IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

D13.1 Integration and Configuration

Due date of deliverable: March 2007

Actual Submission date: March 30th, 2007

Start date of Project: 1st December 2004 **Duration: 28 months**

Lead Contractor of this WP: merlinux

Authors: Carl Friedrich Bolz, Alexandre Fayolle, Guido Wesdorp

Revision: Final

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

Revision History

Date	Name	Reason of Change
2007-01-22	Alexandre Fayolle	Describe Debian Packaging of PyPy
2007-02-01	Carl Friedrich Bolz	Add executive summary and abstract
2007-02-01	Carl Friedrich Bolz	Publishing of intermediate version on the web
2007-02-26	Guido Wesdorp	Added description of the build tool
2007-03-25	Carl Friedrich Bolz	Describe work package integration work
2007-03-29	Holger Krekel	Internal Review and Refinements of Exec Summary
2007-03-30	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

Since both PyPy's translation toolchain and its Python interpreter are very flexible, configuration can be complex and so we have developed extensive configuration and build facilities. The configuration tools allow easy addition of new configuration options as well as specification of dependencies, and make it easy to access option values and to expose options to the user. Build tools enable the user to build a custom version of the PyPy Python interpreter locally and on remote machines. We provide an overview of the integration efforts regarding PyPy's code base and its features. We also describe which packages have been made available to the Debian packaging system, allowing users to install and use the Python interpreter without the need to start the translation process. All integration and configuration work was completed with and resulted in the PyPy 1.0 Milestone release (D14.4).

Purpose, Scope and Related Documents

This document describes the design of PyPy's configuration framework, PyPy's build tools, the integration of the results of other work packages and the creation of Debian packages for PyPy. The document is targeted at potential PyPy developers who want to learn about the configuration framework and how the various parts of PyPy work together.

This document describes the integration of the work of other work packages into a consistent whole. It does not describe the achievements of these work packages in detail. For this, refer to the documents listed below. For an overview on the translation process see the report (D05.1).

Related Documents:

- D02.3 Automated Testing and Development support library (D02.3)
- D06.1 Core Object Optimization Results (D06.1)
- D07.1 Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects (D07.1)
- D08.2 JIT Compiler Architecture (D08.2)
- D09.1 Constraint Satisfaction and Inference Engine, Logic Programming in Python (D09.1)

- D10.1 Aspect-Oriented, Design-by-Contract Programming and RPython static checking (D10.1)
- D11.1 A Case Study On Using PyPy for Embedded Devices (D11.1)
- D12.1 High-Level Backends and Interpreter Feature Prototypes (D12.1)
- D14.4 Milestone 3 report (D14.4)
- D14.5 Development Process (D14.5)

Contents

1 Executive Summary

PyPy's test-driven development process (D14.5) facilitated early integration of features and development areas without requiring larger particular integration phases. However, in order to implement a pervasive configuration and option handling and to resolve some conflicting development branches, several specific refactorings and implementation efforts were necessary.

PyPy's flexibility on the level of its Python interpreter and of its translation framework gives a lot of freedom to users and developers. On the other hand it becomes necessary to deal with the abundance of options that this flexibility induces: the translation toolchain offers various translation aspects that can be weaved into the translated program and the Python interpreter can be adapted to have certain features and to be able to work better in specific situations. Some of these options need to be exposed to end-users of PyPy - people that use the PyPy Python interpreter as their main Python implementation - so that they can adapt PyPy to their wishes.

Options and features may have dependencies or may conflict with other options; not all combinations of values make sense. The configuration framework allows to specify options, dependencies and expose them in all areas of the code base. The ease of adding options makes the integration of most features in the Python interpreter and the translation toolchain easy; adding a new option and conditionally including the feature is enough. The configuration framework is fully documented; features and options from all development areas of PyPy are integrated and packaged with the PyPy 1.0 Milestone (D14.4) release.

Using this configuration system and integrated code base, we have developed build tools in order generate custom version of our Python interpreter. It is easy for PyPy developers to specify the steps necessary to translate an RPython program to another language as well as the dependencies of these steps. For users of PyPy's Python interpreter it is easy to specify which special features they want their translated Python interpreter to have. In addition to the default build tool, which does the requested compilation on the machine it is being run on, we have an experimental version that distributes translation tasks to a networked set of servers. This will eventually enable contributors to donate machine time to PyPy development by making their computer available for such builds.

