PyCC Documentation

Release 1.0.0

Kevin Conway

November 01, 2014

Contents

PyCC is a Python code optimizer. It rewrites your code to make it faster.

CHAPTER 1

PyCC Usage Documentation

PyCC offers two command line utilities: pycc-transform and pycc-compile.

1.1 Transforming Code

PyCC is quite a young project. As a result, you may be hesitant to simply trust the output and skip straight to the compile step. Instead, you may want to see an example of what the optimizer has produced so you can verify that it is, indeed, both valid Python and acceptable code. This is what the *pycc-transform* command is for.

Usage for this command is simple:

> pycc-transform –source=<python_file.py> [–optimizer_option]

All optimization is disabled by default. Use the appropriate flags to enable the optimizations you want to apply. At the time of writing the full options listing for pycc-transform was:

> **usage: pycc-transform [-h] –source SOURCE [–destination DESTINATION]** [–constants]
>
> PyCC Transformer
>
> **optional arguments:**
>
> | | |
> |---|---|
> | **-h, --help** | show this help message and exit |
> | **--source SOURCE** | Path to load the Python source from. |
> | **--destination DESTINATION** | Path to place the optimized result or “stdout”. |
> | **--constants** | Inline constant values. |

Running this command will generate a file called <python_file>_optimized.py that you can view. To print the results directly to the terminal simply add the ‘–destination=”stdout”’ option.

If you run pycc-transform and it produces invalid Python code or it changes the code such that it no longer does what it is supposed to do then please post the original source and optimization options enabled as a bug on the project GitHub page.

1.2 Compiling Code

Once you are comfortable with the way PyCC alters your code you can start running the *pycc-compile* command:

> **usage: pycc-compile [-h] –source SOURCE [–destination DESTINATION]** [–constants]
>
> PyCC Compiler

optional arguments:

-h, --help	show this help message and exit
--source SOURCE	Path to load the Python source from.
--destination DESTINATION	Path to place the optimized result or "stdout".
--constants	Inline constant values.

The compiler can be run on either individual Python modules or it can be pointed at a Python package. By default the script will drop the '.pyc' files right next to the source files just like the normal Python compiler. Alternatively you may use the 'destination' option to direct the '.pyc' files to another directory.

The Python compiler will overwrite existing '.pyc' files every time a source file is updated. The 'destination' option is useful if you want to create a compiled version of a module or package that doesn't get overwritten every time you make a change to the source.

CHAPTER 2

PyCC Optimizer Documentation

Each PyCC optimizer applies some transformation to your source code. All optimizers are disabled by default in both the *pycc-transform* and *pycc-compile* scripts. Below is a list of optimizations that can be enabled, a description of what transformation it applies, and the command line flag needed to enable it.

2.1 Constant In-lining

Flag: –constants

As demonstrated on the main page, this option replaces the use of read-only, constant values with their literal values. This affects variables that are assigned only once within the given scope and are assigned to a number, string, or name value. Name values are any other symbols including True, False, and None.

This transformation does not apply to constants that are assigned to complex types such as lists, tuples, function calls, or generators.

In addition, simple arithmetic operations performed on constant values are automatically calculated and the constant value inserted back.

CHAPTER 3

PyCC Developer Documentation

3.1 Contributing To The Project

Adding a new optimizer to the core project is (hopefully) a straightforward process. You are, of course, welcome to use any kind of workflow you like. For this walkthrough I will be using my own as an example.

3.1.1 Step 1: Create An Extension Object

The first step is getting your extension bootstrapped into the CLI right away. This will allow you to use the *pycc-transform* command as soon as your code is written. Seeing the transformed output early makes it a little easier to debug and iterate.

Start by creating a new module in the *pycc.cli.extensions* package. Normal Python naming rules apply. Try to keep it short and descriptive. Inside the module start with some light scaffolding like:

```
"""Core extension for some new optimizer."""

from __future__ import division
from __future__ import absolute_import
from __future__ import print_function
from __future__ import unicode_literals

from . import interfaces
from ...optimizers import <my_optimizer_module>


class MyOptimizerExtension(interfaces.CliExtension):

    """A CLI extension which enables my optimization."""

    name = 'makeitfast'
    description = 'Makes code go fast.'
    arguments = ()

    @staticmethod
    def optimize(node):
        """Make all the code better and faster."""
        <my_optimizer_module>.optimize(node)
```

This class will act as the entry point for your optimizer and will be used by the CLI tools.

