PyCLIPS Manual

Release 1.0

Francesco Garosi

Feb 22, 2008

E-mail: franz -dot- g -at- tin -dot- it

Abstract

This manual documents the high-level interface to the CLIPS system provided by the `PyCLIPS` module. This module allows the creation of a fully-functional CLIPS environment in a Python session, thus providing access to a well-known expert systems shell programmatically.

`PyCLIPS` incorporates most of the API bindings documented in the *Clips Reference Guide Vol. II: Advanced Programming Guide* (freely downloadable from the CLIPS web site, see below) and embeds these bindings in an Object Oriented layer, incorporating most CLIPS constructs into Python classes. Instances of these classes allow access to the status and functionality of the corresponding CLIPS objects.

Note: This manual is not intended to be a documentation of CLIPS itself. CLIPS is extensively documented in several manuals, which are available on the CLIPS website (see below). A comprehensive tutorial for CLIPS can also be downloaded. A reasonable knowledge of the CLIPS system is necessary to understand and use this module.

See Also:

Python Language Web Site
(http://www.python.org/)
for information on the Python language

CLIPS system web site
(http://www.ghg.net/clips/CLIPS.html)
for information on the CLIPS system

CONTENTS

CHAPTER

ONE

Introduction

1.1 Overview

This module aims to embed a fully functional CLIPS engine in Python, and to give to the developer a more Python-compliant interface to CLIPS without cutting down on functionalities. In fact CLIPS is compiled into the module in its entirety, and most API functions are bound to Python methods. However the direct bindings to the CLIPS library (implemented as the `_clips` submodule) are not described here: each function is described by an appropriate documentation string, and accessible by means of the `help()` function or through the **pydoc** tool. Each direct binding maps to an API provided function. For a detailed reference[1] for these functions see *Clips Reference Guide Vol. II: Advanced Programming Guide*, available for download at the CLIPS website.

`PyCLIPS` is also capable of generating CLIPS text and binary files: this allows the user to interact with sessions of the CLIPS system itself.

An important thing to know, is that `PyCLIPS` implements CLIPS as a separated[2] engine: in the CLIPS module implementation, CLIPS "lives" in its own memory space, allocates its own objects. The module only provides a way to send information and commands to this engine and to retrieve results from it.

1.1.1 Structure

`PyCLIPS` is organized in a package providing several classes and top-level functions. Also, the module provides some objects that are already instanced and give access to some of the CLIPS internal functions and structures, including debug status and engine I/O.

CLIPS is accessible through these classes and functions, that send appropriate commands to the underlying engine and retrieve the available information. Many of the CLIPS classes, constructs and objects are shadowed by Python classes and objects. However, whereas `PyCLIPS` classes provide a comfortable way to create objects that reference the actual engine objects, there is no one-to-one mapping between the two memory spaces: for instance, when a Python object is deleted (via the `del` command), the corresponding CLIPS object will still remain alive in the CLIPS memory space. An appropriate command is necessary to remove the object from the underlying engine, and this is provided by the module interface.

1.1.2 Interactive Usage

The `PyCLIPS` package can also be used interactively, since it can inspect an underlying CLIPS session and give some of the output that CLIPS usually provides when used as an interactive shell.

[1] The order of parameters is changed sometimes, in order to allow a more intuitive use of default parameters in the Python interface: however the meaning of each parameter is described in the function documentation string, and it should not be difficult for the programmer to correctly understand the relationship between a module function and the corresponding CLIPS API.

[2] This has an impact on the way the module can be used, as the engine is only set up once when the module is `import`ed the first time.

A simple interactive session with PyCLIPS follows:

```
>>> import clips
>>> clips.Reset()
>>> clips.Assert("(duck)")
<Fact 'f-1': fact object at 0x00DE4AE0>
>>> clips.BuildRule("duck-rule", "(duck)", "(assert (quack))", "the Duck Rule")
<Rule 'duck-rule': defrule object at 0x00DA7E00>
>>> clips.PrintRules()
MAIN:
duck-rule
>>> clips.PrintAgenda()
MAIN:
   0      duck-rule: f-1
For a total of 1 activation.
>>> clips.PrintFacts()
f-0     (initial-fact)
f-1     (duck)
For a total of 2 facts.
>>> clips.Run()
>>> clips.PrintFacts()
f-0     (initial-fact)
f-1     (duck)
f-2     (quack)
For a total of 3 facts.
```

Users of the CLIPS interactive shell will find the PyCLIPS output quite familiar. In fact the Print*<object>*() functions are provided for interactive use, and retrieve their output directly from the underlying CLIPS engine I/O subsystem, in order to resemble an interactive CLIPS session. Other functions are present to retrieve object names, values and the so called *pretty-print forms* for programmatic use.

1.2 Implementation Structure

This section describes the guidelines and considerations that lead to this CLIPS interface implementation. For the developers which normally use CLIPS as a development environment or expert systems shell, the architecture of the PyCLIPS module will look a little bit different, and in some ways could also seem confusing.

The main topics covered by these sections are the *Implementation of Constructs as Classes*, the *Implementation of CLIPS I/O Subsystem*, the *Configuration and Debug Objects*, the *Coexistence of Global and Environment-Aware Engines* and the *Conventions used for Naming* which explains the rules that helped choose the current naming scheme for classes, functions and objects implemented in PyCLIPS.

1.2.1 Implementation of Constructs as Classes

CLIPS users know that this shell offers several constructs to populate the system memory. These constructs are not described here, since a detailed explaination of the CLIPS language can be found in the official CLIPS documentation. These constructs, many of which have their particular syntax, create "objects" (not necessarily in the sense of OOP[3], although some of these can be Instances of Classes) in the subsystem memory.

The choice of implementing most of these constructs as classes gives to PyCLIPS a more organic structure. Most of the construct classes share similarities which make the interface structure simpler and the access to CLIPS objects

[3] In fact, the word *object* is used here to indicate an item that takes part in a program. For instance, one such object can be a Rule: it is not a proper OO object, but something that in imperative languages would act as a control structure.

more systematic.

Most constructs are implemented as *factory functions* which return instances of Python classes. These Python instances (that shadow the corresponding CLIPS objects), on their turn, have methods and properties which operate directly on the objects they map in the CLIPS subsystem. Methods and properties provide both access and *send messages* to these objects.

An example of this follows:

```
>>> import clips
>>> f0 = clips.Assert("(duck)")
>>> print f0
f-0
>>> print f0.Exists()
True
>>> f0.Retract()
>>> print f0.Exists()
False
```

In the above example, a fact (`(duck)`) is asserted and then retracted. The assertion is done by means of a module-level function (`Assert()`) and the fact is retracted using a method of the shadow object (*f0*). A verification on the CLIPS `fact` object, using the Python `Fact` instance[4] *f0*'s method `Exists()`, shows that after invoking the `Retract()` method of *f0* the `fact` object does no longer exists in the CLIPS subsystem, thus it has been actually retracted.

As stated previously, this does not remove the Python object (a `Fact` instance) from the namespace pertinent to Python itself: as it can be seen from the code snip shown above, `f0` is still a functional instance, and can be queried about the existence of the corresponding object in CLIPS.

Objects in the CLIPS operating space can be referenced by more than a Python object (or even not be referenced at all, if CLIPS creation does not correspond to a Python assignment), as demonstrated by the following code:

```
>>> clips.Reset()
>>> f1 = clips.Assert("(duck)")
>>> clips.Assert("(quack)")
<Fact 'f-2': fact object at 0x00DE8420>
>>> f1
<Fact 'f-1': fact object at 0x00DE3020>
>>> fl = clips.FactList()
>>> f1b = fl[1]
>>> f1b
<Fact 'f-1': fact object at 0x00E08C40>
```

Both *f1* and *f1b* refer to the same object in CLIPS namespace, but their address is different: in fact they are two different Python objects (equality test fails) but correspond to the same `fact` in CLIPS.

See Also:

Clips Reference Guide Vol. I: Basic Programming Guide
the main reference for the CLIPS language

[4] It should be clear that terminology differs semantically from Python system to the CLIPS system: while OOP, to which terms used in Python are coherent, uses the words *method*, *instance* and so on with a particular meaning (to which Python developers are familiar), CLIPS terminology often differs from OOP, sometimes only slightly but at other times more substantially. The reader should note that throughout this manual each term is used – as far as possible – with the meaning that it assumes in its specific environment. In this case, the word *instance* represents the instance of a Python `class`, and is not referred to an entity in CLIPS.

1.2.2 Implementation of CLIPS I/O Subsystem

The CLIPS shell interacts with the user answering to typed in commands with some informational output. An interactive CLIPS session will show this:

```
CLIPS> (reset)
CLIPS> (watch activations)
CLIPS> (defrule duck-rule "the Duck Rule"
   (duck)
=>
   (assert (quack)))
CLIPS> (assert (duck))
==> Activation 0      duck-rule: f-1
<Fact-1>
CLIPS> (run)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (duck)
f-2     (quack)
For a total of 3 facts.
```

Each time a `fact` is asserted, CLIPS outputs a string containing its index, and since we decided to show some debug output about activations, CLIPS produces a line as soon as `duck` is asserted, since `duck-rule` would be activated by this. Although in an interactive session all of the output would go to the terminal, CLIPS logically considers the "streams" for different output types as separated: in fact, debug output (the one generated by the `watch` command) goes to a special stream called `wtrace`. In this special case, for instance, the debug output can be captured by `PyCLIPS` through a special stream-like Python object, which provides a `Read()` function[5]. Comparing the behaviour of two interactive sessions, the former in the CLIPS subsystem and the latter in Python, will help to understand the close relationship between CLIPS I/O and `PyCLIPS` stream objects. CLIPS will interact with the user as follows:

```
CLIPS> (defrule sayhello
   (hello)
=>
   (printout t "hello, world!" crlf))
CLIPS> (assert (hello))
==> Activation 0      sayhello: f-1
<Fact-1>
CLIPS> (run)
hello, world!
```

And the Python counterpart follows:

[5] Note that `Read()` is capitalized: this is because the stream-like objects do not really act as "files" as many objects which can be read in Python do. So it becomes impossible to use these `PyCLIPS` objects where a file-like object is to be used.

```
>>> import clips
>>> clips.DebugConfig.ActivationsWatched = True
>>> r0 = clips.BuildRule("sayhello", "(hello)",
                         '(printout stdout "hello, world!" crlf)')
>>> print r0.PPForm()
(defrule MAIN::sayhello
   (hello)
   =>
   (printout stdout "hello, world!" crlf))

>>> clips.Assert("(hello)")
<Fact 'f-0': fact object at 0x00DE81C0>
>>> t = clips.TraceStream.Read()
>>> print t
==> Activation 0      sayhello: f-0
>>> clips.Run()
>>> t = clips.StdoutStream.Read()
>>> print t
hello, world!
```

The I/O access objects can be used both in interactive and unattended sessions. In this case, they can be useful to retrieve periodical information about CLIPS internal status, since most of the output provided can be easily interpreted in a programmatic way. Also, there is one more stream called *StdinStream* (which has a `Write()` method) that might be useful to send input to the CLIPS engine when some "user interaction" is required[6].

There is no way to create other instances of these streams: the high-level module hides the class used to build these objects. This is because the I/O streams have to be considered like "physical devices" whose use is reserved to the engine to report trace and debug information as well as user requested output.

These I/O streams will be described later in the detail, since each one can be used to report about a specific task.

Note: The streams have to be explicitly read: there is no way to receive a notification from CLIPS that some output has been written. In other words, the `PyCLIPS` engine is not *event driven* in its interaction with Python.

1.2.3 Configuration and Debug Objects

As well as I/O streams, there are two other objects directly provided by `PyCLIPS`. These objects provide access to the CLIPS engine global configuration. Many aspects of the CLIPS engine, that in the command line environment would be configured using particular commands, are accessible via the *EngineConfig* (global engine configuration) object and the *DebugConfig* (global debug and trace configuration) object. For example, we can take the code snip shown above:

```
>>> clips.DebugConfig.ActivationsWatched = True
```

(...)

```
>>> t = clips.TraceStream.Read()
>>> print t
==> Activation 0      sayhello: f-0
```

The `clips.DebugConfig.ActivationsWatched = True` line tells to the underlying subsystem that debug

[6] This only happens actually when CLIPS invokes the `read` or `readline` functions.

information about *rule activations* has to be written to the proper stream (the stream dedicated to debug output in CLIPS is called `wtrace` and is accessible in `PyCLIPS` through the *TraceStream* object).

As it has been said for the I/O streams, these objects cannot be instanced by the user: access to these objects affects global (or at least *environmental*, we will see the difference later) configuration, so it would be of no meaning for the user to create more, possibly confusing, instances of such objects.

1.2.4 Coexistence of Global and Environment-Aware Engines

As of version 6.20, CLIPS API offers the possibility to have several *environments* in which to operate. We can consider environments as separate engines that only share the operating mode, in other words "the code". `PyCLIPS` also implements environments by means of a special `Environment` class. This class implements all the features provided by the top level methods and classes. The `Environment` class reimplements all classes provided by `PyCLIPS`, but – although their behaviour is quite similar – methods of classes provided by `Environment` only affect the CLIPS environment represented by the `Environment` instance itself.

There is normally no need to use environments. However, access to them is provided for CLIPS "gurus" who want to have more than one engine working at the same time. The end user of `PyCLIPS` will see no real difference between a call to a function and its environmental counterpart (defined as *companion function* in the official CLIPS documentation), apart from being called as a member function of an `Environment` object.

A simple example will be explanatory:

```
>>> clips.Clear()
>>> clips.Reset()
>>> e0 = clips.Environment()
>>> e1 = clips.Environment()
>>> e0.Assert("(duck)")
<Fact 'f-0': fact object at 0x00E7D960>
>>> e1.Assert("(quack)")
<Fact 'f-0': fact object at 0x00E82220>
>>> e0.PrintFacts()
f-0     (duck)
For a total of 1 fact.
>>> e1.PrintFacts()
f-0     (quack)
For a total of 1 fact.
```

1.2.5 Using External Functions in CLIPS

`PyCLIPS` gives the ability to users to call Python code from within the CLIPS subsystem. Virtually every function defined in Python can be called from CLIPS code using the special CLIPS function `python-call`. However, since CLIPS has different basic types than Python, in most cases it would be useful for modules that implement function to be called in the CLIPS engine to import the `PyCLIPS` module themselves, in order to be aware of the structures that CLIPS uses.

Functions have to be registered in `PyCLIPS` in order to be available to the underlying engine, and the registration process can dynamically occur at any moment.

A simple example follows:

```
>>> import clips
>>> def py_square(x):
        return x * x
>>> clips.RegisterPythonFunction(py_square)
>>> print clips.Eval("(python-call py_square 7)")
49
>>> print clips.Eval("(python-call py_square 0.7)")
0.49
```

A more detailed description of the features provided by `python-call` can be found in the appendices.

1.2.6 Conventions Used for Naming

In `PyCLIPS`, the simple convention that is used is that all valuable content exposed has a name beginning with a capital letter. Names beginning with a single underscore have normally no meaning for the `PyCLIPS` user. Functions, class names and objects use mixed capitals (as in Java), and *manifest constants* (names used in *lieu* of explicit values to pass instructions to CLIPS functions or properties) are all capitalized, as is usual for the C language.