For users who just want to use the features of PyPy's Python interpreter and do not have specific wishes or don't want to bother with translating themselves (a time consuming process), we provide pre-built versions of the Python Interpreter for the Debian GNU/Linux distribution. We also provided some experimental executables for the Windows Platform with the PyPy 1.0 release.

2 Configuration System

The PyPy configuration system was introduced to replace a number of ad-hoc methods for customizing the PyPy Python interpreter and translation process. It is meant to support the large number of features that can be chosen when running and translating the Python interpreter or, in general, any program written RPython.

2.1 Goals

The design goal for the configuration system was to support a large number of options in various subsystems of the PyPy Python interpreter and translation toolchain. Adding new options

should be possible with little effort to encourage the addition of options to make PyPy strongly customizable. On the other hand, it should be possible to keep track of the many options by structuring them into sensible categories and by making it possible to describe their relationships as data.

The configuration system was constructed in a completely PyPy-unspecific way, to make it usable with other projects. However, it needs to efficiently work with PyPy's translation tool chain, in that only program code needed to satisfy chosen options is produced and put into the final executable.

A further goal for the configuration system is to automatically construct as much documentation on the options as possible, so that adding new options also involves only a low overhead of exposing and documenting it properly to the end-users and developers of PyPy.

2.2 Implementation and Usage

The implementation of the configuration framework makes the fundamental distinction between *option descriptions* (or option *schemata*) and *configurations*. The option description describes what options there are, what values they can take, what the default value of each option is and how the various options relate to each other. A configuration is one particular choice of these options, conforming to a particular option description. Option descriptions are project-specific, whereas configurations are chosen by an end-user according to his needs and choices.

Here is an example snippet for a PyPy specific Option:

```
OptionDescription("std", "Standard Object Space Options", [
    BoolOption("withprebuiltint", "prebuild commonly used int objects",
               default=False,
               requires=[("objspace.std.withsmallint", False)]),
    ...
])
```

The option descriptions contain information about the available options, how they are organized into a tree of names. In addition they contain information of dependencies and conflicts between options, and an option can also merely *suggest* another option. The schemata also contain a command line option for use when automatically generating a command-line parser out of an option description.

Usually at the entry point a configuration object is generated out of a schema and then changed according to the command line arguments the user has given. Every time an option is set, all its dependencies are checked so that the configuration object can never be in an inconsistent state. For the translation toolchain, the entry point where the configuration object is created is the `translate.py` script described below, for the PyPy standard interpreter it is the `py.py` interactive interpreter entry point. From these entry points, the configuration object is passed on to be available everywhere. With the Python Interpreter it is attached to the object space, in the translation toolchain it is an attribute of the translation driver.

As soon as a configuration is instantiated and changed once, its option values can never change again to prevent the class programmer errors that lead to inconsistent option values across a program run.

A further important goal for the configuration system is that checks for specific values for configuration options are done at translation time. This means that the translated PyPy interpreter

does not contain any actual checks for configuration values and also does not contain the code which would be executed if a *different* value for the option in question was used. This makes it possible to use configuration options to achieve an effect similar to what is done with macros or preprocessor options in C.

From all of PyPy's option schemata a documentation page is generated automatically as part of PyPy's documentation. It contains all the information available in the option schemata together with more detailed documentation for each option. These documentation pages for PyPy's standard interpreter and PyPy's translation tool-chain can be found at:

http://codespeak.net/pypy/dist/pypy/doc/config/

There is also API-level documentation about the configuration framework in (CONFIG).

3 Support for Building PyPy

3.1 `translate.py`

`translate.py` is a general tool for translating RPython code to a different language. To specify what should be translated a so-called *target file* is used which determines the entry point of the to-be-compiled code as well as which special command line options the target accepts.

`translate.py` drives the whole translation process: it makes sure that all necessary steps for the compilation are performed in the right order. In addition, `translate.py` offers debugging facilities should a translation fail. Among those are a debugging console to interactively inspect the failed translation process to identify the problems, a flow graph viewer to inspect the state of the translation visually and a forking facility to debug the translation toolchain itself.

`translate.py` is integrated well with the configuration system: a configuration object is created when starting `translate.py` and used throughout the whole translation.

3.2 Build Tool

3.2.1 Introduction and Goals

The build tool consists of a set of scripts that provide experimental functionality to remotely build custom PyPy Python interpreter versions.

The main goal of the build tool is to allow people that don't have a powerful enough machine to build the Python interpreter, or don't have the required programs installed to do so, to build a customized Python interpreter. The target audiences are both developers and end-users: developers can use it to test building PyPy on platforms that they don't have direct access to, and end users can use it to avoid having to set up a suitable build environment.

