The Ultimate Psyco Guide

Release 1.6

Armin Rigo

updated December 3rd, 2007

Email: arigo@tunes.org
Psyco Home page: http://psyco.sourceforge.net/

Abstract

Psyco is a Python extension module which can massively speed up the execution of any Python code.

CONTENTS

CHAPTER

ONE

Installation

1.1 Requirements

Psyco requires:

- **A 32-bit architecture.** A Pentium or any other Intel 386 compatible processor is recommended.
- **Linux, Mac OS/X, Windows, BSD** are known to work.
- **A regular Python installation, version 2.2.2 or up.** Psyco is not a replacement for the Python interpreter and libraries, it works on top of them.

Psyco is not strongly dependent on a particular OS. Precompiled binaries for Windows are contributed by users. Compiling on Linux, Mac OS/X and other Unixes should be easy.

Psyco has an emulation mode that allows it to run on other 32-bit processors but the speed benefits are not big in that case. Psyco does not support the 64-bit x86 architecture, unless you have a Python compiled in 32-bit compatibility mode. **There are no plans to port Psyco to 64-bit architectures.** This would be rather involved. Psyco is only being maintained, not further developed. The development efforts of the author are now focused on PyPy, which includes Psyco-like techniques. http://codespeak.net/pypy/

1.2 Installing

Binaries are now only available for Windows as installers for recent Python versions. You need to compile from the source (see 1.3) in all other cases (it is trivial on non-Windows platforms, though).

Be sure to download the version of Psyco that corresponds to your Python version. Binary releases are at http://sourceforge.net/project/showfiles.php?group_id=41036. Only some pre-compiled versions are available (contributions for missing ones are welcome).

File name	Python versions	Well-tested with
psyco-x.y-win32-py2.2.2.exe	2.2.2 and up	2.2.2 and 2.2.3
psyco-x.y-win32-py2.3.exe	2.3 and up	2.3 and 2.3.3
psyco-x.y-win32-py2.4.exe	2.4 and up	2.4.*
psyco-x.y-win32-py2.5.exe	2.5 and up	2.4

Note that the 2.2.2 version will **not** work on the 2.2 or 2.2.1 Pythons!

1.3 Installing from the sources

You should get the source code from its most current Subversion version at http://codespeak.net/svn/psyco/dist/. The command is:

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

People unfamiliar with Subversion or whose Subversion access is firewalled can use a web grabber (e.g. “wget -r http://codespeak.net/svn/psyco/dist/ -I /svn/psyco/dist“), or download a snapshop at http://wyvern.cs.uni-duesseldorf.de/psyco/psyco-snapshot.tar.gz.

Note that the Subversion tree is considered the latest official release. It evolves slowly and it is more stable than the packaged version. But if you really really prefer official releases (though there is not much point, Psyco being stable and slow-changing nowadays), have a look at http://sourceforge.net/project/showfiles.php?group_id=41036.

To install from the source, run the top-level installation script ‘setup.py’:

```
python setup.py install
```

Warning, many Linux distributions (e.g. Debian) ship with an incomplete Python. You need to install the “python-dev” package for this to work.

As usual, other commands are available, e.g.

```
python setup.py build_ext -i
```

will compile the C source and put the result directly into the ‘py-support/’ subdirectory (no administrator priviledge is required). After this step, the ‘py-support/’ directory is a complete package (exactly as found in the binary distributions) that you can rename to ‘psyco’ and copy around.

1.4 Compiling Psyco in Debug mode

In case of trouble that you suspect to be a bug in Psyco, you can recompile it with additional coherency checks that will be done during the compilation of your Python functions (they should not slow down the execution of the compiled code).

Additionally, Psyco will dump debugging information to the standard error. The debugging version of Psyco can also dump the whole compiled code buffers and associated data structures, using `psyco.dumpcodebuf`.

Follow the instructions in ‘setup.py’ to enable debugging. For fine-grained control under Unix, I use a custom ‘Makefile’ instead of ‘setup.py’; this ‘Makefile’ is present in the CVS tree too.

CHAPTER

TWO

Tutorial

The goal of Psyco is to be extremely easy to use, and ideally it should be completely transparent. If this were the case, this user guide would be empty. This is not the case because it is still experimental, and because you will probably like to keep control over what you trust Psyco to accelerate and how much resources (mainly memory) you are ready to let it spend.

2.1 Quick examples

Psyco comes as a package module that you can use in your applications, althought you will seldom need to use functions that are not exported via the top-level `psyco` module. Try adding the following two lines at the beginning of your main '.py' source file:

```
import psyco
psyco.full()
```

This instructs Psyco to compile and run as much of your application code as possible. This is the simplest interface to Psyco. In good cases you can just add these two lines and enjoy the speed-up. If your application does a lot of initialization stuff before the real work begins, you can put the above two lines after this initialization – e.g. after importing modules, creating constant global objects, etc. Psyco usually doesn't speed up the initialization of a program; it rather tends to make start-up slower. A good place to put these lines is at the beginning of the `__main__` part of the script:

```
if __name__ == '__main__':
    # Import Psyco if available
    try:
        import psyco
        psyco.full()
    except ImportError:
        pass
    # ...your code here...
```

For larger applications, try:

```
import psyco
psyco.profile()
```

Psyco will do some profiling behind the scene to discover which functions are worth being compiled.

```
import psyco
psyco.log()
psyco.profile()
```

Enable logging to a file named 'xxx.log-psyco' by default, where 'xxx' is the name of the script you ran.

```
import psyco
psyco.log()
psyco.profile(0.2)
```

The extra argument `0.2` to `profile` is called the watermark; it means that Psyco will compile all functions that take at least 20% of the time. The exact meaning of the watermark will be given in the reference manual; simply keep in mind that smaller numbers (towards `0.0`) mean more functions compiled, and larger numbers (towards `1.0`) mean less functions compiled. The default watermark is `0.09`.

```
import psyco
psyco.log()
psyco.full(memory=100)
psyco.profile(0.05, memory=100)
psyco.profile(0.2)
```

This example combines the previous ones. All functions are compiled until it takes 100 kilobytes of memory. Aggressive profiling (watermark `0.05`) is then used to select further functions until it takes an extra 100 kilobytes of memory. Then less aggressive profiling (watermark `0.2`) is used for the rest of the execution. **Note:** The limit of 100 kilobytes is largely underestimated by Psyco: your process' size will grow by much more than that before Psyco switches to the next profiler.

