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

D14.2 Tutorials and Guide Through the PyPy Source Code

Due date of deliverable: March 2007

Actual Submission date: March 23rd, 2007

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

Lead Contractor of this WP: Change Maker

Authors: Beatrice Düring (Change Maker)

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
2006-09-05	Beatrice Düring	First draft of text after research
2007-02-07	Beatrice Düring	Redraft after internal review
2007-02-14	Lene Wagner	Review, insert references
2007-02-24	Beatrice Düring	Finalizing all sections
2007-02-28	Michael Hudson	Internal review
2007-02-28	Carl Friedrich Bolz	Publish Interim on the Web Page
2007-03-21	Beatrice Düring	Changes based on external review by Steve Holden
2007-03-22	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

This report summarizes the supporting documentation that has been used for dissemination purposes in the project. The primary tool for face-to-face dissemination with a hands-on purpose has been the “sprint introduction”, an introductory tutorial for Python programmers presented at the start of a sprint. The current version of this tutorial is appended to this report.

The report presents how the on-line documentation at http://codespeak.net/pypy acts as an important guide through not just the source code but also to the coding style used in the development process and the various supporting tools that are being employed in the project. We also present the efforts of the project to ensure, through various mediums and entry points, a creative and flexible strategy of documenting software development. A key part of the on-line documentation (the getting-started document) is appended to this report.

Purpose, Scope and Related Documents

This document gives an insight into various introductory access points into the PyPy project, which have been created and maintained during the project period of December 2004 until March 2007, targeting technical users and possible contributors from the Python community.

This document summarizes the supporting documentation being used in both face-to-face and on-line dissemination contexts for explaining the PyPy architecture and source code and the development methodology being used in the project. This document does not serve as a tutorial or a guide through the source code *per se*.

This document mentions dissemination activities that have been previously reported in the following reports:

- D14.1 report on phase 1
- D14.3 report about milestone/phase 2

Contents

1 Executive Summary 4

2 Face-to-Face dissemination: supporting documentation 4

2.1 Sprints and the necessity of tutorials 4

2.2 Technical talks and architectural diagrams 6

3 On-line dissemination 6

3.1 PyPy User Documentation: strategy and content 6

3.2 Project Documentation: strategy and content 7

3.3 Source Code Documentation: strategy and content 8

4 References 9

5 Glossary of Abbreviations 9

5.1 Technical Abbreviations: 9

5.2 Partner Acronyms: 10

Appendices

- getting-started snapshot from the web page
- current version of the sprint introduction

1 Executive Summary

The PyPy project has, thanks to its EU grant, been able to implement sprint driven development in a more regular and systematic way than was possible before the funding. The EU funding has had a great impact on the dissemination work done in the project. It has meant extending the usual F/OSS on-line documentation with more extensive getting-started support, in-depth technical reports, talks and papers published for various technical and agile conferences and even more unorthodox dissemination methods such as video documentation of talks, tutorial sessions, interviews, sprint work and design discussions.

The methodology of "sprinting" also played a key role when looking at dissemination activities, especially when attracting new contributors. Tutorials as a start up sprint activity have been the practice ever since the PyPy community started sprinting, and have proved to be the most efficient dissemination strategy (shortening the learning curve over time) for face-to-face sessions when combined with working collaboratively with newcomers.

The PyPy project provides, through its extensive on-line documentation (foremost the "getting-started" section), to the reader the ability to:

- Start writing code on top of PyPy, using the tools needed in this particular language environment
- Find and navigate the source code, and suggestions on how to begin contributing to it
- Understand the source code, firstly by understanding the overall architecture of the project but also by reading coding guidelines and testing philosophy
- Read extensively and in-depth on the various components of the PyPy architecture, orienting themselves based on *individual* interest and not the assumptions of the project and its core contributors

The combination of this on-line documentation with sprint participation (including tutorial sessions and working with more experienced PyPy developers) has significantly lowered the learning curve in a technically complex open source project, inspiring contributors to participate and become part of the PyPy community.

2 Face-to-Face dissemination: supporting documentation

2.1 Sprints and the necessity of tutorials

The EU funded part of the PyPy project is designed to develop a flexible and fast Python implementation purely written in Python. The strategy used is "leveraging the community" - evolving the PyPy architecture in close collaboration with the Python community. To fulfill this strategy the main practice is to continue and enhance the Sprint Driven Development process.

The PyPy project grew as a community out of the practice of "sprinting". A "sprint" is a one week face-to-face coding session that supports working accelerated and collaboratively when co-located, minimizing many of the risks of traditional distributed and dispersed F/OSS development[1].

These public coding sprints are a main entry point for single contributors to join the project, and a majority of today's PyPy contributors entered the project in this way. Thus it was obvious that PyPy sprints should be targeted towards enabling newcomers to join[2].

[1] Sprints are an effective way to use established agile practices (pair programming, whole team, test driven development, daily planning meetings etc [XP]) in a distributed and dispersed reality, using incremental and iterative cycles of co-located development. Although using agile practices, a PyPy or Python "sprint" should not be confused with the Agile methodology SCRUM and the meaning of sprints in the SCRUM sense [SCR].
Find a more detailed description of the methodology and practices of sprinting in the PyPy community at http://codespeak.net/pypy/dist/pypy/doc/dev_method.html.

[2] The model of contributing as a Physical Partner of the EU project and the Summer-of-PyPy programme facilitated newcomer participation in PyPy sprints during the PyPy EU project period. Both these models provided opportunities for non-consortium participants to recieve reimbursement of travel costs to the sprints in exchange for contributions to the project.

In order for the sprint newcomers to be productive it is important to get a rudimentary understanding of the architecture and codebase as quick as possible. This is achieved by core developers presenting and going through a tutorial (a slide presentation) on the first morning of the sprint, in order to enable enough understanding for newcomer to pair up with a more experienced developer on an area of matching interests or according to work needed.

By December 2003, one year before the EU-funded part of the project started, the PyPy core group had produced the first version of a PyPy tutorial, a presentation of 12 slides called "PyPy - Architecture overview".

The tutorial was extended to keep up to date regularly, but by the time of the sprint at PyCon 2006 (February) it was rewritten to match the results of both the 0.7 and the 0.8 releases. This version of the tutorial presentation was used during spring 2006 as sprint introduction of sprints arranged at the two largest Python language conferences, PyCon 2006 and EuroPython 2006, both sprints being attended by many newcomers.

This current tutorial presentation, "PyPy - Crash Course/Sprint Intro" is 42 pages long and takes approximately 1 hour to present - please view the entire tutorial session given at PyCon 2006 sprint by Michael Hudson in our video documentation (http://codespeak.net/pypy/dist/pypy/doc/video-index.html). The tutorial is an in-depth tour of the PyPy architecture (the Standard Interpreter, the Translation tools with details on the flow model and analysis and the various translation options available) but it is also showing how code is being implemented in the project specific RPython subset as well as important jargon needed to navigate the codebase.

The purpose of the current tutorial is to support navigation through the codebase by creating an understanding of the PyPy architectural landscape - which is complex enough in itself - without drowning the audience in all the details. However, this high-level overview would not be sufficient enough to get started if it was not almost immediately followed up by pair programming with someone more experienced, which gives the low-level implementation insights. Because of this core practice the goal of the sprint introduction and the tutorial is not to explain and show everything - just enough to support the main work in the sprint which is coding - "learning by doing". Evaluations of the PyCon sprint showed great appreciation for this latest tutorial by the newcomers.

During the duration of the EU project the various versions of this tutorial have been presented at 9 public sprints to approximately 50 sprint newcomers.

Informal evalution with the Summer of PyPy participants on what effect sprints had on their PyPy learning curve yielded the following quotes which give a good picture of how sprints and online documentation complement each other:

> "My approach to PyPy has been a bit non-typical because I was in a hurry to finish my thesis and I had no time to wait for the next sprint, so the online documentation has been my main source of PyPy knowledge, especially the documents available under the "documentation" sections; I didn't need to read the technical reports much. But I have to admit that with only the online docs I couldn't have got started very soon, because a lot of things was still unclear and a lot of unanswered questions arose while coding: what definitely made the difference was the IRC channel, where I could ask question and have answers almost immediately. "
>
> Antonio Cuni
>
> "I mostly agree with him (Antonio Cuni), PyPy is a tough (and cool) project. It isn't the kind of opensource project you administer to master in the weekend and start sending patches to. The documentation was helpful to get a broad view of the entire project and some of its internals but the difference, in my own case, was the sprint I attended ad EuroPython 2006. I had the chance to get in touch with the team and I think I learnt more in those three days than in the previous couple of weeks in loneliness :-)
>
> The sprint really smoothed my learning curve. I received much help from Anders and Armin and I'm grateful to them and the whole team."
>
> Lawrence Oluyede

"It (online documentation) did reduce the learning curve, but in my experience (writing another language on the pypy framework) it didn't help that much in showing what is possible and how easy it is. What I mean is, the documentation that exist is pretty solid, but they don't have any documentation to motivate people in a simple way, much like what you get when you see a screencast of those new web framework (like RoR or TG). But the people on the sprint and on the irc channel have helped me a lot. They really solved my doubts, pointed to the right way and even pitched in when I asked then to."