CLIPS users will perhaps be confused because often the constructs in CLIPS are expressed by keywords containing a `def` prefix. The choice was made in `PyCLIPS` to drop this prefix in many cases: the use of this prefix has a strong logic in the CLIPS language, because in this way the developer knows that a *construct* is used, that is, a *definition* is made. The keyword used to instance this definition, both encapsulates the meaning of `def`inition itself, and also the type of construct that is being defined (e.g. `def`ine a `rule` is `defrule`), thus avoiding making constructs more difficult by means of two separate keywords. In `PyCLIPS`, since the definition happens at class declaration and the instantiation of classes shadows a construct definition when it has already been performed, it seemed unnecessary to keep the prefix: in fact, to follow the above example, it does not seem correct to refer to a rule within the CLIPS subsystem as a "`Defrule`" object, hence it is simply referred to as a `Rule`.

1.2.7 Pickling Errors

Python objects cannot be pickled or unpickled. This is because, since pickling an object would save a reference to a CLIPS entity – which is useless across different `PyCLIPS` sessions – the unpickling process would feed the underlying engine in an unpredictable way, or at least would reference memory locations corresponding to previous CLIPS entities without the engine having them allocated.

One better way to achieve a similar goal is to use the `Save()` or `BSave()` (and related `Load()` or `BLoad()`) to save the engine[7] status in its entirety.

If a single entity is needed, its *pretty-print form* can be used in most cases to recreate it using the `Build()` functions.

1.3 Other Usage Modes

It is also interesting that, by using some particular functions and the provided I/O subsystem, even "pure" CLIPS programs can be executed by `PyCLIPS`, and while the simple output from CLIPS can be read to obtain feedback, the possibility of inspecting the internal CLIPS subsystem state remains.

The following example, taken from the CLIPS website[8], illustrates this: first we take a full CLIPS program, saved as 'zebra.clp', and reported below:

[7] The mentioned functions are also *members* of the `Environment` class, in which case the `Environment` status is saved.

[8] In fact the file has been slightly reformatted for typesetting reasons.

```
;;;======================================================
;;;   Who Drinks Water? And Who owns the Zebra?
;;;
;;;     Another puzzle problem in which there are five
;;;     houses, each of a different color, inhabited by
;;;     men of different nationalities, with different
;;;     pets, drinks, and cigarettes. Given the initial
;;;     set of conditions, it must be determined which
;;;     attributes are assigned to each man.
;;;
;;;     CLIPS Version 6.0 Example
;;;
;;;     To execute, merely load, reset and run.
;;;======================================================

(deftemplate avh (field a) (field v) (field h))

(defrule find-solution
  ; The Englishman lives in the red house.

(avh (a nationality) (v englishman) (h ?n1))
  (avh (a color) (v red) (h ?c1&?n1))

; The Spaniard owns the dog.

  (avh (a nationality) (v spaniard) (h ?n2&~?n1))
  (avh (a pet) (v dog) (h ?p1&?n2))

; The ivory house is immediately to the left of the green house,
  ; where the coffee drinker lives.

  (avh (a color) (v ivory) (h ?c2&~?c1))
  (avh (a color) (v green) (h ?c3&~?c2&~?c1&=(+ ?c2 1)))
  (avh (a drink) (v coffee) (h ?d1&?c3))

; The milk drinker lives in the middle house.

  (avh (a drink) (v milk) (h ?d2&~?d1&3))

; The man who smokes Old Golds also keeps snails.

  (avh (a smokes) (v old-golds) (h ?s1))
  (avh (a pet) (v snails) (h ?p2&~?p1&?s1))

; The Ukrainian drinks tea.

  (avh (a nationality) (v ukrainian) (h ?n3&~?n2&~?n1))
  (avh (a drink) (v tea) (h ?d3&~?d2&~?d1&?n3))

; The Norwegian resides in the first house on the left.

  (avh (a nationality) (v norwegian) (h ?n4&~?n3&~?n2&~?n1&1))

; Chesterfields smoker lives next door to the fox owner.

  (avh (a smokes) (v chesterfields) (h ?s2&~?s1))
  (avh (a pet) (v fox)
    (h ?p3&~?p2&~?p1&:(or (= ?s2 (- ?p3 1)) (= ?s2 (+ ?p3 1)))))
```

```
; The Lucky Strike smoker drinks orange juice.

  (avh (a smokes) (v lucky-strikes) (h ?s3&~?s2&~?s1))
  (avh (a drink) (v orange-juice) (h ?d4&~?d3&~?d2&~?d1&?s3))

; The Japanese smokes Parliaments

  (avh (a nationality) (v japanese) (h ?n5&~?n4&~?n3&~?n2&~?n1))
  (avh (a smokes) (v parliaments) (h ?s4&~?s3&~?s2&~?s1&?n5))

; The horse owner lives next to the Kools smoker,
  ; whose house is yellow.

  (avh (a pet) (v horse) (h ?p4&~?p3&~?p2&~?p1))
  (avh (a smokes) (v kools)
    (h ?s5&~?s4&~?s3&~?s2&~?s1&:(or (= ?p4 (- ?s5 1)) (= ?p4 (+ ?s5 1)))))
  (avh (a color) (v yellow) (h ?c4&~?c3&~?c2&~?c1&?s5))

; The Norwegian lives next to the blue house.

  (avh (a color) (v blue)
    (h ?c5&~?c4&~?c3&~?c2&~?c1&:(or (= ?c5 (- ?n4 1)) (= ?c5 (+ ?n4 1)))))

; Who drinks water?  And Who owns the zebra?

  (avh (a drink) (v water) (h ?d5&~?d4&~?d3&~?d2&~?d1))
  (avh (a pet) (v zebra) (h ?p5&~?p4&~?p3&~?p2&~?p1))

=>
  (assert (solution nationality englishman ?n1)
          (solution color red ?c1)
          (solution nationality spaniard ?n2)
          (solution pet dog ?p1)
          (solution color ivory ?c2)
          (solution color green ?c3)
          (solution drink coffee ?d1)
          (solution drink milk ?d2)
          (solution smokes old-golds ?s1)
          (solution pet snails ?p2)
          (solution nationality ukrainian ?n3)
          (solution drink tea ?d3)
          (solution nationality norwegian ?n4)
          (solution smokes chesterfields ?s2)
          (solution pet fox ?p3)
          (solution smokes lucky-strikes ?s3)
          (solution drink orange-juice ?d4)
          (solution nationality japanese ?n5)
          (solution smokes parliaments ?s4)
          (solution pet horse ?p4)
          (solution smokes kools ?s5)
          (solution color yellow ?c4)
          (solution color blue ?c5)
          (solution drink water ?d5)
          (solution pet zebra ?p5))
  )

(defrule print-solution
  ?f1 <- (solution nationality ?n1 1)
  ?f2 <- (solution color ?c1 1)
```

```
  ?f3 <- (solution pet ?p1 1)
  ?f4 <- (solution drink ?d1 1)
  ?f5 <- (solution smokes ?s1 1)
  ?f6 <- (solution nationality ?n2 2)
  ?f7 <- (solution color ?c2 2)
  ?f8 <- (solution pet ?p2 2)
  ?f9 <- (solution drink ?d2 2)
  ?f10 <- (solution smokes ?s2 2)
  ?f11 <- (solution nationality ?n3 3)
  ?f12 <- (solution color ?c3 3)
  ?f13 <- (solution pet ?p3 3)
  ?f14 <- (solution drink ?d3 3)
  ?f15 <- (solution smokes ?s3 3)
  ?f16 <- (solution nationality ?n4 4)
  ?f17 <- (solution color ?c4 4)
  ?f18 <- (solution pet ?p4 4)
  ?f19 <- (solution drink ?d4 4)
  ?f20 <- (solution smokes ?s4 4)
  ?f21 <- (solution nationality ?n5 5)
  ?f22 <- (solution color ?c5 5)
  ?f23 <- (solution pet ?p5 5)
  ?f24 <- (solution drink ?d5 5)
  ?f25 <- (solution smokes ?s5 5)
  =>
 (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7 ?f8 ?f9 ?f10 ?f11 ?f12 ?f13 ?f14
          ?f15 ?f16 ?f17 ?f18 ?f19 ?f20 ?f21 ?f22 ?f23 ?f24 ?f25)

(format t "HOUSE | %-11s | %-6s | %-6s | %-12s | %-13s%n"
           Nationality Color Pet Drink Smokes)
 (format t "-----------------------------------------------------------------%n")
 (format t "  1   | %-11s | %-6s | %-6s | %-12s | %-13s%n" ?n1 ?c1 ?p1 ?d1 ?s1)
 (format t "  2   | %-11s | %-6s | %-6s | %-12s | %-13s%n" ?n2 ?c2 ?p2 ?d2 ?s2)
 (format t "  3   | %-11s | %-6s | %-6s | %-12s | %-13s%n" ?n3 ?c3 ?p3 ?d3 ?s3)
 (format t "  4   | %-11s | %-6s | %-6s | %-12s | %-13s%n" ?n4 ?c4 ?p4 ?d4 ?s4)
 (format t "  5   | %-11s | %-6s | %-6s | %-12s | %-13s%n" ?n5 ?c5 ?p5 ?d5 ?s5)
 (printout t crlf crlf))

(defrule startup
   =>
 (printout t
 "There are five houses, each of a different color, inhabited by men of"  crlf
 "different nationalities, with different pets, drinks, and cigarettes."  crlf
 crlf
 "The Englishman lives in the red house.  The Spaniard owns the dog."     crlf
 "The ivory house is immediately to the left of the green house, where"   crlf
 "the coffee drinker lives.  The milk drinker lives in the middle house." crlf
 "The man who smokes Old Golds also keeps snails.  The Ukrainian drinks"  crlf
 "tea.  The Norwegian resides in the first house on the left.  The"       crlf)
 (printout t
 "Chesterfields smoker lives next door to the fox owner.  The Lucky"      crlf
 "Strike smoker drinks orange juice.  The Japanese smokes Parliaments."   crlf
 "The horse owner lives next to the Kools smoker, whose house is yellow." crlf
 "The Norwegian lives next to the blue house."      t
 crlf
  "Now, who drinks water?  And who owns the zebra?" crlf crlf)
   (assert (value color red)
           (value color green)
           (value color ivory)
           (value color yellow)
```

```
            (value color blue)
            (value nationality englishman)
            (value nationality spaniard)
            (value nationality ukrainian)
            (value nationality norwegian)
            (value nationality japanese)
            (value pet dog)
            (value pet snails)
            (value pet fox)
            (value pet horse)
            (value pet zebra)
            (value drink water)
            (value drink coffee)
            (value drink milk)
            (value drink orange-juice)
            (value drink tea)
            (value smokes old-golds)
            (value smokes kools)
            (value smokes chesterfields)
            (value smokes lucky-strikes)
            (value smokes parliaments))
   )

(defrule generate-combinations
   ?f <- (value ?s ?e)
   =>
   (retract ?f)
   (assert (avh (a ?s) (v ?e) (h 1))
           (avh (a ?s) (v ?e) (h 2))
           (avh (a ?s) (v ?e) (h 3))
           (avh (a ?s) (v ?e) (h 4))
           (avh (a ?s) (v ?e) (h 5))))
```

then we execute all commands (using the `BatchStar()` function) in the current `Environment` of an interactive `PyCLIPS` session:

```
>>> clips.BatchStar("zebra.clp")
>>> clips.Reset()
>>> clips.Run()
>>> s = clips.StdoutStream.Read()
>>> print s
There are five houses, each of a different color, inhabited by men of
different nationalities, with different pets, drinks, and cigarettes.

The Englishman lives in the red house.  The Spaniard owns the dog.
The ivory house is immediately to the left of the green house, where
the coffee drinker lives.  The milk drinker lives in the middle house.
The man who smokes Old Golds also keeps snails.  The Ukrainian drinks
tea.  The Norwegian resides in the first house on the left.  The
Chesterfields smoker lives next door to the fox owner.  The Lucky
Strike smoker drinks orange juice.  The Japanese smokes Parliaments.
The horse owner lives next to the Kools smoker, whose house is yellow.
The Norwegian lives next to the blue house.

Now, who drinks water?  And who owns the zebra?

HOUSE | Nationality | Color  | Pet    | Drink        | Smokes
-----------------------------------------------------------------
  1   | norwegian   | yellow | fox    | water        | kools
  2   | ukrainian   | blue   | horse  | tea          | chesterfields
  3   | englishman  | red    | snails | milk         | old-golds
  4   | spaniard    | ivory  | dog    | orange-juice | lucky-strikes
  5   | japanese    | green  | zebra  | coffee       | parliaments

>>> clips.PrintFacts()
f-0     (initial-fact)
f-26    (avh (a smokes) (v parliaments) (h 1))
f-27    (avh (a smokes) (v parliaments) (h 2))
f-28    (avh (a smokes) (v parliaments) (h 3))
```

[... a long list of facts ...]

```
f-150   (avh (a color) (v red) (h 5))
For a total of 126 facts.
>>> li = clips.FactList()
>>> for x in li:
... if str(x) == 'f-52':
... f52 = x
>>> f52
<Fact 'f-52': fact object at 0x00E6AA10>
>>> print f52.PPForm()
f-52    (avh (a drink) (v tea) (h 2))
```

You can just copy the program above to a file, say ‘zebra.clp’ as in the example, and follow the same steps to experiment with PyCLIPS objects and with the CLIPS subsystem.

CHAPTER

TWO

Module Contents

This chapter gives a detailed description of top-level functions and constants in the `PyCLIPS` module. It's not intended to be a CLIPS reference: the official CLIPS Reference Guide still remains the main source of documentation for this. This Guide will often be referred to for information about the engine itself, and the user is expected to know the CLIPS language and structure sufficiently with respect to his goals.

Although the `PyCLIPS` user is not supposed to know CLIPS API, it is advisable to have its reference at least at hand to sometimes understand *how* `PyCLIPS` interacts with CLIPS itself. Besides the programmatic approach offered by `PyCLIPS` is vastly different to the C API – with the occasional exception of the intrinsic logic.

We will first describe the top level functions.

2.1 Top Level Functions and Constants

2.1.1 Constants

Several constants are provided in order to configure CLIPS environment or to instruct some functions to behave in particular ways.

Scope Save Constants

Constants to decide what to save in CLIPS dump files (see the `SaveInstances()`, `BSaveInstances()` and `SaveFacts()` functions).

Constant	Description
`LOCAL_SAVE`	save objects having templates defined in current `Module`
`VISIBLE_SAVE`	save all objects visible to current `Module`

Salience Evaluation Constants

Constants to tell the underlying engine when salience has to be evaluated (see the *EngineConfig.SalienceEvaluation* property).

Constant	Description
`WHEN_DEFINED`	evaluate salience on rule definition (the default)
`WHEN_ACTIVATED`	evaluate salience on rule activation (*dynamic salience*)
`EVERY_CYCLE`	evaluate salience on every execution cycle (*dynamic salience*)

Conflict Resolution Strategy Constants

Constants to specify the way the underlying engine should resolve conflicts among rules having the same salience (see the *EngineConfig.Strategy* property).

