation error)
Integer	$\Longleftrightarrow$	int	Supports big integers
Rational	$\Longleftrightarrow$	fractions.`Fraction()`	
Float	$\Longleftrightarrow$	float	
@(none)	$\Longleftrightarrow$	None	
@(true)	$\Longleftrightarrow$	True	
@(false)	$\Longleftrightarrow$	False	
Atom	$\longleftarrow$	`enum.Enum()`	Name of Enum instance
Atom	$\Longleftrightarrow$	String	Depending on `py_string_as` option
String	$\longrightarrow$	String	
string(Text)	$\longrightarrow$	String	*Text* is an atom, string, code- or char list
#(Term)	$\longrightarrow$	String	*stringify* using `write_canonical/1` if not atomic
prolog(Term)	$\longrightarrow$	`janus.Term()`	Represents any Prolog term
Term	$\longleftarrow$	`janus.Term()`	
List	$\longrightarrow$	List	
List	$\longleftarrow$	Sequence	
List	$\longleftarrow$	Iterator	Note that a Python *Generator* is an *Iterator*
py_set(List)	$\Longleftrightarrow$	Set	
-()	$\Longleftrightarrow$	()	Python empty Tuple
-(a,b,...)	$\Longleftrightarrow$	(a,b,...)	Python Tuples. Note that a Prolog *pair* `A-B` maps to a Python (binary) tuple.
Dict	$\Longleftrightarrow$	Dict	Default for SWI-Prolog
{k:v,...}	$\Longleftrightarrow$	Dict	Compatibility when using `py_dict_as({})`
py({})	$\longleftarrow$	{}	Empty dict when using `py_dict_as({})`
{k:v,...}	$\longrightarrow$	Dict	Compatibility (see above)
py({k:v,...})	$\longrightarrow$	Dict	Compatibility (see above)
eval(Term)	$\longrightarrow$	Object	Evaluate Term as first argument of `py_call/2`
`py_obj` blob	$\Longleftrightarrow$	Object	Used for any Python object not above
Compound	$\longrightarrow$	-	for any term not above (type error)

The interface supports unbounded integers and rational numbers. Large integers (> 64 bits) are converted using a hexadecimal string as intermediate. SWI-Prolog rational numbers are mapped to

the Python class *fractions:Fraction*.[1]

The conversion #(Term) allows passing anything as a Python string. If *Term* is an atom or string, this is the same as passing the atom or string. Any other Prolog term is converted as defined by `write_canonical/1`. The conversion `prolog`(*Term*) creates an instance of `janus.Term()`. This class encapsulates a copy of an arbitrary Prolog term. The SWI-Prolog implementation uses the `PL_record()` and `PL_recorded()` functions to store and retrieve the term. *Term* may be any Prolog term, including *blobs*, *attributed variables*. Cycles and subterm sharing in *Term* are preserved. Internally, `janus.Term()` is used to represent Prolog exeptions that are raised during the execution of `janus.query_once()` or `janus.query()`.

Python Tuples are array-like objects and thus map best to a Prolog compound term. There are two problems with this. One is that few systems support compound terms with arity zero, e.g., `f` and many systems have a limit on the *arity* of compound terms. Using Prolog *comma lists*, e.g., `(a,b,c)` does not implement array semantics, cannot represent empty tuples and cannot disambiguate tuples with one element from the element itself. We settled with compound terms using the `-` as functor to make the common binary tuple map to a Prolog *pair*.

3 Janus by example - Prolog calling Python

This section introduces Janus calling Python from Prolog with examples.

3.1 Janus calling spaCy

The spaCy package provides natural language processing. This section illustrates the Janus library using spaCy. Typically, spaCy and the English language models may be installed using

```
> pip install spacy
> python -m spacy download en
```

After spaCy is installed, we can define `model/1` to represent a Python object for the English language model using the code below. Note that by tabling this code as shared, the model is loaded only once and is accessible from multiple Prolog threads.

```
:- table english/1 as shared.

english(NLP) :-
    py_call(spacy:load(en_core_web_sm), NLP).
```

Calling `english`(*X*) results in *X* = `<py_English>(0x7f703c24f430)`, a *blob* that references a Python object. *English* is the name of the Python class to which the object belongs and *0x7f703c24f430* is the address of the object. The returned object implements the Python *callable* protocol, i.e., it behaves as a function with additional properties and methods. Calling the model with a string results in a parsed document. We can use this from Prolog using the built-in `__call__` method:

[1] Currently, mapping rational numbers to fractions uses a string as intermediate representation and may thus be slow.

```
?- english(NLP),
   py_call(NLP:'__call__'("This is a sentence."), Doc).
NLP = <py_English>(0x7f703851b8e0),
Doc = [<py_Token>(0x7f70375be9d0), <py_Token>(0x7f70375be930),
       <py_Token>(0x7f70387f8860), <py_Token>(0x7f70376dde40),
       <py_Token>(0x7f70376de200)
      ].
```

This is not what we want. Because the spaCy `Doc` class implements the *sequence* protocol it is translated into a Prolog list of spaCy `Token` instances. The `Doc` class implements many more methods that we may wish to use. An example is `noun_chunks`, which provides a Python *generator* that enumerates the noun chunks found in the input. Each chunk is an instance of `Span`, a sequence of `Token` instances that have the property `text`. The program below extracts the noun chunks of the input as a non-deterministic Prolog predicate. Note that we use `py_object(true)` to get the parsed document as a Python object. Next, we use `py_iter/2` to access the members of the Python *iterator* returned by `Doc.noun_chunks` as Python object references and finally we extract the text of each noun chunk as an atom. The SWI-Prolog (atom) garbage collector will take care of the *Doc* and *Span* Python objects. Immediate release of these objects can be enforced using `py_free/1`.[2]

```
:- use_module(library(janus)).

:- table english/1.

english(NLP) :-
    py_call(spacy:load(en_core_web_sm),NLP).

noun(Sentence, Noun) :-
    english(NLP),
    py_call(NLP:'__call__'(Sentence), Doc, [py_object(true)]),
    py_iter(Doc:noun_chunks, Span, [py_object]),
    py_call(Span:text, Noun).
```

After which we can call

```
?- noun("This is a sentence.", Noun).
Noun = 'This' ;
Noun = 'a sentence'.
```

The subsequent section 4 documents the Prolog library `janus`.

4 library(janus): Call Python from Prolog

This library implements calling Python from Prolog. It is available directly from Prolog if the janus package is bundled. The library provides access to an *embedded* Python instance. If SWI-Prolog is

[2] Janus implementations are not required to implement Python object reference garbage collection.

embedded into Python using the Python package `janus-swi`, this library is provided either from Prolog or from the Python package.

Normally, the Prolog user can simply start calling Python using `py_call/2` or friends. In special cases it may be needed to initialize Python with options using `py_initialize/3` and optionally the Python search path may be extended using `py_add_lib_dir/1`.

py_version *[det]*

Print version info on the embedded Python installation based on Python `sys.version`. If a Python *virtual environment* (venv) is active, indicate this with the location of this environment found.

py_call(+*Call*) *[det]*
py_call(+*Call, -Return*) *[det]*
py_call(+*Call, -Return, +Options*) *[det]*

Call Python and return the result of the called function. *Call* has the shape '[Target][:Action]*', where *Target* is either a Python module name or a Python object reference. Each *Action* is either an atom to get the denoted attribute from current *Target* or it is a compound term where the first argument is the function or method name and the arguments provide the parameters to the Python function. On success, the returned Python object is translated to Prolog. *Action* without a *Target* denotes a buit-in function.