Leonardo Santaquada

"A bit. There was no or little documentation regarding parts of pypy where I've started, so it was more about feedback from IRC than docs. Sprint participation helped a lot. Otherwise I don't think I would be able to enter the PyPy project at all."

Majiek Fijalkowski

2.2 Technical talks and architectural diagrams

Separately from these tutorials, technical presentations have been given frequently, especially when meeting a specific technical audience at a specific event - such as conferences. The difference is here that these presentation events have been arranged as dissemination activities for audiences that were not meant to be participating in the coding sprints. The project also presented technical talks at universities purely for dissemination purposes, not in connection with conferences or sprints.

During these events, as opposed to the tutorial presentation, the goal has been more to convey a high level summary of the project's motivations, architecture, implementation and goals. The goal then was to introduce the audience to the concepts behind PyPy and invite interested people to join the community, read documentation, join sprints etc.

Of course, the usage of and the presentation style of both technical talks and tutorials varies among the core developers holding them, along with possibly different mentoring approaches and interests. It is safe to say though that the supporting practice of mentoring and tutorial sessions is a "fixed" component of any sprint where there are newcomers participating - what varies is the style of presenting, ranging from formal to more informal. Note also that tutorial sessions are given purely when having sprints and face-to-face interaction. For newcomers that do not have the possibility to join a sprint the opportunities to get into PyPy in the on-line documentation range from reading the documentation (and the tutorial/presentations, trying the demos), watching video documentation and asking questions on the main developer mailing list, or better yet, on the main developer IRC channel for real time mentoring.

3 On-line dissemination

The on-line PyPy dissemination is done through the main PyPy website, at

http://codespeak.net/pypy/dist/pypy/doc/index.html

Here we are presenting the main subsections of it, explaining their different focus and content.

3.1 PyPy User Documentation: strategy and content

The "PyPy User Documentation" targets programmers using PyPy as their Python implementation to built applications on it, presenting tools and features. In distinction to potential contributors to the PyPy implementation itself, no deep understanding of the PyPy source code is needed. The user documentation provides both very high level information like FAQs, the "new features" section - an in-depth explanation of PyPy's specific features - and the "getting started" section.

"Getting started" covers all aspects from obtaining the PyPy source code, short introductions into its main features and to the related tools, prototypes and testing procedures used when developing PyPy. Thus, the "getting-started" documentation also acts as the main release documentation, the current version catering to the 0.99 release.

However, apart from giving an insight into PyPy features, "getting-started" documentation also helps with getting into the actual source code, up to a certain degree. In this sense, it acts as an on-line tutorial. The reader is guided towards interesting starting points to use PyPy and is shown examples on how to try out components of the architecture (such as running translation and the various backends). But a directory reference also helps the interested reader to trace between the documentation and the various components of the PyPy architecture and the actual source code.

As such, the "getting-started" section is the most important on-line documentation available because it is written to make sure that:

- future users can download and start working with PyPy
- enabling an understanding of PyPy's features and surrounding tools and thus a sustainable interest in the project
- new users and contributors get the setup they need in order to be able to start "hacking away", even without participating in a sprint and getting face-to-face mentoring as described above.

This PyPy strategy and the usual F/OSS contribution ladder is discussed in the paper "Sprint-Driven Development: working, learning and the process of enculturation in the PyPy community", which was the result of research done by the socGSD project during a PyPy sprint [SOC].

Feedback from a user of the "getting-started" page:

> "BTW, I think the getting started page is about the clearest, best such page I have ever seen."
>
> ...
>
> "The result of the good writing/design at
>
> http://codespeak.net/pypy/dist/pypy/doc/getting-started.html#downloading-running-the-pypy-0-9-release was that I was able to *quickly* see exactly what I had to do. I think the gray text boxes were very helpful: they showed that only easy typing was required, and they broke the text up into easily understandable pieces.
>
> I know how hard this kind of introductory writing is, and how important it is. I've spent years trying to do as well with the Leo introduction, but haven't :-) The problem is that everything must be discussed first, and you have to tell everything essential and **nothing else**. Perhaps another way of saying all this is that the intro text made no assumptions about what I would know.
>
> One more thing. The contents at the top, and the clear organization of the entire page, made the entire process less intimidating."
>
> Edvard K. Ream

3.2 Project Documentation: strategy and content

The PyPy project documentation has the primary purpose of orienting readers to the project context of PyPy. Thanks to the EU funding and the dissemination work being done as part of that work PyPy has extensive documentation in the form of talks (slide presentations) at various conferences and reports submitted to the

Commission as part of presenting the technical results in the form of deliverables. Also as part of the EU-funded project video documentation was produced - covering talks at various conferences, interviews with core developers from the CPython and PyPy communities, a tutorial presentation as well as filmed snippets from various sprints covering design discussions and how work is done during PyPy sprints. By 27th February 2007, 6955 successful downloads of 40 different torrents have been made.

The main strategy behind this extensive documentation *about* the project is to create a "whole brained" approach to documentation and create a lower threshold for entering the project by not just documenting the code but also the process. Because the sprint driven development is such a core practice, infusing and driving the entire development process, it is important that newcomers understand how this is done - and also to *see* that the project employs an "open door policy" via the public coding sprints.

As part of the project documentation the readers can find information about the results at previous sprints (sprint reports), a summary of the development methodology being used, coding guidelines, license information, a glossary for PyPy specific terminology and the extensive talks and technical reports "library" - where also interim version of the reports are being published for community feedback. An architectural overview helps readers to understand the basic high-level components of the PyPy platform so to better understand the language and content of the reports and talks.

The technical reports being submitted to the Commission serve the purpose of presenting the technical results of the EU-funded part of the PyPy project. Because the main target group for dissemination is the Python community it was a clear strategy to write the technical reports so that they also served a secondary purpose - to provide to the community a more extensive documentation that many other non-funded F/OSS communities do not have the capacity or resources to produce even if there was need for it. Through dialogue on the development mailing lists the core developers have come to understand that this extensive documentation, in the form of the technical reports, have been appreciated and have been perceived as useful.

There is also cases where the Project Documentation has been useful for academic research purposes. The socGSD team at University of Limerick, Ireland, performed studies of the August 2006 sprint in Limerick and used the Project Documentation as well as the email archives (dating back to 2003) for their research [SOC]. Researchers involved in the Calibre project (http://www.calibre.ie) have also used the documentation for research purposes on how F/OSS projects work.

3.3 Source Code Documentation: strategy and content

The technical complexity of the PyPy platform and the various options and features available provides a challenge when documenting. "Source Code Documentation" aims at providing in-depth context and understanding of the major components of the PyPy architecture. This documentation is mainly directed towards readers that know what area they want to investigate in order to better understand the source code, and readers that may or may not be contributing but are curious as to how certain components have been implemented design-wise in the PyPy platform.

As a documentation equivalent of second-line or back-end support ("getting-started" acting as front-line support) the documentation per component is quite extensive - explaining not only why the component is needed and what services it performs within the PyPy architecture but also summarizing the main design ideas and decisions made when implementing the component. This is combined with practical examples of options, variations, exceptions and tests involved. The technical reports submitted to the Commission and published under "Project Documentation" could be viewed as a complement to the "Source Code Documentation" since it has the same goal of creating an in-depth technical presentation of components and features - albeit with more background and project context as well as academic references and bibliography included.

The "Source Code Documentation" provides a detailed presentation of components of the PyPy architecture such as Object Spaces, Byte Code Interpreter, JIT generation in PyPy as well as information about various aspects of the translation tool chain as well as providing documentation for the CLI backend.

It is important to stress that the Source Code Documentation has evolved since 2003, in many cases as the result of concrete feedback from the users within the community. The quote above from a community member is a good example of the kind of feedback that help shaping the documentation.

4 References

5 Glossary of Abbreviations

The following abbreviations may be used within this document:

5.1 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 DirectMedia 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

References

[XP] Beck, K. (1999). Extreme programming explained: Embrace change. Reading, MA: Addison-Wesley. Highsmith, J. (2002). Agile software development ecosystems. Boston, MA: Pearsons Education. Highsmith, J. & Cockburn, A. (2001). Agile Software Development: The Business of Innovation. In Computer 34(2).

[SCR] Schwaber, K. & Beedle, M. (2002). Agile Software Development with Scrum. Upper Saddle River, NJ: Prentice-Hall.

[SOC] Avram, G., Sigfridsson, A., Sheehan, A. & Sullivan, D.K. (2007) Sprint-Driven Development: working, learning and the process of enculturation in the PyPy community. Paper accepted for The Third International Conference on Open Source Systems, Limerick, Ireland.