Constant	Description
DEPTH_STRATEGY	newly activated rule comes first
BREADTH_STRATEGY	newly activated rule comes last
LEX_STRATEGY	strategy based on *time tags* applied to facts
MEA_STRATEGY	strategy based on *time tags* applied to facts
COMPLEXITY_STRATEGY	newly activated rule comes before rules of lower *specificity*
SIMPLICITY_STRATEGY	newly activated rule comes before rules of higher *specificity*
RANDOM_STRATEGY	assign place in agenda randomly

In the table above, the term *specificity* refers to the number of comparisons in the LHS of a rule.

Class Definition Default Mode Constants

Constants to specify default mode used for classes definition. Please refer to *Clips Reference Guide Vol. I: Basic Programming Guide* for details about the usage of the two modes, as the meaning is quite complex and outside the scope of this manual (see the *EngineConfig.ClassDefaultsMode* property).

Constant	Description
CONVENIENCE_MODE	set the default class mode to convenience
CONSERVATION_MODE	set the default class mode to conservation

Message Handler Type Constants

Constants to define the execution time, the purpose and the behaviour of *message handlers*: see function BuildMessageHandler() and the following members of the Class class: AddMessageHandler() and FindMessageHandler(). The following table summarizes the analogous one found in *Clips Reference Guide Vol. I: Basic Programming Guide*.

Constant	Description
AROUND	only to set up an environment for the message
AFTER	to perform auxiliary work after the primary message
BEFORE	to perform auxiliary work before the primary message
PRIMARY	to perform most of the work for the message

Template Slot Default Type Constants

It's possible to inspect whether or not a Template slot has been defined to have a default value, and in case it is, if its default value is *static* (that is, constant) or *dynamically generated* (for instance, using a function like gensym). See the documentation of Template for more details.

Constant	Description
NO_DEFAULT	the slot has no default value
STATIC_DEFAULT	the default value is a constant
DYNAMIC_DEFAULT	the default value is dynamically generated

Notice that NO_DEFAULT evaluates to False, so it's legal to use the Template.Slot.HasDefault() function just to test the presence of a default value. Please also note that NO_DEFAULT is only returned when the default value for a slot is set to ?NONE as stated in the *Clips Reference Guide Vol. II: Advanced Programming Guide*.

2.1.2 Functions

AgendaChanged()
test whether or not Agenda has changed since last call.

Assert(*o*)
Assert a Fact (already created or from a string). This perhaps needs some explanation: CLIPS allows the creation of facts based on Templates, and in PyCLIPS this is done by instancing a Fact with a Template argument. The resulting Fact slots can then be modified and the object can be used to make an assertion, either by using the Fact Assert() function or this version od Assert().

BLoad(*filename*)
Load the constructs from a binary file named *filename*. Binary files are not human-readable and contain all the construct information.

BLoadInstances(*filename*)
Load Instances from binary file named *filename*. Binary files are not human-readable and contain all the construct information.

BSave(*filename*)
Save constructs to a binary file named *filename*.

BSaveInstances(*filename* [, *mode=LOCAL_SAVE*])
Save Instances to binary file named *filename*. The *mode* parameter can be one of LOCAL_SAVE (for all Instances whose Definstances are defined in current Module) or VISIBLE_SAVE (for all Instances visible to current Module).

BatchStar(*filename*)
Execute commands stored in text file named as specified in *filename*.

BrowseClasses(*name*)
Print the list of Classes that inherit from specified one.

Build(*construct*)
Build construct given in argument as a string. The string must enclose a full construct in the CLIPS language.

BuildClass(*name, text* [, *comment*])
Build a Class with specified name and body. *comment* is the optional comment to give to the object. This function is the only one that can be used to create Classes with multiple inheritance.

BuildDeffacts(*name, text* [, *comment*])
Build a Deffacts object with specified name and body. *comment* is the optional comment to give to the object.

BuildDefinstances(*name, text* [, *comment*])
Build a Definstances having specified name and body. *comment* is the optional comment to give to the object.

BuildFunction(*name, args, text* [, *comment*])
Build a Function with specified name, arguments and body. *comment* is the optional comment to give to the object. *args* can be either a blank-separated string containing argument names, or a sequence of strings corresponding to argument names. Such argument names should be coherent to the ones used in the function body (that is, *text*). The argument list, if expressed as a string, should *not* be surrounded by brackets. *None* can also be used as the argument list if the function has no arguments.

BuildGeneric(*name, text* [, *comment*])
Build a Generic with specified name and body. *comment* is the optional comment to give to the object.

BuildGlobal(*name,* [, *value*])
Build a Global variable with specified *name* and *value*. The *value* parameter can be of any of the types supported by CLIPS: it can be expressed as a Python value (with type defined in Python: the module will

try to pass to CLIPS a value of an according type), but for types that normally do not exist in Python (such as `Symbols`) an explicit conversion is necessary. If the *value* is omitted, then the module assigns *Nil* to the variable.

BuildInstance(*name, defclass* [, *overrides*])
Build an `Instance` of given `Class` overriding specified `slots`. If no `slot` is specified to be overridden, then the `Instance` will assume default values.

BuildMessageHandler(*name, class, args, text* [, *type, comment*])
Add a new *message handler* to the supplied class, with specified name, body (the *text* argument) and argument list: this can be specified either as a sequence of variable names or as a single string of whitespace separated variable names. Variable names (expressed as strings) can also be *wildcard parameters*, as specified in the *Clips Reference Guide Vol. I: Basic Programming Guide*. The *type* parameter should be one of *AROUND*, *AFTER*, *BEFORE*, *PRIMARY* defined at the module level: if omitted it will be considered as *PRIMARY*. The body must be enclosed in brackets, as it is in CLIPS syntax. The function returns the *index* of the *message handler* within the specified `Class`.

BuildModule(*name* [, *text, comment*])
Build a `Module` with specified name and body. *comment* is the optional comment to give to the object. The current `Module` is set to the new one.

BuildRule(*name, lhs, rhs* [, *comment*])
Build a `Rule` object with specified name and body. *comment* is the optional comment to give to the object. The *lhs* and *rhs* parameters correspond to the *left-hand side* and *right-hand side* of a `Rule`.

BuildTemplate(*name, text* [, *comment*])
Build a `Template` object with specified name and body. *comment* is the optional comment to give to the object.

Call(*func, args*)
Call a CLIPS internal `Function` with the given argument string. The *args* parameter, in its easiest form, can be a list of arguments separated by blank characters using CLIPS syntax. There are other forms that can be used, depending on how many arguments the called function requires: if it accepts a single argument, the caller can just specify the argument[1] possibly cast using one of the *wrapper classes* described below. When the function accepts multiple arguments it is possible to specify them as a sequence of values (either a list or a tuple) of basic[2] values. It is always preferable to convert these values using the *wrapper classes* in order to avoid ambiguity, especially in case of string arguments.

ClassList()
Return the list of `Class` names.

Clear()
Clear current `Environment`.

ClearPythonFunctions()
Unregister all user defined Python functions from `PyCLIPS`.

ClearFocusStack()
Clear focus stack.

CurrentEnvironment()
Return an `Environment` object representing current CLIPS *environment*. This is useful for switching between *environment*s in a `PyCLIPS` session. Please note that almost all *environment* operations are disallowed on the returned object until another `Environment` is selected as current: all operations on current `Environment` should be performed using the *top level* module functions.

[1] It must be of a type compatible with CLIPS. If a string is supplied, however, it will be considered as a list of arguments separated by whitespace: in order to explicitly pass a string it has either to be converted or to be specified surrounded by double quotes.

[2] Complex values, as *multifield*, are not supported: CLIPS does not allow external calls with non-constant arguments and there is no possibility to build a *multifield* in place without an explicit function call.

CurrentModule()
Return current module as a `Module` object.

DeffactsList()
Return a list of `Deffacts` names in current `Module`.

DefinstancesList()
Retrieve list of all `Definstances` names.

Eval(*expr*)
Evaluate expression passed as argument. Expressions that only have *side effects* (e.g. `printout` expressions) return `None`.

ExternalTracebackEnabled()
Return `True` if functions called from within CLIPS using the engine function `python-call` will print a standard traceback to *sys.stderr* on exceptions, `False` otherwise. This function is retained for backwards compatibility only: please use the *ExternalTraceback* flag of the *DebugConfig* object to enable or disable this feature instead.

FactList()
Return list of `Facts` in current `Module`.

FactListChanged()
Test whether `Fact` list is changed since last call.

FindClass(*name*)
Find a `Class` by name.

FindDeffacts(*name*)
Find a `Deffacts` by name.

FindDefinstances(*name*)
Find a `Definstances` by name.

FindFunction(*name*)
Find a `Function` by name.

FindGeneric(*name*)
Find a `Generic` function by name.

FindGlobal(*name*)
Find a `Global` variable by name.

FindInstance(*name*)
Find an `Instance` in all `Modules` (including imported).

FindInstanceLocal(*name*)
Find an `Instance` in non imported `Modules`.

FindModule(*name*)
Find a `Module` in list by name.

FindRule(*name*)
Find a `Rule` by name.

FindTemplate(*name*)
Find a `Template` by name.

FocusStack()
Get list of `Module` names in focus stack.

FunctionList()
Return the list of `Function` names.

GenericList()
Return the list of `Generic` names.

GlobalList()
Return the list of `Global` variable names.

GlobalsChanged()
Test whether or not `Global` variables have changed since last call.

InitialActivation()
Return first `Activation` object in current CLIPS `Environment`.

InitialClass()
Return first `Class` in current CLIPS `Environment`.

InitialDeffacts()
Return first `Deffacts` in current CLIPS `Environment`.

InitialDefinstances()
Return first `Definstances` in current CLIPS `Environment`.

InitialFact()
Return first `Fact` in current CLIPS `Environment`.

InitialFunction()
Return first `Function` in current CLIPS `Environment`.

InitialGeneric()
Return first `Generic` in current CLIPS `Environment`.

InitialGlobal()
Return first `Global` variable in current CLIPS `Environment`.

InitialInstance()
Return first `Instance` in current CLIPS `Environment`.

InitialModule()
Return first `Module` in current CLIPS `Environment`.

InitialRule()
Return first `Rule` in current CLIPS `Environment`.

InitialTemplate()
Return first `Template` in current CLIPS `Environment`.

InstancesChanged()
Test if `Instances` have changed since last call.

Load(*filename*)
Load constructs from the specified file named *filename*.

LoadFacts(*filename*)
Load `Facts` from the specified file named *filename*.

LoadFactsFromString(*s*)
Load `Facts` from the specified string.

LoadInstances(*filename*)
Load `Instances` from file named *filename*.

LoadInstancesFromString(*s*)
Load `Instances` from the specified string.

MessageHandlerList()
Return list of `MessageHandler` constructs.

MethodList()
Return the list of all methods.

ModuleList()
Return the list of Module names.

PopFocus()
Pop focus.

PrintAgenda()
Print Rules in Agenda to standard output.

PrintBreakpoints()
Print a list of all breakpoints to standard output.

PrintClasses()
Print a list of all Classes to standard output.

PrintDeffacts()
Print a list of all Deffacts to standard output.

PrintDefinstances()
Print a list of all Definstances to standard output.

PrintFacts()
Print Facts to standard output.

PrintFocusStack()
Print focus stack to standard output.

PrintFunctions()
Print a list of all Functions to standard output.

PrintGenerics()
Print list of Generic functions to standard output.

PrintGlobals()
print a list of Global variables to standard output

PrintInstances([*class*])
Print a list of Instances to standard output. If the *class* argument is omitted, all Instances in the subsystem will be shown. The *class* parameter can be a Class object or a string containing a Class name.

PrintMessageHandlers()
Print a list of all MessageHandlers.

PrintModules()
Print a list of Modules to standard output.

PrintRules()
Print a list of Rules to standard output.

PrintSubclassInstances([*class*])
Print subclass Instances to standard output for the Class specified. If the *class* argument is omitted, all instances in the subsystem will be shown. The *class* parameter can be a string containing a Class name or a Class object.

PrintTemplates()
Print Template names to standard output.

RefreshAgenda()
Refresh Agenda Rules for current Module.

RegisterPythonFunction(*callable* [, *name*])

Register the function `callable` for use within CLIPS via the engine function `python-call`. If the parameter *name* of type `str` is not given, then the *__name__* attribute of the first argument will be used. *name* is the name that will be used in CLIPS to refer to the function. See appendix for a more detailed explanation.

ReorderAgenda()
Reorder `Agenda` `Rules` for current `Module`.

Reset()
Reset current `Environment`.

RestoreInstancesFromString(*s*)
Restore `Instances` from the specified string.

RuleList()
Return a list of `Rule` names in current `Module`.

Run([*limit*])
Execute `Rules` up to *limit* (which is an `int` if given). If *limit* is omitted, then no limitation is assumed and the program runs countinuously until all rules are executed. The function returns the number of rules that have been fired[3].

Save(*filename*)
Save constructs to the file specified by *filename*. The constructs are saved in text (human readable) form.

SaveFacts(*filename* [, *mode=LOCAL_SAVE*])
Save current `Facts` to file specified by *filename*. The *mode* parameter can be one of `LOCAL_SAVE` (for all `Facts` whose `Templates` are defined in current `Module`) or `VISIBLE_SAVE` (for all `Facts` visible to current `Module`).

SaveInstances(*filename* [, *mode=LOCAL_SAVE*])
Save `Instances` to file specified by *filename*. The *mode* parameter can be one of `LOCAL_SAVE` (for all `Instances` whose `Definstances` are defined in current `Module`) or `VISIBLE_SAVE` (for all `Instances` visible to current `Module`).

SendCommand(*cmd* [, *verbose=False*])
Send a command to the underlying CLIPS engine, as if it was typed at the console in an interactive CLIPS session. This command could actually be useful when embedding a CLIPS shell in a Python program. Please note that other input than commands, in such a case, should be entered using the *StdinStream* input stream. If *verbose* is set to `True` the possible[4] command output is sent to the appropriate output stream.

SetExternalTraceback([*enabled=True*])
Allow or disallow functions called from within the CLIPS engine using `python-call` to print a standard traceback to *sys.stderr* in case an exception occurs. Please note that this does not mean that a real exception arises, as there is no possibility to catch Python exceptions in CLIPS code. In such case all failing Python functions will return the symbol `FALSE` to CLIPS[5]. This behaviour is initially set to `False`, as it is useful only for debugging purposes. This function is retained for backwards compatibility only: please use the *ExternalTraceback* flag of the *DebugConfig* object to enable or disable this feature instead.

ShowGlobals()
Print list of `Global` variables and their values to standard output (the `PrintGlobals()` functions only prints out `Global` variable names).

TemplateList()
Return a list of `Template` names.

UnregisterPythonFunction(*name*)

[3]This means, for instance, that continuously using this function and checking whether or not the result is less than the specified limit can give more control over the running CLIPS subprogram, eventually giving the ability to actively check for the end of the program.

[4]Except for the CLIPS printing functions, as for instance `printout`, that issue an output even when the flag is not set.

[5]This is not always an error condition because a function can intentionally return boolean values to CLIPS. However the CLIPS engine will report an error message which can be read from the error stream.