If you want to keep control over what gets compiled, you can select individual functions:

```
import psyco
psyco.bind(myfunction1)
psyco.bind(myfunction2)
```

This only selects the two given functions for compilation. Typically, you would select functions that do CPU-intensive computations, like walking lists or working on strings or numbers. Note that the functions called by the functions you select will also be compiled.

```
import psyco
g = psyco.proxy(f)
g(args)            # Psyco-accelerated call
f(args)            # regular slow call
```

Unlike `psyco.bind`, `psyco.proxy` does not affect your original function at all, but only creates a Psyco-accelerated copy of it (a "proxy"). Useful if you only want to use Psyco at a very specific moment. One can also argue that `f=psyco.proxy(f)` looks more Pythonic than `psyco.bind(f)`.

2.2 Zope's TAL: a real example with benchmarks

As an example of a good real-life candidate for Psyco acceleration, I chose **TAL**, the simpler of the two templating languages of Zope (http://www.zope.org). TAL is an extension of HTML in which you can write templates which are used to generate real HTML pages dynamically. The translation from templates to real pages is a typical data-manipulation algorithm that is much faster in C than in Python, but much more obfuscated too. TAL is currently implemented in Python only.

For these tests, we will use the file 'TAL.markbench.py' of a Zope 2 installation (if you did not install Zope, the file is in the 'lib/python' subdirectory of the source archive). You may have to create a file named '.path' to be able to run 'markbench.py', as explained when you first try to run it.

On a Dual Pentium Linux running Python 2.2.2, using the Zope 2.6.1 distribution, I get the following times (lower numbers mean faster runs; I have added the horizontal line at test 9 for emphasis):

```
##:        ZPT        TAL       DTML
01:          0          0          0
02:          0          0          0
03:          2          1          0
04:          9          6          3
05:         16         12          4
06:         19         13         10
07:         15         10          1
08:         10          7          1
09: ------ 123 ------- 90 ------- 32
10:         14          6          3
11:         28         18         10
12:         20         15          6
```

If I add the following line at the top of 'markbench.py' the times drop massively — the longest-running tests run two to three times faster!

```
import psyco; psyco.log(); psyco.full()
```

Results:

```
##:        ZPT        TAL       DTML
01:          0          0          0
02:          0          0          0
03:          1          0          0
04:          5          2          2
05:          8          4          4
06:         11          5         10
07:          7          4          1
08:          6          2          1
09: ------- 61 ------- 34 ------- 32
10:         10          3          2
11:         17          7         10
12:         11          6          6
```

Here you see in the DTML test the first rule about Psyco: some code is obviously not seen by Psyco at all. In other words for some reason this code runs in the normal interpreter only. There are a lot of potential reasons for this, which range from Psyco failing to "steal" the code from the Python interpreter before it executes it to code using unsupported constructs (see appendix A.3).

```
import psyco; psyco.log(); psyco.profile()
```

Results:

```
##:         ZPT         TAL        DTML
01:           0           0           0
02:           0           0           0
03:           4           3           0
04:          15          11           3
05:          20          13           5
06:          16           5          12
07:           7           5           1
08:           5           2           1
09: ------- 65 ------- 35 ------- 40
10:          11           3           2
11:          18           8          10
12:          12           6           6
```

If you use the profiler instead of compiling everything, the small tests won't run much faster, but the long-running ones will be about as fast as above. It means that the profiler is a powerful tool that can be useful on arbitrarily large programs. However, in the case of a large program in which you know where most of the processor time is spent, you can choose to selectively compile this part:

```
from TALInterpreter import TALInterpreter
import psyco; psyco.bind(TALInterpreter)
```

Results:

```
##:         ZPT         TAL        DTML
01:           0           0           0
02:           0           0           0
03:           2           0           0
04:           9           2           2
05:          16           4           4
06:          19           5          10
07:          14           4           1
08:          10           3           1
09: ------ 122 ------- 33 ------- 32
10:          13           3           3
11:          28           8          10
12:          20           6           6
```

Here you can see that only the TAL test is accelerated. If you have a Zope server using TAL, you could try to add the above two lines to it. You may get very interesting speed-ups for a very reasonable memory overweight! (untested: please tell me about it — there is no real Zope server I can play with myself.)

Another ceveat of Psyco is that it is not always easy to know which module or class Psyco has just acclerated in `psyco.full()` mode. It took me some time to discover what I should bind to accelerate specifically the ZPT test. Here is how: I ran Psyco in `psyco.log();psyco.profile()` mode, running only the ZPT test, and inspected the log file. Indeed, when a function is about to be accelerated, Psyco writes a line '`tag function: xxx`'. In this case I figured out from the 'markbench.log-psyco' file that I had to write this:

```
import TAL.TALInterpreter
import psyco; psyco.bind(TAL.TALInterpreter.TALInterpreter.interpret)
```

Results:

```
##:        ZPT        TAL       DTML
01:          0          0          0
02:          0          0          0
03:          1          2          0
04:          5          6          2
05:          9         11          4
06:         11         13         10
07:          8         10          1
08:          6          7          1
09: ------- 66 ------- 90 ------- 32
10:         10          6          4
11:         18         18         10
12:         12         15          6
```

2.3 Old-style classes vs. Built-in types

Althought Psyco can plug itself over various versions of the interpreter, there are some features that depend on specific extensions.

A major introduction of Python 2.2 are the so-called new-style classes, which have a (positive) performance impact on Psyco. So you should, as much as possible, let your classes be built-in types and not old-style classes (which is done by inheriting from another new-style class, or from `object` for root classes).

In addition, Psyco 1.4 provides a compact alternative to Python's new-style instances: the `psyco.compact` root class. Currently, you have to use or inherit from `psyco.compact` explicitely. This lets Psyco produce faster code and reduces the memory used by your instances. Even better, if you add the line

```
from psyco.classes import *
```

at the top of the module(s) that define the classes, then not only will all your classes automatically inherit from `psyco.compact`, but all the methods defined in your classes will automatically be compiled as if you had called `psyco.bind()` on them.