Note that the tool is not PyPy-specific: it can be used for other projects that have complex build requirements, too, as long as they meet a certain set of criteria (of which the most important is that they use Subversion as their VCS).

For the PyPy project, a meta server is running on codespeak.net (D02.1). This means that at any time, if someone wants to participate or request a compilation, they can start the appropriate script and connect.

3.2.2 Configuration

To make the tool as accessible as possible, emphasis is put on ease-of-use. Requesting a build can be done by running a simple command-line script which, even though it supports the same large set of command-line options as translate.py does, uses a sensible set of defaults which rarely need to be adjusted. And providing a buildserver is even easier: there's also a single command-line script to connect, but it doesn't require any arguments.

Configuration is rarely required: there is a configuration file in the tool's directory, but it only contains per-project configuration and is set up properly for PyPy (having codespeak.net set as meta server).

3.2.3 Implementation

People can donate machine time by connecting to a special meta server and registering their machine as a 'build server', passing information about what builds can be executed on it. This machine can then be used by 'build clients' which request a compilation from the system.

If a compilation is requested, information from the request is compared to the configuration of the build servers, and if any of the build servers' configuration matches the request, the build is delegated to that build server. If no build server that meets the criteria is available, the request will be queued until a suitable machine is connected or done compiling.

Once a build is complete, the user is notified, either via email or (depending on the mode the compilation request script is started in) directly as a response to the request on the command line. If it was successful, a link to a ZIP file containing the build is returned. If the build failed, an error report is returned.

3.2.4 Components

The build tool consists of three basic components: the meta server, the build server and a compilation request script. The meta server component is responsible for managing build servers and compilation requests, the build server component holds system information and functionality to load the project's code from Subversion and actually build (compile) the project, and the compilation request script connects to the meta server to request a compilation.

All components communicate using part of the py-lib called 'execnet', a simple and flexible ad-hoc networking library. (PYEXEC)

A separate component is a web server that displays status information about the meta server, connected build servers and the builds that are done or in progress. This server is run separately from the meta server, connecting to it over the network in a similar way as the other components. A web server for the codespeak.net meta server is available at

http://codespeak.net/pypy/buildstatus/.

4 Combined PyPy Codebase

The following sections describe the work done to integrate the results of the work-packages WP6-WP12 into the PyPy standard interpreter and the translation toolchain as well as the interaction between the features of these work-packages. For most features it was enough to

integrate them with the configuration framework, for others it was necessary to hook them into other parts of PyPy, for example into the translation process.

The compatibility matrix in Figure 1. gives an overview about compatibility between the various features. Reasons for incompatibilities are given in the below work package specific subsections.

The integration of these features was completely done as part of the 1.0 release of PyPy on March 27th, 2007. See the release announcement (RELEASE) for more details.

4.1 WP 6: Core Object Optimizations

Nearly all the optimizations of this work package are code changes on the level of the bytecode interpreter and the standard object space. This makes them very easy to integrate by just adding new options in the configuration framework and putting checks for these options into the code. It also makes the optimizations completely independent of any translation aspect such as choice of backend, GC, stackless, JIT-compiler, etc.

The only optimization of WP6 that interferes with translation aspects was for *tagged integers* (which is also the only optimization that needs special support in the RTyper). Tagged integers cannot work with statically and strongly typed high-level target environments such as the CLR and the Java VM for fundamental reasons. Also they currently do not work together with our own garbage collectors, only with the Boehm collector. The latter, however, is not a fundamental restriction and could be fixed if necessary.

4.2 WP 7: Stackless Features, Garbage Collection and Optimizations

The work in WP7 changes the translation process in rather fundamental ways: the stackless transformation, the integration of the garbage collectors and the low level optimizations are all completely separate steps during the translation process and therefore needed deep integration into the translation process. All these were in addition integrated into the configuration system, so that the user can easily choose which GC to use, whether to use stackless or not and which optimizations to use.

An additional task of WP7 was the addition of interfaces for task-switching and explicit stack manipulation to PyPy's Python interpreter. This was done by the addition of a new `_stackless` module which contains various interfaces such as Greenlets and Coroutines.

