
# 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**

# Release Automated Testing Framework with html/pdf Reports and Runtime http-Server for Introspection Aiding Debugging and Understanding of PyPy Internals

## Due date of deliverable: March 2007

## Actual Submission date: 12th March 2007

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

**Lead Contractor of this WP: merlinux**

**Authors: Holger Krekel, Guido Wesdorp (merlinux)**

**Revision: Interim**

**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 |
|---|---|---|
| Dec 01 2006 | Carl Friedrich Bolz | created a rough structure, outline |
| Jan 21 2006 | Holger Krekel | Draft of exec summary, outline & abstract |
| Jan 28 2006 | Carl Friedrich Bolz | Publishing of intermediate version on the web |
| Mar 10 2006 | Holger Krekel | Filled and refined all sections |
| Mar 10 2006 | Guido Wesdorp | More refinement, small additions |
| Mar 10 2006 | Holger Krekel | Review, refactoring for Interim Version |
| Mar 12 2006 | Carl Friedrich Bolz | Publish Interim Version on the website |

## Abstract

This report describes PyPy's easy-to-use testing and development support tools. We discuss unique features like ad-hoc distributed testing and distributed program execution, as well as flexibility in configuration, execution and reporting. Also we describe the various testing needs that arose and have been satisfied in the context of PyPy as a tool set and framework for implementing dynamic languages.

## Contents

# 1 Executive Summary

Development and research within the PyPy project relies and depends on automated testing and a test-driven development approach. The Python community traditionally uses project specific extensions to the Python version of the JUnit Java test library (itself derived from Smalltalk's SUnit) to unit-test applications. The project specific extensions usually deal with extending the test collection, test execution and test reporting process. Most of these extensions are not easy to reuse by other projects. Moreover, PyPy, being a full Python implementation and a full compilation tool chain, has special needs for amending and extending these testing aspects.

For these reasons, the PyPy developers wrote `py.test`: a highly configurable, project-independent tool that aims at flexibility and ease of customization as well as providing a more modern API and taking a more flexible approach to testing. Its support for per-project, reusable extensions promotes sharing of code between projects and developers, and makes it usable not only to test code execution, but also for documentation and integrity tests.

`py.test` minimizes the overhead needed to implement a project-specific testing process. Particular emphasis lies on providing helpful and detailed feedback to the developer in case of test failures. When such failures occur, there are several facilities to observe the full context and introspect the running program.

Usage is not limited to testing Python code, or tied to a particular PyPy interpreter implementation. In fact, it is possible to run compliance tests for other interpreters such as the emerging one for JavaScript, or use different combinations of PyPy's interpreter and object space. Moreover, the testing architecture can and has been extended to drive GUI and acceptance tests.

More recently, we pioneered ad-hoc distributed testing, which allows to distribute PyPy's thousands of tests to multiple computers. Since running tests relating to translation of code snippets down to C or .NET code is particularly time consuming, the advent of ad-hoc distributed testing has helped to considerably speed up development cycles. Another more recent addition is a documentation generation tool that incorporates information gathered while running tests into the documentation.

Throughout 2004 to 2007 we incrementally developed the `py lib` by means of a test-driven approach. Many projects and developers and entities began using it from the publically accessible codespeak Subversion repository. It has been presented at multiple Python Community conferences and beginning 2007 the first full release was prepared and published.

The development of the `py lib` along with its tools will continue after the EU project phase, and there are opportunities on extending, consulting and teaching `py.test` and the benefits of using a thoroughly employed testing process and tool, relaying the experiences obtained during the execution and implementation of the PyPy EU project. It is certain that the py lib will continue to evolve as several individuals, merlinux and an emerging community are devoted to improving and maintaining it.

# 2 Introduction

## 2.1 Purpose of this Document

This document provides basic information for developers and prospective users about PyPy's development support "py" library. Particularly, we highlight features and reference starting points for using the `py.test` testing tool, with a particular focus on its past and future uses in the PyPy project.

## 2.2 Scope of this Document

This document provides a conceptual overview of PyPy's testing tool and development support library. For detailed technical information, we refer to the exhaustive web and API documentation.

## 2.3 Related Documents

- D02.1 Configuration and Maintenance of Development Tools and web
- D12.1 High-Level Backends and Interpreter Feature Prototypes
- D13.1 Integration and Configuration
- D14.5 Development Process

# 3 py lib: development support library

The py lib provides a consistent set of rather independent support tools and code for development and testing. For the purposes of this report, we discuss a few innovative areas, briefly mention other features and otherwise refer to available online documentation.

## 3.1 py.test: the testing tool

A main benefit of using `py.test` is that it's easy to learn: it hardly requires any boilerplate code and a developer can immediately start to write and run tests for his application. As the number of tests grows, and new testing needs arise in a project, more features and extension possibilities can be implemented, usually without modifying already written test code.

A main difference to other approaches is py.test's separation between test code and test configuration and extensions. For example, to write new reporters or to check for memory leakage around test methods, no intrusive changes to the actual test classes and functions are required.

Particular focus lies on reporting about and interacting with the application context when a test failure is encountered. One may enable additional reporting of program variable values, or drop to an interactive Debugger to introspect variables and the execution call stack.

Using `py.test` does not require to hook test classes into framework class hierarchies. In fact, the test collection process discovers tests by recursively searching for test files, classes and

methods, expecting certain naming conventions. These naming conventions are modifiable on various levels. Systematic hooks and temporary test directory management allow fine grained and easy control over test setup and state on module, class or method level.