3.1.2 Step 2: Adding Arguments

If you know you want to have CLI arguments to customize the behaviour of your optimizer then you need to modify the *arguments* property of your extension class. All arguments you add to the tuple should be instances of the *interfaces.Arg* class which takes in a *name*, *type*, and *description* at initialization. For example, if you wanted to add an integer argument:

```
arguments = (interfaces.Arg('num-times', int, 'Go x times faster!'),)
```

This will add a CLI argument called 'makeitfast-num-times'. Each argument will be prepended by the extension name when it appears on the command line. However, when you accept this argument in your *optimize* method you should simply use the argument name with underscores instead of dashes:

```
def optimize(node, num_times=1):

    for x in range(num_times):

        <my_optimizer_module>.optimize(node)
```

All arguments will be passed in as keyword arguments.

3.1.3 Step 3: Adding An Empty Optimizer

Now create a module with a similar name to your extension module in the *pycc.optmizers* package. In this module define and empty function called *optimize*:

```
def optimize(node):

    pass
```

This is where you will implement the actual optimization logic. Leave it blank just for a moment.

3.1.4 Step 4: Add An Entry Point

In the setup.py file you will find a list of setuptools entry points which link to other extensions. Add one under *pycc.optimizers* that points back to the extension class you created in step 1.

```
entry_points={
    'console_scripts': [
        'pycc-transform = pycc.cli.transform:main',
        'pycc-compile = pycc.cli.compile:main',
    ],
    'pycc.optimizers': [
        'pycc_constant_inliner = pycc.cli.extensions.constants:ConstantInlineExtension',
        'pycc_makeitfaster = pycc.cli.extensions.makeitfast:MyOptimizerExtension',
    ],
},
```

This will register your new extension with the CLI. Now if you do a *pycc-transform –help* you will see a flag, or flags, added to the CLI that represent your new addition.

3.1.5 Step 5: Optimize

All the rest is on you. Implement the body of the *optimize* function in your *optimizers* module and see the results. All optimization and modifications of the AST should be done in-place.

How you go about implementing the optimizer is up to you. There are, however, some tools in PyCC which may prove useful. A full listing of those tools can be found in the asttools and astwrapper modules.

3.1.6 Step 6: Test And Lint

Before you submit your pull request, make sure it passes all the automated tests. TravisCI will run them for you, but you can also use the tox setup packaged with this project. Make sure your code passes PEP8, pyflakes, and tests in all Python environments (2.6 - 3.4).

You should also add to the tests as you develop your optimizer. Use the existing tests as a guide if you are unsure where to start. Just make sure you've given a best effort to make sure the optimizer works correctly. If you spend a significant amount of time trying to overcome and edge case or bug you should most certainly make a test that replicates the issue so another developer doesn't change your code and cause a regression.

3.2 AST Tools

3.3 Developing Third Party Extensions

PyCC is designed to treat all optimizers, even the core ones, as extensions. This makes all the above information applicable to writing your own third part extension.

The major differences are the, obviously, you will be working in your own code base rather than this project directly. Since that is the case, the organization, style, testing framework, and etc. are all up to you. This project places no constriction on how you develop your own, independent code.

The only exception to this is the extension interface. While you do not have to use the base classes or tools from PyCC, the extension you expose _must_ match the standard interface.

The basic requirements for the interface are:

- Must have a 'name' property with a short, unique name for the extension.
- Must have a 'description' property with a short description of the extension.
- Must have an 'arguments' property which is an iterable.
- Each item present in 'arguments' must expose the following properties:
  - 'name'

    Name of the argument as it appears on the command line.
  - 'description'

    Help message that describes what the flag does.
  - 'type'

    Type object (int, str, etc.) that will be used to type cast the value of the flag.
- Must expose a function called 'optimize'. This method must:
  - Accept an *ast.AST* node as the first parameter.
  - Accept keyword arguments that match the items given in 'arguments' above. Note: dashes are replaced with underscores.

Beyond this, the only thing your project must do is provide an entry point under the *pycc.optimizer* group which points to your extension interface.

CHAPTER 4

pycc

4.1 pycc package

4.1.1 Subpackages

pycc.asttools package

Submodules

pycc.asttools.compiler module

pycc.asttools.name module

Utilities for working with ast.Name nodes.

class pycc.asttools.name.**NameVisitorMixin**
Bases: object

Generates ast.Name nodes for a given tree.