The work in WP7 was originally targeted exclusively at low-level backends, which makes most of it not usable together with the Java or the CLR backend. Especially the garbage collectors are not usable for the fundamental reason that both the JVM and the CLR have their own garbage collector, making PyPy's garbage collectors unnecessary in these environments. The stackless transformation currently only works with the C backend, but this is more of a temporary restriction. We did not work so far on porting it to object-oriented backends, though this should not be overly hard, and it does not work with the LLVM backend for minor technical reasons. For the low level optimizations written as part of WP7 it depends: some of them were ported to work with object-oriented backends (inlining, malloc removal), some cannot work for fundamental reasons (stack allocation).

4.3 WP 8: Just-in-Time Compiler Generation

Generating a Just-In-Time compiler out of an interpreter is an advanced and complicated transformation which made it necessary to integrate it deeply into the translation process by

PyPy Compatibility Matrix

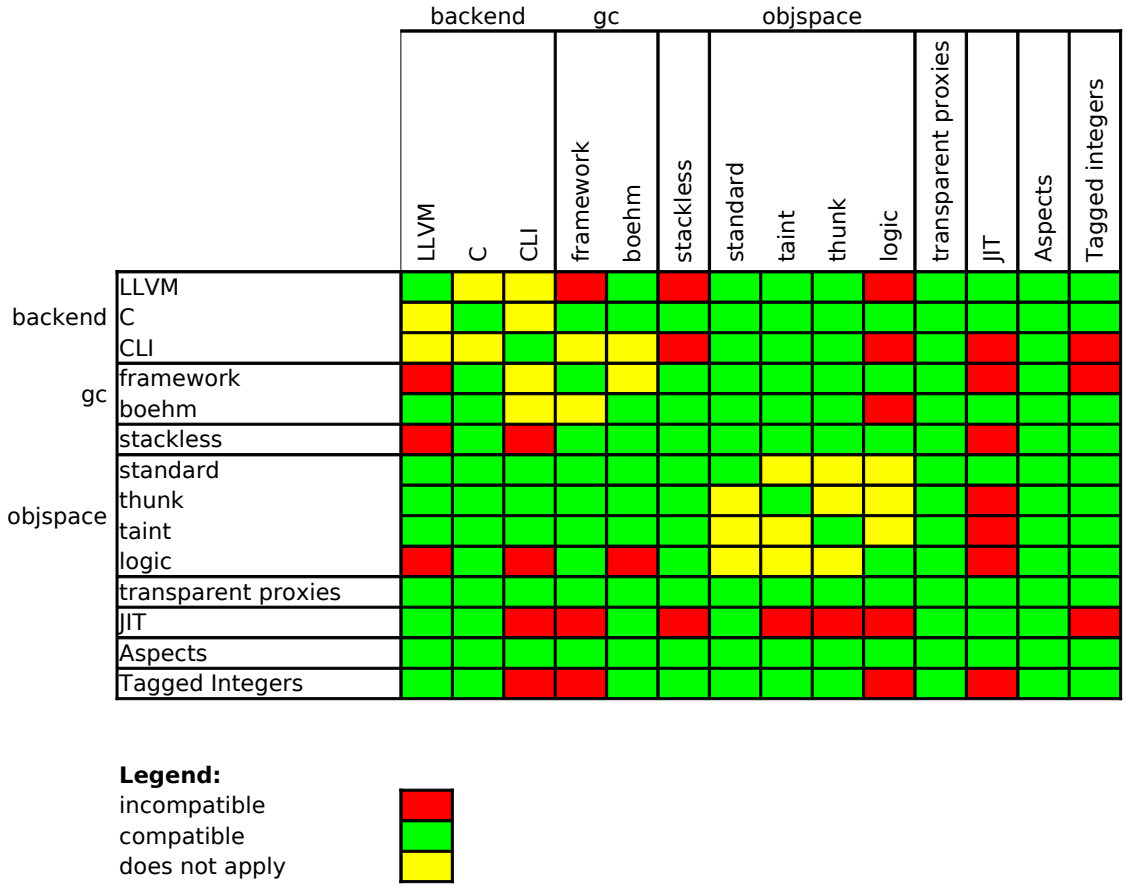

Figure 1: PyPy feature compatibility matrix

adding several new steps to it. In addition a new option was added to choose whether a JIT should be generated or not.

Since the JIT work is quite challenging, it was decided to not concentrate on making the JIT work with configurations other than the default one (using the standard object space, the Boehm GC, no stackless, no tagged integers). However, some features still work well together with the JIT, such as transparent proxies (WP12), aspect oriented programming (WP10) and most optimizations of WP6.

For fundamental reasons the JIT can currently only be translated using the C and the LLVM backend, since it is built to work with the low level type system only.