PyPy - Getting Started

Contents

1 What is PyPy ?

2 Just the facts

- 2.1 Downloading & running the PyPy 0.99 release
- 2.2 Svn-check out & run the latest PyPy as a two-liner
- 2.3 Understanding PyPy's architecture
- 2.4 Running all of PyPy's tests
- 2.5 Filing bugs or feature requests

3 Interesting Starting Points in PyPy

- 3.1 Main entry point
  - 3.1.1 The py.py interpreter
  - 3.1.2 py.py options
- 3.2 Special PyPy features
  - 3.2.1 Interpreter-level console
  - 3.2.2 Tracing bytecode and operations on objects
  - 3.2.3 Lazily computed objects
  - 3.2.4 Logic programming
  - 3.2.5 Aspect Oriented Programming
  - 3.2.6 Running the tests
  - 3.2.7 Demos
- 3.3 Trying out the translator
  - 3.3.1 Trying out the type annotator
  - 3.3.2 Translating the flow graph to C code
  - 3.3.3 Translating the flow graph to LLVM code
  - 3.3.4 Translating the flow graph to Javascript code
  - 3.3.5 Translating the flow graph to CLI code
  - 3.3.6 A slightly larger example
- 3.4 Translating the PyPy interpreter
  - 3.4.1 Translating with the thunk object space
  - 3.4.2 Translating using the LLVM backend
  - 3.4.3 Translating using the CLI backend
    - 3.4.3.1 Trying the experimental .NET integration
  - 3.4.4 Translate using the Build Tool

3.5 Where to start reading the sources
3.6 Additional Tools for running (and hacking) PyPy
3.6.1 graphviz & pygame for flowgraph viewing (highly recommended)
3.6.2 CTypes (highly recommended)
3.6.3 CLISP
3.6.4 py.test and the py lib
4 Getting involved

1 What is PyPy ?

PyPy is an implementation of the Python programming language written in Python itself, flexible and easy to experiment with. Our long-term goals are to target a large variety of platforms, small and large, by providing a compiler toolsuite that can produce custom Python versions. Platform, memory and threading models are to become aspects of the translation process - as opposed to encoding low level details into the language implementation itself. Eventually, dynamic optimization techniques - implemented as another translation aspect - should become robust against language changes. more...

2 Just the facts

2.1 Downloading & running the PyPy 0.99 release

Download one of the following release files and unpack it:
pypy-0.99

- download one of
  - pypy-0.99.0.tar.bz2 (unix line endings) or
  - pypy-0.99.0.tar.gz (unix line endings) or
  - pypy-0.99.0.zip (windows line-endings) and unpack it
- alternatively run
  - `svn co http://codespeak.net/svn/pypy/release/0.99.x pypy-0.99.x` (the 0.99 maintenance branch)

to get it from the subversion repository. Then change to the `pypy-0.99.0` or `pypy-0.99.x` directory and execute the following command line:

```
python pypy/bin/py.py
```

This will give you a PyPy prompt, i.e. a very compliant Python interpreter implemented in Python. PyPy passes around 95% of CPythons core language regression tests. Because this invocation of PyPy still runs on top of CPython, it runs around 2000 times slower than the original CPython.
However, since the 0.7.0 release it is possible to use PyPy to translate itself to lower level languages after which it runs standalone, is not dependant on CPython anymore and becomes faster.

2.2 Svn-check out & run the latest PyPy as a two-liner

If you want to play with the ongoing development PyPy version you can check it out from the repository using subversion. Download and install subversion if you don't allready have it. Then you can issue on the command line (DOS box or terminal):

```
svn co http://codespeak.net/svn/pypy/dist pypy-dist
```

This will create a directory named `pypy-dist`, and will get you the PyPy source in `pypy-dist/pypy` and documentation files in `pypy-dist/pypy/doc`.

After checkout you can get a PyPy interpreter via:

```
python pypy-dist/pypy/bin/py.py
```

have fun :-)

We have some help on installing subversion for PyPy. Have a look at interesting starting points for some guidance on how to continue.

2.3 Understanding PyPy's architecture

For in-depth information about architecture and coding documentation head over to the documentation section where you'll find lots of interesting information. Additionally, in true hacker spirit, you may just start reading sources .

2.4 Running all of PyPy's tests

If you want to see if PyPy works on your machine/platform you can simply run PyPy's large test suite with:

```
cd pypy
python test_all.py directory-or-files
```

test_all.py is just another name for py.test which is the testing tool that we are using and enhancing for PyPy. Note that running all the tests takes a very long time, and enormous amounts of memory if you are trying to run them all in the same process; test_all.py is only suitable to run a subset of them at a time. To run them all we have an autotest driver that executes the tests directory by directory and produces pages like the following one:

http://wyvern.cs.uni-duesseldorf.de/pypytest/summary.html

2.5 Filing bugs or feature requests

You may file bug reports on our issue tracker which is also accessible through the 'issues' top menu of the PyPy website. using the development tracker has more detailed information on specific features of the tracker.

3 Interesting Starting Points in PyPy

The following assumes that you have successfully downloaded and extracted the PyPy release or have checked out PyPy using svn. It assumes that you are in the top level directory of the PyPy source tree, e.g. pypy-x.x (if you got a release) or pypy-dist (if you checked out the most recent version using subversion).

3.1 Main entry point

3.1.1 The py.py interpreter

To start interpreting Python with PyPy, use Python 2.3 or greater:

```
cd pypy
python bin/py.py
```

After a few seconds (remember: this is running on top of CPython), you should be at the PyPy prompt, which is the same as the Python prompt, but with an extra ">".

Now you are ready to start running Python code. Most Python modules should work if they don't involve CPython extension modules. Here is an example of determining PyPy's performance in pystones:

```
>>>> from test import pystone
>>>> pystone.main(10)
```

The parameter is the number of loops to run through the test. The default is 50000, which is far too many to run in a non-translated PyPy version (i.e. when PyPy's interpreter itself is being interpreted by CPython).

3.1.2 py.py options

To list the PyPy interpreter command line options, type:

```
cd pypy
python bin/py.py --help
```

py.py supports most of the options that CPython supports too (in addition to a large amount of options that can be used to customize py.py). As an example of using PyPy from the command line, you could type:

```
python py.py -c "from test import pystone; pystone.main(10)"
```

Alternatively, as with regular Python, you can simply give a script name on the command line:

```
python py.py ../../lib-python/2.4.1/test/pystone.py 10
```

See our configuration sections for details about what all the commandline options do.

3.2 Special PyPy features

3.2.1 Interpreter-level console

There are quite a few extra features of the PyPy console: If you press <Ctrl-C> on the console you enter the interpreter-level console, a usual CPython console. You can then access internal objects of PyPy (e.g. the object space) and any variables you have created on the PyPy prompt with the prefix `w_`:

```
>>>> a = 123
>>>> <Ctrl-C>
*** Entering interpreter-level console ***
>>> w_a
W_IntObject(123)
```

Note that the prompt of the interpreter-level console is only '>>>' since it runs on CPython level. If you want to return to PyPy, press <Ctrl-D> (under Linux) or <Ctrl-Z>, <Enter> (under Windows).

You may be interested in reading more about the distinction between interpreter-level and app-level.

3.2.2 Tracing bytecode and operations on objects

You can use the trace object space to monitor the interpretation of bytecodes in connection with object space operations. To enable it, set `__pytrace__=1` on the interactive PyPy console:

```
>>>> __pytrace__ = 1
Tracing enabled
>>>> a = 1 + 2
|- <<<< enter <inline>a = 1 + 2 @ 1 >>>>
|- 0    LOAD_CONST    0 (W_IntObject(1))
|- 3    LOAD_CONST    1 (W_IntObject(2))
|- 6    BINARY_ADD
  |-    add(W_IntObject(1), W_IntObject(2))   -> W_IntObject(3)
|- 7    STORE_NAME    0 (a)
  |-    hash(W_StringObject('a'))   -> W_IntObject(-468864544)
  |-    int_w(W_IntObject(-468864544))   -> -468864544
|-10    LOAD_CONST    2 (<W_NoneObject()>)
|-13    RETURN_VALUE
|- <<<< leave <inline>a = 1 + 2 @ 1 >>>>
```

3.2.3 Lazily computed objects

One of the original features provided by PyPy is the "thunk" object space, providing lazily-computed objects in a fully transparent manner:

```
cd pypy
python bin/py.py -o thunk

>>>> from __pypy__ import thunk
>>>> def longcomputation(lst):
....     print "computing..."
....     return sum(lst)
....
>>>> x = thunk(longcomputation, range(5))
>>>> y = thunk(longcomputation, range(10))
```

from the application perspective, `x` and `y` represent exactly the objects being returned by the `longcomputation()` invocations. You can put these objects into a dictionary without triggering the computation:

```
>>>> d = {5: x, 10: y}
>>>> result = d[5]
>>>> result
computing...
10
>>>> type(d[10])
computing...
<type 'int'>
>>>> d[10]
45
```

It is interesting to note that this lazy-computing Python extension is solely implemented in a small objspace/thunk.py file consisting of around 200 lines of code. Since the 0.8.0 release you can translate PyPy with the thunk object space.