Remove the function referred as *name* within the CLIPS engine from the set of functions that can be called via `python-call` calls.

Among other exception types, arising in cases that can also occur in Python, the `PyCLIPS` module can raise exceptions specific to CLIPS identified by the following:

exception `ClipsError`
Exception raised when an operation fails in the CLIPS subsystem: normally it occurs when CLIPS finds an error, when an iteration is over or when an invalid value is passed to CLIPS. This exception is accompanied by explanatory text preceded by an alphanumeric code that can be used to programmatically identify the error.

exception `ClipsMemoryError`
Severe memory error raised when the CLIPS subsystem is unable to allocate the needed memory. In normal circumstances, when an error of this type occurs, the CLIPS system has become inconsistent and the only way to recover is exiting. This exception is raised only in order to allow a developer to notify the user of the impossibility to continue.

See Also:

Clips Reference Guide Vol. II: Advanced Programming Guide
contains detailed information about CLIPS API and internal structures

Clips Reference Guide Vol. I: Basic Programming Guide
the main reference for the CLIPS language

CHAPTER

THREE

Classes and Objects

As previously stated in the introduction, `PyCLIPS` provides classes and objects to access CLIPS "*entities*". It could be preferable to refer to counterparts "living" in the CLIPS subsystem as *entities* than as *objects*, because it is common practice in OOP to give the name of "*objects*" to class instances: since CLIPS has its own object oriented structure (in fact there are *classes* in CLIPS, and therefore also *instances* of these), calling these structures simply *objects* may generate confusion.

Entities in CLIPS are generated by *constructs*, one for each type of entity. In Python, the common way to create something is to instance an `object` for a certain `class`. So it seemed straightforward to make a class for each of these entities, and to substitute the constructs used in CLIPS to create entities with *factory functions*. These functions are defined at module level, and have names of the type `Build`*`Entity`*`()` where *`Entity`* is the type of entity that has to be created. The only exception for this are `Fact` objects, which are created in several ways from `Template`s or `Assert`ions.

There is another way to create entities in the CLIPS subsystem, that is directly using the `Build()` function with a full CLIPS construct as string argument. However, this function does not return anything to the caller, so the created entity has to be sought after creation to obtain a reference.

The `Build`*`Entity`*`()` functions and the `Assert()` function return objects of proper types (whose detailed list is given below) which shadow the corresponding entities in the CLIPS space.

Note: Many objects in `PyCLIPS` have common features[1], such as a *factory function* as stated above, or methods returning their name or their so-called *pretty-print form*: in the following detailed documentation only the first occurrence of a feature will be described thoroughly.

3.1 Wrapper Classes

There are some simple classes that deserve a special mention in the `PyCLIPS` module, used to represent in Python namespace the basic types in CLIPS. These *wrappers* are used to differentiate values that CLIPS returns from other values that live in the Python space. However these classes are equivalent to their Python counterparts, and there is no need to pass objects converted to these classes to the module functions. Here is a list containing the class names and their equivalents in Python:

Class	Type	Python Equivalent
`Integer`	*ClipsIntegerType*	`int`
`Float`	*ClipsFloatType*	`float`
`String`	*ClipsStringType*	`str`
`Symbol`	*ClipsSymbolType*	`str`
`InstanceName`	*ClipsInstanceNameType*	`str`
`Multifield`	*ClipsMultifieldType*	`list`

[1] The current `PyCLIPS` implementation still does not make use of inheritance, although it's likely that a future release will do.

A special object named *Nil* is defined, and is equivalent to `Symbol('nil')` in comparisons and slot assignments. It is provided to make code more readable in such situations. It has to be noticed that also *Nil* evaluates to *False* in boolean tests: this also yields for the explicit `Symbol('nil')` and `Symbol('FALSE')` definitions[2].

3.2 Template

`Templates` are used to build `Fact` objects, that is, they provide a systematic way to construct `Facts` sharing a common pattern, and the only way to define `Facts` that have named `Slots` (the equivalent of record *fields* or *structure members* in other programming languages).

class `Template`

This represents a copy of a `deftemplate` construct in the CLIPS subsystem, and not a true `deftemplate` entity. More than one `Template` object in Python can refer to the same `deftemplate` entity in the CLIPS subsystem.

`BuildFact``()`

Build a `Fact` object using this `Template` without asserting it. The created `Fact` *Slots* can be modified and the `Fact` asserted using its `Assert` method.

`Deletable`

Read-only property to verify if this `Template` can be deleted from the CLIPS subsystem.

`InitialFact``()`

Return initial `Fact` in list created using this `Template`.

`Module`

Read-only property to retrieve the CLIPS name of the `Module` where the `Template` is defined.

`Name`

Read-only property returning the name in CLIPS of this `Template`. The name identifies this entity in the CLIPS subsystem, and has nothing to do with the name given to the corresponding object in Python.

`Next``()`

Return next[3] `Template` in the list of all `Templates`. *None* is returned at the end of the list.

`NextFact`(*fact*)

Return next `Fact` in list created using this `Template`, using the supplied *fact* as offset.

`PPForm``()`

Return the *pretty-print form* of this `Template`. *Pretty-print forms* are often the code necessary to build a construct in CLIPS, formatted in a way that makes it quite readable. The result of the `PPForm()` method can be used as the argument for the `Build()` top level function to rebuild the construct once the `Environment` has been cleared[4].

`Remove``()`

Remove the entity corresponding to this `Template` from the CLIPS subsystem. This does not remove the corresponding Python object that has instead to be deleted via the `del` statement or garbage collected.

`Slots`

`Deftemplate` `slots` information. This is itself an object, having many methods, and deserves a special explaination.

`AllowedValues`(*name*)

Return a list of allowed values for `slot` specified by *name*.

[2] In this `Symbol` is different from `str` as only the empty string evaluates to false in Python. However, it seemed closer to the assumption that symbols in CLIPS are not to be considered as "*literals*" (they are more similar to implicitly defined variables) to implement such behaviour, that can be reverted with an explicit conversion to `str`.

[3] CLIPS stores its objects (or entities) in ordered lists, so it makes sense to "iterate" over these lists. However this implementation of `PyCLIPS` does not implement *iterators* (as known in Python) on these classes: a way to do this is currently under examination.

[4] Actually *pretty-print forms* use fixed size buffers to build the representing string: when such a form is too complex, the default buffer size of 8192 bytes can be insufficient. In this case the *PPBufferSize* property of the *Memory* object can be used to allow the creation of properly sized buffers.

Cardinality(*name*)
Return *cardinality* for `slot` specified by *name*.

DefaultValue(*name*)
Return *cardinality* for `slot` specified by *name*.

Exists(*name*)
Return `True` if `slot` specified by *name* exists, `False` otherwise.

HasDefault(*name*)
Return one of the following values: `NO_DEFAULT` if the default value is set to `?NONE`, `STATIC_DEFAULT` when the default value is static and `DYNAMIC_DEFAULT` when it is dynamically generated (eg. `gensym`).

IsMultifileld(*name*)
Return `True` if `slot` specified by *name* is a `Multifield` value, `False` otherwise.

IsSinglefield(*name*)
Return `True` if `slot` specified by *name* is a single field value, `False` otherwise.

Names()
Return the list of `slot` names.

Range(*name*)
Return *numerical range information* for `slot` specified by *name*.

Types(*name*)
Return names of *primitive types* for `slot` specified by *name*.

Watch
Read-only property to verify if this `Template` is being watched.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for `Template`s is `BuildTemplate()`, which has been discussed above.

3.3 Fact

`Fact`s are one of the main entities in CLIPS, since it is whether a `Fact` exists or not of that drives the subsystem in the decision to fire or not certain `Rule`s. `Fact`s, as seen above, can be created in several ways, that is either by directly asserting sentences in string form, or by building them first from `Template`s and then asserting them.

class Fact
This represents a copy of a fact definition in the CLIPS subsystem, and not a true fact entity. More than one Fact objects in Python can refer to the same fact entity in the CLIPS subsystem. Many CLIPS functions return a `Fact` object, but most `Fact` objects obtained from CLIPS are *read-only*[5]. Read-only `Fact`s cannot be reasserted or modified in *Slots*, and are provided for "informational" purposes only.

The argument can be a string with the same format of the `Assert()` function seen in the previous chapter: in this case the fact is created and asserted. Otherwise the argument can be a `Template` object, and in this case the resulting `Fact` can be modified and then asserted via the `Assert()` member function.

Assert()
Assert this `Fact`. Only `Fact`s that have been constructed from `Template`s can be `Assert`ed using this method: read-only `Fact`s can only be inspected with the other methods/properties.

AssignSlotDefaults()
Assign default values to *Slots* of this `Fact`.

CleanPPForm()
Return only the second part of this `Fact`s *pretty-print form* – which can be used to build the `Fact` itself as described above.

[5] Actually, the main rule is that if a `Fact` has been `Assert`ed then it is read-only. Note that all *shadow representations* of CLIPS asserted `fact` entities are read-only.

Exists
Is `True` if this `Fact` has been asserted (and never retracted), `False` otherwise.

ImpliedSlots
The list of all *implied Slots* for this `Fact`.

Index
Read-only property returning the index in CLIPS of this `Fact`. As for other entities the *Name* is a unique identifier, as is the *Index* for `Facts`.

Next`()`
Return next `Fact` in the list of all `Facts`. This list is not based on `Modules`, but global to the CLIPS subsystem.

PPForm`()`
Return the *pretty-print form* of this `Fact`. In this case, only the second part of the returned string (the one between parentheses) can be used to build the `Fact` via the `Assert()` function[6].

PPrint`(`[*ignoredefaults*]`)`
Print the `Fact` to the standard output. When *ignoredefaults* is set to `True` (the default), slots containing the default values are omitted.

Relation
Return only the name of the *relation* that identifies this `Fact`[7] as a `Symbol`.

Retract`()`
Retract this `Fact`: in other words, remove the corresponding entity from the CLIPS subsystem. As in `Remove()` seen above, this does not delete the corresponding Python object. *None* is returned at the end of the list.

Slots
Dictionary of `Fact` *Slots*. This member *behaves* like a `dict`, but is not related to such objects. In fact, the values of `slots` are accessible using a `dict`-like syntax (square brackets), but not all the members of `dict` are implemented.

Please note that `Facts` have slightly different methods than classes representing other entities in CLIPS: an instance of `Fact` is created using the module-level `Assert()` function, and removed using the `Retract()` member function: this syntax, closer to the original CLIPS form, was seen as the preferred method instead of using a name such as `BuildFact()` for creation and a `Remove()` member because of the particular nature of `Fact` related to other types of entity.

Here is an example of usage of `Fact` and `Template` objects:

[6] This is also not always true: as said before, there is no way to `Assert Facts` that have named slots using a string if there is not a `deftemplate` for this kind of `Fact`. However, once a `Template` with the specified slots has been created, this becomes possible.

[7] The authors of CLIPS call a *relation* the first field of the `Fact` itself, although it is not needed to actually represent a real relationship.

```
>>> import clips
>>> clips.Reset()
>>> t0 = clips.BuildTemplate("person", """
    (slot name (type STRING))
    (slot age (type INTEGER))
""", "template for a person")
>>> print t0.PPForm()
(deftemplate MAIN::person "template for a person"
   (slot name (type STRING))
   (slot age (type INTEGER)))

>>> f1 = clips.Fact(t0)
>>> f1_slotkeys = f1.Slots.keys()
>>> print f1_slotkeys
<Multifield [<Symbol 'name'>, <Symbol 'age'>]>
>>> f1.Slots['name'] = "Grace"
>>> f1.Slots['age'] = 24
>>> print f1.PPForm()
f-0     (person (name "Grace") (age 24))
>>> clips.PrintFacts()
f-0     (initial-fact)
>>> f1.Assert()
<Fact 'f-1': fact object at 0x00E0CB10>
>>> print f1.Exists()
True
>>> clips.PrintFacts()
f-0     (initial-fact)
f-1     (person (name "Grace") (age 24))
For a total of 2 facts.
>>> f1.Retract()
>>> print f1.Exists()
False
>>> clips.PrintFacts()
f-0     (initial-fact)
For a total of 1 fact.
```

Please note that slot names are implemented as `Symbol`s, and the list of *Slots* is returned as a `Multifield`. Also note that the `Fact` *f1*, that has been constructed from a `Template` (and not yet `Assert`ed) object and then modified using the *Slots* property, can be `Assert`ed while other `Fact`s built from construct strings cannot.

3.4 Deffacts

A `Deffacts` is used to modify the "initial structure" of a CLIPS environment, by allowing some `Fact`s to be `Assert`ed by default each time the `Reset()` function is called.

class `Deffacts`

This represents a copy of a `deffacts` construct in the CLIPS subsystem, and not a true `deffacts` entity. More than one `Deffacts` object in Python can refer to the same `deffacts` entity in the CLIPS subsystem.

`Deletable`

Read-only property to verify if this `Deffacts` can be deleted.

`Module`

Read-only property to retrieve the CLIPS name of the `Module` where the `Deffacts` is defined.

`Name`

Read-only property returning the name in CLIPS of this `Deffacts`.

Next()
Return next Deffacts in the list of all Deffacts. *None* is returned at the end of the list.

PPForm()
Return the *pretty-print form* of this Deffacts.

Remove()
Remove the entity corresponding to this Deffacts from the CLIPS subsystem.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for Deffacts is BuildDeffacts(), which has been discussed above.

3.5 Rule

The construct defines rules to be activated and then *fired* whenever particular conditions are met. This construct is in fact the counterpart of the defrule construct in CLIPS. Normally conditions that fire Rules are Facts Asserted during a session.

class Rule
This represents a copy of a defrule construct in the CLIPS subsystem, and not a true defrule entity. More than one Rule object in Python can refer to the same defrule entity in the CLIPS subsystem.

Breakpoint
Set or remove a breakpoint from this Rule.

Deletable
Read-only property to verify if this Rule can be deleted.

Module
Read-only property to retrieve the CLIPS name of the Module where the Rule is defined.

Name
Read-only property returning the name in CLIPS of this Rule.

Next()
Return next Rule in the list of all Rules. *None* is returned at the end of the list.

PPForm()
Return the *pretty-print form* of this Rule.

PrintMatches()
Print partial matches of this Rule to standard output.

Refresh()
Refresh this Rule.

Remove()
Remove the entity corresponding to this Rule from the CLIPS subsystem.

WatchActivations
Set or reset debugging of *activations* for this Rule.

WatchFirings
Set or reset debugging of *firings* for this Rule.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for Rules is BuildRule(), which has been discussed above.

An example – derived from the ones present in the standard CLIPS documentation – may be useful here:

```
>>> clips.Reset()
>>> r1 = clips.BuildRule("duck-rule", "(duck)",
                         "(assert (quack))", "The Duck rule")
>>> print r1.PPForm()
(defrule MAIN::duck-rule "The Duck rule"
   (duck)
   =>
   (assert (quack)))

>>> clips.PrintFacts()
f-0     (initial-fact)
For a total of 1 fact.
>>> clips.PrintRules()
MAIN:
duck-rule
>>> f1 = clips.Assert("(duck)")
>>> clips.PrintAgenda()
MAIN:
   0      duck-rule: f-1
For a total of 1 activation.
>>> clips.PrintFacts()
f-0     (initial-fact)
f-1     (duck)
For a total of 2 facts.
>>> clips.Run()
>>> clips.PrintFacts()
f-0     (initial-fact)
f-1     (duck)
f-2     (quack)
For a total of 3 facts.
```

3.6 Activation

`Rule Activation`s are only returned by the CLIPS subsystem, and cannot be created – thus there is no *factory function* for these objects. CLIPS provides `Activation` objects to keep the program flow under control.

class `Activation`

This represents a copy of an `activation` object in the CLIPS subsystem, and not a true `activation` entity. More than one `Activation` object in Python can refer to the same `activation` entity in the CLIPS subsystem.

