Using a separate (private) Python distribution with a Python for Delphi application

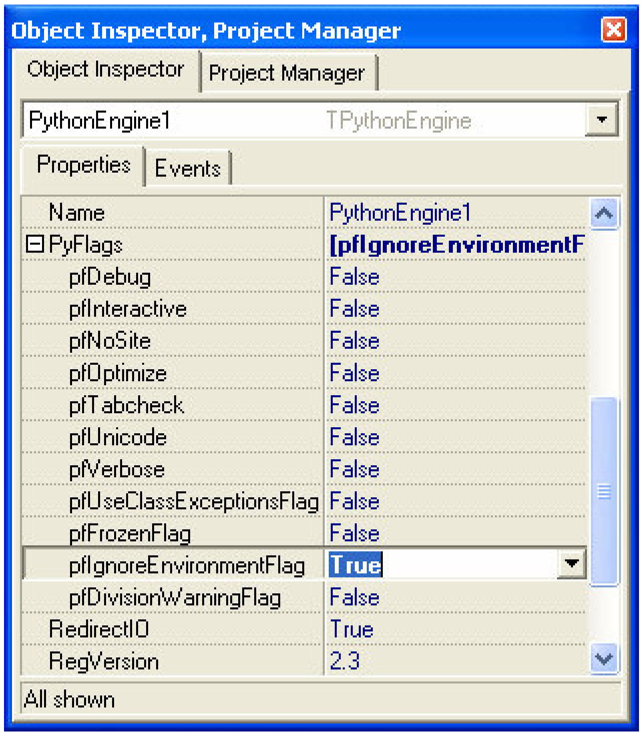
Purpose:
If your application needs to make sure it uses the intended version of Python that was tested against, and not rely on any installed version in the system, here are the steps to follow.

Definitions:

Python folder =	folder where a specific version of Python has been installed (for instance 'c:\python23')
System32 folder =	folder contained in the windows folder of your operating system (for instance 'c:\windows\system32')
Application folder =	folder where you intend to install your application (for instance 'c:\program files\MyCompany\MyProduct')

Setup TPythonEngine

In your P4D application, make sure that the TPythonEngine component has the flag pfIgnoreEnvironmentFlag set in the PyFlags property:


Object Inspector, Project Manager
Object Inspector
Project Manager
PythonEngine1 TPythonEngine
Properties
Events
All shown

Name	PythonEngine1
⊟PyFlags	[pfIgnoreEnvironmentF
pfDebug	False
pfInteractive	False
pfNoSite	False
pfOptimize	False
pfTabcheck	False
pfUnicode	False
pfVerbose	False
pfUseClassExceptionsFlag	False
pfFrozenFlag	False
pfIgnoreEnvironmentFlag	True
pfDivisionWarningFlag	False
RedirectIO	True
RegVersion	2.3

This will avoid that another Python path may be used, if an environment variable 'PYTHONHOME' has already been defined in the system.

According to Python documentation:
"The embedding application can steer the search by calling `Py_SetProgramName(`*file*`)` *before* calling `Py_Initialize()`. Note that PYTHONHOME still overrides this and PYTHONPATH is still inserted in front of the standard path."

P4D does it for you before calling Py_Initialize:

```
  if Assigned(Py_SetProgramName) then
  begin
    FProgramName := ParamStr(0);
    Py_SetProgramName(PChar(FProgramName));
  end;
```

Note that `ParamStr(0)` is the first parameter of the command line that created this process, and thus your executable.

Copy the Python folder content into your application folder

Note that you should copy the whole folder structure (lib, libs, dlls…) unless you want to minimize the size of your distribution. In that case, you can look at http://www.tarind.com/depgraph.html for a tool that will generate a dependency graph of all your Python modules.
Note that this tool relies on the modulefinder module included in the standard distribution, that you can use for your own tools.
Note however that if your application generates Python code or doesn't store the code into files, this may not work 100% accurately, unless you run the tool from your application, after everything is properly initialized.