Warning: In Python, instances of classes that inherit from a built-in type are subject to some semantic differences that you should know about. These are described in http://www.python.org/2.2.2/descrintro.html. An immediate difference is that if *x* contains an instance of such a new-style class, then `type(x)` will be `x.__class__`- instead of `types.InstanceType`.

For more information see sections 3.4 and 3.5.

2.4 Known bugs

Almost any Python code should execute correctly when run under Psyco, including code that uses arbitrary C extension modules. Some features of Python which are marked as "not supported" by Psyco will not cause Psyco to fail to produce the correct result; instead, it will revert to the regular Python interpreter and the code will run non-accelerated.

All exceptions to this rule are bugs, but there are a couple of them because they would be either very hard to fix or fixing them would severely hurt performance. They correspond to uncommon situations. The current list is in appendix A.1.

There are also performance bugs: situations in which Psyco slows down the code instead of accelerating it. It is difficult to make a complete list of the possible reasons, but here are a few common ones:

- The built-in `map` and `filter` functions must be avoided and replaced by list comprehension. For example, `map(lambda x: x*x, lst)` should be replaced by the more readable but more recent syntax `[x*x` `for x in lst]`.

- The compilation of regular expressions doesn't seem to benefit from Psyco. (The execution of regular expressions is unaffected, since it is C code.) Don't enable Psyco on this module; if necessary, disable it explicitely, e.g. by calling `psyco.cannotcompile(re.compile)`.

CHAPTER

THREE

User Reference Guide

This chapter describes in detail what you will find in the `psyco` package module.

The various functions described here are all concerned with how you tell Psyco what it should compile and what it should leave to the standard interpreter to execute. Compiling everything is often overkill for medium- or large-sized applications. The drawbacks of compiling too much are in the time spent compiling, plus the amount of memory that this process consumes. It is a subtle balance to keep.

3.1 Selecting the functions to compile

With the following functions you can explicitely control what parts of your application get compiled.

`bind`(*object, rec=10*)

Tells Psyco that the given *object* should be compiled for faster execution.

If *object* is a Python function, `bind` modifies the function object in-place (specifically, its `func_code` attribute is changed). All future calls to the function will go through Psyco. Remember that in Python no behavior can depend on which reference to the function you use to make the call; in other words, after

```
g = f
bind(f)
```

calls to both *f* and *g* will go through Psyco.

The *object* argument can also be:

- a method, either bound or unbound, implemented in Python; it underlying implementation is bound.
- a class (or anything with a `__dict__` attribute); in this case, all functions and methods found in `c.__dict__` are bound (which means, for classes, all methods defined in the class but not in its parents).

Note that compilation never actually starts before a function is first called.

All functions called by a compiled function are also compiled, as well as the functions called by these, and so on, up to some limit (by default 10) that you can specify as the optional second argument.

`proxy`(*function-or-method, rec=10*)

Makes a Psyco-enabled copy of *object*.

This function returns a new object, which is a copy of the *function-or-method* with the same behavior but going through Psyco. The function or method must be implemented in Python. The argument itself is not affected; if you call it, it will still be run by the regular interpreter.

Thus, '`bind(f)`' is similar to '`f=proxy(f)`', except that any reference to *f* that might previously have been taken (for example by a '`from spam import *`' statement) are unaffected in the second case.

`unbind`(*object*)

Cancels the action of `bind`.

Note that Psyco currently cannot release the memory occupied by the compilation of a function.

unproxy (*proxy-object*)

Reverse of `proxy`.

`unproxy(proxy(f))` is a new function object equal to *f*. Calling it does no longer invoke Psyco.

cannotcompile (*object*)

Prevents the given function to be ever compiled by Psyco. *object* can also be a method object or directly a code object. Raises `psyco.error` if the object is already being compiled. Note that this does not prevent functions called by *object* to be compiled, if they are bound or proxied or if the profiler chooses to compile them.

setfilter (*func or None*)

Sets a global filter function: "func(co)" will be called once per code object *co* considered by Psyco. If it returns *False*, the code object will not be compiled. Note that "func(co)" will not be called if Psyco already started to compile the code object before you called `setfilter`, nor if the code object contains constructs that prevents Psyco from compiling it anyway. The return value of `setfilter` is the previous filter function or *None*.

The same function will never be compiled over and over again, so that you can freely mix calls to `bind`, `proxy`, `unbind` and `unproxy` and to the profile-based routines described in the next section.

3.2 Profile-based compilation

This section lists the functions that use the Python profiler and line tracer to discover which functions it should compile. These routines are best to use if your application doesn't have a clear "slow spot" where a lot of time is spent algorithmically. If it does, the previous section is all you need.

full (*memory=None, time=None, memorymax=None, timemax=None*)

Compile as much as possible.

Very efficient for small scripts and applications. Might be overkill for large applications: the memory blow-up and the time spent compiling a large amount of code could outweight the benefits.

The (optional) arguments are described in the next sections.

profile (*watermark=0.09, halflife=0.5, pollfreq=20, parentframe=0.25, memory=None, time=None, memorymax=None, timemax=None*)

Do profiling to detect the functions that consume a lot of interpretation time, and only compile those.

As far as I can tell, the best solution for large applications. If a function takes a long time to run, it will be detected and compiled *while it is still running,* in the middle of its execution. Psyco takes it away from the standard interpreter.

The (optional) arguments are used for fine-tuning. They are described in the next sections.

background (*watermark=0.09, halflife=0.5, pollfreq=100, parentframe=0.25, memory=None, time=None, memorymax=None, timemax=None*)

This is similar to `psyco.profile`, but does not use active profiling. It only starts a background thread that periodically checks where the execution point is in all the other threads, and collects statistics based on this.

While quite less accurate, this method has the advantage of adding only a very small overhead to the standard execution and might be suited to large applications with an occasional algorithmically-intensive function.

The (optional) arguments are used for fine-tuning. They are described in the next sections. Try putting a '`psyco.background()`' call in your 'site.py' file, with a larger *watermark*. Psyco will stay in the shadow and you should not notice it at all until you do some heavy work in Python.

runonly (*memory=None, time=None, memorymax=None, timemax=None*)

Run the compiled functions but don't compile new ones.

The usefulness of this routine is unknown. See below.

stop ()

Stop the current profiler and purge the profiler queue (described below).