`Name`

Retrieve `Activation` name.

`Next``()`

Return next `Activation` in the list of all `Activation`s. *None* is returned at the end of the list.

`PPForm``()`

Return the *pretty-print form* of `Activation`.

`Salience`

Retrieve `Activation` *salience*[8].

`Remove``()`

Remove this `Activation` from CLIPS.

[8] *Salience* is a value that represents the *priority* of a `rule` in CLIPS.

The name of this entity in CLIPS is also returned by the string coercion function.

3.7 Global

`Global` objects represent *global variables* in CLIPS, which are normally built using the `defglobal` construct. To define a new `Global` variable the `BuildGlobal()` function must be used, which returns a new object.

class `Global`

A `Global` object represents a copy of a `defglobal` construct in the CLIPS subsystem, and not a true `defglobal` entity. More than one `Global` object in Python can refer to the same `defglobal` entity in the CLIPS subsystem.

`Deletable`
Verify if this `Global` can be deleted.

`Module`
Read-only property to retrieve the CLIPS name of the `Module` where the `Global` is defined.

`Name`
Retrieve `Global` name. The returned value is a `Symbol` containing the name of the global variable in the CLIPS subsystem.

`Next``()`
Return next `Global` in the list of all global variables. *None* is returned at the end of the list.

`PPForm``()`
Return the *pretty-print* form of `Global`.

`Remove``()`
Remove this `Global` from CLIPS subsystem.

`Value`
Set or retrieve `Global` value. The returned value can be of many types, depending on the type of value contained in the corresponding CLIPS global variable.

`ValueForm``()`
Return a "*printed*" form of `Global` value. The *printed* form is the one that would be used in CLIPS to represent the variable itself.

`Watch`
Set or retrieve `Global` debug status.

Some examples follow to show the use of `Global` objects:

```
>>> g_x = clips.BuildGlobal("x", 15)
```

This is equivalent to the CLIPS declaration:

```
CLIPS> (defglobal ?*x* = 15)
```

Some of the `Global` methods are illustrated here:

```
>>> g_x
<Global 'x': defglobal object at 0x00E09960>
>>> print g_x
x
>>> g_x.Value
<Integer 15>
>>> print g_x.Value
15
>>> print g_x.ValueForm()
?*x* = 15
```

The name of this entity in CLIPS is also returned by the string coercion function.

3.8 Function

Objects of this type represent newly defined *functions* (usually via the CLIPS `deffunction` construct) in the CLIPS subsystem. In fact the `BuildFunction()` function described above, which returns a `Function` object, corresponds to the `deffunction` construct.

class `Function`

This represents a copy of a `deffunction` construct in the CLIPS subsystem, and not a true `deffunction` entity. More than one `Function` object in Python can refer to the same `deffunction` entity in the CLIPS subsystem.

`Call`(*[*args]*)

Call this `Function` with the given arguments, if any. If one only argument is passed and it is a `str`, then it is considered a "list of whitespace separated arguments[9]" and follows the CLIPS syntax: in order to pass a single string it has to be explicitly cast to the `String` *wrapper class*. Conversion to *wrapper classes* is however recommended for all passed arguments.

`Deletable`

Verify if this `Function` can be deleted.

`Module`

Read-only property to retrieve the CLIPS name of the `Module` where the `Function` is defined.

`Name`

Retrieve `Function` name.

`Next`()

Return next `Function` in the list of all CLIPS functions. *None* is returned at the end of the list.

`PPForm`()

Return the *pretty-print form* of `Function`.

`Remove`()

Remove this `Function`.

`Watch`

Set or retrieve `Function` debug status.

The name of this entity in CLIPS is also returned by the string coercion function.

Note: Objects of this class are *callable* themselves using the syntax `object([arg1 [, ... [argN]])`, where the arguments follow the same rules as in the `Call` method.

[9] See the syntax for the toplevel function with the same name.

3.9 Generic

`Generic`s (in CLIPS called *generic functions*) are similar to `Function`s, but they add *generic programming* capabilities to the CLIPS system. Python programmers will find them similar to Python functions, since *overloading* is possible within the corresponding construct.

Each different implementation (for different argument sets) of a *generic function* is called a `Method`, and the `Generic` class provides several ways to inspect the various `Method`s. `Method`s are identified by an *index*.

class `Generic`

This represents a copy of a `defgeneric` construct in the CLIPS subsystem, and not a true `defgeneric` entity. More than one `Generic` objects in Python can refer to the same `defgeneric` entity in the CLIPS subsystem.

`AddMethod`(*restrictions, actions*[, *midx*[, *comment*]])

Add a `Method` to this `Generic`. The structure of this function resembles the one of the `Build<entity>()` functions: in fact this method of `Generic` actually implements the `defmethod` construct which is present in CLIPS. For proper documentation of this construct, see the CLIPS reference: the *restrictions* parameter (which represents the `Method` *parameter restrictions*) must be expressed *without* parentheses; the *actions* parameter must be expressed as in the `defmethod` construct, that is with all the necessary parentheses pairs. *midx* is the `Method` index when it has to be forced (optionally). The example below should be explanatory. *restrictions* can also be expressed as a sequence of tuples, in each of which the first element is the argument name (with its proper prefix) as a string and the following ones are the actual restrictions, either in string form or as CLIPS primitive types – which can be specified using `PyCLIPS` *wrapper classes* types, see above.

`Call`([**args*])

Call this `Generic` with the given arguments, if any. If one only argument is passed and it is a `str`, then it is considered a "list of whitespace separated arguments" and follows the CLIPS syntax: in order to pass a single string it has to be explicitly cast to the `String` *wrapper class*. Conversion to *wrapper classes* is however recommended for all passed arguments.

`Deletable`

Verify if this `Generic` can be deleted.

`InitialMethod`()

Return the index of first `Method` in this `Generic`.

`MethodDeletable`(*midx*)

Test whether or not specified `Method` can be deleted from this `Generic`.

`MethodDescription`(*midx*)

Return the synopsis of specified `Method` *restrictions*.

`MethodList`()

Return the list of `Method` indices for this `Generic`.

`MethodPPForm`(*midx*)

Return the *pretty-print form* of specified `Method`.

`MethodRestrictions`(*midx*)

Return the *restrictions* of specified `Method` in this `Generic` object: the *midx* parameter must be an `Integer` or `int` indicating the `Method` index.

`MethodWatched`(*midx*)

Test whether or not specified `Method` is being watched.

`Module`

Read-only property to retrieve the CLIPS name of the `Module` where the `Generic` is defined.

`Name`

Retrieve `Generic` name.

NextMethod(*midx*)
Return the index of next `Method` in this `Generic` given the start index as an `Integer` or `int`.

PPForm()
Return the *pretty-print form* of `Generic`.

PrintMethods()
Print out `Method` list for this `Generic`.

Remove()
Remove this `Generic`.

RemoveMethod(*midx*)
Remove specified `Method` from this `Generic`.

UnwatchMethod(*midx*)
Deactivate watch on specified `Method`.

Watch
Set or retrieve `Generic` debug status.

WatchMethod(*midx*)
Activate watch on specified `Method`.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for `Generic`s is `BuildGeneric()`, which has been discussed above.

Note: Objects of this class are *callable* themselves using the syntax `object([arg1 [, ... [argN]])`, where the arguments follow the same rules as in the `Call` method.

An example for this class follows.

```
>>> import clips
>>> addf = clips.BuildGeneric("my-addf", "my generic add function")
>>> addf.AddMethod("(?a STRING)(?b STRING)", "(str-cat ?a ?b)")
>>> addf.AddMethod("(?a INTEGER)(?b INTEGER)", "(+ ?a ?b)")
>>> addf.PrintMethods()
my-addf #1  (STRING) (STRING)
my-addf #2  (INTEGER) (INTEGER)
For a total of 2 methods.
>>> print addf.MethodPPForm(1)
(defmethod MAIN::my-addf
   ((?a STRING)
    (?b STRING))
   (str-cat ?a ?b))

>>> print addf.PPForm()
(defgeneric MAIN::my-addf "my generic add function")

>>> print clips.Eval('(my-addf 5 13)')
18
>>> print clips.Eval('(my-addf "hello,"(my-addf " " "world!"))')
hello, world!
>>> print clips.Eval('(my-addf "hello" 13)')
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in ?
    print clips.Eval('(my-addf "hello" 13)')
  File ".../_clips_wrap.py", line 2472, in Eval
    return _cl2py(_c.eval(expr))
ClipsError: C10: unable to evaluate expression
>>> s = clips.ErrorStream.Read()
>>> print s
[GENRCEXE1] No applicable methods for my-addf.
```

Please note how the *error stream* (*ErrorStream*) can be used to retrieve a more explanatory text for the error. The *error stream* can be very useful during interactive debugging PyCLIPS sessions to fix errors.

3.10 Class

Class objects are class definition constructs, the most important feature of the *COOL*[10] sublanguage of CLIPS. As in other OOP environments, classes represent in CLIPS new data types (often resulting from aggregation of simpler data types) which have particular ways of being handled. Normally, as in Python, these particular ways are called *methods*[11], while in CLIPS they are called *message handlers*, since to apply a method to a CLIPS object (in fact, the Instance of a Class in PyCLIPS) a *message* has to be sent to that object.

class Class

This represents a copy of a defclass construct in the CLIPS subsystem, and not a true defclass entity. More than one Class object in Python can refer to the same defclass entity in the CLIPS subsystem.

Abstract

Verify if this Class is *abstract* or not.

AddMessageHandler(*name, args, text* [, *type, comment*])

Add a new *message handler* to this class, with specified name, body (the *text* argument) and argument list: this can be specified either as a sequence of variable names or as a single string of whitespace separated variable names. Variable names (expressed as strings) can also be *wildcard parameters*, as specified in the *Clips Reference Guide Vol. I: Basic Programming Guide*. The *type* parameter should be one of *AROUND*, *AFTER*, *BEFORE*, *PRIMARY* defined at the module level: if omitted it will be considered as *PRIMARY*. The body must be enclosed in brackets, as it is in CLIPS syntax. The function returns the *index* of the *message handler* within this Class.

AllMessageHandlerList()

Return the list of MessageHandler constructs of this Class including the ones that have been inherited from the superclass.

BuildInstance(*name,* [*overrides*])

Build an Instance of this Class with the supplied name and overriding specified slots. If no slot is specified to be overridden, then the Instance will assume default values.

BuildSubclass(*name, text* [, *comment*])

Build a subclass of this Class with specified name and body. *comment* is the optional comment to give to the object.

Deletable

Verify if this Class can be deleted.

Description()

Return a summary of Class description.

InitialInstance()

Return initial Instance of this Class. It raises an error if the Class has no subclass Instances.

InitialSubclassInstance()

Return initial instance of this Class including its subclasses. It raises an error if the Class has no subclass Instances.

IsSubclassOf(*o*)

Test whether this Class is a subclass of specified Class.

IsSuperclassOf(*o*)

Test whether this Class is a superclass of specified Class.

[10] Acronym for CLIPS Object-Oriented Language.

[11] Note that the term Method has been used for function overloading in the definition of Generic functions.

MessageHandlerDeletable(*index*)
Return true if specified `MessageHandler` can be deleted.

MessageHandlerIndex(*name* [, *htype*])
Find the specified `MessageHandler`, given its *name* and type (as the parameter *htype*). If type is omitted, it is considered to be *PRIMARY*.

MessageHandlerName(*index*)
Return the name of specified `MessageHandler`.

MessageHandlerList()
Return the list of `MessageHandler` constructs for this `Class`.

MessageHandlerPPForm(*index*)
Return the *pretty-print form* of `MessageHandler`.

MessageHandlerType(*index*)
Return the type of the `MessageHandler` specified by the provided *index*.

MessageHandlerWatched(*index*)
Return watch status of specified `MessageHandler`.

Module
Read-only property to retrieve the CLIPS name of the `Module` where the `Class` is defined.

Name
Retrieve `Class` name.

Next()
Return next `Class` in the list of all CLIPS `classes`. *None* is returned at the end of the list.

NextInstance(*instance*)
Return next `Instance` of this `Class`. Returns *None* if there are no `Instance`s left.

NextMessageHandlerIndex(*index*)
Return index of next `MessageHandler` with respect to the specified one.

NextSubclassInstance(*instance*)
Return next instance of this `Class`, including subclasses. Returns *None* if there are no `Instance`s left.

PPForm()
Return the *pretty-print form* of `Class`.

PreviewSend(*msgname*)
Print list of `MessageHandler`s suitable for specified message.

PrintAllMessageHandlers()
Print the list of all `MessageHandler`s for this `Class` including the ones that have been inherited from the superclass.

PrintMessageHandlers()
Print the list of `MessageHandler`s for this `Class`.

RawInstance(*name*)
Create an empty `Instance` of this `Class` with specified name.

Reactive
Verify if this `Class` is *reactive* or not.

Remove()
Remove this `Class`.

RemoveMessageHandler(*index*)
Remove `MessageHandler` specified by the provided *index*.

Slots
`Class slots` information. This is itself an object, having many methods, and deserves a special explaination.

AllowedClasses(*name*)
Return a list of allowed class names for `slot` specified by *name*.

AllowedValues(*name*)
Return a list of allowed values for `slot` specified by *name*.

Cardinality(*name*)
Return *cardinality* for `slot` specified by *name*.

DefaultValue(*name*)
Return the default value for `slot` specified by *name*.

Exists(*name*)
Return `True` if `slot` specified by *name* exists, `False` otherwise.

ExistsDefined(*name*)
Return `True` if `slot` specified by *name* is defined in this `Class`, `False` otherwise.

Facets(*name*)
Return *facet names* for `slot` specified by *name*.

HasDirectAccess(*name*)
Return `True` if `slot` specified by *name* is directly accessible, `False` otherwise.

IsInitable(*name*)
Return `True` if `slot` specified by *name* is *initializable*, `False` otherwise.

IsPublic(*name*)
Return `True` if `slot` specified by *name* is *public*, `False` otherwise.

IsWritable(*name*)
Return `True` if `slot` specified by *name* is *writable*, `False` otherwise.

Names()
Return the list of `slot` names.

NamesDefined()
Return the list of `slot` names explicitly defined in this `Class`.

Range(*name*)
Return *numerical range information* for `slot` specified by *name*.

Sources(*name*)
Return *source class names* for `slot` specified by *name*.

Types(*name*)
Return names of *primitive types* for `slot` specified by *name*.

Subclasses()
Return the names of subclasses of this `Class`.

Superclasses()
Return the names of superclasses of this `Class`.

UnwatchMessageHandler(*index*)
Turn off debug for specified `MessageHandler`.