Arguments to Python functions use the Python conventions. Both *positional* and *keyword* arguments are supported. Keyword arguments are written as `Name = Value` and must appear after the positional arguments.

Below are some examples.

```
% call a built-in
?- py_call(print("Hello World!\n")).
true.

% call a built-in (alternative)
?- py_call(builtins:print("Hello World!\n")).
true.

% call function in a module
?- py_call(sys:getsizeof([1,2,3]), Size).
Size = 80.

% call function on an attribute of a module
?- py_call(sys:path:append("/home/bob/janus")).
true

% get attribute from a module
?- py_call(sys:path, Path)
Path = ["dir1", "dir2", ...]
```

Given a class in a file `dog.py` such as the following example from the Python documentation

```
class Dog:
    tricks = []

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)
```

We can interact with this class as below. Note that `$Doc` in the SWI-Prolog toplevel refers to the last toplevel binding for the variable *Dog*.

```
?- py_call(dog:'Dog'("Fido"), Dog).
Dog = <py_Dog>(0x7f095c9d02e0).

?- py_call($Dog:add_trick("roll_over")).
Dog = <py_Dog>(0x7f095c9d02e0).

?- py_call($Dog:tricks, Tricks).
Dog = <py_Dog>(0x7f095c9d02e0),
Tricks = ["roll_over"]
```

If the principal term of the first argument is not `Target:Func`, The argument is evaluated as the initial target, i.e., it must be an object reference or a module. For example:

```
?- py_call(dog:'Dog'("Fido"), Dog),
   py_call(Dog, X).
   Dog = X, X = <py_Dog>(0x7fa8cbd12050).
?- py_call(sys, S).
   S = <py_module>(0x7fa8cd582390).
```

Options processed:

py_object(*Boolean***)**
: If `true` (default `false`), translate the return as a Python object reference. Some objects are *always* translated to Prolog, regardless of this flag. These are the Python constants `None`, `True` and `False` as well as instances of the Python base classes `int`, `float`, `str` or `tuple`. Instances of sub classes of these base classes are controlled by this option.

py_string_as(+*Type***)**
: If *Type* is `atom` (default), translate a Python String into a Prolog atom. If *Type* is `string`, translate into a Prolog string. Strings are more efficient if they are short lived.

py_dict_as(+*Type*)

One of `dict` (default) to map a Python dict to a SWI-Prolog dict if all keys can be represented. If `{}` or not all keys can be represented, *Return* is unified to a term `{k:v, ...}` or `py({})` if the Python dict is empty.

Compatibility PIP. The options `py_string_as` and `py_dict_as` are SWI-Prolog specific, where SWI-Prolog Janus represents Python strings as atoms as required by the PIP and it represents Python dicts by default as SWI-Prolog dicts. The predicates `values/3`, `keys/2`, etc. provide portable access to the data in the dict.

py_iter(+*Iterator*, -*Value*) *[nondet]*

py_iter(+*Iterator*, -*Value*, +*Options*) *[nondet]*

True when *Value* is returned by the Python *Iterator*. Python iterators may be used to implement non-deterministic foreign predicates. The implementation uses these steps:

1. Evaluate *Iterator* as `py_call/2` evaluates its first argument, except the `Obj:Attr = Value` construct is not accepted.
2. Call `__iter__` on the result to get the iterator itself.
3. Get the `__next__` function of the iterator.
4. Loop over the return values of the *next* function. If the Python return value unifies with *Value*, succeed with a choicepoint. Abort on Python or unification exceptions.
5. Re-satisfaction continues at (4).

The example below uses the built-in iterator `range()`:

```
?- py_iter(range(1,3), X).
X = 1 ;
X = 2.
```

Note that the implementation performs a *look ahead*, i.e., after successful unification it calls '**next**()' again. On failure the Prolog predicate succeeds deterministically. On success, the next candidate is stored.

Note that a Python *generator* is a Python *iterator*. Therefore, given the Python generator expression below, we can use `py_iter(squares(1,5),X)` to generate the squares on backtracking.

```
def squares(start, stop):
     for i in range(start, stop):
         yield i * i
```

Arguments

Options	is processed as with `py_call/3`.

Compatibility PIP. The same remarks as for `py_call/2` apply.

bug *Iterator* may not depend on janus.`query()`, i.e., it is not possible to iterate over a Python iterator that under the hoods relies on a Prolog non-deterministic predicate.

py_setattr(+*Target, +Name, +Value*) *[det]*

Set a Python attribute on an object. If *Target* is an atom, it is interpreted as a module. Otherwise it is normally an object reference. `py_setattr/3` allows for *chaining* and behaves as if defined as

```
py_setattr(Target, Name, Value) :-
    py_call(Target, Obj, [py_object(true)]),
    py_call(setattr(Obj, Name, Value)).
```

Compatibility PIP

py_is_object(@*Term*) *[semidet]*

True when *Term* is a Python object reference. Fails silently if *Term* is any other Prolog term.

Errors `existence_error(py_object, Term)` is raised of *Term* is a Python object, but it has been freed using `py_free/1`.

Compatibility PIP. The SWI-Prolog implementation is safe in the sense that an arbitrary term cannot be confused with a Python object and a reliable error is generated if the references has been freed. Portable applications can not rely on this.

py_is_dict(@*Term*) *[semidet]*

True if *Term* is a Prolog term that represents a Python dict.

Compatibility PIP. The SWI-Prolog version accepts both a SWI-Prolog dict and the `{k:v,...}` representation. See `py_dict_as` option of `py_call/2`.

py_free(+*Obj*) *[det]*

Immediately free (decrement the reference count) for the Python object *Obj*. Further reference to *Obj* using e.g., `py_call/2` or `py_free/1` raises an `existence_error`. Note that by decrementing the reference count, we make the reference invalid from Prolog. This may not actually delete the object because the object may have references inside Python.

Prolog references to Python objects are subject to atom garbage collection and thus normally do not need to be freed explicitly.

Compatibility PIP. The SWI-Prolog implementation is safe and normally reclaiming Python object can be left to the garbage collector. Portable applications may not assume garbage collection of Python objects and must ensure to call `py_free/1` exactly once on any Python object reference. Not calling `py_free/1` leaks the Python object. Calling it twice may lead to undefined behavior.

py_with_gil(*:Goal*) *[semidet]*

Run *Goal* as `once(Goal)` while holding the Phyton GIL (*Global Interpreter Lock*). Note that all predicates that interact with Python lock the GIL. This predicate is only required if we wish to make multiple calls to Python while keeping the GIL. The GIL is a *recursive* lock and thus calling `py_call/1`,2 while holding the GIL does not *deadlock*.

py_gil_owner(-*Thread*) *[semidet]*

True when the Python GIL is owned by *Thread*. Note that, unless *Thread* is the calling thread, this merely samples the current state and may thus no longer be true when the predicate succeeds. This predicate is intended to help diagnose *deadlock* problems.

Note that this predicate returns the Prolog threads that locked the GIL. It is however possible that Python releases the GIL, for example if it performs a blocking call. In this scenario, some other thread or no thread may hold the gil.

py_func(+*Module, +Function, -Return*) *[det]*
py_func(+*Module, +Function, -Return, +Options*) *[det]*

Call Python *Function* in *Module*. The SWI-Prolog implementation is equivalent to `py_call(Module:Function, Return)`. See `py_call/2` for details.