Calling this function prevents Psyco from compiling any new function. It will still execute the ones that have already been compiled.

Note: most comments on the performance are based on small tests and extrapolations. Just don't trust what I guess and try yourself. Any comment about which routine you found most useful or what value you often give to parameters are welcome!

3.2.1 Charge profilers

The "profilers" invoked by the above functions `profile` or `background` work on the following principles. During the regular execution of your application, Psyco measures the time spent by your functions. Individual functions are selected for compilation based on these data.

Python 2.2.2 (and up) maintains a counter of executed bytecode instructions; this number is the most accurate (well, the least inaccurate) method I found out to guess how much a function could benefit from being run by Psyco. This number counts the number of bytecode instructions (or elementary steps) taken during the interpretation, which gives a "time" estimate. This "time" does not include things like time spent waiting for data to be read or written to a file, for example, which is a good thing because Psyco cannot optimize this anyway.

The "time" thus charged to various functions is accumulated, until some limit is reached. To favour functions that have recently be seen running over old-timers, the charge of each function "decays" with time, following an exponential law as if the amount of time was stored as a dissipating electric charge. The limit at which a function is compiled is given as a fraction of the total charge; in other words, when a function's charge reaches at least `xx` percents of the total current charge, it is compiled. As all the charges decay with time, reaching such a limit is easier than it may seem; for example, if the limit is set at 10%, and if the execution stays for several seconds within the same 10 functions, then at least one of them will eventually reach the limit. In practice, it seems to be a very good way to measure the charge; a few CPU-intensive functions will very quickly reach the limit, even if the program has already been running for a long time.

The functions `profile` and `background` take the following optional keyword arguments:

watermark The limit, as a fraction of the total charge, between `0.0` and `1.0`. The default value is `0.09`, or 9%.

halflife The time (in seconds) it takes for the charge to drop to half its initial value by decay. After two half-lifes, the charge is one-fourth of the initial value; after three, one-eighth; and so on. The default value is `0.5` seconds.

pollfreq How many times per second statistics must be collected. This parameter is central to `background`'s operation. The default value is `100`. This is a maximum for a number of operating systems, whose `sleep` function is quite limited in resolution.

pollfreq also applies to `profile`, whose main task is to do active profiling, but which collects statistics in the background too so that it can discover long-running functions that never call other functions. The default value in this case is `20`.

parentframe When a function is charged, its parent (the function that called it) is also charged a bit. The purpose is to discover functions that call several other functions, each of which does a part of the job, but none of which does enough to be individually identified and compiled. When a function is charged some amount of time, its the parent is additionally charged a fraction of it. The parent of the parent is charged too, and so on recursively, althought the effect is more and more imperceptible. This parameter controls the fraction. The default value of `0.25` means that the parent is charged one-fourth of what a child is charged. **Note:** Do not use values larger than `0.5`.

Note: All default values need tuning. Any comments about values that seem to give better results in common cases are welcome.

Profiling data is not stored on disk (currently). Profiling starts anew each time you restart your application. This is consistent with the way the "electric charges" approach works: any charge older than a few half-lifes gets very close to zero anyway. Moreover, in the current implementation, a full reset of the charges occurs every 120 half-lifes, but the effect should go unnoticed.

3.2.2 Limiting memory usage

Psyco can and will use large amounts of memory. To give you more control, the effect of the various "profilers" selected by the functions `full`, `profile` and `background` can be limited.

Here are the corresponding optional keyword arguments.

memory Stop when the profiler has caused the given amount of memory to be consumed in code compilation (in kilobytes).

Note that the value is very approximative because it only takes some of the data structures of Psyco in account. The actual amount of memory consumed by Psyco will be much larger than that.

time Only run the profiler for the given amount of seconds.

memorymax Stop when the memory consumed by Psyco reached the limit (in kilobytes). This limit includes the memory consumed before this profiler started.

timemax Stop after the given number of seconds has elapsed, counting from now.

The difference with *time* is only apparent when the profiler does not actually start immediately, if it is queued as explained in the next section.

3.2.3 The profilers queue

Profilers can be chained. By calling several of the above profiling functions (or maybe the same function several times with different arguments), only the first one is started. The second one will only be started when the first one reaches one of its limits, and so on. For example:

```
import psyco
psyco.full(memory=100)
psyco.profile()
```

causes Psyco to compile everything until it takes 100 kilobytes of memory, and then only compile based on collected statistics. Remember that the limit of 100 kilobytes is largely underestimated by Psyco: your process' size will grow by much more than that before Psyco switches to profiling.

If all profilers reach their limits, profiling is stopped. Already-compiled functions will continue to run faster.

You may try to add a line '`psyco.runonly()`' after the last profiler, if all your profilers may stop (i.e. all of them have limits). It might help, or it might not. Feedback appreciated.

Note: Besides the limits, external events may prevent a specific profiler from running. When this occurs, Psyco falls back to the next profiler but may come back later when the external condition changes again. This occurs if some other program (e.g. the `pdb` debugger) sets a profiling or line-by-line tracing hook. Psyco always gracefully leaves the priority to other programs when they want to occupy Python's hooks.

3.3 Exceptions and warnings

exception `error`

The `psyco.error` exception is raised in case of a Psyco-specific error during the call of one of the above functions or during profiling. It is never automatically raised within one of your function that is accelerated by Psyco.

exception `warning`

The `psyco.warning` warning is issued in the hopefully rare cases where the meaning of your functions may change because of Psyco. Unlike `psyco.error`, this warning is typically issued while Psyco is running your own Python code, during a call to one of the functions described in appendix A.2.

3.4 The `psyco.classes` module

The `psyco.classes` module defines a practical metaclass that you can use in your applications.

A metaclass is the type of a class. Just like a class controls how its instances behave, a metaclass controls how its classes behave. The purpose of the present metaclass is to make your classes more Psyco-friendly.

class `psymetaclass`

The new metaclass. This is a subclass of `psyco.compacttype`. Any user-defined class using this metaclass will be a new-style class and will automatically bind all the methods defined in the class body for compilation with Psyco. It will also automatically inherit from `psyco.compact` (see section 3.5).

class `__metaclass__`

A synonym for `psymetaclass`.

class `psyobj`

An empty class whose metaclass is `psymetaclass`, suitable for subclassing in the same way that the built-in `object` is the generic empty new-style class which you can subclass.