3.2.4 Logic programming

People familiar with logic programming languages will be interested to know that PyPy optionally supports logic variables and constraint-based programming. Among the many interesting features of logic programming -- like unification -- this subsumes the thunk object space by providing a more extensive way to deal with laziness.

Try it out:

```
cd pypy
python bin/py.py -o logic

>>>> X = newvar()         # a logic variable
>>>> bind(X, 42)          # give it a value
>>>> assert X / 2 == 21   # then use it
>>>> assert type(X) is int

>>>> X, Y, Z = newvar(), newvar(), newvar()  # three logic vars
>>>> unify({'hello': Y, 'world': Z}, X)      # a complex unification
>>>> bind(Y, 5)                              # then give values to Y
>>>> bind(Z, 7)                              # ... and Z
>>>> X
{'hello': 5, 'world': 7}

>>>> bind(Z, 8)
RuntimeError: Cannot bind twice
```

Read more about Logic Object space features.

3.2.5 Aspect Oriented Programming

PyPy provides some *Aspect Oriented Programming* facilities.

Try it out:

```
cd pypy
python bin/py.py

>>>> from aop import Aspect, before, PointCut
>>>> class DemoAspect:
....     __metaclass__ = Aspect
....     @before(PointCut(func="^open$").call())
....     def before_open(self, joinpoint):
....         print "opening", joinpoint.arguments()
....
>>>> aspect = DemoAspect()
>>>> f = open("/tmp/toto.txt", 'w')
```

To read more about this, try the aop module documentation.

3.2.6 Running the tests

The PyPy project uses test-driven-development. Right now, there are a couple of different categories of tests which you can run. To run all the unit tests:

```
cd pypy
python test_all.py
```

(this is not recommended, since it takes hours and uses huge amounts of RAM). Alternatively, you may run subtests by going to the correct subdirectory and running them individually:

```
python test_all.py interpreter/test/test_pyframe.py
```

`test_all.py` is actually just a synonym for py.test which is our external testing tool. If you have installed that you can as well just issue `py.test DIRECTORY_OR_FILE` in order to perform test runs or simply start it without arguments to run all tests below the current directory.

Finally, there are the CPython regression tests which you can run like this (this will take hours and hours and hours):

```
cd lib-python/2.4.1/test
python ../../../pypy/test_all.py
```

or if you have installed py.test then you simply say:

```
py.test
```

from the lib-python/2.4.1/test directory. You need to have a checkout of the testresult directory. Running one of the above commands tells you how to proceed.

3.2.7 Demos

The demo/ directory contains examples of various aspects of PyPy, ranging from running regular Python programs (that we used as compliance goals) over experimental distribution mechanisms to examples translating sufficiently static programs into low level code.

3.3 Trying out the translator

The translator is a tool based on the PyPy interpreter which can translate sufficiently static Python programs into low-level code. To be able to use it you need to:

- Download and install Pygame if you do not already have it.
- Have an internet connection. The flowgraph viewer connects to codespeak.net and lets it convert the flowgraph by a patched version of Dot Graphviz that does not crash. This is only needed if you want to look at the flowgraphs.
- Use Python-2.4 for using translation tools because python2.5 is (as of revision 39130) not fully supported.

To start the interactive translator shell do:

```
cd pypy
python bin/translatorshell.py
```

Test snippets of translatable code are provided in the file `pypy/translator/test/snippet.py`, which is imported under the name `snippet`. For example:

```
>>> t = Translation(snippet.is_perfect_number)
>>> t.view()
```

After that, the graph viewer pops up, that lets you interactively inspect the flowgraph. To move around, click on something that you want to inspect. To get help about how to use it, press 'H'. To close it again, press 'Q'.

3.3.1 Trying out the type annotator

We have a type annotator that can completely infer types for functions like `is_perfect_number` (as well as for much larger examples):

```
>>> t.annotate([int])
>>> t.view()
```

Move the mouse over variable names (in red) to see their inferred types.

3.3.2 Translating the flow graph to C code

The graph can be turned into C code:

```
>>> t.rtype()
>>> f = t.compile_c()
```

The first command replaces the operations with other low level versions that only use low level types that are available in C (e.g. int). To try out the compiled version:

```
>>> f(5)
False
>>> f(6)
True
```

3.3.3 Translating the flow graph to LLVM code

To translate for LLVM (low level virtual machine) you must first have LLVM installed with version 1.9 - the how to install LLVM provides some helpful hints. Please note that you do not need the CFrontend to compile, make tools-only.

The LLVM backend is still experimental. However, it is very close to C backend functionality. At the point of writing, it is mostly missing stackless and threading support as well as the possibility to use reference counting. Calling compiled LLVM code from CPython is more restrictive than the C backend - the return type and the arguments of the entry function must be ints, floats or bools. The emphasis of the LLVM backend is to compile standalone executables - please see the pypy/translator/llvm/demo directory for examples.

Here is a simple example to try:

```
>>> t = Translation(snippet.my_gcd)
>>> a = t.annotate([int, int])
>>> t.rtype()
>>> f = t.compile_llvm()
>>> f(15, 10)
5
```

3.3.4 Translating the flow graph to Javascript code

The Javascript backend is still experimental but was heavily improved during last years Google summer of code. It contains some rudimentary support for the document object model and a good integration with PyPy's unittesting framework. Code can be tested with the Spidermonkey commandline javascript interpreter in addition to a multitude of javascript capable browsers. The emphasis of the Javascript backend is to compile RPython code into javascript snippets that can be used in a range of browsers. The goal is to make it more and more capable to produce full featured web applications. Please see the pypy/translator/js/test directory for example unittests.

Here is a simple example to try:

```
>>> t = Translation(snippet.my_gcd)
>>> a = t.annotate([int, int])
>>> source = t.source_js()
```

If you want to know more about the JavaScript backend please refer to the JavaScript docs.

3.3.5 Translating the flow graph to CLI code

Use the CLI backend to translate the flowgraphs into .NET executables: `gencli` is quite mature now and can also compile the whole interpreter. You can try out the CLI backend from the interactive translator shell:

```
>>> def myfunc(a, b): return a+b
...
>>> t = Translation(myfunc)
>>> t.annotate([int, int])
>>> f = t.compile_cli()
>>> f(4, 5)
9
```

The object returned by `compile_cli` is a wrapper around the real executable: the parameters are passed as command line arguments, and the returned value is read from the standard output.

Once you have compiled the snippet, you can also try to launch the executable directly from the shell; you can find the executable in one of the `/tmp/usession-*` directories:

```
$ mono /tmp/usession-<username>/main.exe 4 5
9
```

To translate and run for CLI you must have the SDK installed: Windows users need the .NET Frameword SDK 2.0, while Linux and Mac users can use Mono.

3.3.6 A slightly larger example

There is a small-to-medium demo showing the translator and the annotator:

```
cd demo
python bpnn.py
```

This causes `bpnn.py` to display itself as a call graph and class hierarchy. Clicking on functions shows the flow graph of the particular function. Clicking on a class shows the attributes of its instances. All this information (call graph, local variables' types, attributes of instances) is computed by the annotator.

As soon as you close the PyGame window, the function is turned into C code, compiled and executed.

3.4 Translating the PyPy interpreter

(**Note**: for some hints on how to translate PyPy under Windows, see the windows document)

Not for the faint of heart nor the owner of a very old machine: you can translate the whole of PyPy to low level C code. This is the largest and ultimate example of source that our translation toolchain can process:

```
cd pypy/translator/goal
python translate.py --run targetpypystandalone.py
```

By default the translation process will try to use the Boehm-Demers-Weiser garbage collector for the translated PyPy (Use `--gc=framework` to use our own exact mark-n-sweep implementation which at the moment is slower but doesn't have external dependencies). Otherwise, be sure to install Boehm before starting the translation (e.g. by running `apt-get install libgc-dev` on Debian).

This whole process will take some time and quite a lot of memory. To reduce the memory footprint of the translation process you can use the option `--lowmem`:

```
python translate.py --run targetpypystandalone.py --lowmem
```

With this option the whole process should be runnable on a machine with 512Mb of RAM. If the translation is finished running and after you closed the graph you will be greeted (because of `--run` option) by the friendly prompt of a PyPy executable that is not running on top of CPython any more:

```
[translation:info] created: ./pypy-c
[translation:info] Running compiled c source...
debug: entry point starting
debug:  argv -> ./pypy-c
debug: importing code
debug: calling code.interact()
Python 2.4.1 (pypy 0.7.1 build 18929) on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>> 1 + 1
2
>>>>
```

With the default options, you can find the produced executable under the name `pypy-c`. Type `pypy-c --help` to see the options it supports -- mainly the same basic options as CPython. In addition, `pypy-c --info` prints the translation options that where used to produce this particular executable. This executable contains a lot of things that are hard-coded for your particular system (including paths), so it's not really meant to be installed or redistributed at the moment.

