Pythran Developper Guide

Do not hang around in Pythran code base without your developper guide! It is the compass that will guide you in the code jungle!

Disclaimer

This document is a never ending work-in-progress draft. Please contribute!

Coding Style

All Python code must be conform to the PEP8, and the `pep8` command must not yield any message when run on our database. Additionnaly, avoid backslashes, and try to make your code as concise as possible.

C++ code use spaces (no tabs) and a tab width of 4.

File Hierarchy

Listing the top level directory yields the following entries:

setup.py

The files that describels what gets installed, that holds `PyPI` entries and such.

doc/

If you're reading this document, you know what it's all about! `MANUAL` is the user documentation and `DEVGUIDE` is the developper documentation.

LICENSE

Boring but important stuff.

MANIFEST.in

Describe additionnal stuff to package there.

README

Quick introduction and description of _pythran_. The `README.rst` file is just a symbolic link that pleases `github` and `PyPI`.

pythran/

The source of all things.

pythran/tests/

The source of all issues.

scripts/

Where python scripts calling the `pythran` module lies.

Validation

`pythran` uses the `unittest` module to manage test cases. The whole validation suite is run through the command:

```
$> python setup.py test
```

If you have *py.test <http://pytest.org/latest/>* from debian package *python-pytest-xdist* in your `PYTHONPATH`, the test suite will run using all available cores. Otherwise it might run **very** slowly, almost half an hour on a decent laptop :'(.

Note that it is still possible to use the `unittest` module directly, for instance to pass a subset of the test suite:

```
$> PYTHONPATH=.:pythran/tests:$PYTHONPATH python -m unittest test_math
```

runs all the tests found in `pythran/tests/test_math.py`. The command:

```
$> PYTHONPATH=. py.test -n 8 pythran/tests/test_list.py
```

does almost the same with `py.test`.

There are two kinds of tests in `pythran`:

1. unit tests that test a specific feature of the implementation. Such tests are listed as method of a class deriving from `test_env.TestEnv` and must call the `run_test(function_to_translate, *effective_parameters, **name_to_signature)` method [1]. It translates `function_to_translate` into a native function using the type annotations given in the `name_to_signature` dictionnary, runs both the python and the native version with `effective_parameters` as arguments and asserts the results are the same.
2. test cases that are just plain python modules to be converted in native module by `pythran`. It is used to test complex situations, codes or benchmarks found on the web etc. They are just translated, not run. These test cases lie in `pythran/tests/cases/` and are listed in `pythran/tests/test_cases.py`.

C++ runtime

The C++ code generated by `pythran` relies on a specific backend, `pythonic++`. It is a set of headers that mimics python's intrinsic and collections behavior in C++. It lies in `pythran/pythonic++/`. All headers are `#included` in the header `pythran/pythonic++/pythonic++.h`. Core features lie in `pythran/pythonic++/core/` and extra modules lie in `pythran/pythonic++/modules`.

Each extra module defines a new namespace, like `pythonic::math` or `pythonic::random`. The `PROXY` and `VPROXY` macros are used to convert functions into functors, the difference between the two being that `VPROXY` allows its argument to be modified.

Benchmarking and Testing

Stand-alone algorithms are put into `pythran/tests/cases`. They must be valid pythran input (including spec annotations). To be taken into account by the validation suite, they must be listed in `pythran/tests/test_cases.py`. To be taken into account by the benchmarking suite, they must have a line starting with the `#runas` directive. Check `pythran/tests/matmul.py` for a complete example.

To run the benchmark suite, one can rely on:

```
$> python setup.py bench --mode=<mode>
```

where <*mode*> is one among:

python

Uses the interpreter used to run `setup.py`.

pythran

Uses the pythran compiler.

pythran+omp

Uses the pythran compiler in OpenMP mode.

All measurements are made using the `timeit` module. The number of iterations is customizable through the `--nb-iter` switch.

How to

Add support for a new module:

1. **Provide its C++ implementation in `pythran/pythonic++/modules`.**

   `pythran/pythonic++/modules/math.h` and `pythran/pythonic++/modules/list.h` are good example to referer to.

2. **Provide its description in `pythran/tables.py`. Each function, method**

   or variable must be listed there with the appropriate description.

3. **Provide its test suite in `pythran/tests/` under the name**

   `test_my_module.py`. One test case per function, method or variable is great.

Add a new analysis:

1. Subclass one of `ModuleAnalysis`, `FunctionAnalysis` or `NodeAnalysis`.
2. List analysis required by yours in the parent constructor, they will be built automatically and stored in the attribute with the corresponding uncameled name.
3. Write your analysis as a regular `ast.NodeVisitor`. The analysis result must be stored in `self.result`.
4. Use it either from another pass's constructor, or throught the `passmanager.gather` function.

Push changes into the holy trunk:

1. Use the `github` interface and the pull/push requests features
2. **Make your dev available on the web and asks for a merge on the IRC**

   channel `#pythran`

[1] See examples in `pythran/tests/test_base.py` for more details.