> **Compatibility** PIP. See `py_call/2` for notes. Note that, as this implementation is based on `py_call/2`, *Function* can use *chaining*, e.g., `py_func(sys, path:append(dir), Return)` is accepted by this implementation, but not portable.

py_dot(+*ObjRef, +MethAttr, -Ret*) *[det]*
py_dot(+*ObjRef, +MethAttr, -Ret, +Options*) *[det]*

Call a method or access an attribute on the object *ObjRef*. The SWI-Prolog implementation is equivalent to `py_call(ObjRef:MethAttr, Return)`. See `py_call/2` for details.

> **Compatibility** PIP. See `py_func/3` for details.

values(+*Dict, +Path, ?Val*) *[semidet]*

Get the value associated with *Dict* at *Path*. *Path* is either a single key or a list of keys.

> **Compatibility** PIP. Note that this predicate handle a SWI-Prolog dict, a {k:v, ...} term as well as py({k:v, ...}.

keys(+*Dict, ?Keys*) *[det]*

True when *Keys* is a list of keys that appear in *Dict*.

> **Compatibility** PIP. Note that this predicate handle a SWI-Prolog dict, a {k:v, ...} term as well as py({k:v, ...}.

key(+*Dict, ?Key*) *[nondet]*

True when *Key* is a key in *Dict*. Backtracking enumerates all known keys.

> **Compatibility** PIP. Note that this predicate handle a SWI-Prolog dict, a {k:v, ...} term as well as py({k:v, ...}.

items(+*Dict, ?Items*) *[det]*

True when *Items* is a list of Key:Value that appear in *Dict*.

> **Compatibility** PIP. Note that this predicate handle a SWI-Prolog dict, a {k:v, ...} term as well as py({k:v, ...}.

py_shell

Start an interactive Python REPL loop using the embedded Python interpreter. The interpreter first imports `janus` as below.

```
from janus import *
```

So, we can do

```
?- py_shell.
...
>>> query_once("writeln(X)", {"X":"Hello world"})
Hello world
{'truth': True}
```

If possible, we enable command line editing using the GNU readline library.

When used in an environment where Prolog does not use the file handles 0,1,2 for the standard streams, e.g., in `swipl-win`, Python's I/O is rebound to use Prolog's I/O. This includes Prolog's command line editor, resulting in a mixed history of Prolog and Pythin commands.

py_pp(+*Term*) *[det]*
py_pp(+*Term, +Options*) *[det]*
py_pp(+*Stream, +Term, +Options*) *[det]*

Pretty prints the Prolog translation of a Python data structure in Python syntax. This exploits `pformat()` from the Python module `pprint` to do the actual formatting. *Options* is translated into keyword arguments passed to pprint.`pformat()`. In addition, the option `nl(Bool)` is processed. When `true` (default), we use pprint.`pp()`, which makes the output followed by a newline. For example:

```
?- py_pp(py{a:1, l:[1,2,3], size:1000000},
         [underscore_numbers(true)]).
{'a': 1, 'l': [1, 2, 3], 'size': 1_000_000}
```

Compatibility PIP

py_object_dir(+*ObjRef, -List*) *[det]*
py_object_dict(+*ObjRef, -Dict*) *[det]*

Examine attributes of an object. The predicate `py_object_dir/2` fetches the names of all attributes, while `py_object_dir/2` gets a dict with all attributes and their values.

Compatibility PIP

py_obj_dir(+*ObjRef, -List*) *[det]*
py_obj_dict(+*ObjRef, -Dict*) *[det]*

deprecated Use `py_object_dir/2` or `py_object_dict/2`.

py_type(+*ObjRef, -Type:atom*) *[det]*

True when *Type* is the name of the type of *ObjRef*. This is the same as `type(ObjRef).__name__` in Python.

Compatibility PIP

py_isinstance(+*ObjRef, +Type*) *[semidet]*

True if *ObjRef* is an instance of *Type* or an instance of one of the sub types of *Type*. This is the same as `isinstance(ObjRef)` in Python.

Arguments

Type	is either a term `Module:Type` or a plain atom to refer to a built-in type.

Compatibility PIP

py_module_exists(+*Module*) *[semidet]*

True if *Module* is a currently loaded Python module or it can be loaded.

Compatibility PIP

py_hasattr(+*ModuleOrObj, ?Name*) *[nondet]*

True when *Name* is an attribute of Module. The name is derived from the Python built-in `hasattr()`. If *Name* is unbound, this enumerates the members of `py_object_dir/2`.

Arguments

ModuleOrObj	If this is an atom it refers to a module, otherwise it must be a Python object reference.

Compatibility PIP

py_import(+*Spec, +Options*) *[det]*

Import a Python module. Janus imports modules automatically when referred in `py_call/2` and related predicates. Importing a module implies the module is loaded using Python's `__import__()` built-in and added to a table that maps Prolog atoms to imported modules. This predicate explicitly imports a module and allows it to be associated with a different name. This is useful for loading *nested modules*, i.e., a specific module from a Python package as well as for avoiding conflicts. For example, with the Python `selenium` package installed, we can do in Python:

```
>>> from selenium import webdriver
>>> browser = webdriver.Chrome()
```

Without this predicate, we can do

```
?- py_call('selenium.webdriver':'Chrome'(), Chrome).
```

For a single call this is fine, but for making multiple calls it gets cumbersome. With this predicate we can write this.

```
?- py_import('selenium.webdriver', []).
?- py_call(webdriver:'Chrome'(), Chrome).
```

By default, the imported module is associated to an atom created from the last segment of the dotted name. Below we use an explicit name.

```
?- py_import('selenium.webdriver', [as(browser)]).
?- py_call(browser:'Chrome'(), Chrome).
```

Errors `permission_error(import_as, py_module, As)` if there is already a module associated with As.

py_module(+*Module:atom, +Source:string*) *[det]*

Load *Source* into the Python module *Module*. This is intended to be used together with the `string` *quasi quotation* that supports long strings in SWI-Prolog. For example:

```
:- use_module(library(strings)).
:- py_module(hello,
             {|string||
              | def say_hello_to(s):
              |     print(f"hello {s}")
              |}).
```

Calling this predicate multiple times with the same *Module* and *Source* is a no-op. Called with a different source creates a new Python module that replaces the old in the global namespace.

Errors `python_error(Type, Data)` is raised if Python raises an error.

py_initialize(+*Program, +Argv, +Options*) *[det]*

Initialize and configure the embedded Python system. If this predicate is not called before any other call to Python such as `py_call/2`, it is called *lazily*, passing the Prolog executable as *Program*, passing *Argv* from the Prolog flag `py_argv` and an empty *Options* list.

Calling this predicate while the Python is already initialized is a no-op. This predicate is thread-safe, where the first call initializes Python.

In addition to initializing the Python system, it

- Adds the directory holding `janus.py` to the Python module search path.
- If Prolog I/O is not connected to the file handles 0,1,2, it rebinds Python I/O to use the Prolog I/O.

Arguments

Options	is currently ignored. It will be used to provide additional configuration options.

py_lib_dirs(-*Dirs*) *[det]*

True when *Dirs* is a list of directories searched for Python modules. The elements of *Dirs* are in Prolog canonical notation.

Compatibility PIP

py_add_lib_dir(+*Dir*) *[det]*

py_add_lib_dir(+*Dir, +Where*) *[det]*

Add a directory to the Python module search path. In the second form, *Where* is one of `first` or `last`. `py_add_lib_dir/1` adds the directory as `last`. The property `sys:path` is not modified if it already contains *Dir*.

Dir is in Prolog notation. The added directory is converted to an absolute path using the OS notation using `prolog_to_os_filename/2`.

If *Dir* is a *relative* path, it is taken relative to Prolog source file when used as a *directive* and relative to the process working directory when called as a predicate.