If you exit the interpreter you get a pygame window with all the flowgraphs plus a pdb prompt. Moving around in the resulting flow graph is difficult because of the sheer size of the result. For this reason, the debugger prompt you get at the end has been enhanced with commands to facilitate locating functions and classes. Type `help graphs` for a list of the new commands. Help is also available on each of these new commands.

The `translate.py` script itself takes a number of options controlling what to translate and how. See `translate.py -h`. Some of the more interesting options are:

- `--text`: don't show the flowgraph after the translation is done. This is useful if you don't have pygame installed.
- `--stackless`: this produces a pypy-c that includes features inspired by Stackless Python.
- `--gc=boehm|ref|framework|stacklessgc`: choose between using the Boehm-Demers-Weiser garbage collector, our reference counting implementation or our own implementation of a mark and sweep collector, with two different approaches for finding roots (as we have seen Boehm's collector is the default).

Find a more detailed description of the various options in our configuration sections.

You can also use the translate.py script to try out several smaller programs, e.g. a slightly changed version of Pystone:

```
cd pypy/translator/goal
python translate.py targetrpystonedalone
```

This will produce the executable "targetrpystonedalone-c".

3.4.1 Translating with the thunk object space

It is also possible to experimentally translate a PyPy version using the "thunk" object space:

```
cd pypy/translator/goal
python translate.py targetpypystandalone.py --objspace=thunk
```

the examples in lazily computed objects should work in the translated result.

3.4.2 Translating using the LLVM backend

To create a standalone executable using the experimental LLVM compiler infrastructure:

```
./translate.py --text --batch --backend=llvm --raisingop2direct_call target-
pypystandalone.py
```

3.4.3 Translating using the CLI backend

To create a standalone .NET executable using the CLI backend:

```
./translate.py --text --batch --backend=cli targetpypystandalone.py
```

The executable and all its dependecies will be stored in the ./pypy-cli-data directory. To run pypy.NET, you can run ./pypy-cli-data/main.exe. If you are using Linux or Mac, you can use the convenience ./pypy-cli script:

```
$ ./pypy-cli
debug: entry point starting
debug:  argv ->
debug: importing code
debug: calling code.interact()
Python 2.4.1 (pypy 0.9.0 build 38134) on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
 >>>> 1 + 1
 2
 >>>>
```

Unfortunately, at the moment it's not possible to do the full translation using only the tools provided by the Microsoft .NET SDK, since `ilasm` crashes when trying to assemble the pypy-cli code due to its size. Microsoft .NET SDK 2.0.50727.42 is affected by this bug; other version could be affected as well: if you find a version of the SDK that works, please tell us.

Windows users that want to compile their own pypy-cli can install Mono: if a Mono installation is detected the translation toolchain will automatically use its `ilasm2` tool to assemble the executables.

3.4.3.1 Trying the experimental .NET integration

You can also try the still very experimental `clr` module that enables integration with the surrounding .NET environment.

You can dynamically load .NET classes using the `clr.load_cli_class` method. After a class has been loaded, you can instantiate and use it as it were a normal Python class. Special methods such as indexers and properties are supported using the usual Python syntax:

```
>>>> import clr
>>>> ArrayList = clr.load_cli_class('System.Collections', 'ArrayList')
>>>> obj = ArrayList()
>>>> obj.Add(1)
0
>>>> obj.Add(2)
1
>>>> obj.Add("foo")
2
>>>> print obj[0], obj[1], obj[2]
1 2 foo
>>>> print obj.Count
3
```

At the moment the only way to load a .NET class is to explicitly use `clr.load_cli_class`; in the future they will be automatically loaded when accessing .NET namespaces as they were Python modules, as IronPython does.

3.4.4 Translate using the Build Tool

If you don't have access to the required resources to build PyPy yourself, or want to build it on a platform you don't have access to, you can use the 'build tool', a set of components that allow translating and compiling PyPy on remote servers. The build tool provides a meta server that manages a set of connected build servers; clients can request a build by specifying target options and Subversion revision and path, and the meta server will do its best to get the request fullfilled.

The meta server is made available on codespeak.net and there should always be a number of build servers running. If you want to request a compilation, just issue:

```
$ ./bin/startcompile.py [options] <email>
```

in the root of the PyPy package, replacing '[options]' with the options you desire and '<email>' with your email address. Type:

```
$ ./bin/startcompile.py --help
```

to see a list of available options.

To donate build server time (we'd be eternally thankful! ;) just run:

```
$ ./bin/buildserver.py
```

without arguments. Do mind that you need a decent machine with enough RAM (about 2 GB is recommended) and that you need to remove older 'build-*' dirs from /tmp from time to time if you decide to participate.

Note: currently Windows is not supported.

3.5 Where to start reading the sources

PyPy is made from parts that are relatively independent from each other. You should start looking at the part that attracts you most (all paths are relative to the PyPy toplevel directory). You may look at our directory reference or start off at one of the following points:

- pypy/interpreter contains the bytecode interpreter: bytecode dispatcher in pyopcode.py, frame and code objects in eval.py and pyframe.py, function objects and argument passing in function.py and argument.py, the object space interface definition in baseobjspace.py, modules in module.py and mixedmodule.py. Core types supporting the bytecode interpreter are defined in typedef.py.

- pypy/interpreter/pyparser contains a recursive descent parser, and input data files that allow it to parse both Python 2.3 and 2.4 syntax. Once the input data has been processed, the parser can be translated by the above machinery into efficient code.

- pypy/interpreter/astcompiler contains the compiler. This contains a modified version of the compiler package from CPython that fixes some bugs and is translatable. That the compiler and parser are translatable is new in 0.8.0 and it makes using the resulting binary interactively much more pleasant.

- pypy/objspace/std contains the Standard object space. The main file is objspace.py. For each type, the files `xxxtype.py` and `xxxobject.py` contain respectively the definition of the type and its (default) implementation.

- pypy/objspace contains a few other object spaces: the thunk, trace and flow object spaces. The latter is a relatively short piece of code that builds the control flow graphs when the bytecode interpreter runs in it.

- pypy/translator contains the code analysis and generation stuff. Start reading from translator.py, from which it should be easy to follow the pieces of code involved in the various translation phases.
- pypy/annotation contains the data model for the type annotation that can be inferred about a graph. The graph "walker" that uses this is in pypy/annotation/annrpython.py.
- pypy/rpython contains the code of the RPython typer. The typer transforms annotated flow graphs in a way that makes them very similar to C code so that they can be easy translated. The graph transformations are controlled by the stuff in pypy/rpython/rtyper.py. The object model that is used can be found in pypy/rpython/lltypesystem/lltype.py. For each RPython type there is a file rxxxx.py that contains the low level functions needed for this type.

3.6 Additional Tools for running (and hacking) PyPy

We use some optional tools for developing PyPy. They are not required to run the basic tests or to get an interactive PyPy prompt but they help to understand and debug PyPy especially for the ongoing translation work.

3.6.1 graphviz & pygame for flowgraph viewing (highly recommended)

graphviz and pygame are both neccessary if you want to look at generated flowgraphs:

graphviz: http://www.graphviz.org/Download.php

pygame: http://www.pygame.org/download.shtml

3.6.2 CTypes (highly recommended)

ctypes (version 0.9.9.6 or later) is required if you want to translate PyPy to C. See the download page of ctypes.

3.6.3 CLISP

The CLISP backend is optional and not quite uptodate with the rest of PyPy. Still there are a few examples you can try our backend out on. Here is a link to a LISP implementation that should basically work:

http://clisp.cons.org/

3.6.4 py.test and the py lib

The py library is used for supporting PyPy development and running our tests against code and documentation as well as compliance tests. You don't need to install the py library because it ships with PyPy and pypy/test_all.py is an alias for `py.test` but if you want to have the `py.test` tool generally in your path, you might like to visit:

http://codespeak.net/py/dist/download.html

4 Getting involved

PyPy employs an open development process. You are invited to join our pypy-dev mailing list or look at the other contact possibilities. We are also doing coding Sprints which are separatedly announced and often happen around Python conferences such as EuroPython or Pycon. Take a look at the list of upcoming events to plan where to meet with us.

PyPy
Crash Course/Sprint Intro

Michael Hudson, based on Holger's old intro
25 August 2006

Something to look at if I'm too boring

- "getting started" has a lot of good stuff, including where to get the source, links to subversion clients and entry points:

http://codespeak.net/pypy/dist/pypy/doc/getting-started.html

What is PyPy?

- PyPy is:
  - An implementation of Python
  - A very flexible compiler framework
  - An open source project (MIT license)
  - A STREP ("Specific Targeted REsearch Project"), partially funded by the EU
  - A lot of fun!