WatchInstances
Set or retrieve debug status for this `Class Instance`s.

WatchMessageHandler(*index*)
Turn on debug for specified `MessageHandler`.

WatchSlots
Set or retrieve `Slot` debug status.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for `Class`es is `BuildClass()`, which has been discussed above.

3.11 Instance

`Instance` objects represent *class instances* (that is, *objects* in the OOP paradigm) that live in the CLIPS subsystem. Messages can be sent to those objects and values can be set and retrieved for the *slots* defined in the related *class*, where the meaning of `slot` has been described in the section above.

class `Instance`

This represents a copy of an `instance` object in the CLIPS subsystem, and not a true `instance` entity. More than one `Instance` object in Python can refer to the same `instance` entity in the CLIPS subsystem.

`Class`

Retrieve the `Class` of this `Instance`: this property actually refers to a `Class` object, so all of its methods are available.

`DirectRemove``()`

Directly remove this `Instance`, without sending a message.

`GetSlot`(*slotname*)

Retrieve the value of `Slot` specified as argument. The synonym `SlotValue` is retained for readability and compatibility. Please notice that these functions are provided in order to be more coherent with the behaviour of CLIPS API, as CLIPS C interface users know that a function like `GetSlot` usually bypasses message passing, thus accessing `slots` directly. The possibilities offered by `GetSlot` are also accessible using the *Slots* property described below.

`IsValid``()`

Determine if this `Instance` is still valid.

`Name`

Retrieve the `Instance` name.

`Next``()`

Return next `Instance` in the list of all CLIPS `instances`. It returns *None* if there are no `Instances` left.

`PPForm``()`

Return the *pretty-print form* of `Instance`.

`PutSlot`(*slotname, value*)

Set the value of specified `slot`. The *value* parameter should contain a value of the correct type, if necessary cast to one of the *wrapper classes* described above if the type could be ambiguous. The synonym `SetSlotValue` is provided for readability and compatibility. What has been said about `GetSlot` also yields for the hereby described function, as the possibilities offered by `PutSlot` are also accessible using the *Slots* property described below.

`Remove``()`

Remove this `Instance` (passing a message).

`Send`(*msg* [, *args*])

Send the provided *message* with the given arguments to `Instance`. The *args* parameter (that is, *message arguments*), should be a string containing a list of arguments separated by whitespace, a tuple containing the desired arguments or a value of a basic type. Also in the second case the tuple elements have to be of basic types. The function returns a value depending on the passed message.

`Slots`

Dictionary of `Instance` `slots`. This member *behaves* like a `dict`, but is not related to such objects. In fact, the values of `slots` are accessible using a `dict`-like syntax (square brackets), but not all the members of `dict` are implemented. The functionality of `PutSlot` and `GetSlot` is superseded by this property.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for `Instances` is `BuildInstance()`, which has been discussed above.

Here is an example of usage of `Instance` and `Class` objects:

```
>>> import clips
>>> clips.Build("""(defclass TEST1
    (is-a USER)
    (slot ts1 (type INSTANCE-NAME))
    (multislot ts2))""")
>>> c = clips.FindClass("TEST1")
>>> print c.PPForm()
(defclass MAIN::TEST1
   (is-a USER)
   (slot ts1
      (type INSTANCE-NAME))
   (multislot ts2))

>>> clips.Reset()
>>> i = clips.BuildInstance("test1", c)
>>> i.Slots['ts2'] = clips.Multifield(['hi', 'there'])
>>> i.Slots['ts1'] = i.Name
>>> print i.PPForm()
[test1] of TEST1 (ts1 [test1]) (ts2 "hi" "there")
```

3.12 Definstances

As there are `deffacts` for `fact` objects, `instances` are supported in CLIPS by the `definstances` construct: it allows certain default `Instances` to be created each time a `Reset()` is issued. In `PyCLIPS` this construct is provided via the `Definstances` class.

class `Definstances`

This represents a copy of the `definstances` construct in the CLIPS subsystem, and not a true `definstances` entity. More than one `Definstances` object in Python can refer to the same `definstances` entity in the CLIPS subsystem.

`Deletable`

Verify if this `Definstances` can be deleted.

`Module`

Read-only property to retrieve the CLIPS name of the `Module` where the `Definstances` is defined.

`Name`

Retrieve `Definstances` name.

`Next``()`

Return next `Definstances` in the list of all CLIPS `definstances`. *None* is returned at the end of the list.

`PPForm``()`

Return the *pretty-print form* of this `Definstances` object.

`Remove``()`

Delete this `Definstances` object from CLIPS subsystem.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for `Definstances` is `BuildDefinstances()`, which has been discussed above.

3.13 Module

Modules are a way, in CLIPS, to organize constructs, facts and objects. There is a big difference between *modules* and *environments*[12]: one should think of a Module as a *group* of definitions and objects, which can interoperate with entities that are defined in other Modules. The Module class provides methods, similar to the ones defined at top level, to directly create entities as part of the Module itself, as well as methods to examine Module contents. Also, Module objects have methods that instruct the related CLIPS module to become *current*, so that certain operations can be performed without specifying the module to which they have to be applied.

class Module

This represents a copy of a defmodule construct in the CLIPS subsystem, and not a true defmodule entity. More than one Module object in Python can refer to the same defmodule entity in the CLIPS subsystem.

Name

Return the name of this Module.

Next()

Return next Module in the list of all CLIPS modules. *None* is returned at the end of the list.

PPForm()

Return the *pretty-print form* of this Module.

SetCurrent()

Make the module that this object refers the current Module.

SetFocus()

Set focus to this Module.

For the following methods:

```
TemplateList(), FactList(), DeffactsList(), ClassList(), DefinstancesList(),
GenericList(), FunctionList(), GlobalList(), BuildTemplate(),
BuildDeffacts(), BuildClass(), BuildDefinstances(), BuildGeneric(),
BuildFunction(), BuildGlobal(), BuildRule(), BuildInstance(),
PrintTemplates(), PrintDeffacts(), PrintRules(), PrintClasses(),
PrintInstances(), PrintSubclassInstances(), PrintDefinstances(),
PrintGenerics(), PrintFunctions(), PrintGlobals(), ShowGlobals(),
PrintAgenda(), PrintBreakpoints(), ReorderAgenda(), RefreshAgenda()
```

please refer to the corresponding function defined at module level, keeping in mind that these methods perform the same task but within the Module where they are executed.

The name of this entity in CLIPS is also returned by the string coercion function. The *factory function* for Modules is BuildModule(), which has been discussed above.

3.14 Environment

This class represents an *environment*, and implements almost all the module level functions and classes. The only objects appearing at PyCLIPS level and *not* at Environment level are the CLIPS I/O subsystem *streams*, which are shared with the rest of the CLIPS engine.

Environment objects are not a feature of CLIPS sessions (as stated above), thus there is no way to identify them in CLIPS using a *symbol*. So Environment objects do not have a *Name* property. Instead, CLIPS provides a way to identify an *environment* through an integer called *index*.

[12] Besides the discussion above, also notice that in a "pure" CLIPS session there is no concept of *environment* at all: the use of environment is reserved to those who embed CLIPS in another program, such as PyCLIPS users.

class Environment

Please refer to top level functions, variables and classes for information on contents of `Environment` objects. The extra methods and properties follow below.

Index

Retrieve the *index* identifying this `Environment` internally in CLIPS.

SetCurrent()

Make the *environment* that this object refers the current `Environment`.

Further explanations about `Environment` objects can be found in the appendices.

3.15 Status and Configuration Objects

As seen in the introduction, there are a couple of objects that can be accessed to configure the underlying CLIPS engine and to retrieve its status. These are the *EngineConfig* and *DebugConfig* objects. The reason why configuration and status functions have been grouped in these objects is only cosmetic: in fact there is no counterpart of *EngineConfig* and *DebugConfig* in CLIPS. It was seen as convenient to group configuration and debug functions in two main objects and to make them accessible mainly as *properties* in Python, instead of populating the module namespace with too many *get/set* functions.

There is also an object, called *Memory*, which gives information about memory utilization and allow the user to attempt to free memory used by the CLIPS engine and no longer needed.

A description of what the above objects (which can not be instanced by the user of `PyCLIPS`[13]) actually expose follows.

3.15.1 Engine Configuration

The *EngineConfig* object allows the configuration of some features of the underlying CLIPS engine. Here are the properies provided by *EngineConfig*:

AutoFloatDividend

Reflects the behaviour of CLIPS `get/set-auto-float-dividend`. When `True` the dividend is always considered to be a floating point number within divisions.

ClassDefaultsMode

Reflects the behaviour of CLIPS `get/set-class-defaults-mode`. Possible values of this flag are `CONVENIENCE_MODE` and `CONSERVATION_MODE`. See *Clips Reference Guide Vol. II: Advanced Programming Guide* for details.

DynamicConstraintChecking

Reflects the behaviour of CLIPS `get/set-dynamic-constraint-checking`. When `True`, *function calls* and *constructs* are checked against constraint violations.

FactDuplication

Reflects the behaviour of CLIPS `get/set-fact-duplication`. When `True`, `facts` can be reasserted when they have already been asserted[14].

IncrementalReset

Reflects the behaviour of CLIPS `get/set-incremental-reset`. When `True` newly defined `rules` are updated according to current `facts`, otherwise new `rules` will only be updated by `facts` defined after their construction.

[13] Besides removal of class definitions, a *singleton*-styled implementation mechanism prevents the user from creating further instances of the objects.

[14] This does not change the behaviour of the `Fact` class, which prohibits reassertion anyway. However, `facts` that would be asserted through firing of rules and would generate duplications will not raise an error when this behaviour is set.

ResetGlobals
Reflects the behaviour of CLIPS `get/set-reset-globals`. When `True` the `Global` variables are reset to their initial value after a call to `Reset()`.

SalienceEvaluation
Reflects the behaviour of CLIPS `get/set-salience-evaluation`. Can be one of `WHEN_DEFINED`, `WHEN_ACTIVATED`, `EVERY_CYCLE`. See the previous chapter and *Clips Reference Guide Vol. II: Advanced Programming Guide* for more information.

SequenceOperatorRecognition
Reflects the behaviour of CLIPS `get/set-sequence-operator-recognition`. When `False`, `Multifield` values in function calls are treated as a single argument.

StaticConstraintChecking
Reflects the behaviour of CLIPS `get/set-static-constraint-checking`. When `True`, *slot values* are checked against constraint violations.

Strategy
Reflects `get/set-strategy` behaviour. Can be any of the following values: `RANDOM_STRATEGY`, `LEX_STRATEGY`, `MEA_STRATEGY`, `COMPLEXITY_STRATEGY`, `SIMPLICITY_STRATEGY`, `BREADTH_STRATEGY` or `DEPTH_STRATEGY`. See the previous chapter and *Clips Reference Guide Vol. II: Advanced Programming Guide* for more information.

3.15.2 Debug Settings

The *DebugConfig* object provides access to the debugging and trace features of CLIPS. During a CLIPS interactive session debug and trace messages are printed on the system console (which maps the `wtrace` I/O *router*). Users of the trace systems will have to poll the *TraceStream* to read the generated messages.

In CLIPS, the process of enabling trace features on some class of entities is called *to watch* such a class; this naming convention is reflected in `PyCLIPS`. Note that specific objects can be *watched*: many classes have their own *Watch* property to enable or disable debugging on a particular object.

Also, CLIPS provides a facility to log all debug information to physical files: this is called *to dribble* on a file. *Dribbling* is possible from *DebugConfig* via the appropriate methods.

The names of methods and properties provided by this object are quite similar to the corresponding commands in CLIPS, so more information about debugging features can be found in *Clips Reference Guide Vol. I: Basic Programming Guide*.

ActivationsWatched
Flag to enable or disable trace of `Rule` activations and deactivations.

CompilationsWatched
Flag to enable or disable trace of construct definition progress.

DribbleActive()
Tell whether or not *dribble* is active.

DribbleOff()
Turn off *dribble* and close the *dribble* file.

DribbleOn(*fn*)
Enable *dribble* on the file identified by provided filename *fn*.

ExternalTraceback
Flag to enable or disable printing traceback messages to Python *sys.stderr* if an error occurs when the CLIPS engine calls a Python function. Please note that the error is not propagated to the Python interpreter. See the appendices for a more detailed explaination.

FactsWatched
Flag to enable or disable trace of `Fact` assertions and retractions.

FunctionsWatched
Flag to enable or disable trace of start and finish of `Functions`.

GenericFunctionsWatched
Flag to enable or disable trace of start and finish of `Generic` functions.

GlobalsWatched
Flag to enable or disable trace of assignments to `Global` variables.

MethodsWatched
Flag to enable or disable trace of start and finish of `Methods` within `Generic` functions.

MessageHandlersWatched
Flag to enable or disable trace of start and finish of `MessageHandlers`.

MessagesWatched
Flag to enable or disable trace of start and finish of *messages*.

RulesWatched
Flag to enable or disable trace of `Rule` firings.

SlotsWatched
Flag to enable or disable trace of changes to `Instance Slots`.

StatisticsWatched
Flag to enable or disable reports about timings, number of `facts` and `instances`, and other information after `Run()` has been performed.

UnwatchAll`()`
Turn off *watch* for all items above.

WatchAll`()`
Watch all items above.

Note: Other CLIPS I/O streams besides *TraceStream* can be involved in the trace process: please refer to the CLIPS guides for details.

3.15.3 Memory Operations

This object provides access to the memory management utilities of the underlying CLIPS engine. As said above, it allows the reporting of memory usage and the attempt to free memory that is used not for computational purposes. Also, a property of this object affects the engine behaviour about whether or not to cache some information. Here is what the object exposes:

Conserve
When set to `True`, the engine does not cache *pretty-print forms* to memory, thus being more conservative.

EnvironmentErrorsEnabled
When set to `True`, the engine is enabled to directly write fatal environment errors to the console (`stderr`). This kind of messages is in most of the cases printed when the program exits, so it can be annoying. The behaviour is disabled by default.

Free`()`
Attempt to free as much memory as possible of the one used by the underlying CLIPS engine for previous computations.

PPBufferSize
Report the size (in bytes) of the buffers used by `PyCLIPS` to return *pretty-print forms* or similar values. By

default this is set to 8192, but the user can modify it using values greater than or equal to 256. Greater values than the default can be useful when such forms are used to reconstruct CLIPS entities and definitions are so complex that the default buffer size is insufficient.

Requests
Read-only property reporting the number of memory request made by the engine to the operating system since PyCLIPS has been initialized.

Used
Read-only property reporting the amount, in kilobytes, of memory used by the underlying CLIPS engine.

NumberOfEnvironments
Read-only property reporting the number of currently allocated Environments.

3.16 I/O Streams

In order to be more embeddable, CLIPS defines a clear way to redirect its messages to locations where they can be easily retrieved. CLIPS users can access these locations for reading or writing by specifying them as *logical names* (namely stdin, stdout, wtrace, werror, wwarning, wdialog, wdisplay, wprompt)[15]. PyCLIPS creates some special unique objects[16], called *I/O streams* throughout this document, to allow the user to read messages provided by the underlying engine. Most of these objects have only one method, called Read(), that consumes CLIPS output and returns it as a string: this string contains all output since a previous call or module initialization. The only exception is *StdinStream* whose single method is Write() and it accepts a string[17] as parameter. As CLIPS writes line-by-line, the string resulting from a call to Read() can contain newline characters, often indicating subsequent messages.