Here's the list of builtin modules that are embedded into the python dll
(you can get it by inspecting sys.builtin_module_names):
('__builtin__', '__main__', '_codecs', '_hotshot', '_locale', '_random',
'_weakref', 'array', 'audioop', 'binascii', 'cPickle', 'cStringIO',
'cmath', 'errno', 'exceptions', 'gc', 'imageop', 'imp', 'itertools', 'marshal',
'math', 'md5', 'msvcrt', 'nt', 'operator', 'pcre', 'regex', 'rgbimg',
'rotor', 'sha', 'signal', 'strop', 'struct', 'sys', 'thread', 'time',
'xreadlines', 'xxsubtype', 'zipimport')

You also have to be careful if your code relies on the datetime module which is located in C:\\Python23\\DLLs\\datetime.pyd

Copy the System32\PythonXX.dll into your application folder

This will guarantee that P4D will use the correct Python dll, because when a dll is in the same folder as the host application, it will be loaded first, without inspecting the PATH environment variable or the System32 folder.

Note that when deploying your application on Windows 98, you have to make sure that the file msvcr71.dll is deployed too or P4D won't be able to load the python dll.
Note that if you want to specify the exact filename of the Python dll, you can specify the folder where it is located using the DllPath property of the TPythonEngine (you can initialize the property in the OnBeforeLoad event):

```
PythonEngine1.DllPath := ExtractFilePath(Application.ExeName);
```

Copy your P4D executable into your application folder.

In that case Python will extract the path of your exe and try to deduce some critical folders like lib, libs, dlls. If it can find them it will assume that it found the Python location, without relying to the Registry.

Here's what you can read in sources\PC\getpathp.c:

- When running python.exe, or any other .exe in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is **deduced**, and the core paths in the registry are **ignored**. Other "application paths" in the registry are always read.
- When Python is hosted in another exe (different directory, embedded via COM, etc), the Python Home will **not be deduced**, so the core path from the **registry** is used. Other "application paths" in the registry are always read.
- If Python can't find its home and there is no registry (eg, frozen exe, some very strange installation setup) you get a path with some default, but relative, paths.

Verifying the origin of imported modules

You can check from where your modules where imported by executing the following Python script from your P4D application:

```
import sys
import os

print "Python path =", sys.path
print
print "Python modules already imported:"
for m in sys.modules.values():
  if m:
    print "   ", m
print
print "PYTHONHOME =", os.getenv('PYTHONHOME')
print "PYTHONPATH =", os.getenv('PYTHONPATH')
```

This will list the python path, the imported modules and their filename, and finally the Python environment variables.

Conclusion

As we disabled usage of environment variables, your path won't be affected by the presence of PYTHONHOME and/or PYTHONPATH.
Python will use the path of your executable as the core path of Python, if it can find what it expects (some core modules), and thus ignore the registry settings.
Finally, Python will add to your path any application path that comes from the registry (any subkey of SOFTWARE\Python\PythonCore\2.3\PythonPath, for instance win32 or win32com that comes with PythonWin).
If you don't want to be affected by those application paths, you have to edit the 'path' global variable in the 'sys' module by code and remove what you don't want or, even simpler, overwrite it with what you expect (sys.path = [MyHome+'lib', MyHome+'libs', MyHome+'dlls'…]). You should do this in the OnSysPathInit event of TPythonEngine, that is triggered immediately after Python has been initialized (Py_Initialize) and before TPythonEngine looks up some specific modules, like datetime.

Example:

```
procedure TForm1.PythonEngine1SysPathInit(Sender : TObject;
PathList : PPyObject);
var
  folder : PPyObject;
begin
  with GetPythonEngine do
  begin
    folder := PyString_FromString('c:\myapp\mymodules');
    PyList_Append(PathList, folder); // or insert, or clear…
    Py_XDecRef(folder);
  end;
end;
```

Note that Python won't use the application paths by default in its initialization. They simply allow you to import packaged modules more simply.

Note that the OnSysPath event can help you customize the Python path and have your modules located anywhere in the filesystem, even with a regular Python installation.

Note that you can also do the changes after Python has been initialized, in the OnAfterInit event of TPythonEngine, or using the InitScript property of TPythonEngine that contains a Python script executed immediately after Python is ready, but this would be too late for locating properly the datetime.pyd module.