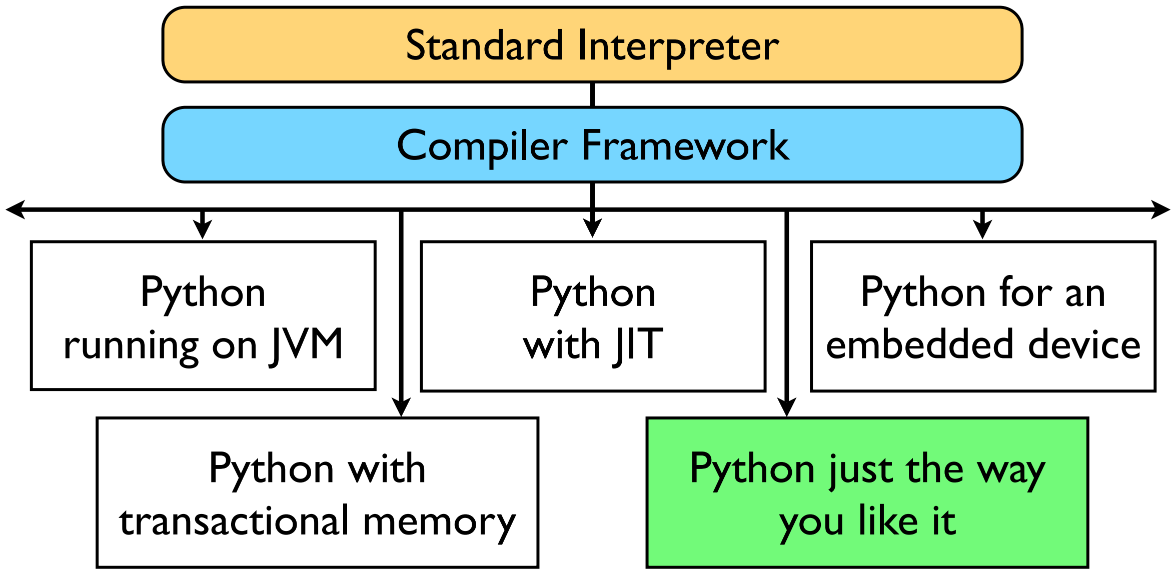
Motivation

- PyPy grew out of a desire to modify/extend the *implementation* of Python, for example to:
  - increase performance (psyco-style JIT compilation, better garbage collectors)
  - add expressiveness (stackless-style coroutines, logic programming)
  - ease porting (to new platforms like the JVM or CLI or to low memory situations)

Lofty goals, but first...

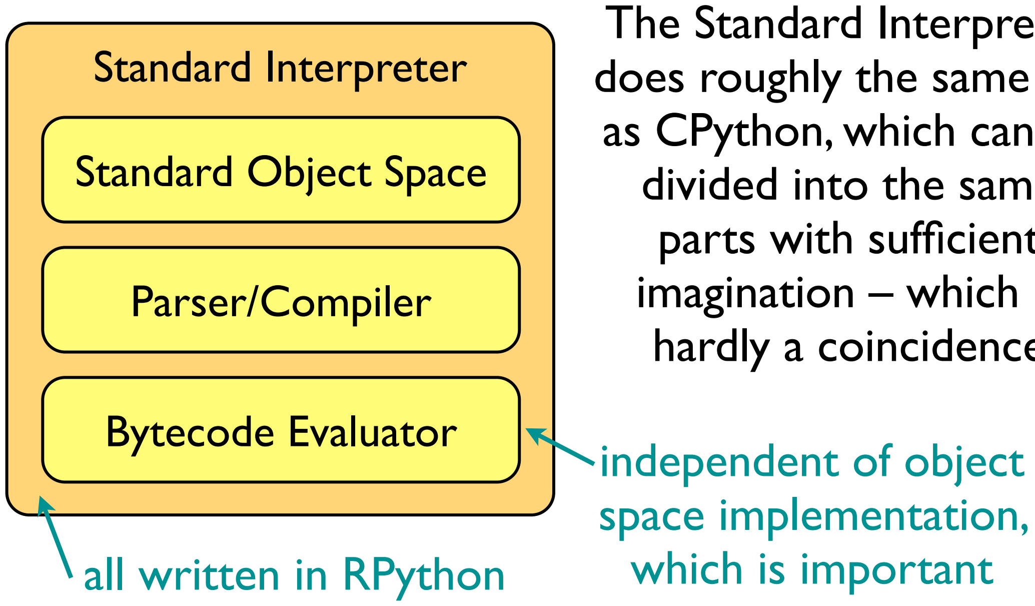
- CPython is a fine implementation of Python but:
  - it's written in C, which makes porting to, for example, the CLI hard
  - while psyco and stackless exist, they are very hard to maintain as Python evolves
  - some implementation decisions are very hard to change (e.g. refcounting)

Enter the PyPy platform



Standard Interpreter
Compiler Framework
Python running on JVM
Python with JIT
Python for an embedded device
Python with transactional memory
Python just the way you like it

Standard Interpreter



The Standard Interpreter does roughly the same job as CPython, which can be divided into the same parts with sufficient imagination – which is hardly a coincidence

The “What is RPython?” question

- Restricted Python, or RPython, first and foremost it *is* Python
- It is a subset of Python that is static enough – *after initialization code has run* – for our analysis tools to cope with
- Somewhat Java-like – classes, methods, no pointers, no operator overloading

The “What is RPython?” question

- The definition of RPython is basically “what our compiler can analyze” – so changes (slowly) as toolchain does
- The property of “being RPython” belongs to entire programs and not, say, functions or modules because the annotator performs a global analysis

Some Jargon

- There are many levels in PyPy
- Two of the more important, defined in terms of running PyPy on top of CPython:
  - “interp-level”: code that will be executed by CPython (and maybe get translated to C)
  - “app-level”: code that will be executed by PyPy’s bytecode evaluator, not CPython’s

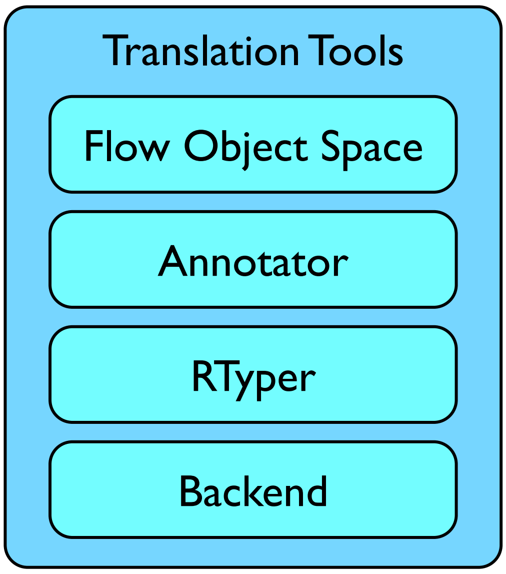
Interp/App-level

- The standard interpreter is written in a mixture of app-level and interp-level code
- Can call from one to the other
- Advantages of app-level: can use full power of Python, less code
- Advantages of interp-level: faster, closer to the metal, can run "early"

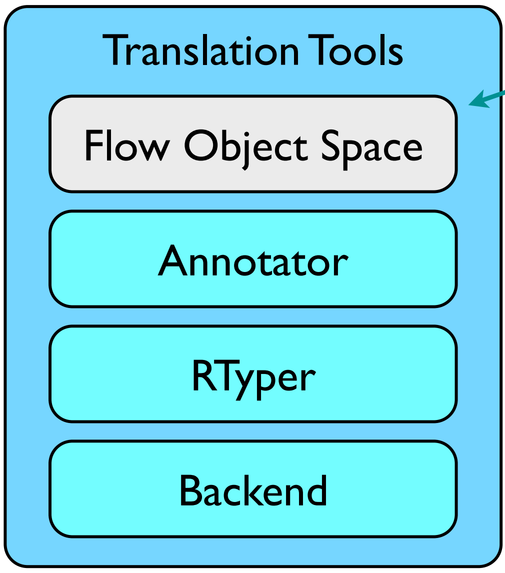
Status

- Standard Interpreter very complete, passes a large majority (>95%) of CPython's core regression tests
- Standard Object Space and bytecode evaluator now very stable
- Only remaining work: make it faster! (not *necessarily* by working on its code, though)

Translation Tools



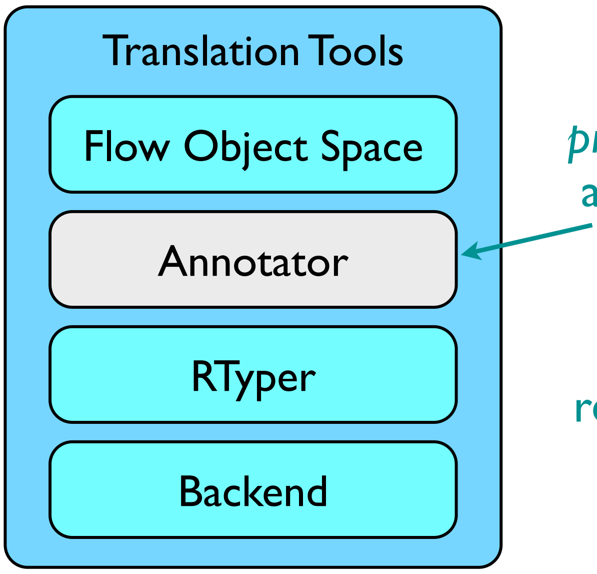
Translation Tools



Analyzes a single code object to deduce control flow

We have a funky pygame flow graph viewer that we use to view these flow graphs (demo)