Here is a list of the *I/O streams* provided by PyCLIPS, along with a brief description of each.

Stream	Description
StdoutStream	where information is usually printed out (eg. via (printout t ...))
TraceStream	where trace information (see *watch*) goes
ErrorStream	where CLIPS error messages are written in readable form
WarningStream	where CLIPS warning messages are written in readable form
DialogStream	where CLIPS informational messages are written in readable form
DisplayStream	where CLIPS displays information (eg. the output of the (facts) command)
PromptStream	where the CLIPS prompt (normally CLIPS>) is sent
StdinStream	where information is usually read by CLIPS (eg. via (readline))

Some of the provided *I/O streams* are actually not so relevant for PyCLIPS programmers: for instance, it is of little use to read the contents of *PromptStream* and *DisplayStream*. In the latter case, in fact, there are other inspection functions that provide the same information in a more structured way than text. However they are available to provide a closer representation of the programming interface and allow CLIPS programmers to verify if the output of *CLIPS-oriented* calls (see the paragraph about Build() and Eval() in the appendices) really do what they are expected to.

[15] CLIPS also defines t as a *logical name*: as stated in *Clips Reference Guide Vol. II: Advanced Programming Guide* this indicates stdin in functions that read text and stdout in function that print out. In PyCLIPS, for all functions that print out to t the user must read from *StdoutStream*.

[16] PyCLIPS in fact defines one more I/O stream, called temporary, which is used internally to retrieve output from CLIPS that shouldn't go anywhere else. PyCLIPS users however are not supposed to interact with this object.

[17] The current implementation converts the argument to a string, so other types can be accepted.

3.17 Predefined `Class`es

`PyCLIPS` defines[18], some `Class` objects, that is the ones that are present in CLIPS itself by default. They are defined in order to provide a compact access to CLIPS "stock" classes: most of these objects are of little or no use generally (although they can be handy when testing for class specification or generalization), but at least one (*USER_CLASS*) can be used to make code more readable.

Namely, these `Class`es are:

Python Name	**CLIPS defclass**
FLOAT_CLASS	FLOAT
INTEGER_CLASS	INTEGER
SYMBOL_CLASS	SYMBOL
STRING_CLASS	STRING
MULTIFIELD_CLASS	MULTIFIELD
EXTERNAL_ADDRESS_CLASS	EXTERNAL-ADDRESS
FACT_ADDRESS_CLASS	FACT-ADDRESS
INSTANCE_ADDRESS_CLASS	INSTANCE-ADDRESS
INSTANCE_NAME_CLASS	INSTANCE-NAME
OBJECT_CLASS	OBJECT
PRIMITIVE_CLASS	PRIMITIVE
NUMBER_CLASS	NUMBER
LEXEME_CLASS	LEXEME
ADDRESS_CLASS	ADDRESS
INSTANCE_CLASS	INSTANCE
INITIAL_OBJECT_CLASS	INITIAL-OBJECT
USER_CLASS	USER

The following code, shows how to use the "traditional" `BuildClass()` factory function and how to directly subclass one of the predefined `Class` object. In the latter case, probably, the action of subclassing is expressed in a more clear way:

```
>>> import clips
>>> C = clips.BuildClass("C", "(is-a USER)(slot s)")
>>> print C.PPForm()
(defclass MAIN::C
   (is-a USER)
   (slot s))

>>> D = clips.USER_CLASS.BuildSubclass("D", "(slot s)")
>>> print D.PPForm()
(defclass MAIN::D
   (is-a USER)
   (slot s))
```

Although it actually does not save typing (the statement is slightly longer), the second form can be used to produce more readable Python code.

Note: These objects are actually useful *only* when the package is fully imported, that is using the `import clips` form: importing the symbols at global level (in the form `from clips import *`) does in fact create some namespace problems. Since in the latter case the names that represent stock classes are only references to the ones defined

[18] At the module level only: defining these objects at the *environment* level could cause aliasing current CLIPS enviroment. On the other hand, if these objects were implemented in a way that checked for aliasing, access to the actual entities would be surely slower only favouring compactness of user code.

at module level, PyCLIPS cannot change them when the actual classes are relocated in the CLIPS memory space, for instance when Clear is called.

APPENDIX

A

Usage Notes

A.1 Environments

As seen in the detailed documentation, `PyCLIPS` also provides access to the *environment* features of CLIPS, through the `Environment` class. `Environment` objects provide almost all functions normally available at the top level of `PyCLIPS`, that is importing `clips` into a Python script. `Environment` object methods having the same name of functions found at the top level of `PyCLIPS`, have the same effect of the corresponding function – but restricted to the logical environment represented by the object itself[1]. Normally in a true CLIPS session users would not use environments, so the concept of environment may, in many cases, not be useful.

There are also functions (namely: `CurrentEnvironment()` and `Environment.SetCurrent()`) that allow the user to switch environment and to use top level functions and classes in any of the created environments. This is useful in cases where environments are used, because due to the double nature of CLIPS API (see *Clips Reference Guide Vol. II: Advanced Programming Guide* about *companion functions* for standard API), objects defined in environments have slightly different types than corresponding top level objects – since these types are *environment-aware*. However environment aware classes define exactly the same methods as the top level counterparts, so their logical use is the same.

Note: Please note that the `CurrentEnvironment()` function returns a fully functional `Environment` object. However the system prevents[2] invasive access via *environment-aware* functions to current *environment*: user code should *always* use functions defined at module level to access current *environment*. The `Environment` methods become safe as soon as another `Environment` has become current – and in this case its methods and properties will raise an error.

`Environments` are a limited resource: this is because it is impossible in `PyCLIPS` to destroy a created `Environment`. In order to reuse `Environments` it may be useful to keep a reference to each of them for the whole `PyCLIPS` session. If there is an attempt to create more `Environments` than allowed, an exception is raised.

A.2 Multiple Ways

There is more than one way to use the `PyCLIPS` module, since it exposes almost all the API functions of CLIPS, seen in fact as a library, to the Python environment.

The module `PyCLIPS` provides some functions, that is `Build()` and `Eval()`, that let the user directly issue commands in the CLIPS subsystem from a Python program. In fact, the use of `Build()` allows definition of constructs in CLIPS[3], and `Eval()` lets the user evaluate values or call code directly in the subsystem. So for instance rules can

[1] In fact, the Python submodule that implements the `Environment` class is generated automatically: the process can be examined by looking at the code in 'setup.py' and 'clips/_clips_wrap.py'.

[2] Raising an appropriate exception.

[3] Note that the `Build()` function does not return any value or object, so you will have to call `Find`*`Construct`*`()` to find entities created using the `Build()` function.

be built in PyCLIPS using

```
>>> import clips
>>> clips.Build("""
(defrule duck-rule "the Duck Rule"
   (duck)
   =>
   (assert (quack)))
""")
>>> clips.PrintRules()
MAIN:
duck-rule
```

and evaluate a sum using

```
>>> n = clips.Eval("(+ 5 2)")
>>> print n
7
```

Also, the user is allowed to call functions that do not return a value using Eval()[4], as in the following example:

```
>>> clips.Eval('(printout t "Hello, World!" crlf)')
>>> print clips.StdoutStream.Read()
Hello, World!
```

There is another function, namely SendCommand(), that sends an entire CLIPS command (it has to be a full, correct command, otherwise PyCLIPS will issue an exception): as Build() it does not return any value or object[5]. However this can be particularly useful when the user needs to implement an interactive CLIPS shell within an application built on PyCLIPS. Unless the application is mostly CLIPS oriented (if for instance Python is used just as a "glue" script language) probably the use of this function has to be discouraged, in favour of the code readability that – at least for Python programmers – is provided by the Python oriented interface.

Using SendCommand() it becomes possible to write:

[4] There is a discussion about functions that only have *side effects* in CLIPS, such as printout, in *Clips User's Guide*, that is, the CLIPS tutorial.

[5] Some information about the command result can be retrieved reading the appropriate output streams.

```
>>> import clips
>>> clips.SendCommand("""
(defrule duck-rule "the Duck Rule"
   (duck)
   =>
   (assert (quack)))
""")
>>> clips.SendCommand("(assert (duck))")
>>> clips.Run()
>>> clips.PrintFacts()
f-0     (duck)
f-1     (quack)
For a total of 2 facts.
>>> clips.PrintRules()
MAIN:
duck-rule
```

The most important caveat about `SendCommand()` is that CLIPS accepts some kinds of input which normally have to be considered incorrect, and `PyCLIPS` does neither return an error value, nor raise an exception: for instance, it is possible to pass a symbol to CLIPS to the command line as in

```
CLIPS> thing
thing
```

and in this case CLIPS "evaluates" the symbol, printing it to the console as a result of the evaluation. `PyCLIPS` does not automatically capture evaluation output, and just accepts a symbol (or other commands that can be evaluated) as input without any production:

```
>>> import clips
>>> clips.SendCommand("thing")
```

but, turning on the verbosity flag:

```
>>> clips.SendCommand("thing", True)
>>> print clips.StdoutStream.Read()
thing
```

Of course, `PyCLIPS` complains if something incorrect is passed to the `SendCommand()` function and raises an exception as previously stated. However the exception is accompanied by a rather non-explanatory text. The *ErrorStream* object should be queried in this case in order to obtain some more information about the error:

```
>>> clips.SendCommand("(assert (duck)") # no right bracket

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in -toplevel-
    clips.SendCommand("(assert (duck)") # no right bracket
  File ".../_clips_wrap.py", line 2575, in SendCommand
    _c.sendCommand(s)
ClipsError: C09: unable to understand argument
>>> print clips.ErrorStream.Read()

[PRNTUTIL2] Syntax Error:  Check appropriate syntax for RHS patterns.
```

Obviously `SendCommand()` can lead to serious errors if not used with some kind of interaction.

The point of this paragraph is that, for entity definition (evaluation can only be performed using the `Eval()` or `Call()` functions), the `PyCLIPS` module provides a full set of specific `Build`*`Entity`*`()` functions which also return appropriate objects corresponding to specific entities. So, the task of building a *rule* in CLIPS (in fact, a `Rule` object in Python) could preferably be performed directly using the `BuldRule()` function, that is:

```
>>> clips.Clear()
>>> clips.Reset()
>>> r0 = clips.BuildRule("duck-rule", "(duck)", "(assert (quack))",
                         "the Duck Rule")
>>> print r0.PPForm()
(defrule MAIN::duck-rule "the Duck Rule"
   (duck)
   =>
   (assert (quack)))

>>> clips.PrintRules()
MAIN:
duck-rule
```

thus with the same effect as with the `Build()` function, but obtaining immediately a reference to the rule entity in CLIPS as a Python object. Similar examples could be provided for the `SendCommand()` function, using the appropriate constructs or commands that can be used to achieve the same goals.

This allows the user to choose between at least two programming styles in `PyCLIPS`: the former, more CLIPS oriented, relies heavily on the use of the `Build()`, `Eval()` and `SendCommand()` functions, and is probably more readable to CLIPS developers. The latter is somewhat closer to Python programming style, based on the creation of objects of a certain nature by calling specific Python functions. The advice is to avoid mixing the two styles unless necessary, since it can make the code quite difficult to understand.

A.3 Python Functions in CLIPS

In `PyCLIPS` it is possible to execute Python functions from within CLIPS embedded constructs and statements. This allows the extension of the underlying inference engine with imperative functionalities, as well as the possibility to retrieve information from the Python layer asynchronously with respect to Python execution. Of course this possibility enables some enhancements of the CLIPS environment, but – as a drawback – it also opens the way to errors and misunderstandings.

Usage of Python external functions is fairly simple: the user should register functions that will be called from within the CLIPS subsystem in `PyCLIPS` using the `RegisterPythonFunction()` toplevel function. If no alternate

name for the function is specified, then the Python name will be used[6]. If necessary, Python function names can be deregistered using `UnregisterPythonFunction()` and `ClearPythonFunctions()` utilities. Once a function is registered it can be called from within the CLIPS engine using the following syntax:

```
(python-call <funcname> [arg1 [arg2 [ ... [argN]]]])
```

and will return a value (this allows its use in assignments) to the CLIPS calling statement. In the call, `<funcname>` is a *symbol* (using a string will result in an error) and the number and type of arguments depends on the actual Python function. When arguments are of wrong type or number, the called function fails. Using the previously illustrated `py_square` example, we have:

```
>>> clips.RegisterPythonFunction(py_square)
>>> clips.SetExternalTraceback(True) # print traceback on error
>>> print clips.Eval("(python-call py_square 7)")
49
>>> print clips.Eval('(python-call py_square "a")')
Traceback (most recent call last):
  File ".../_clips_wrap.py", line 2702, in <lambda>
    f = lambda *args: _extcall_retval(func(*tuple(map(_cl2py, list(args)))))
  File "<pyshell#85>", line 2, in py_square
TypeError: can't multiply sequence to non-int
FALSE
>>> print clips.Eval("(python-call py_square 7 7)")
Traceback (most recent call last):
  File ".../_clips_wrap.py", line 2702, in <lambda>
    f = lambda *args: _extcall_retval(func(*tuple(map(_cl2py, list(args)))))
TypeError: py_square() takes exactly 1 argument (2 given)
FALSE
```

It is important to know, in order to avoid errors, that the Python interpreter that executes functions from within CLIPS is exactly the same that calls the `PyCLIPS` function used to invoke the engine: this means, for example, that a Python function called in CLIPS is subject to change the state of the Python interpreter itself. Moreover, due to the nature of CLIPS external function call interface, Python functions called in CLIPS will never raise exceptions[7] in the Python calling layer.

Here are some other issues and features about the nature of the external function call interface provided by `PyCLIPS`:

Functions should be CLIPS-aware: when CLIPS calls a Python external function with arguments, these are converted to values that Python can understand using the previously described *wrapper classes*. Thus, for instance, if the Python function is given an integer argument, then an argument of type `Integer` (not `int`) will be passed as actual parameter. This means that in most cases `PyCLIPS` has to be imported by modules that define external Python functions.

Actual parameters cannot be modified: there is no way to pass values back to CLIPS by modifying actual parameters. The possibility to use `Multifield` parameters as lists should not deceive the user, as every modification performed on `Multifield`s that Python receives as parameters will be lost after function completion. A way to handle this is to treat parameters as *immutable* values.

External functions should always return a value: functions always return a value in CLIPS, even in case of an error. This can be clearly seen in the following chunk of CLIPS code:

[6] An example of function registration has been provided in the introduction.