4.4 WP 9 Logic and Constraint Programming

Due to the generic support for the addition of new object spaces in the configuration framework, integrating the logic spaces and PyPy standard interpreter modules with the PyPy standard interpreter - both of which were WP9 tasks - was rather easy. The harder integration work in WP9 was done on the garbage collector and the stackless transformation: the computation space facilities needed special support for clonable coroutines - coroutines that can be cloned by creating a copy of all data that was allocated while the coroutine is running. This feature needed special support in the garbage collector as well as special support in the `_stackless` module.

Since the logic object space depends on the stackless transformation it does not interact too well with some translation aspects: as described above, the stackless transformer works only with the C backend and not with the LLVM and CLI ones. In addition the specific GC support makes the logic object space dependent on our own garbage collectors and means that it does not work with the Boehm collector, which seems unlikely to change in the future. This means it is also unlikely to ever work with an object-oriented backend such as CLI or JVM, since these environments include garbage collectors that don't have the necessary features.

4.5 WP 10 Aspect-Oriented Programming

The aspect-oriented programming work of WP10 was done on the level of the Python parser that PyPy's Python interpreter includes. This means that there was not much integration work required to use aspects in PyPy. It also means that using aspects is completely independent of any of the other PyPy features, those of the Python interpreter as well as those of the translation toolchain. Since the changes that aspects do happen purely on the AST level, the PyPy bytecode interpreter and object space never see any difference.

The drawback of this approach is that one has to be very careful that aspects don't affect code that one does not expect them to affect.

4.6 WP 12 High-Level Backends and Interpreter Feature Prototypes

High level backends were one of the hardest features to integrate since they required deep changes in the RTyper to transform graphs to using a newly introduced object-oriented type-system. Integrating them also required changes in the translation toolchain, since it added new steps to the translation process that were actually replacing the current ones. Also the addition of various backends added steps to the translation process. All these new steps in the

translation process were fully integrated with the configuration framework so that it is easy to choose a backend and that the choice of type system is done consistently with the backend.

Transparent proxies were easy to add since they are a feature that is purely added at the level of the standard object space, and required no changes in the translation toolchain. This in fact means that transparent proxies interact well with all translation aspects and also with all other optimizations of the standard object space.

The taint object space is an object space that proxies the standard object space. It was easy to integrate because of the general machinery for adding new object spaces in the standard interpreter. Since the taint space proxies the standard object space, it can replace it in almost all situation and work together with with all translation aspects apart from the JIT because the JIT has some standard object space specific code (which should be fixable easily).

5 Debian Packaging

5.1 Goals of Providing Debian Packages

To allow developers and end users to easily install PyPy's Python interpreter and translation toolchain, we decided to use Debian GNU/Linux's packaging system. The reason behind this choice was its popularity: Debian is a popular Linux distribution, and various other distributions, such as Ubuntu, are based upon it. This means that by choosing this system, we target a wide range of users with minimal effort.

5.2 A Debian package

A Debian package is a binary file, containing all the required data and meta-data to install a particular software. The software is provided in binary form (though it is possible to download a so-called "source package" with the raw source code for closer examination), eliminating the need for the user to compile the code locally, which can be impractical if the number of build dependencies is high, or if the compiling requires a large amount of system resources. The package's meta-data includes some critical dependency information, which allows the user to install the software and all the associated required packages in one single operation.

Packages which are part of the Debian project gain benefit from the Debian project infrastructure. This includes a bug tracking system, which, although it is mostly meant to communicate about issues in the packages themselves, is also widely used by users as a proxy to a program author's own bug tracker. This makes sense since the maintainer of a Debian package is generally in close contact with the author of the program, and will sometimes be able to patch the program to fix bugs, or to produce a fixed version of the program using patches available from the author's source tracker. Another important feature of the Debian infrastructure is the autobuilders: Debian supports 11 hardware architectures, with the appropriate automated package compiling machines for each architecture. Software packaged for Debian can potentially, with the help of developers who dedicate part of their time to supporting these architectures, get ported to hardware platforms to which the author does not have access, enlarging its user base as well as its quality, since porting often involves fixing platform-specific bugs.

Since the Debian packaging system is so popular, it has been used as a basis for other distributions, which means that the packages likely install without changes on distributions like

DeMuDi which was produced by the EU funded AGNULA project, skolelinux, and Ubuntu, or can be ported extremely easily.

All in all, targeting Debian-based distributions means that PyPy will get a high visibility among a large public of open source developers, and that it will become available to a large user base. Programs taking advantage of the unique PyPy functionalities such as massive parallelism with stackless, or logic programming, will be easy to deploy thanks to the availability of these packages.