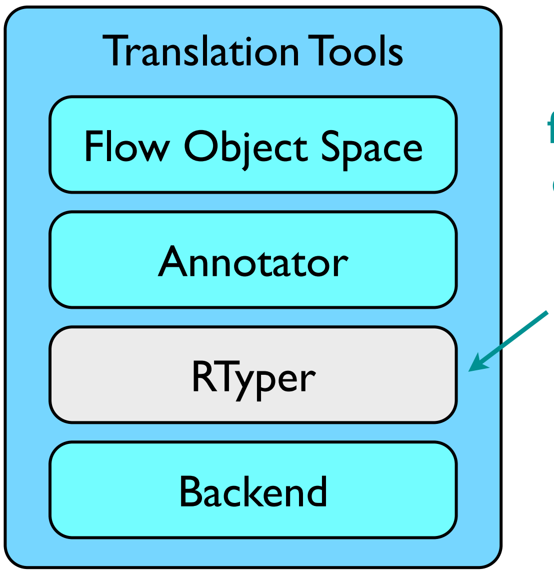
Translation Tools



Analyzes an *entire program* to deduce type and other information

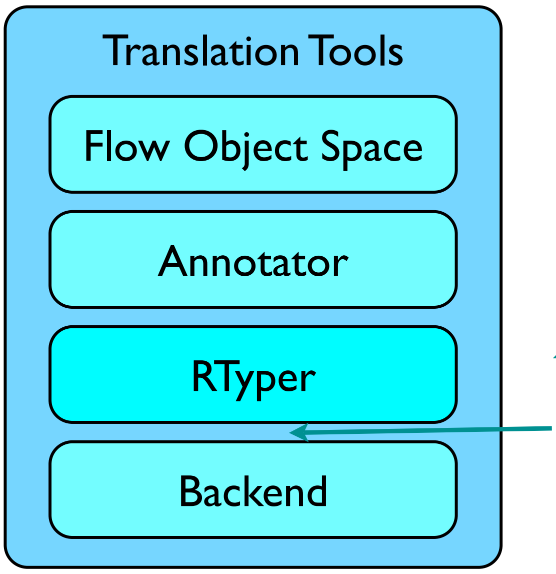
Uses abstract interpretation, rescheduling and other funky stuff

Translation Tools



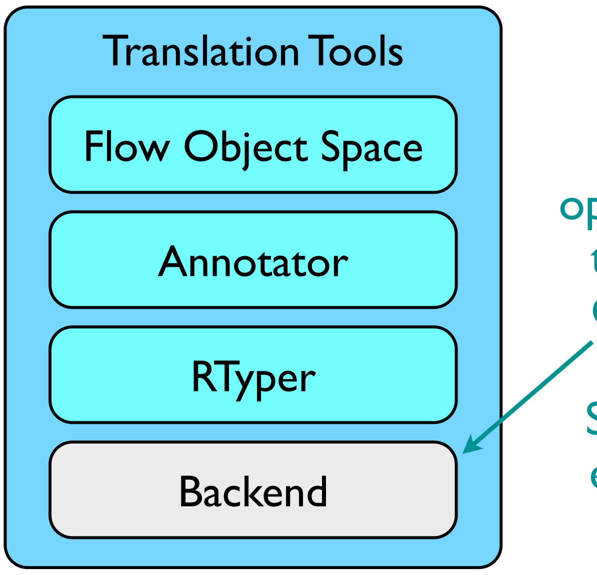
Uses the information found by the annotator to decide how to lay out the types used by the input program in memory, and translates high level operations to lower level more pointer-ish operations

Translation Tools



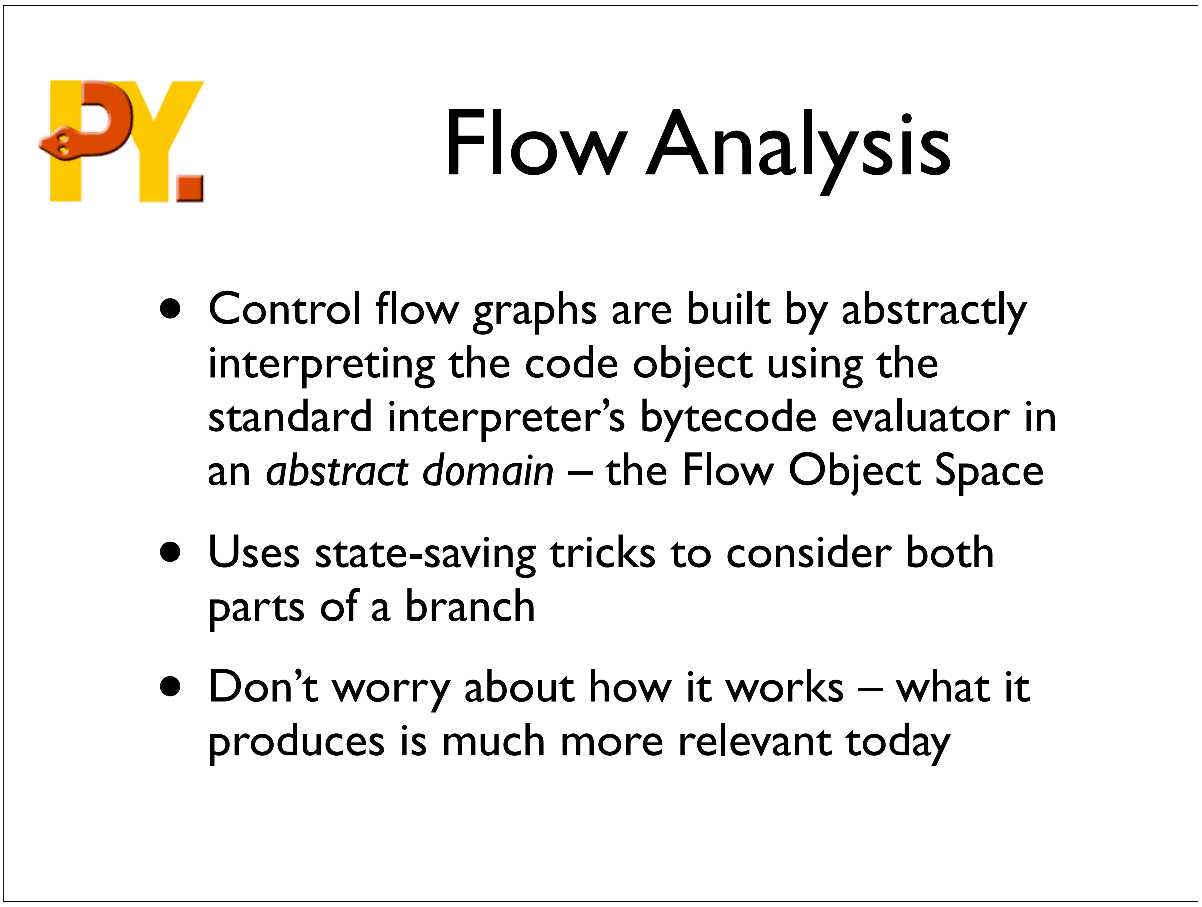
"Stuff" happens here – optional optimizations, the "stackless transform", insertion of explicit exception handling and memory management/GC code