Compatibility PIP. Note that SWI-Prolog uses POSIX file conventions internally, mapping to OS conventions inside the predicates that deal with files or explicitly using `prolog_to_os_filename/2`. Other systems may use the native file conventions in Prolog.

4.1 Handling Python errors in Prolog

If `py_call/2` or one of the other predicates that access Python causes Python to raise an exception, this exception is translated into a Prolog exception of the shape below. The library defines a rule for `print_message/2` to render these errors in a human readable way.

`error(python_error(`*ErrorType, Value*`), _)`

Here, *ErrorType* is the name of the error type, as an atom, e.g., `'TypeError'`. *Value* is the exception object represented by a Python object reference. The `janus` defines the message formatting, which makes us end up with a message like below.

```
?- py_call(nomodule:noattr).
ERROR: Python 'ModuleNotFoundError':
ERROR:   No module named 'nomodule'
ERROR: In:
ERROR:   [10] janus:py_call(nomodule:noattr)
```

The Python *stack trace* is handed embedded into the second argument of the `error`(*Formal, ImplementationDefined*). If an exception is printed, printing the Python backtrace, is controlled by the Prolog flags `py_backtrace` (default `true`) and `py_backtrace_depth` (default `4`).

Compatibility PIP. The embedding of the Python backtrace is SWI-Prolog specific.

4.2 Calling and data translation errors

Errors may occur when converting Prolog terms to Python objects as defined in section 2. These errors are reported as `instantiation_error`, `type_error`(*Type, Culprit*) or `domain_error`(*Domain, Culprit*).

Defined **domains** are:

py_constant
: In a term `@`(*Constant*), *Constant* is not `true`, `false` or `none`. For example, `py_call(print(@error))`.

py_keyword_arg
: In a call to Python, a non keyword argument follows a keyword argument. For example, `py_call(m:f(1,x=2,3), R)`

py_string_as
: The value for a `py_string_as`(*As*) option is invalid. For example, `py_call(m:f(), R, [py_string_as(float)])`

py_dict_as
: The value for a `py_dict_as`(*As*) option is invalid. For example, `py_call(m:f(), R, [py_dict_as(list)])`

py_term
: A term being translated to Python is unsupported. For example, `py_call(m:f(point(1,2)), R).`

Defined **types** are:

py_object
: A Python object reference was expected. For example, `py_free(42)`

rational
: A Python `fraction` instance is converted to a Prolog rational number, but the textual conversion does not produce a valid rational number. This can happen if the Python `fraction` is subclassed and the `__str__()` method does not produce a correct string.

py_key_value
: Inside a `{k:v, ...}` representation for a dictionary we find a term that is not a key-value pair. For example, `py_call(m:f({a:1, x}), R)`

py_set
: Inside a `py_set`(*Elements*), *Elements* is not a list. For example, `py_call(m:f(py_set(42)), R).`

py_target
: In `py_call`(*Target:FuncOrAttrOrMethod*), *Target* is not a module (atom) or Python object reference. For example, `py_call(7:f(), R).`

py_callable
: In `py_call`(*Target:FuncOrAttrOrMethod*), *FuncOrAttrOrMethod* is not an atom or compound. For example, `py_call(m:7, R).`

4.3 Janus and virtual environments (venv)

An embedded Python system does not automatically pick up Python virtual environments. It is supposed to setup its own environment. Janus is sensitive to Python `venv` environments. Running under such as environment is assumed if the environment variable `VIRTUAL_ENV` points at a directory that holds a file `pyvenv.cfg`. If the virtual environment is detected, the actions in the list below are taken.[3]

[3]This is based on observing how Python 3.10 on Linux responds to being used inside a virtual environment. We do not know whether this covers all platforms and versions.

- Initialize Python using the `-I` flag to indicate *isolation*.
- Set `sys.prefix` to the value of the `VIRTUAL_ENV` environment variable.
- Remove all directories with base name `site-packages` or `dist-packages` from `sys.path`.[4]
- Add `$VIRTUAL_ENV/lib/pythonX.Y/site-packages` to `sys.path`, where *X* and *Y* are the major and minor version numbers of the embedded Python library. If this directory does not exist we print a diagnostic warning.
- Add a message to `py_version/0` that indicates we are using a virtual environment and from which directory.

5 Calling Prolog from Python

The Janus interface can also call Prolog from Python. Calling Prolog from Python is the basis when embedding Prolog into Python using the Python package `janus_swi`. However, calling Prolog from Python is also used to handle *call backs*. Mutually recursive calls between Python and Prolog are supported. They should be handled with some care as it is easy to crash the process due to a stack overflow.

Loading janus into Python is realized using the Python package `janus-swi`, which defines the module `janus_swi`. We do not call this simply `janus` to allow coexistence of Janus for multiple Prolog implementations. Unless you plan to interact with multiple Prolog systems in the same session, we advise importing janus for SWI-Prolog as below.

```
import janus_swi as janus
```

If Python is embedded into SWI-Prolog, the Python module may be imported both as `janus` and `janus_swi`. Using `janus` allows the same Python code to be used from different Prolog systems, while using `janus_swi` allows the same code to be used both for embedding Python into Prolog and Prolog into Python. In the remainder of this section we assume the Janus functions are available in the name space `janus`.

The Python module `janus` provides utility functions and defines the classes `janus.query()`, `janus.apply()`, `janus.Term()`, `janus.Undefined()` and `janus.PrologError()`.

The Python calling Prolog interface consist of four primitives, distinguishing deterministic vs. non-deterministic Prolog queries and two different calling conventions which we name *functional notation* and *relational notation*. The *relational* calling convention specifies a Prolog query as a string with an *input dict* that provides (input) bindings for part of the variables in the query string. The results are represented as a dict that holds the bindings of the output variables and the truth value (see section 5.4). For example:

```
>>> janus.query_once("Y is sqrt(X)", {'X':2})
{'truth': True, 'Y': 1.4142135623730951}
```

[4]Note that `-I` only removes the personal packages directory, while the Python executable removes all, so we do the same.

The functional notation calling convention specifies the query as a module, predicate name and input arguments. It calls the predicate with one argument more than the number of input arguments and translates the binding of the output argument to Python. For example

```
>>> janus.apply_once("user", "plus", 1, 2)
3
```

The table below summarizes the four primitives.

	Relational notation	**Functional notation**
det	`janus.query_once()`	`janus.apply_once()`
nondet	`janus.query()`	`janus.apply()`

We start our discussion by introducing the `janus.query_once(`*query,inputs*`)` function for calling Prolog goals as `once/1`. A Prolog goal is constructed from a string and a dict with *input bindings* and returns *output bindings* as a dict. For example

```
>>> import janus_swi as janus
>>> janus.query_once("Y is X+1", {"X":1})
{'Y': 2, 'truth': True}
```

Note that the input argument may also be passed literally. Below we give two examples. We **strongly advise against using string interpolation** for three reasons. Firstly, the query strings are compiled and cached on the Prolog sided and (thus) we assume a finite number of distinct query strings. Secondly, string interpolation is sensitive to *injection attacks*. Notably inserting quoted strings can easily be misused to create malicious queries. Thirdly and finally, serializing and deserializing the data is generally slower then using the input dictionary, especially if the data is large. Using a dict for input and output together with a (short) string to denote the goal is easy to use and fast.

```
>>> janus.query_once("Y is 1+1", {})     # Ok for "static" queries
{'Y': 2, 'truth': True}
>>> x = 1
>>> janus.query_once(f"Y is {x}+1", {}) # WRONG, See above
{'Y': 2, 'truth': True}
```