See http://www.python.org/2.2.2/descrintro.html for more information about old-style vs. new-style classes. Be aware of the changes in semantics, as recalled in section 2.3 (Psyco tutorial).

By using `psyco.classes.__metaclass__` as the metaclass of your commonly-used classes, Psyco will call `psyco.bind` for you on all methods defined in the class (this does not include inherited parent methods or methods added after the class is first created). Additionally, these classes are "compact" (section 3.5), i.e. Psyco can produce quite fast code to handle instances of these classes and they use less memory each.

3.4.1 Examples

The following examples both define a Psyco-friendly class `X` and bind the method `f`.

```
from psyco.classes import __metaclass__
class X:
    def f(x):
        return x+1
```

'`import __metaclass__`' affects all classes defined in the module. To select individual classes instead:

```
import psyco.classes
class X(psyco.classes.psyobj):
    def f(x):
        return x+1
```

3.5 The `psyco.compact` type

Starting from version 1.4, Psyco exports two extension types: `psyco.compact` and `psyco.compacttype`. (In some future version, Psyco may be able to patch the features implemented by these types directly into the standard interpreter's built-in types).

class `psyco.compact`

A base class whose instances are stored more compactly in memory. Can be directly instantiated or subclassed. The instances behave roughly like normal Python instances, i.e. you can store and read attributes, and put methods and other attributes in the subclasses, including Python's special methods (`__init__`, `__str__`, etc.). Their advantages over regular instances are:

- They are more compact in memory, with a footprint similar to that of Python's `__slots__`--constrained instances. For example, an instance with two integer attributes uses around 180 bytes in plain Python, and only around 44 if you use __slots__ or with psyco.compact. The advantage over __slots__ is that you don't have to know all possible attributes in advance.

- In non-Psyco-accelerated function, there is a space/time trade-off: attribute accesses are a bit slower. However,
- Psyco-accelerated functions handle psyco.compact instances extremely efficiently, both in space and in time. For example, an assignment like x.attr=(a,b,c) stores the individual values of a, b and c directly into the instance data without building a tuple in memory.

There are a few visible differences and limitations:

1. Explicit use of __slots__ is forbidden.
2. Weak references to instances are not allowed. (Please write me if you would like me to remove this limitation.)
3. Instances are not based on a real dictionary; the __dict__ attribute returns a dictionary proxy (which however supports all dict operations, including writes).
4. When assigning to __dict__, a copy of the data is stored into the instance; the dict and the instance do not reflect each other's future changes (they do in plain Python).
5. Instances have a __members__ attribute which lists the current attributes, in the order in which they have been set.
6. For internal reasons, the subclasses' __bases__ attribute will always contain psyco.compact as the last item, even if one or several other base classes were specified.
7. Three methods __getslot__, __setslot__ and __delslot__ show up for internal purposes.

class psyco.compacttype

The metaclass of psyco.compact. It inherits from type. It means that when you subclass psyco.compact, your classes are by default of type psyco.compacttype instead of type. The sole purpose of this metaclass is to force psyco.compact to appear in the subclasses' bases, and to check for __slots__.

Note that you should not mix psyco.compacttype classes and normal classes in the same hierarchy. Although this might work, the instances will still each have an unused allocated dictionary.

3.5.1 Examples

```
import psyco
class Rectangle(psyco.compact):
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def getarea(self):
        return self.w * self.h

r = Rectangle(6, 7)
assert r.__dict__ == {'w': 6, 'h': 7}
print r.getarea()
```

The above example runs without using Psyco's compilation features. The r instance is stored compactly in memory. (Thus this feature of Psyco probably works on any processor, but this hasn't been tested.)

The same example using the metaclass:

```
import psyco
class Rectangle:
    __metaclass__ = psyco.compacttype
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def getarea(self):
        return self.w * self.h

r = Rectangle(6, 7)
assert r.__dict__ == {'w': 6, 'h': 7}
print r.getarea()
```

3.6 Logging

`log`(*logfile='', mode='w', top=10*)

This function enables logging. Psyco will write information to the file *logfile*, opened in the given *mode* (which should be `'w'` for overwriting or `'a'` for appending).

If unspecified, *logfile* defaults to the base script name (i.e. the content of `sys.argv[0]` without the final '.py') with a '.log-psyco' extension appened.

See below for the meaning of *top*.

A log file has the structure outlined in the following example (from 'test/bpnn.py'):

```
11:33:53.19  Logging started, 12/22/02                   %%%%%%%%%%%%%%%%%%%%
11:33:53.29  ActivePassiveProfiler: starting                           %%%%%
11:33:53.40   ______
        #1   |95.1 %|  active_start              ...psyco\profiler.py:258
        #2   | 0.9 %|  ?                         ...\lib\traceback.py:1
        #3   | 0.8 %|  ?                                      bpnn.py:8
        #4   | 0.7 %|  time                                   bpnn.py:22
        #5   | 0.4 %|  seed                      ...222\lib\random.py:140
        #6   | 0.3 %|  ?                         ...\lib\linecache.py:6
        #7   | 0.3 %|  write                     ...s\psyco\logger.py:22
        #8   | 0.3 %|  __init__                               bpnn.py:48
        #9   | 0.2 %|  go                        ...psyco\profiler.py:31
11:33:53.62  tag function: backPropagate                                  %
11:33:53.62  tag function: update                                         %
11:33:53.67  tag function: train                                          %
11:33:54.12   ______
        #1   |58.4 %|  active_start              ...psyco\profiler.py:258
        #2   | 2.5 %|  random                    ...222\lib\random.py:168
        #3   | 2.1 %|  __init__                               bpnn.py:48
        #4   | 2.1 %|  demo                                   bpnn.py:167
        #5   | 2.0 %|  dumpcharges               ...s\psyco\logger.py:56
        #6   | 1.2 %|  do_profile                ...psyco\profiler.py:299
        #7   | 1.2 %|  rand                                   bpnn.py:36
        #8   | 0.9 %|  makeMatrix                             bpnn.py:40
        #9   | 0.8 %|  time                                   bpnn.py:22
        #10  | 0.6 %|  ?                                      bpnn.py:8
(...cut...)
11:33:55.50   ______
        #1   |42.5 %|  active_start              ...psyco\profiler.py:258
        #2   | 8.3 %|  random                    ...222\lib\random.py:168
        #3   | 6.7 %|  dumpcharges               ...s\psyco\logger.py:56
        #4   | 6.6 %|  __init__                               bpnn.py:48
        #5   | 4.0 %|  rand                                   bpnn.py:36
        #6   | 3.4 %|  demo                                   bpnn.py:167
        #7   | 2.9 %|  makeMatrix                             bpnn.py:40
        #8   | 2.3 %|  do_profile                ...psyco\profiler.py:299
        #9   | 1.3 %|  time                                   bpnn.py:22
        #10  | 1.0 %|  test                                   bpnn.py:140
11:33:55.50  tag function: random                                         %
11:33:55.94  memory usage: 220+ kb                                        %
11:33:55.94  program exit, 12/22/02                      %%%%%%%%%%%%%%%%%%%%
```