When the tests are ran, `py.test` modifies the `assert` statement so that it provides detailed information about the involved values which result in a failed Assertion. Most other approaches require the use of a (limited) set of methods to express assertions about values. With `py.test`, assertions about truth values are regular Python expressions.

`py.test` allows to modify all of its test collection, execution and reporting aspects through the use of per-project or cross-project `conftest.py` files. Using this facility, we were able to remotely run Windows GUI tests from a Linux machine in an ad-hoc manner, one of the facilities of interest to commercial entities.

Recently we introduced ad-hoc distributed testing. By running tests on several hosts at once, test runs receive a considerable speed up. Showcasing both PyPy's and the py lib's flexibility, a new web application generated with PyPy's translation tool chain and JavaScript backend visualizes (see D12 report) test progress and allows to interactively view failures from distributed test runs. The only prerequisite for using distributed testing is a remote computer allowing access to a runnable Python interpreter, for example through the popular Secure Shell server.

For a more detailed feature list, architecture and extension possibilities, please refer to the py.test documentation.

## 3.2 Ad-hoc Distribution of Program Code

Driven partially by requirements of the testing tool, the py library provides a new remote execution model which makes full use of Python as a very dynamic language, being able to compile and execute program text at run time. Through its py.execnet namespace and lean API it provides means to seamlessly instantiate and interact with other processes, both local ones and remote processes instantiated through a Secure Shell or through socket connections.

The communication protocol between the sending and the receiving side is determined by the sender. There are no installation requirements on the remote side, except for a runnable Python interpreter. Once set up, one may send program functions and fragments to the remote side and send and receive basic Python objects via **Channels**, overall providing a convenient model to exchange structured data by locally modifiable protocols.

`py.test` itself uses this way of distributing programs to perform ad-hoc distributed testing, but also for co-ordinating translation build jobs and for system maintenance tasks. It has received positive feedback at conferences and from individuals, and is at the verge of providing an interesting model for distributing and controlling processes and applications for larger networks of heterogeneous platforms.

## 3.3 Rich API Documentation

With its public release, all of the specified exported API of the py library is documented online. In addition to the information provided by many other tools, information collected from test runs provides insights into types of arguments, return values as well as call traces. This documentation is generated using `apigen`, a recently added feature to the py lib.

In dynamic languages like Python it can be hard to find out in what ways an API can be used, what argument types a function can handle, and what return values it can produce.

In well-tested environments, most of this information can be derived by monitoring test runs. Apigen monitors all function calls during test runs to provide such runtime information in the documentation. This adds valuable information to the API documentation, and additionally serves as an indicator for the quality of the tests.

### 3.4 Other Features

Besides `py.test`, the py lib provides support for higher level access to system, language and control flow aspects:

- Path objects for uniformly accessing local file systems and subversion file systems.
- greenlets for lightweight concurrent programming (used by the distributed execution prototype in D12.1)
- py.code for high level access to Python Code and Traceback information, substituted and extended by PyPy to connect it to its own Interpreter objects.
- py.log for logging events to various channels
- py.xml for generating and rendering XML objects, used for generating web pages.
- and various smaller useful features

### 3.5 Availability and Releases

The `py lib` is and has been permanently publically accessible through codespeak's Subversion repository (see D02.1 for more information) since 2004. Several outside developers stated that they enjoy the ability to follow the development of the tool and have encountered few problems just using the latest repository version - the same approach taken for developing the PyPy interpreter and compiler code base.

In February 2007, the 0.9.0 version was released to the public, together with extensive introductory and API documentation. It contains all main features, including ad-hoc distributed testing, reported through a PyPy generated AJAX web application. It can be used on Windows, Mac OS and Linux platforms and runs on top of CPython 2.3, 2.4. and 2.5.

## 4 Usage of py lib Facilities in PyPy

PyPy pervasively uses the py library for multiple purposes, such as accessing the file system, event logging, to distribute translation build jobs, and to generate web pages and PDF Reports using its py.rest tool.

PyPy makes use of and extends the `py.test` tool at multiple levels, providing methods to write different test types:

- Interpreter level tests: instrumented and set up with an Object Space, these tests run directly on CPython and test low level PyPy interpreter or translation aspects.

- Application level tests: these tests are instrumented and set up with particular Object Spaces and PyPy's own Python Interpreter, running itself on CPython. Additionally, these tests can run directly on a translated PyPy version (which of course is considerably faster as it avoids the double interpretation overhead).
- Compliance tests: the CPython regression tests are instrumented to run either through PyPy's interpreter, or directly on a translated PyPy version.
- JavaScript tests: ECMA regression tests (written in JavaScript) are instrumented to run on top of the emerging JavaScript PyPy interpreter.
- Documentation tests: web site generation and full link and reference integrity tests are executed from a special extension of `py.test`.

These test types co-exist in the PyPy project and can be invoked through the `py.test` tool. Also, PyPy developers have extended reporting facilities to provide revision and nightly build testing information on web pages, to allow detection and introspection of problems that require longer running processes.

All of of the test types can be seamlessly distributed to multiple hosts, allowing a developer to perform large test runs before committing a change to the code base - a practice that is hard to follow if such test runs take too long.

# 5 Glossary of Abbreviations

The following abbreviations may be used within this document:

## 5.1 Technical Abbreviations:

| | |
|---|---|
| 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. |

## 5.2 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 |