The *output dict* contains all named Prolog variables that (1) are not in the input dict and (2) do not start with an underscore. For example, to get the grandparents of a person given `parent/2` relations we can use the code below, where the *_GP* and *_P* do not appear in the output dict. This both saves time and avoids the need to convert Prolog data structures that cannot be represented in Python such as variables or arbitrary compound terms.

```
>>> janus.query_once("findall(_GP, parent(Me, _P), parent(_P, _GP), GPs)",
                {'Me':'Jan'})["GPs"]
[ 'Kees', 'Jan' ]
```

In addition to the variable bindings the dict contains a key `truth`[5] that represents the truth value of evaluating the query. In normal Prolog this is a Python Boolean. In systems that implement *Well Founded Semantics*, this may also be an instance of the class `janus.Undefined()`. See section 5.4 for details. If evaluation of the query failed, all variable bindings are bound to the Python constant `None` and the `truth` key has the value `False`. The following Python function returns `True` if the Prolog system supports unbounded integers and `False` otherwise.

```
def hasBigIntegers():
    janus.query_once("current_prolog_flag(bounded,false)")['truth']
```

While `janus.query_once()` deals with semi-deterministic goals, the class `janus.query()` implements a Python *iterator* that iterates over the solutions of a Prolog goal. The iterator may be aborted using the Python `break` statement. As with `janus.query_once()`, the returned dict contains a `truth` field. This field cannot be `False` though and thus is either `True` or an instance of the class `janus.Undefined()`

```
import janus_swi as janus

def printRange(fr, to):
    for d in janus.query("between(F,T,X)", {"F":fr, "T":to}):
        print(d["X"])
```

The call to janus.`query()` returns an object that implements both the iterator protocol and the context manager protocol. A context manager ensures that the query is cleaned up as soon as it goes out of scope - Python typically does this with for loops, but there is no guarantee of when cleanup happens, especially if there is an error. (You can think of a `with` statement as similar to Prolog's `setup_call_cleanup/3`.) Using a context manager, we can write

```
def printRange(fr, to):
    with janus.query("between(F,T,X)", {"F":fr, "T":to}) as d_q:
        for d in d_q:
            print(d["X"])
```

Iterators may be nested. For example, we can create a list of tuples like below.

```
def double_iter(w,h):
    tuples=[]
    for yd in janus.query("between(1,M,Y)", {"M":h}):
        for xd in janus.query("between(1,M,X)", {"M":w}):
            tuples.append((xd['X'],yd['Y']))
    return tuples
```

or, using context managers:

[5]Note that variable bindings always start with an uppercase latter.

```
def doc_double_iter(w,h):
    tuples=[]
    with janus.query("between(1,M,Y)", {"M":h}) as yd_q:
        for yd in yd_q:
            with janus.query("between(1,M,X)", {"M":w}) as xd_q:
                for xd in xd_q:
                    tuples.append((xd['X'],yd['Y']))
    return tuples
```

After this, we may run

```
>>> demo.double_iter(2,3)
[(1, 1), (2, 1), (1, 2), (2, 2), (1, 3), (2, 3)]
```

In addition to the *iterator* protocol that class `janus.query()` implements, it also implements the methods `janus.query.next()` and `janus.query.close()`. This allows for e.g.

```
    q = query("between(1,3,X)")
    while ( s := q.next() ):
        print(s['X'])
    q.close()
```

or

```
  try:
      q = query("between(1,3,X)")
      while ( s := q.next() ):
          print(s['X'])
  finally:
      q.close()
```

The `close()` is called by the context manager, so the following is equivalent:

```
    with query("between(1,3,X)") as q:
        while ( s := q.next() ):
            print(s['X'])
```

But, **iterators based on Prolog goals are fragile**. This is because, while it is possible to open and run a new query while there is an open query, the inner query must be closed before we can ask for the next solution of the outer query. We illustrate this using the sequence below.

```
>>> q1 = query("between(1,3,X)")
>>> q2 = query("between(1,3,X)")
>>> q2.next()
```

```
{'truth': True, 'X': 1}
>>> q1.next()
Traceback (most recent call last):
...
swipl.Error: swipl.next_solution(): not inner query
>>> q2.close()
>>> q1.next()
{'truth': True, 'X': 1}
>>> q1.close()
```

Failure to close a query typically leaves SWI-Prolog in an inconsistent state and further interaction with Prolog is likely to crash the process. Future versions may improve on that. To avoid this, it is recommended that you use the query with a context manager, that is using the Python constwith statement.

`dict` **janus.query_once**(*query, inputs={}, keep=False, truth_vals=TruthVals.PLAIN_TRUTHVALS*)
Call *query* using *bindings* as `once/1`, returning a dict with the resulting bindings. If *bindings* is omitted, no variables are bound. The *keep* parameter determines whether or not Prolog discards all backtrackable changes. By default, such changes are discarded and as a result, changes to backtrackable global variables are lost. Using `True`, such changes are preserved.

```
>>> query_once("b_setval(a, 1)", keep=True)
{'truth': 'True'}
>>> query_once("b_getval(a, X)")
{'truth': 'True', 'X': 1}
```

If *query* fails, the variables of the query are bound to the Python constant `None`. The *bindings* object includes a key `truth`[6] that has the value `False` (query failed, all bindings are `None`), `True` (query succeeded, variables are bound to the result converting Prolog data to Python) or an instance of the class `janus.Undefined()`. The information carried by this instance is determined by the `truth` parameter. Below is an example. See section 5.4 for details.

```
>>> import janus_swi as janus
>>> janus.query_once("undefined")
{'truth': Undefined}
```

See also `janus.cmd()` and `janus.apply_once()`, which provide a fast but more limited alternative for making ground queries (`janus.cmd()`) or queries with leading ground arguments followed by a single output variable.

Compatibility PIP.

`dict` **janus.once**(*query, inputs={}, keep=False, truth_vals=TruthVals.PLAIN_TRUTHVALS*)
Deprecated. Renamed to `janus.query_once()`.

[6] As this name is not a valid Prolog variable name, this cannot be ambiguous.

`Any` **janus.apply_once**(*module, predicate, *input, fail=obj*)

Functional notation style calling of a deterministic Prolog predicate. This calls `module:predicate`(*Input ..., Output*), where *Input* are the Python *input* arguments converted to Prolog. On success, *Output* is converted to Python and returned. On failure a `janus.PrologError()` exception is raised unless the `fail` parameter is specified. In the latter case the function returns *obj*. This interface provides a comfortable and fast calling convention for calling a simple predicate with suitable calling conventions. The example below returns the *home directory* of the SWI-Prolog installation.

```
>>> import janus_swi as janus
>>> janus.apply_once("user", "current_prolog_flag", "home")
'/home/janw/src/swipl-devel/build.pdf/home'
```

Compatibility PIP.

`Truth` **janus.cmd**(*module, predicate, *input*)

Similar to `janus.apply_once()`, but no argument for the return value is added. This function returns the *truth value* using the same conventions as the `truth` key in `janus.query_once()`. For example:

```
>>> import janus_swi as janus
>>> cmd("user", "true")
True
>>> cmd("user", "current_prolog_flag", "bounded", "true")
False
>>> cmd("user", "undefined")
Undefined
>>> cmd("user", "no_such_predicate")
Traceback (most recent call last):
  File "/usr/lib/python3.10/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<console>", line 1, in <module>
janus.PrologError: '$c_call_prolog'/0: Unknown procedure: no_such_predicate/(
```

The function `janus.query_once()` is more flexible and provides all functionality of `janus.cmd()`. However, this function is faster and in some scenarios easier to use.