5.3 Provided Packages

Packaging a program for Debian often means providing several binary packages, which is also known as "splitting" the program. This is necessary because of technical reasons and end user needs: for instance, two different packages cannot contain the same file (or else, they need to conflict with each other, and cannot therefore be installed simultaneously). Another reason for this requirement are development packages: some users are only interested in using the program in a production context and do not want to install development tools on their computers, it is therefore necessary to provide a separate development package for developers that do (want to) have the required development tools and libraries in its dependency set. Providing multiple binary packages can also be a means of providing different flavors of a program, compiled with different configuration settings, in order to suit different user needs.

With regard to the PyPy Debian packaging, the choice was made to split the source package in 6 packages: pypy-dev, pypy, pypy-stackless, pypy-logic, pypy-doc and pypy-lib. Another important package on which work was contributed is python-codespeak-lib - the Debian package for the py-lib, which was developed as part of WP2. The work consisted of updating the existing package to suit the changes which where applied between the packaged snapshot and the 1.0 release of py-lib.

5.3.1 pypy-dev

This package provides the PyPy translation toolchain, which requires a Python interpreter to run. It is meant to be used by people wanting to translate the PyPy standard interpreter, or to compile programs written in RPython. It depends on the pypy-lib and python-codespeak-lib packages, and on various third-party packages required for the translation process to succeed.

It's very useful for core PyPy developers working from the source Subversion repository to install this package, since it provides an easy way to install all the dependencies required to work on PyPy on a Debian system.

5.3.2 pypy, pypy-stackless, pypy-logic

These packages provide the PyPy standard interpreter compiled using the C backend. The pypy package provides a default interpreter with no special additional functionality, and the pypy-stackless and pypy-logic packages provide access to the stackless functionalities and the logic object space, respectively. All three packages depend on the pypy-lib package.

These packages are meant to be installed by people who want to use the PyPy standard interpreter, but cannot or do not want to translate it themselves because they do not have

access to a computer which is powerful enough to run the translation (or do not want to install the necessary tools). They will also be used indirectly by people who want to install another program packaged for Debian which requires one of these interpreters to work.

5.3.3 pypy-lib

This package provides a version of the standard Python library suitable for use with the PyPy standard interpreter, both interpreted and translated.

This package is there to provide files required by the pypy-dev, pypy, pypy-stackless and pypy-logic packages.

5.3.4 pypy-doc

This package contains the documentation in raw text and HTML for the PyPy project. As such, it is strongly recommended to install it when installing the other packages, and especially the pypy-dev package.

5.3.5 python-codespeak-lib

This package contains py-lib and its documentation. Installing it makes some high level commands available, such as the py.test tool (D02.3). It is also a dependency of the pypy-dev package.

6 Glossary of Abbreviations

The following abbreviations may be used within this document:

6.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.

GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple Direct-Media Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

6.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

(D02.1) *Configuration and Maintenance of Development Tools and Website*, PyPy EU-Report, 2007

(D02.3) *Automated Testing and Development support library*, PyPy EU-Report, 2007

(D05.1) *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

(D06.1) *Core Object Optimization Results*, PyPy EU-Report, 2007

(D07.1) *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2007

(D08.2) *JIT Compiler Architecture*, PyPy EU-Report, 2007

(D09.1) *Constraint Satisfaction and Inference Engine, Logic Programming in Python*, PyPy EU-Report, 2007

(D10.1) *Aspect-Oriented, Design-by-Contract Programming and RPython static checking*, PyPy EU-Report, 2007

(D11.1) *A Case Study On Using PyPy for Embedded Devices*, PyPy EU-Report, 2007

(D12.1) *High-Level Backends and Interpreter Feature Prototypes*, PyPy EU-Report, 2007

(D14.4) *PyPy Third Milestone Report*, PyPy EU-Report, 2007

(D14.5) *Documentation of the Development Process and Sprint Driven Development*, PyPy EU-Report, 2007

(PYEXEC) *py.execnet documentation*:
http://codespeak.net/py/dist/execnet.html,
py.execnet api documentation:
http://codespeak.net/py/dist/apigen/api/execnet.html

(CONFIG) *PyPy's configuration handling*:
http://codespeak.net/pypy/dist/pypy/doc/configuration.html

(RELEASE) *PyPy 1.0: JIT compilers for free and more*, release announcement, March 27th 2007,
http://codespeak.net/pypy/dist/pypy/doc/release-1.0.0.html