Translation Tools



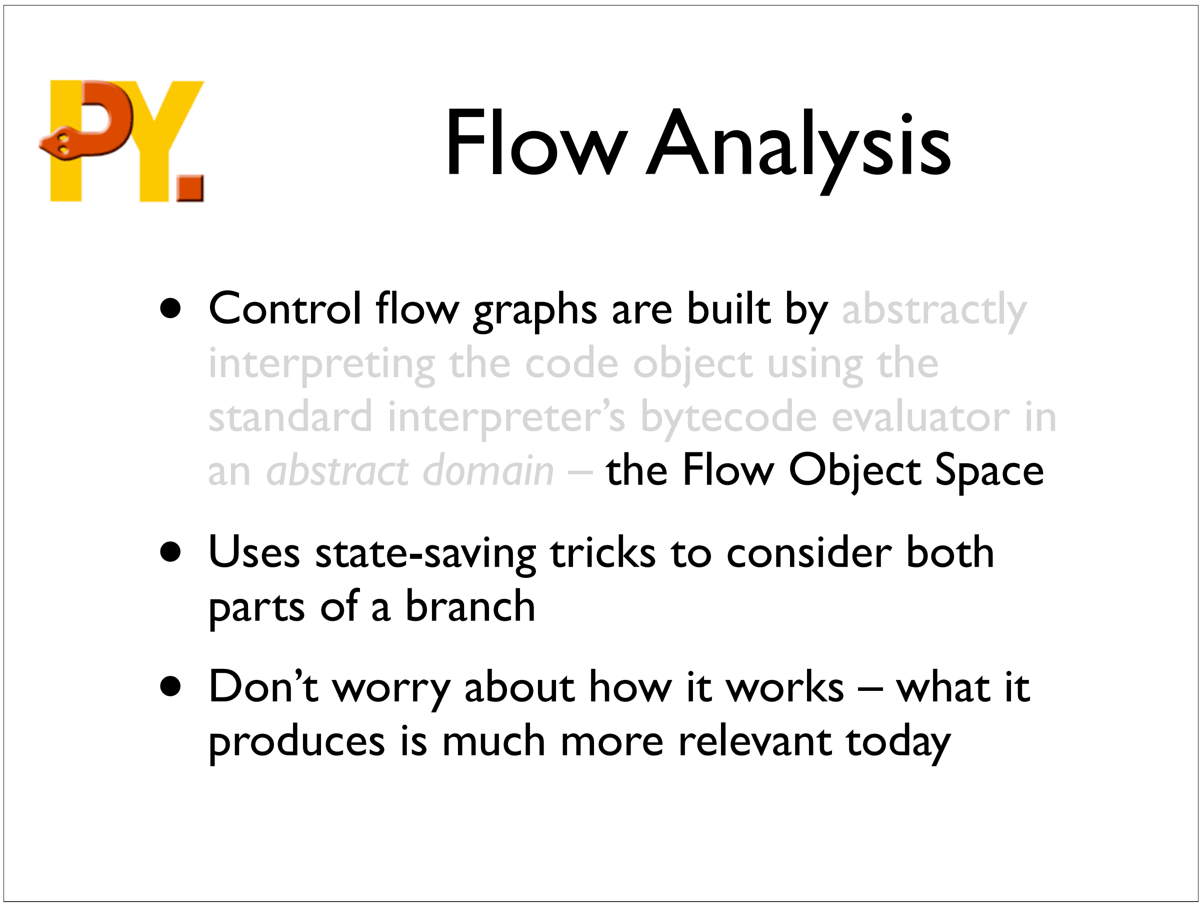
Translates low level operations and types from the RTyper to C, LLVM, CLI, JavaScript, … code

Sounds like it should be easy, in fact a bit painful



Flow Analysis

- Control flow graphs are built by abstractly interpreting the code object using the standard interpreter’s bytecode evaluator in an *abstract domain* – the Flow Object Space
- Uses state-saving tricks to consider both parts of a branch
- Don’t worry about how it works – what it produces is much more relevant today



Flow Analysis

- Control flow graphs are built by abstractly interpreting the code object using the standard interpreter’s bytecode evaluator in an *abstract domain* – the Flow Object Space
- Uses state-saving tricks to consider both parts of a branch
- Don’t worry about how it works – what it produces is much more relevant today

The Flow Model

- All defined in `pypy.objspace.flow.model`
- Values are either `Variable`s or `Constant`s
- A function's control flow graph is described by a `FunctionGraph`
- This contains `Block`s and `Link`s
- Blocks contain a list of `SpaceOperation`s

The Flow Model

- `SpaceOperation`s have an `opname`, a `result` variable and a list of `args`.
- The graph is naturally produced in *Static Single Information* form, which is very handy for later analysis
- This means each `Variable` is used in exactly one block and is renamed if it needs to be passed across a link

The Flow Model

- Some examples:
  - `z=x.y` → `SpaceOperation("getattr", [v_x, Constant("y")], v_z)`
  - `c=a+b` → `SpaceOperation("add", [v_a, v_b], v_c)`
  - `t=f(u)` → `SpaceOperation("simple_call", [Constant(f), v_u], v_t)`

```
z=x.y  → SpaceOperation("getattr",
                 [v_x, Constant("y")], v_z)
c=a+b  → SpaceOperation("add",
                 [v_a, v_b], v_c)
t=f(u) → SpaceOperation("simple_call",
                 [Constant(f), v_u], v_t)
```

The Annotator

- Type annotation is a fairly widely known concept – it associates variables with information about which values they might take at run time
- An unusual feature of PyPy's approach is that the annotator works on live objects
- This means it never sees initialization code, so that can use `exec` and other insane tricks

The Annotator

- Works by abstractly interpreting (a popular phrase :) the control flow graphs produced by the flow analysis
- Annotation starts at an entry point and discovers as it proceeds which functions are needed
- Read "Compiling dynamic language implementations" on the web site for more than is on these slides

The Annotator

- Works a block at a time, maintaining a pile of blocks that need analysis
- As analysis proceeds, the information about a block may get invalidated – in other words, the annotation reschedules as needed
- A fix-point approach:

```
while work_to_do: do_work()
```

The Annotation Model

- Does not modify the graphs; end result is essentially a big dictionary mapping `Variable`s to instances of a subclass of `pypy.annotation.model.SomeObject`.
- Important subclasses are `SomeInteger`, `SomeList`, `SomeInstance`, `SomePBC` (“some pre-built constant”, includes classes and functions)

The RTyper

- RTyper takes as input an annotated RPython program (e.g. our Python implementation)
- Performs “representation selection” and converts high-level operations to low-level
- Can target a C-ish language or an object oriented platform such as .NET/CLI or Squeak (for real) or the JVM (would be nice)

The RTyper

- Originally we tried to do the job the RTyper does at the same time as source generation
- Failed
- Miserably
- It does a job that's not part of the standard "Introduction to Compilers 101" course

Representation Selection

- The fact that the annotator performs a global analysis gives us a novel opportunity
- For example, in:
  ```
  l = range(10)
  for x in l: print l
  ```
  can represent the return value of range as just start/stop/step, but if we know the return value of range() is going to be mutated we just return a normal list

lltypes

- `pypy.rpython.lltypesystem.lltype` contains a collection of Python classes that describe (and implement!) a C-like memory model with `Struct`s, `Array`s, `Pointer`s, `Signed`s (integers), `GcStruct`s, `Float`s... all subclasses of `LowLevelType`
- Convention is that variables holding instances of lltypes are in ALLCAPS:

```
    TYPE = GcStruct(“T”, (“x”, Signed))
```

Representation Selection

- The RTyper attaches an attribute “`concretetype`” containing an lltype to all `Constant`s and `Variable`s
- During the process of RTyping, however, an instance of `pypy.rpython.rmodel.Repr` is created and associated with each `Variable`’s annotation, which knows how to translate operations involving the `Variable`

Translating High Level to Low Level

- The high level operations such as "`add`" apply to different types; you can add strings, floats or integers and continually having to distinguish is annoying
- Better to have monomorphic operations `int_add`, `float_add`, `str_add` (well...)
- Some operations are more complex, e.g. instantiation of a class

Translating High Level to Low Level

- For each operation:
  - an instance of a subclass of `Repr` is created/found for each argument's annotation
  - and these are asked what low-level operation(s) the high level operation should be replaced with

Translating High Level to Low Level, example

- Start with, say, `SpaceOperation(“add”, [v_x, v_y], v_z)` with `v_x` and `v_y` (and `v_z`) all annotated as `SomeInteger()`
- `rtyper.getrepr(v_x)` returns an instance of `IntegerRepr` (same for `v_y`)
- We end up calling a method `rtype_add` on a “`pairtype`” which makes a “`int_add`” operation

Source Generation

- Maintained backends: C, LLVM, JavaScript, .NET/CLI, Squeak, Common Lisp (in roughly descending order of maintainedness)
- All proceed in two phases:
  - Traverse the forest of rtyped graphs, computing names for everything
  - Spit out the code

Dealing with the world

- One of Python's strengths is the ease with which external C libraries can be wrapped
- Existing C extensions cannot work with PyPy
- But don't need C any more for this job – ctypes!

rctypes

- rctypes is our name for the code which allows RPython programs to use ctypes (in a sufficiently static way)
- Example:

```
time_t = ctypes_platform.getsimpletype(
   'time_t', '#include <time.h>', c_long)
time = libc.time
time.argtypes = [POINTER(time_t)]
time.restype = time_t
```

The Extension Compiler

- In addition to using this to wrap external libraries for PyPy, it can be used to do the same for CPython
- This is `bin/compilemodule`
- It’s not yet “the nicest thing in the world”

Things I haven’t talked about

- Garbage collection policies
- Stackless
- Backend optimizations
- The JIT
- Exceptions in the flow model
- Specialization of functions/annotation policies
- Low level helpers
- Constraint solving
- Geninterp
- Pairtypes
- Multimethods in the Standard Object Space

Coding Issues

- See `http://codespeak.net/pypy/dist/pypy/doc/coding-guide.html`
- But in summary: PEP 8, test driven development (using py.test)
- Away from sprints, much talking in `#pypy`

Thank you for your time!

Subversion awaits!