Compatibility PIP.

`None` **janus.consult**(*file, data=None, module='user'*)

Load Prolog text into the Prolog database. By default, *data* is `None` and the text is read from *file*. If *data* is a string, it provides the Prolog text that is loaded and *file* is used as *identifier* for source locations and error messages. The *module* argument denotes the target module. That is where the clauses are added to if the Prolog text does not define a module or where the exported predicates of the module are imported into.

If *data* is not provided and *file* is not accessible this raises a Prolog exception. Errors that occur during the compilation are printed using `print_message/2` and can currently not be captured easily. The script below prints the train connections as a list of Python tuples.

```
import janus_swi as janus

janus.consult("trains", """
train('Amsterdam', 'Haarlem').
train('Amsterdam', 'Schiphol').
""")

print([d['Tuple'] for d in
       janus.query("train(_From,_To),Tuple=_From-_To")])
```

Compatibility PIP. The `data` and `module` keyword arguments are SWI-Prolog extensions.

`None` **janus.prolog**()
Start the interactive Prolog toplevel. This is the Python equivalent of `py_shell/0`.

5.1 Janus iterator query

Class `janus.query()` is similar to the `janus.query_once()` function, but it returns a Python *iterator* that allows for iterating over the answers to a non-deterministic Prolog predicate.

The iterator also implements the Python context manaager protocol (for the Python `with` statement).

`query` **janus.query**(*query, inputs*={}*, keep=False*)
As `janus.query_once()`, returning an *iterator* that provides an answer dict as `janus.query_once()` for each answer to *query*. Answers never have `truth` `False`. See discussion above.

Compatibility PIP. The `keep` is a SWI-Prolog extension.

`Query` **janus.Query**(*query, inputs*={}*, keep=False*)
Deprecated. This class was renamed to `janus.query(.)`

`dict|None` **janus.query.next**()
Explicitly ask for the next solution of the iterator. Normally, using the `query` as an iterator is to be preferred. See discussion above. `q.next()` is equivalent to `next(q)` except it returns `None` if there are no more values instead of raising the `StopIteration` exception.

`None` **janus.query.close**()
Close the query. Closing a query is obligatory. When used as an iterator, the Python destructor (`__del__()`) takes care of closing the query. However, Python does not guarantee when the destructor will be called, so it is recommended that the context manager protocol is used (with the Python `with` statement), which closes the query when the query goes out of scope or when an error happens.

Compatibility PIP.

5.2 Janus iterator apply

Class `janus.apply()` is similar to `janus.apply_once()`, calling a Prolog predicate using functional notation style. It returns a Python *iterator* that enumerates all answers.

`apply` **janus.apply**(*module, predicate, *input*)

As `janus.apply_once()`, returning an *iterator* that returns individual answers. The example below uses Python *list comprehension* to create a list of integers from the Prolog built-in `between/3`.

```
>>> list(janus.apply("user", "between", 1, 6))
[1, 2, 3, 4, 5, 6]
```

Compatibility PIP.

`any|None` **janus.apply.next**()

Explicitly ask for the next solution of the iterator. Normally, using the `apply` as an iterator is to be preferred. See discussion above. Note that this calling convention cannot distinguish between the Prolog predicate returning `@none` and reaching the end of the iteration.

`None` **janus.apply.close**()

Close the query. Closing a query is obligatory. When used as an iterator, the Python destructor (`__del__()`) takes care of closing the query.

Compatibility PIP.

5.3 Janus access to Python locals and globals

Python provides access to dictionaries holding the local variables of a function using `locals()` as well as the global variables stored as attributes to the module to which the function belongs as `globals()`. The Python C API provides `PyEval_GetLocals()` and `PyEval_GetGlobals()`, but these return the scope of the Janus API function rather than user code, i.e., the global variables of the `janus` module and the local variables of the running Janus interface function.

Python code that wishes Prolog to access its scope must pass the necessary scope elements (local and global variables) explicitly to the Prolog code. It is possible to pass the entire local and or global scope by the output of `locals()` and/or `globals()`. Note however that a dict passed to Prolog is translated to its Prolog representation. This representation may be prohibitively large and does not allow Prolog to modify variables in the scope. Note that Prolog can access the global scope of a module as attributes of this module, e.g.

```
increment :-
    py_call(demo:counter, V0),
    V is V0+1,
    py_setattr(demo, counter, V).
```

5.4 Janus and Prolog truth

In traditional Prolog, queries *succeed* or *fail*. Systems that implement tabling with *Well Founded Semantics* such as XSB and SWI-Prolog define a third truth value typically called *undefined*. Undefined results may have two reasons; (1) the program is logically inconsistent or (2) *restraints* have been applied in the derivation.

Because classical Prolog truth is dominant, we represent the success of a query using the Python booleans `True` and `False`. For undefined answers we define a class `janus.Undefined()` that may represent different levels of detail on why the result is undefined. The notion of *generic undefined* is represented by a unique instance of this class. The three truth values are accessible as properties of the `janus` module.

janus.true
: This property has the Python boolean `True`

janus.false
: This property has the Python boolean `False`

janus.undefined
: This property holds a unique instance of class `janus.Undefined()`

5.4.1 Janus classed Undefined and TruthVal

The class `janus.Undefined()` represents an undefined result under the *Well Founded Semantics*.

`Undefined` **janus.Undefined**(*term=None*)
: Instances are never created explicitly by the user. They are created by the calls to Prolog initiated from `janus.query_once()` and `janus.query()`.

  The class has a single property class `term` that represents either the *delay list* or the *residual program*. See `janus.TruthVal()` for details.

`Enum` **janus.TruthVal**()
: This class is a Python *enumeration*. Its values are passed as the optional *truth* parameter to `janus.query_once()` and `janus.query()`. The defined instances are

  NO_TRUTHVALS
  : Undefined results are reported as `True`. This is quite pointless in the current design and this may go.

  PLAIN_TRUTHVALS
  : Return undefined results as `janus.undefined`, a unique instance of the class `janus.Undefined()`.

  DELAY_LISTS
  : Return undefined results as an instance of class `janus.Undefined()`. thats holds the delay list in Prolog native representation. See `call_delays/2`.

  RESIDUAL_PROGRAM
  : Return undefined results as an instance of class `janus.Undefined()`. thats holds the *residual program* in Prolog native representation. See `call_residual_program/2`.

The instances of this enumeration are available as attributed of the `janus` module.

For example, given Russel's paradox defined in Prolog as below.

```
:- module(russel, [shaves/2]).

:- table shaves/2.

shaves(barber,P) :- person(P),  tnot(shaves(P,P)).
person(barber).
person(mayor).
```

From Python, we may ask who shaves the barber in four ways as illustrated below. Note that the Prolog representations for `janus.DELAY_LISTS` and `janus.RESIDUAL_PROGRAM` use the `write_canonical/1` notation. They may later be changed to use a more human friendly notation.

```
# Using NO_TRUTHVALS
>>> janus.query_once("russel:shaves(barber, X)", truth_vals=janus.NO_TRUTHVALS)
{'truth': True, 'X': 'barber'}

# Using default PLAIN_TRUTHVALS (default)
>>> janus.query_once("russel:shaves(barber, X)")
{'truth': Undefined, 'X': 'barber'}

# Using default DELAY_LISTS
>>> janus.query_once("russel:shaves(barber, X)", truth_vals=janus.DELAY_LISTS)
{'truth': :(russel,shaves(barber,barber)), 'X': 'barber'}

# Using default RESIDUAL_PROGRAM
>>> janus.query_once("russel:shaves(barber, X)", truth_vals=janus.RESIDUAL_PROGRA
{'truth': [:-(:(russel,shaves(barber,barber)),tnot(:(russel,shaves(barber,barber)
```

5.5 Janus class Term

Class `janus.Term()` encapsulates a Prolog term. Similarly to the Python object reference (see `py_is_object/1`), the class allows Python to represent arbitrary Prolog data, typically with the intend to pass it back to Prolog.