[7] Exceptions can arise *during* the Python function execution, and can be caught inside the function code. However, for debugging purposes, there is the possibility to force `PyCLIPS` print a standard traceback whenever an error occurs in a Python function called by CLIPS.

```
CLIPS> (div 1 0)
[PRNTUTIL7] Attempt to divide by zero in div function.
1
```

where, although an error message is printed to the console, the value `1` is returned by the system. In the same way, CLIPS expects Python external functions to return a value. `PyCLIPS` solves this issue by converting a return value of `None` (which is the real return value for Python functions that simply `return`) into the symbol `nil`, that has a meaning similar to the one of `None` for Python. Also, functions that raise uncaught exceptions will in fact return a value to the underlying CLIPS engine: in this case the returned value is the symbol `FALSE`, and an error message is routed to the error stream – thus, it can be retrieved using `ErrorStream.Read()`. The following example imitates the `div` CLIPS example above:

```
>>> import clips
>>> exceptor = lambda : 1 / 0
>>> clips.RegisterPythonFunction(exceptor, 'exceptor')
>>> clips.SetExternalTraceback(True) # print traceback on error
>>> clips.Eval('(python-call exceptor)')
Traceback (most recent call last):
  File ".../_clips_wrap.py", line 2702, in <lambda>
    f = lambda *args: _extcall_retval(func(*tuple(map(_cl2py, list(args)))))
  File "<pyshell#79>", line 1, in <lambda>
ZeroDivisionError: integer division or modulo by zero
<Symbol 'FALSE'>
```

Return values must be understood by CLIPS: only values that can be converted to CLIPS base types can be returned to the inference engine. This includes all values that can be converted to `PyCLIPS` *wrapper classes*. In fact it can be considered a good practice to cast return values to `PyCLIPS` *wrapper classes* when the main purpose of a function is to be called from within CLIPS.

Python functions act as generic functions: due to the nature of Python, functions are generally polymorphic:

```
>>> def addf(a, b):
        return a + b
>>> print addf("egg", "spam")
eggspam
>>> print addf(2, 4)
6
```

The intrinsic polymorphism of Python functions is kept within the CLIPS subsystem:

```
>>> import clips
>>> clips.RegisterPythonFunction(addf)
>>> print clips.Eval('(python-call addf "egg" "spam")')
eggspam
>>> print clips.Eval('(python-call addf 2 4)')
6
```

Thus Python functions act in a way that is similar to `generic`s.

APPENDIX

B

The `clips._clips` Submodule

It has been said throughout the whole document that there are two different "natures" of `PyCLIPS`, called the *high level module* and the *low level module*. The former has been described in detail here, and is supposed to be the main interface to CLIPS. However, since all communication between Python and the CLIPS subsystem is implemented in the *low level* part, some users might find it useful to access this interface instead of the *higher level* one. It is not the intention of this manual to provide documentation for the *low level* `clips._clips` interface, but only to give some indications to users who already have experience of the CLIPS C interface.

Submodule `clips._clips` provides low-level classes that have the same names as their counterparts in CLIPS, such as `fact`, `deftemplate` and so on. Also, `clips._clips` defines an `environment` class to refer to *environment* addresses.

Almost all functions described in *Clips Reference Guide Vol. II: Advanced Programming Guide* are ported to Python, except for missing features described below: the name is the same as in the reference guide, except for the first letter that is not capitalized[1]. CLIPS "top level" functions have been ported to `clips._clips` as well as *companion functions* that accept an `environment` instance as the first argument[2], whose name begins with `env_` followed by the same name as the corresponding top level function.

Low level functions are documented by themselves through *documentation strings*, that describe purpose, arguments and return values. For instance, let's show the documentation of a function:

```
>>> import clips
>>> cl = clips._clips
>>> print cl.listDeftemplates.__doc__
listDeftemplates(logicalname [, module])
list deftemplates to output identified by logicalname
arguments:
  logicalname (str) - the logical name of output
  module (module) - the module to inspect, all modules if omitted
```

Most low level function documentation strings, i.e. the ones given for functions that are not trivially identifiable with the CLIPS API counterparts[3], have this form and describe in detail arguments and return values. Users who want to benefit of these functions, can display *documentation strings* as a reference.

The underlying engine is the same, there is no separation of environment between the two interfaces. Operations performed at *lower level* are obviously reflected in the *higher level* layer, as in the following example:

[1] There can be some exceptions: the most important one is the assertion function, since Python already has an `assert` keyword. This function is called `assertFact` instead.

[2] When trying to show the *documentation string* of these functions, the first argument is not described because their code has been generated automatically.

[3] Some functions have a documentation string that actually refers to the CLIPS API itself, explicitly containing the words "*equivalent of C API*" and the C function name: *Clips Reference Guide Vol. II: Advanced Programming Guide* is especially useful in this case.

```
>>> clips.Clear()
>>> cl.facts("stdout")
>>> s = clips.StdoutStream.Read()
>>> print s
None
>>> print cl.assertString.__doc__
assertString(expr) -> fact
assert a fact into the system fact list
returns: a pointer to the asserted fact
arguments:
  expr (str) - string containing a list of primitive datatypes
>>> f = cl.assertString("(duck)")
>>> clips.PrintFacts()
f-0     (duck)
For a total of 1 fact.
```

so the two interfaces can be used interchangeably.

APPENDIX

C

Error Codes

It has been discussed above, that some of the `PyCLIPS` functions can raise a CLIPS specific exception, namely `ClipsError`. Some of the exceptions of this type (in fact, the ones raised by the underlying CLIPS engine and caught at the lower level), come with an error code in the accompanying text. A brief description of exceptions that arise at low level follows:

Code	Description
`P01`	An object could not be created, due to memory issues
`C01`	The engine could not create a system object
`C02`	The referred object could not be found in subsystem
`C03`	An attempt was made to modify an unmodifiable object
`C04`	A file could not be opened
`C05`	The current environment could not be retrieved
`C06`	The engine was unable to return the required value
`C07`	Parse error in the passed in CLIPS file
`C08`	Syntax or parse error in the passed in CLIPS expression
`C09`	Syntax or parse error in the passed in argument
`C10`	Expression could not be evaluated, maybe because of syntax errors
`C11`	An object could not be removed from the CLIPS subsystem
`C12`	A fact could not be asserted, maybe because of missing `deftemplate`
`C13`	Iteration beyond last element in a list
`C14`	A CLIPS function has been called unsuccessfully
`C15`	Attempt to modify an already asserted fact was made
`C16`	Cannot destroy environment while it is current
`C90`	Other errors: specific description given
`C97`	The feature is present when a higher version of CLIPS is used
`C98`	An attempt was made to use an unimplemented feature (see below)
`C99`	Generic CLIPS error, last operation could not be performed
`C00`	Generic CLIPS error, no specific cause could be reported
`S01`	Attempt to access a fact that is no longer valid or has been deleted
`S02`	Attempt to access an instance that is no longer valid or has been deleted
`S03`	Clear operation on an environment has failed
`S04`	Attempt to access an environment that has been deleted
`S05`	Attempt to operate on alias of current environment
`S06`	Attempt to create more environments than allowed
`S00`	Generic internal system error
`R01`	The logical buffer (I/O Stream) has been misused
`R02`	The logical buffer with given name could not be found
`R03`	Could not write to a read-only logical buffer

These codes can be extracted from the exception description and used to determine errors – for instance, in an `if ... elif ...` control statement. Some of these errors are caught by the `PyCLIPS` high-level layer and interpreted in

different ways (e.g. the `C13` error is used to generate lists or to return `None` after last element in a list).

There are also some CLIPS specific exceptions that can be thrown at the higher level: they are identified by a code beginning with the letter `M`. A list of these exceptions follows, along with their description:

Code	Description
`M01`	A constructor could not create an object
`M02`	An object could not be found in the CLIPS subsystem
`M03`	An attempt was made to `pickle` an object
`M99`	Wrong Python version, module could not initialize

Finally, there is a particular error that occurs in case of *fatal* memory allocation failures, which is identified by a particular exception, namely `ClipsMemoryError`. This excepion is raised with the following code and has the following meaning:

Code	Description
`X01`	The CLIPS subsystem could not allocate memory

In this case the calling program *must* exit, as the underlying engine has reached an unstable state. An exception different from the standard `ClipsError` has been provided in order to allow quick and effective countermeasures[1].

[1] Please note that in some cases, and depending on how the operating system treats memory allocation failures, the Python interpreter too could loose stability in case of memory shortage.

APPENDIX

D

Multithreading

The CLIPS engine is a separate subsystem, as stated many times before. In other words, it maintains a state independently from what happens in the Python interpreter. This also means that, since CLIPS was never conceived to be used as a multithreaded library, multiple threads should not try to access the engine concurrently. The Python interpreter, due to its nature, does not actually allow concurrent calls to the low-level module, so it is safe to create concurrent threads that interact with `PyCLIPS`.

However this has the side effect that, during a time consuming task (such as calling `Run()` on a complex set of rules and facts) the calling thread may block the other ones.

A partial solution to this, to allow multiple threads to switch more reactively, is to call `Run()` with the *limit* parameter, which specifies the number of rules to be fired at once. Of course this allows subsequent calls to the CLIPS engine to modify its state, and consequently enables execution paths that could be different from the ones that a full CLIPS `(run)` would normally cause. Obviously this only applies to the `Run()` function.

The consideration also implies that multithreaded Python applications should take care of examining the engine state before interacting with it, especially when splitting `Run()` (which normally modifies the state) in multiple calls.

APPENDIX

E

Missing Features

Most of the CLIPS API is implemented in `PyCLIPS`. The *lower level* interface (which directly maps CLIPS exposed functions as described in *Clips Reference Guide Vol. II: Advanced Programming Guide*) can be accessed using the `clips._clips` submodule. Almost all the functions defined here have a counterpart in the CLIPS API, and a combined use of documentation strings and *Clips Reference Guide Vol. II: Advanced Programming Guide* itself can allow you to directly manipulate the CLIPS subsystem as you would have done by embedding it in a C program. However, there are some functions that appear in `dir(clips._clips)` but, when called, issue an error:

```
>>> clips._clips.addClearFunction()
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in ?
    clips._clips.addClearFunction()
ClipsError: C98: unimplemented feature/function
```

and in fact, even their documentation string reports

```
>>> print clips._clips.addClearFunction.__doc__
unimplemented feature/function
```

even if the name of the function is defined and is a *callable*.

A list of such functions follows:

Function	Type
addClearFunction	execution hook
addPeriodicFunction	execution hook
addResetFunction	execution hook
removeClearFunction	execution hook
removePeriodicFunction	execution hook
removeResetFunction	execution hook
addRunFunction	execution hook
removeRunFunction	execution hook
decrementFactCount	reference count handler
incrementFactCount	reference count handler
decrementInstanceCount	reference count handler
incrementInstanceCount	reference count handler
setOutOfMemoryFunction	memory handler hook
addEnvironmentCleanupFunction	execution hook
allocateEnvironmentData	memory handler
deallocateEnvironmentData	memory handler
destroyEnvironment	environment destructor
getEnvironmentData	memory handler

The description of these functions is outside the scope of this guide, and can be found in *Clips Reference Guide Vol. II: Advanced Programming Guide* of CLIPS. It is not likely that these functions will be implemented even in future versions of PyCLIPS since Python programmers are usually not interested in dealing with low level memory handling (which is the primary use of the memory oriented functions), and tasks like reference count handling are performed directly by Python itself (for Python objects which shadow CLIPS entities) and by the low level PyCLIPS submodule. Also, the functions referred to above as *execution hooks* often have to deal with CLIPS internal structures at a very low level, so they would be of no use in a Python program.

Other API functions, which are used internally by PyCLIPS (for instance the InitializeEnvironment() C function), are not implemented in the module.

Note: Some of the features (either in current and possibly in further versions of PyCLIPS) may depend on the version of CLIPS that is used to compile the module. Using the most recent stable version of CLIPS is recommended in order to enable all PyCLIPS features. Features that are excluded from the module because of this reason will issue an exception, in which the exception text reports the following: "C97: higher engine version required". Moreover, the CLIPS engine version may affect the behaviour of some functions. Please consider reading the documentation related to the used CLIPS version when a function does not behave as expected.

APPENDIX

F

Installing PyCLIPS

F.1 Installation

To install PyCLIPS you should also download the full CLIPS source distribution. You will find a file called 'CLIPSsrc.zip' at the CLIPS download location: you should choose to download this instead of the UNIX compressed source, since the setup program itself performs the task of extracting the files to an appropriate directory with the correct line endings. The ZIP file format has been chosen in order to avoid using different extraction methods depending on the host operating system.

PyCLIPS uses distutils or setuptools for its installation. So in all supported systems the module can be easily set up once the whole source has been extracted to a directory and the CLIPS source code has been put in the same place, by means of the following command:

```
# python setup.py install
```

In fact recent versions of PyCLIPS will attempt to download the latest supported CLIPS source directly from the PyCLIPS web site if no CLIPS source package is found. Otherwise no attempt to connect to the Internet will be made. The 'README' file provides more up-to-date and detailed information on the setup process.

On UNIX, if you have a system-wide Python distribution, your privileges for installation should be the same as the Python owner.

The CLIPS library itself is compiled for a specific platform, since 'setup.py' modifies the 'setup.h' file in the CLIPS distribution.

PyCLIPS is known to build and pass the tests on *Linux (x86 and x86_64)*[1], *Win32* (many flavours of it have been tested), *Sun Solaris* with 32-bit gcc, *FreeBSD*, *Mac OS X* with *Fink* and, using a customized build process, has been ported to the *Sharp Zaurus (SA-1110)* platform.

F.2 Requirements

PyCLIPS requires Python 2.4 or above to function: it uses decorators to check and enforce types where needed, and in some places it also uses modern aspects of the Python API.

At least version 6.23 of CLIPS is required: it allows the definition and use of *environments*, and the function and macro definitions are more conformant to the ones described in *Clips Reference Guide Vol. II: Advanced Programming Guide*. Of course features present in CLIPS 6.24 are not available when using the previous CLIPS version, so if there is no particular reason to use it, please compile PyCLIPS with CLIPS 6.24, which also fixes some bugs.

[1] The x86_64 platforms requires some optional patches to be applied.

APPENDIX

G

License Information

The following is the license text, which you can obtain by issuing a

```
>>> import clips
>>> print clips.license
```

at the Python prompt once the `PyCLIPS` module has been installed.

```
==========================================================================
License Information (LGPL)
==========================================================================

(c) 2002-2008 Francesco Garosi/JKS
The author's copyright is expressed through the following notice, thus
giving effective rights to copy and use this software to anyone, as shown
in the license text.

NOTICE:
This software is released under the terms of the GNU Lesser General Public
license; a copy of the text has been released with this package (see file
_license.py, where the license text also appears), and can be found on the
GNU web site, at the following address:

           http://www.gnu.org/copyleft/lesser.html

Please refer to the license text for any license information. This notice
has to be considered part of the license, and should be kept on every copy,
integral or modified, of the source files. The removal of the reference to
the license will be considered an infringement of the license itself.

Portions of the code provided with this package may have been released
under different license terms: in this case it is expressed in the source
code piece itself. Parts of this source package (eg. the entire CLIPS
source distribution) are provided under possibly different license terms,
and different restrictions may apply. These source files are provided as
the original author(s) packaged them, thus all license information is
supplied.

If you received the package in binary form, please consult the original
CLIPS license, which you can find at the CLIPS web site:

                http://clipsrules.sourceforge.net
```

for the licensing terms regarding use of the CLIPS library.

==========

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION
0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.
b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
d) If a facility in the modified Library refers to a function or a table of

data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License.Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License.If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying

library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original

licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free

Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.