The first column is a time (hours, minutes, seconds, hundredths). Most lines end in a number of percent signs; the more percent signs, the more important the line is, so that you can for example do a '`grep %%%bpnn.log-psyco`' to see the lines of importance 3 or more. As an exception, lines produced by the Psyco C core (as opposed to the Python glue, e.g. the profiler logic) end in '`% %`' (percent, space, percent).

The most common lines you will find in logs are:

#1, #2, #3,... List (on several lines) of the functions which currently have the highest charge. You can typically use this to tune the watermark (section 3.2.1). The displayed list is limited to the first 10 items by default; this can be customized with the *top* argument of `psyco.log`.

memory usage: *x*+ kb Psyco's current notion of how much memory is consumes for the emitted machine code and supporting data structures. This is a rouch estimation of the memory overhead (the + sign is supposed to remind you that this figure is highly underestimated). Use this info to tune the memory limits (section 3.2.2).

unsupported opcode *x* at *y*:*z* The function *y* cannot be compiled. Look up the opcode number *x* in the table of appendix A.3.

tag function: ***x*** The function charge has reached the watermark. Its code object is compiled. Execution of the function goes on in the compiled version.

bind function: ***x*** The function charge has reached the watermark and it is bound (with `psyco.bind`). Only when function tagging is impossible (when doing passive profiling only). Bound functions are only compiled the next time the function is called, which means that any work that is still done before the current call returns will be done uncompiled. The profiler will continue to charge the function for that work and possibly bind the same function several times (with no bad consequences).

cannot find function ***x*** **in** ***y*** The profiler's attempt to use `psyco.bind` failed because the function object could not be found. Indeed, the profiler charges code objects, and `psyco.bind` only applies to function objects. Looking for a function when we only have the code object is difficult (not to mention that some code objects may have been created by `compile` or `execfile` and are not owned by any function whatsoever). Psyco currently only locates the functions defined at the module top-level and the methods of top-level classes. (This is one reason why profilers tag and don't bind if they can.)

profiling stopped, binding ***n*** **functions** When active profiling stops, the profiler calls `psyco.bind` on all currently tagged functions. If you feel that this operation is too heavy (e.g. makes a visible pause in your real-time application) use the `psyco.runonly` profiler to prevent this from occuring. Normally it only applies if you have set limits on all the profilers queued so far.

disabled (*L* limit reached) The profiler is stopped because it reached the limit *L*. The next queued profiler can start.

disabled by psyco.error Failed to set the profiling or tracing hooks. Normally means that some other code (e.g. a debugger) is already using them. Psyco profiling will restart when the other program let go of the hooks.

resetting stats Profiling charges are reset to zero. Occurs periodically.

no locals() in functions bound by Psyco Logged the first time the corresponding `psyco.warning` is issued. This is just a reminder. To locate the problem, use the information Python prints to the terminal.

unsupported ** argument in call to ***x*** The call to the function *x* cannot be compiled because of the `**` argument. This is a limitation of Psyco; try to avoid `**` arguments (or write to me and insist on this feature).

unsupported free or cell vars in ***x*** Psyco currently cannot compile functions that use nested scopes.

3.7 Machine code inspection

Psyco works by directly emitting machine code for the processor. Typically, it will write a large number of small blocks of code, with numerous jumps from one to the other.

As you can guess, debugging this machine code with traditional tools was difficult, so I had to write helpers. They can be found in the 'py-utils' subdirectory. You need a version of Psyco compiled in debugging mode; see section 1.4 for instructions on how to build it.

`dumpcodebuf``()`

This function (to be called near the end of your program) dumps all the machine code and supporting data structures into a file 'psyco.dump'. This function has no effect if Psyco was not compiled in debugging mode.

Run the script **httpxam.py** with as argument the name of the directory which contains the 'psyco.dump' file to examine. This script formats the data as HTML pages and presents them via a web server built on Python's standard `SimpleHTTPServer`. When it is running, point your browser to http://127.0.0.1:8000.

httpxam.py probably only works on Linux. It requires the programs **objdump** or **ndisasm** to disassemble the code and **nm** to list symbol addresses in the Python and Psyco executables.

The cross-calling code buffers are presented as cross-linked HTML pages. Bold lines show the targets of another jump. If preceeded by a blank line, a bold line shows that another code buffer jumps directly at this position. The end of the buffer is often garbage; it is not code, but data added there (typically for the promotion of a value). There are various kind of code buffers, depending (only) on why Psyco produced it:

normal normal mainstream compilation

respawn execution jumps here when an error occurs, but never did so yet

respawned replaces a respawn buffer when execution has jumped here

unify small buffer whose purpose is to jump back to an existing one

load_global called when a change is detected in a global variable

coding_pause not yet compiled, will be compiled if execution jumps here

APPENDIX

A

Caveats

A.1 Known bugs

Apart from speed, functions are supposed to run identically under Psyco and under Python, with the following known exceptions:

- The functions `locals`, `eval`, `execfile`, `vars`, `dir` and `input` should work as expected starting from Psyco 1.3, but see section A.2.
- **Frame objects** are emulated. The `sys._getframe` function returns an instance of a custom class which emulates the standard frame objects' behavior as much as possible. The frames corresponding to a Psyco-accelerated frame have some placeholder attributes, notably `f_locals`. *There is no way to read the local variables of a Psyco-accelerated frame.* Actually, only the `f_code`, `f_globals`, `f_back` and `f_lineno` fields are well-tested. Also keep in mind that if you obtain a real frame object (which you can do with some other mean than `sys._getframe`, e.g. via a traceback object), the `f_back` chained list will not include the Psyco-accelerated frames.
- The compiled machine code does not include the regular polling done by Python, meaning that a `KeyboardInterrupt` will not be detected before execution comes back to the regular Python interpreter. Your program cannot be interrupted if caught into an infinite Psyco-compiled loop. (This could be fixed if requested.)
- Infinite recursions are not correctly detected. They are likely to result in segmentation faults (or whatever a stack overflow triggers on your system) instead of Python's nice `RuntimeError`. Similarily, circularities among data structures can cause troubles (e.g. printing or comparing lists that contain themselves).
- In Python 2.6 parts of the `warnings` module were rewritten in C. As a result, the reported warnings may be misplaced (they cannot show a Psyco-compiled frame).
- In Python 2.6, creating namedtuple classes (not instances) in a Psyco-optimized function gives the class the wrong `__module__` attribute, for the same reason as above: it is created by C code, so it ignores the Psyco frame hooks.

At other points, Psyco makes assumptions that may be wrong (and will cause damage if they turn out to be):

- **Built-ins** are assumed never to change. Global variables can change, of course, but you must not add or remove a global variable to shadow or expose a built-in (at least not after a module is initialized).
- Do not **dynamically change the methods** of the new-style classes (classes that inherit from a built-in type).
- Psyco assumes that **types never change**. This is basically wrong (you can assign to `__class__`). This might cause Psyco to randomly believe that instances are still of their previous type.
- Do not use Psyco together with **restricted execution** (the `rexec` module). (Given that `rexec` is deprecated and not safe in the first place, not using it is probably a good idea anyway.)

Some minor points:

- The error message coming with exceptions will occasionally differ from Python's (but not the exception class).
- The `is` operator might occasionally work unexpectedly on immutable built-in objects across function calls. For example, in

  ```
  def save(x):
    global g
    g = x
  def test(i):
    n = i+1
    save(n)
    return n is g
  ```

  there is no guarantee with Psyco that the integer object stored in *g* is identical to the object seen as *n* by the operator `is` (althought they would of course be equal). Well, `is` is not supposed to be too useful for immutable objects anyway. There are interesting exceptions, but these work as expected. Consider the above `test` function as broken because it should be (but is not) equivalent to

  ```
  def test(i):
    f(i+1)
    return (i+1) is g
  ```

- I did not test these artificial examples of tricky code accessing a list while it is being sorted. The potential problem here is that Psyco assumes that the type of an object never changes, while Python (before 2.3) changes the type of the list to make it immutable during the sort.
- Running out of memory during compilation is hard to recover from. I made it a fatal error.
- Occasionally, objects become immortal. An example of such a situation that comes in mind is the initial value of a then-changing global variable. In general, however, this concerns objects that are immortal anyway (like a global variable that does not change or constants in the code).

A.2 Patched functions

When Psyco starts, it replaces a few functions from the `__builtin__` and `sys` modules with a version of its own. This trick fails if you made a copy of one of these functions elsewhere before Psyco has a chance to replace it, because the old copy will not behave properly in the presence of Psyco.

Built-in function	Notes
`globals`	
`locals`	(1)
`vars`	(1) when called with no argument
`dir`	(1) when called with no argument
`eval`	(1)(2) when called with a single argument
`execfile`	(1) when called with a single argument
`input`	(1)
`sys._getframe`	(3)

Notes:

(1) A function run by Psyco has no native `locals` dictionary. Psyco 1.3 and above can emulate it properly if a certain optimization (early dead variable deletion) is disabled. Psyco should turn off this optimization automatically for functions where it detects a call to one of the above built-in functions, but this detection is a guess over the function's bytecode. It means that certain indirect calls can be missed. If this occurs at run-time, a `psyco.warning` is issued and the emulated `locals` dictionary is empty.

(2) Note that it is common to find Python programs that use dynamic code evaluation for an effect that can be obtained by calling an ad-hoc built-in function instead. For example, `eval('self.'+attr)` is better written as `getattr(self, attr)` and `exec 'import '+module` is better written as `__import__(module, globals(), locals(), [])`.

(3) Frames corresponding to Psyco-evaluated functions are incomplete, as described in section A.1.

Additionally, the `exec` statement is not supported yet, as seen in section A.3.

A.3 Unsupported Python constructs

Psyco only compiles functions. It will not accelerate any code that runs outside any function, like:

- top-level module code[1]
- the code defining a class — i.e. the execution of the `class` statement. Methods themselves are accelerated just fine when you actually call them.
- the code run by an `exec` statement or `execfile` or `eval`.

You can always work around the above limitations by creating functions and calling them instead of directly executing a time-consuming source. For example, instead of writing a short test script like

```
some_big_list = ...
for x in some_big_list:
    do_something()
```

write instead

```
def process_list(lst):
    for x in lst:
        do_something()
process_list(...)
```

As another example, a function like

```
def f(some_expr):
    for x in range(100):
        print eval(some_expr)    # where some_expr can depend on x
```

should instead be written

```
def f(some_expr):
    my_func = eval("lambda x: " + some_expr)    # -> a function object
    for x in range(100):
        print my_func(x)
```

In addition, inside a function, some syntactic constructs are not supported by Psyco. It does not mean that a function using them will fail; it merely means that the whole function will not be accelerated. The following table lists the unsupported constructs, along with the corresponding bytecode instruction name and number. Log files only report the bytecode instruction number, which you need to look up here.

[1] Support for top-level module code is possible but disabled by default in recent versions of Psyco; contact me for more information.

Bytecode	Instruction name	Appears in
82	LOAD_LOCALS	(1) class definitions
84	IMPORT_STAR	(5) from xx import *
85	EXEC_STMT	(2) exec xx
86	YIELD_VALUE	(3) generators
90	STORE_NAME	(5) outside functions
91	DELETE_NAME	(5) outside functions
101	LOAD_NAME	(5) outside functions
134	MAKE_CLOSURE	(4) nested scopes
135	LOAD_CLOSURE	(4) nested scopes
136	LOAD_DEREF	(4) nested scopes
137	STORE_DEREF	(4) nested scopes

Notes:

(1) Psyco cannot accelerate class definitions, i.e. the execution of the body of the class statement – i.e. the creation of the class object itself. This does not prevent it from accelerating methods in the class.