`Term` **janus.Term**(**args*)

Instances are never created explicitly by the user. An instance is created by handling a term `prolog`(*Term*) to the data conversion process. As a result, we can do

```
?- py_call(janus:echo(prolog(hello(world))), Obj,
           [py_object(true)]).
Obj = <py_Term>(0x7f7a14512050).
?- py_call(print($Obj)).
```

```
hello(world)
Obj = <py_Term>(0x7f7a14512050).
```

`Term` **janus.Term.__str__**()
Return the output of `print/1` on the term. This is what is used by the Python function `print()`.

`Term` **janus.Term.__repr__**()
Return the output of `write_canonical/1` on the term.

5.6 Janus class PrologError

Class `janus.PrologError()`, derived from the Python class *Exception* represents a Prolog exception that typically results from calling `janus.query_once()`, `janus.apply_once()`, `janus.query()` or `janus.apply()`. The class either encapsulates a string on a Prolog exception term using *janus.Term*. Prolog exceptions are used to represent errors raised by Prolog. Strings are used to represent errors from invalid use of the interface. The default behavior gives the expected message:

```
>>> x = janus.query_once("X is 3.14/0")['X']
Traceback (most recent call last):
  ...
janus.PrologError: //2: Arithmetic: evaluation error: 'zero_divisor'
```

At this moment we only define a single Python class for representing Prolog exceptions. This suffices for error reporting, but does not make it easy to distinguish different Prolog errors. Future versions may improve on that by either subclassing *janus.PrologError* or provide a method to classify the error more easily.

`PrologError` **janus.PrologError**(*TermOrString*)
The constructor may be used explicitly, but this should be very uncommon.

`String` **janus.PrologError.__str__**()
Return a human readable message for the error using `message_to_string/2`

`String` **janus.PrologError.__repr__**()
Return a formal representation of the error by means of `write_canonical/1`.

6 Janus and threads

Where SWI-Prolog support native preemptively scheduled threads that exploit multiple cores, Python has a single interpreter that can switch between native threads.[7] Initially the Python interpreter is associated with the thread that created it which, for janus, is the first thread calling Python. The Prolog

[7] Actually, you can create multiple Python interpreters. It is not yet clear to us whether that can help improving on concurrency.

thread that initiated Janus may terminate. This does not affect the embedded Python interpreter and this interpreter may continue to be used from other Prolog threads.

Janus ensures it holds the Python GIL when interacting with the Python interpreter. If Python calls Prolog, the GIL is released using `Py_BEGIN_ALLOW_THREADS`.

- Multiple Prolog threads can make calls to Python. The access to Python is *serialized*. If a Prolog thread does not want other threads to use Python it can use `py_with_gil/1`. When multiple Prolog threads make many calls to Python performance tends to drop significantly.
- Multiple Python threads can make calls to Prolog. While Prolog is working on the query, the Python interpreter may switch to other Python threads.

6.1 Calling Prolog from a Python thread

Prolog may be called safely from any Python thread. The Prolog execution is embraced with `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS`, which implies that Python is allowed to switch to another thread while Prolog is doing its work.

If the calling Python thread is not the one that initiated Janus, `janus.query_once()` and `janus.query()` attach and detach a temporary Prolog engine using `PL_thread_attach_engine()` and `PL_thread_destroy_engine()`. This is relatively costly. In addition we allow associating a Prolog engine persistently with the calling thread.

`int` **janus.engine**()
: Return the identifier of the Prolog engine associated to the current thread, -1 if no engine is attached or -2 if this version of Prolog does not support engines.

`int` **janus.attach_engine**()
: Attach a Prolog engine to the current thread using `PL_thread_attach_engine()`. On success, return the integer thread id of the created Prolog engine.[8]

  If the thread already has an engine the *attach count* is incremented and the current engine id is returned. The engine is detached after a matching number of calls to `janus.detach_engine()`

`None` **janus.detach_engine**()
: Decrement the *attach count* of the attached Prolog engine. Destroy the engine if this count drops to zero. Raises an exception of the calling thread is not attached to a Prolog engine.

6.2 Python and Prolog deadlocks

In a threaded environment, Python calls must be guarded by `PyGILState_Ensure()` and `PyGILState_Release()` that ultimately lock/unlock a *mutex*. Unfortunately there is no `PyGILState_TryEnsure()` and therefore we may create deadlocks when Prolog locks are involved. This may either apply to explicit Prolog locks from `with_mutex/2` and friends or implicit locks on e.g. I/O streams. The classical scenario is thread A holding the Python GIL and wanting to call Prolog code that locks a mutex M, while thread B holds M and wishes to make a Python call and this tries to lock the GIL. The predicate `py_gil_owner/1` can be used to help diagnosing such issues.

[8] The current implementation passes `NULL` to `PL_thread_attach_engine()`. Future versions may provide access to the creation attributes.

7 Janus and signals

If Prolog is embedded into Python, SWI-Prolog is started with the `--no-signals`, i.e., SWI-Prolog does not install any signal handlers. This implies that signals are handled by Python. Python handles signals synchronously (as SWI-Prolog) when executing byte code. As Prolog execution does not involve Prolog execution, running a program like below cannot be in interrupted

```
import janus_swi as janus
janus.query_once("repeat,fail")
```

If your program makes possibly slow Prolog queries and you want signal handling, you can enable a *heartbeat*.

`None` **janus.heartbeat**(*count=10000*)
: Ask Prolog to call a dummy function every *count inferences*. This allows Python to handle signals. Lower numbers for *count* improve responsiveness at the cost of slowing down Prolog. Note that Prolog calls to *foreign code* count as one inference. Signal handling is completely blocked if Prolog is blocked in foreign code.

To complete the picture, some Python exceptions are propagated through Prolog by mapping them into a Prolog exception and back again. This notably concerns

SystemExit(*code*)
: This Python exception is mapped to the Prolog exception `unwind(halt(`*code*`))` and back again when Prolog returns control back to Python.

KeyboardInterrupt
: This Python exception is mapped to the Prolog exception `unwind(`*keyboard_interrupt*`)`

8 Janus versions

The current version as an integer can be accessed as `janus.version`. The integer uses the same conventions as the SWI-Prolog flag `version` and is defined as $10,000 * Major + 100 * Minor + Patch$. In addition, the module defines the following functions:

`str` **janus.version_str**()
: Return the Janus version as a string *Major.Minor.Patch*.

`None` **janus.version**()
: Print information about Janus and SWI-Prolog version.

9 Janus as a Python package

The Janus GIT repo provides `setup.py`. Janus may be installed as a Python package after downloading using

```
pip install .
```

pip allows for installation from the git repository in a one-liner as below.

```
pip install git+https://github.com/SWI-Prolog/packages-swipy.git#egg=janus_swi
```

Installing janus as a Python package requires

- The swipl program in the default search path. The setup.py runs swipl --dump-runtime-variables to obtain the installation locations of the various Prolog components. On Windows, if swipl is not on %PATH%, setup.py tries the registry to find the default binary installation.
- A C compiler that can be used by pip. The janus interface has been tested to compile using GCC, Clang and Microsoft Visual C++.

After successful installation we should be able to use Prolog directly from Python. For example:

```
python
>>> from janus_swi import *
>>> query_once("writeln('Hello world!')")
Hello world!
{'truth': True}
>>> [a["D"] for a in query("between(1,6,D)")]
[1, 2, 3, 4, 5, 6]
>>> prolog()
?- version.
Welcome to SWI-Prolog (threaded, 64 bits, version 9.1.12-8-g70b70a968-DIRTY)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
...
?-
```

10 Prolog and Python

Prolog is a very different language than imperative languages. An interesting similarity is the notion of *backtracking* vs. Python *iterators*.

To be extended.

11 Janus performance evaluation

Below is a table to give some feeling on the overhead of making calls between Prolog and Python. These figures are roughly the same as the figures for the XSB/Python interface. All benchmarks have been executed on AMD3950X running Ubuntu 22.04, SWI-Prolog 9.1.11 and Python 3.10.6.

Action	Time (seconds)
Echo list with 1,000,000 elements	0.12
Call Pyton `demo:int()` from Prolog 1,000,000 times	0.44
Call Pyton `demo:sumlist3(5,[1,2,3])` from Prolog 1,000,000 times	1.4
Call Prolog `Y is X+1` from Python 1,000,000 times	1.9
Iterate from Python over Prolog goal `between(1, 1 000 000, X)`	1.1
Iterate over Python iterator `range(1,1000000)` from Prolog	0.17

12 Python or C/C++ for accessing resources?

Using Python as an intermediate to access external resources allows writing such interfaces with less effort by a much wider community. The resulting interface is often also more robust due to well defined data conversion and sound memory management that you get for free.

Nevertheless, Python often accesses resources with a C or C++ API. We can also create this bridge directly, bypassing Python. That avoids one layer of data conversion and preserves the excellent multi-threading capabilities of SWI-Prolog. As is, Python operations are synchronized using the Python *GIL*, a global lock that allows for only a single thread to use Python at the same time.[9]

Writing an interface for SWI-Prolog is typically easier that for Python/C because memory management is easier. Where we need to manage reference counts to Python objects through all possibly paths of the C functions, SWI-Prolog `term_t` merely has to be allocated once in the function. All failure parts will discard the Prolog data automatically through backtracking and all success paths will do so through the Prolog garbage collector.[10]

Summarizing, Janus is ideal to get started quickly. Applications that need to access C/C++ resources and need either exploit all cores of your hardware or get the best performance on calls or exchanging data should consider using the C or C++ interfaces of SWI-Prolog.

13 Janus platforms notes

Janus relies on the C APIs of Prolog and Python and functions therefore independent from the platform. While the C, Python and Prolog code the builds Janus is platform independent, dynamically loading Prolog into Python or Python into Prolog depends on versions as well as several properties of the dynamic linking performed by the platform. In the sections below we describe some of the issues.

13.1 Janus on Windows

We tested the Windows platform using SWI-Prolog binaries from https://www.swi-prolog.org/Downloads.html and Python downloaded from https://www.python.org/downloads/windows/. The SWI-Prolog binary provides `janus.dll` which is linked to `python3.dll`, a "stable API" based wrapper that each Python 3 binary

[9] There are rumors that Python's multi threading will be able to use multiple cores.

[10] Using a Python C++ interface such as pybind11 simplifies memory management for a Python interface.

distribution provides in addition to `python3xx.dll`. Calling Python from Prolog is supported out of the box, provided the folder holding `python3.dll` is in the search `%PATH%`.

The Python package can be installed using `pip` as described in section 9. Once built, this package finds SWI-Prolog on `%PATH%` or using the registry and should be fairly independent from the Prolog version as long as it is version 9.1.12 or later.

13.2 Janus on Linux

On Linux systems we bind to the currently installed Prolog and Python version. This should work smoothly from source. Janus is included in the PPA distribution for Ubuntu as well as in the Docker images. It is currently not part of the SNAP distribution.

See section 9 for for building the `janus_swi` Python package.

13.3 Janus on MacOS

Unfortunately MacOS versions of Python do not ship with the equivalent of `python3.dll` found on Windows. This implies we can only compile our binaries against a specific version of Python. We will use the default Python binary for that, which is installed in `/Library/Frameworks/Python.framework/`

The Macports version is also linked against an explicit version of Python, in this case provided by Macports.

The Python package `janus_swi` may be compiled against any version of Python selected by `pip`. See section 9 for details.

14 Compatibility to the XSB Janus implementation

We aim to provide an interface that is close enough to allow developing Prolog code that uses Python and visa versa. Differences between the two Prolog implementation make this non-trivial. SWI-Prolog has native support for *dicts*, *strings*, *unbounded integers*, *rational numbers* and *blobs* that provide safe pointers to external objects that are subject to (atom) garbage collection.

We try to find a compromise to make the data conversion as close as possible while supporting both systems as good as possible. For this reason we support creating a Python dict both from a SWI-Prolog dict and from the Prolog term `py({k1:v1, k2:v2, ...})`. With `py` defined as a prefix operator, this may be written without parenthesis and is thus equivalent to the SWI-Prolog dict syntax. The `janus` library provides access predicates that are supported by both systems and where the SWI-Prolog version supports both SWI-Prolog dicts and the above Prolog representation. See `items/2`, `values/3`, `key/2` and `items/2`.

Calling Python from Prolog provides a low-level and a more high level interface. The high level interface is realized by `py_call/[2,3]` and `py_iter/[2,3]`. We realize the low level interfaces `py_func/[3,4]` and `py_dot/[4,5]` on top of `py_call/2`. The interface for calling Prolog from Python is settled on the five primitives described in section 5.

We are discussing to minimize the differences. Below we summarize the known differences.

- SWI-Prolog represents Phyton dicts as Prolog dicts. XSB uses a term py({k:v, ...}), where the `py()` wrapper is optional. The predicate `py_is_dict/1` may be used to test that a Prolog term represents a Python dict. The predicates `values/3`, `keys/2`, `key/2` and `items/2` can be used to access either representation.

- SWI-Prolog allows for `prolog`(*Term*) to be sent to Python, creating an instance of `janus.Term()`.
- SWI-Prolog represents Python object references as a *blob*. XSB uses a term. The predicate `py_is_object/1` may be used to test that a Prolog term refers to a Python object. In XSB, the user *must* call `py_free/1` when done with some object. In SWI-Prolog, either `py_free/1` may be used or the object may be left to the Prolog (atom) garbage collector.
- Prolog exceptions passed to Python are represented differently.
- When calling Prolog from Python and relying on well founded semantics, only *plain truth values* (i.e., `janus.undefined`) are supported in a portable way. *Delay lists*, providing details on why the result is undefined, are represented differently.

14.1 Writing portable Janus modules

This section will be written after the dust has settled. Topics

- Dealing with Python dicts
- Dealing with Prolog modules
- Dealing with Prolog references to Python objects
- More?

15 Status of Janus

The current version of this Janus library must be considered *beta* code.

- The design is stable
- Naming and functionality are almost stable.
- Testing is not exhaustive.

References

[Andersen & Swift, 2023] Carl Andersen and Theresa Swift. The janus system: A bridge to new prolog applications. In David Scott Warren, Verónica Dahl, Thomas Eiter, Manuel V. Hermenegildo, Robert A. Kowalski, and Francesca Rossi, editors, *Prolog: The Next 50 Years*, volume 13900 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2023.

[Swift & Andersen, 2023] Theresa Swift and Carl Andersen. The janus system: Multi-paradigm programming in prolog and python. *CoRR*, abs/2308.15893, 2023.

Index