(2) Functions using this construct cannot be accelerated.

(3) Generators (i.e. any function using the `yield` keyword) cannot be accelerated currently. If there is enough interest I can consider implementing them. This includes generator expressions (Python 2.4). Warning! The function containing a generator expression will be compiled by Psyco, but not the generator expression itself. If the latter calls other functions compiled by Psyco, then performance will be very bad: calling from Psyco to Python to Psyco comes with a significant overhead.

(4) Using nested scopes (i.e. variables shared by a function and an inner sub-function) will prevent both the outer and the inner function to be accelerated. This too could be worked around if there is enough interest, at least for accelerating the unrelated parts of the functions – the accesses to the shared variables themselves might be difficult to optimize.

(5) These constructs can appear in class definitions (see (1)) or at the module top-level. It is possible to enable support for module top-level code, but not recommended; instead, try to put all the code you want accelerated in function bodies.

APPENDIX

B

Performance expectations

Psyco can compile code that uses arbitrary object types and extension modules. Operations that it does not know about will be compiled into direct calls to the C code that implements them. However, some specific operations can be optimized, and sometimes massively so — this is the core idea around which Psyco is built, and the reason for the sometimes impressive results.

The other reason for the performance improvement is that the machine code does not have to decode the pseudo-code ("bytecode") over and over again while interpreting it. Removing this overhead is what compilers classically do. They also simplify the frame objects, making function calls more efficients. So does Psyco. But doing *only* this would be ineffective with Python, because each bytecode instruction still has a lot of run-time decoding to do (typically, looking up the type of the arguments in tables, invoking the corresponding operation and building a resulting Python object).

The type-based look-ups and the successive construction and destruction of objects for all intermediate values is what Psyco can most successfully cancel, but it needs to be taught about a type and its operations before it can do so.

We list below the specifically optimized types and operations. Possible performance gains are just wild guesses; specialization is known to give often-good-but-hard-to-predict gains. Remember, all operations not listed below work well — they just cannot be much accelerated.

A **performance killer** is the usage of the built-in functions `map` and `filter`. **Never** use them with Psyco. Replace them with list comprehensions (see 2.4). The reason is that entering code compiled by Psyco from non-Psyco-accelerated (Python or C) code is quite slow, slower than a normal Python function call. The `map` and `filter` functions will typically result in a very large number of calls from C code to a `lambda` expression foolishly compiled by Psyco. An exception to this rule is when using `map` or `filter` with a built-in function, when they are typically slightly faster than list comprehension because the only difference is then that the loop is performed by C code instead of by Psyco-generated code. Still, I generally recommend that you forget about `map` and `filter` and use the "Pythonic" way.

Virtual-time objects are objects that, when used as intermediate values, are simply not be built at run-time at all. The noted performance gains only apply if the object can actually remain virtualized. Any unsupported operation will force the involved objects to be normally built.

Type	Operations	Notes
Any built-in type	reading members and methods	(1)
Built-in function and method	call	(1)
Integer	truth testing, unary + - ~ `abs()`, binary + - * | & << >> ^, comparison	(2)
Dictionary	`len()`	(4)
Float	truth testing, unary + - `abs()`, binary + - * /, comparison	(5)
Function	call	(6)
Sequence iterators	`for`	(7)
List	`len()`, item get and set, concatenation	(8)
Long	all arithmetic operations	(9)
Instance method	call	(1)
String	`len()`, item get, slicing, concatenation	(10)
Tuple	`len()`, item get, concatenation	(11)
Type	call	
array.array	item get, item set	(15)

Built-in function	Notes
`range`	(8)
`xrange`	(13)
`chr`, `ord`	(10)
`id`	
`type`	
`len`, `abs`, `divmod`	
`apply`	(14)
the whole `math` module	(16)
`map`, `filter`	**not supported**(17)

Notes:

(1) In the common '`object.method(args)`' the intermediate bound method object is never built; it is translated into a direct call to the function that implements the method. For C methods, the underlying `PyMethodDef` structure is decoded at compile-time. Algorithms doing repetitive calls to methods of e.g. lists or strings can see huge benefits.

(2) Virtual-time integers can be 100 times faster than their regular counterpart.

(4) Complex data structures are not optimized yet, beyond (1). In a future version it is planned to allow these structures to be re-implemented differently by Psyco, with an implementation that depends on actual run-time usage.

(5) Psyco does not know about the Intel FPU instruction set. It emits calls to C functions that just add or multiply two `doubles` together. Virtual-time floats are still about 10 times faster than Python.

(6) Virtual-time functions occur when defining a function inside another function, with some default arguments.

(7) Sequence iterators are virtual-time, making `for` loops over sequences as efficient as what you would write in C.

(8) Short lists and `ranges` of step 1 are virtualized. A `for` looping over a range is as efficient as the common C `for` loop. For the other cases of lists see (4).

(9) Minimal support only. Objects of this type are never virtualized. The majority of the CPU time is probably spent doing the actual operation anyway, not in the Python glue.

(10) Virtual-time strings come in many flavors: single characters implemented as a single byte; slices implemented as a pointer to a portion of the full string; concatenated strings implemented as a (possibly virtual) list of the strings this string is the join of. Text-manipulation algorithms should see massive speed-ups.

(11) Programs manipulating small tuples in local variables can see them completely virtualized away. In general however, the gains with tuples are mostly derived from the various places where Python (and Psyco that mimics it) internally manipulates tuples.

(13) Psyco can optimize `range` well enough to make `xrange` useless. Indeed, with no specific support `xrange` would be less efficient than `range`! Currently `xrange` is almost identical to `range`.

(14) Without keyword argument dictionary.

(15) Type codes `'I'` and `'L'` are not supported. Type code `'f'` does not support item assignment. The speed of a complex algorithm using an array as buffer (like manipulating an image pixel-by-pixel) should be very high; closer to C than plain Python.

(16) Missing: `frexp`, `ldexp`, `log`, `log10`, `modf`. See note (5).

(17) Systematically avoid `map` and `filter` and replace them with list comprehension (section 2.4).

INDEX