Surrogate ast.Name nodes will be given for names created by other actions such as function/definitions and function parameters.

visit_ClassDef(*node*)
Get a surrogate name from a function or class definition.

visit_FunctionDef(*node*)
Get a surrogate name from a function or class definition.

visit_Global(*node*)
Get a surrogate name from a global or nonlocal statement.

visit_Name(*node*)
Return ast.Name values unaltered.

visit_Nonlocal(*node*)
Get a surrogate name from a global or nonlocal statement.

visit_alias(*node*)
Get a surrogate name using an alias.

visit_arg(*node*)
Get a surrogate name from an argument specification.

visit_arguments(*node*)
Get surrogate name nodes from function arguments.

This method only handles the *args and **kwargs variable names. In PY2 the rest of the children for this node names. In PY3+ they are args and handled by another visitor.

pycc.asttools.name.**declaration**(*node*, *start=None*)
Find the AST node where a name is first declared.

The search begins in the same scope as the given ast.Name node. To search within a custom path pass in an AST node as the 'start' parameter.

pycc.asttools.name.**name_source**(*node*, *declared=None*)
Get the NAME_SOURCE for the given name.

If 'declared' is not given it will be derived from the given name.

pycc.asttools.parse module

Utilities for generating AST objects from source.

pycc.asttools.parse.**parse**(*source*, *filename=u'<unknown>'*, *mode=u'exec'*)
An ast.parse extension.

This function behaves identically to the standard ast.parse except that it adds parent and sibling references to each node.

pycc.asttools.references module

Utilities for adding node references to AST nodes.

pycc.asttools.references.**add_parent_references**(*node*)
Add a parent backref to all child nodes.

pycc.asttools.references.**add_sibling_references**(*node*)
Add sibling references to all child nodes.

pycc.asttools.references.**copy_location**(*new_node*, *old_node*)
An ast.copy_location extension.

This function behaves identically to the standard ast.copy_location except that it also copies parent and sibling references.

pycc.asttools.references.**get_top_node**(*node*)
Get the top level ast node by backtracking through the parent refs.

pycc.asttools.scope module

Containers for variable scope rules.

pycc.asttools.scope.**child_scopes**(*node*)
Generate child AST nodes that represent lexical scopes.

pycc.asttools.scope.**is_scope**(*node*)
True if the ast node is a scope else False.

pycc.asttools.scope.**parent_scope**(*node*)
Find the AST node that represents the first enclosing scope.

pycc.asttools.scope.**scope_type**(*node*)
Get the type of scope represented by a node.

pycc.asttools.visitor module

Utilities for iterating over child AST nodes.

class pycc.asttools.visitor.**NodeTransformer**(**args*, ***kwargs*)
Bases: pycc.asttools.visitor.NodeVisitor

Alternate implementation of ast.NodeTransformer.

This implementation subclasses the non-recursive NodeVisitor from this module. This implementation should behave identically to that of the standard behaviour in how it replaces nodes.

However this implementation differs in behaviour as layed out in the documentation for the non-recursive NodeVisitor. Another major difference in behaviour is that the visit method does not return the modified AST. Instead you must rely on the fact that the node is modified in place. The standard implementation also modifies the AST in place, it simply returned the resulting value for convenience.

This implementation relies on the parent and sibling references provided by the asttools.references module.

modified
Determine whether or not the source was altered.

visit()
Visit the nodes.

class pycc.asttools.visitor.**NodeVisitor**(*node*)
Bases: object

Alternate visitor implementation.

This NodeVisitor interface differs slightly from the standard implementation. Primarily, the AST node to visit is given during initialization rather than at visit time. This has the effect of making instances of this NodeVisitor only usable with one AST and only one time. Subsequent calls will have no effect.

This implementation provides a "generic_visit" method which can be used within individual visit methods in order to evaluate all the children of a node. This method may be used but should not be overwritten.

This implementation provides no return value from the visit method. This makes it not well suited for use in a compiler.

The benefit of this implementation is that it does not leverage function recursion. Rough benchmarks show this to be somewhere between two and four times faster than the default implementation.

generic_visit(*node*)
Queue up all child nodes for visiting.

visit()
Visit the nodes.

This method will always start with the node given at initialization.

class pycc.asttools.visitor.**NodeVisitorIter**(*node*)
Bases: pycc.asttools.visitor.NodeVisitor

NodeVisitor subclass which produces all visitor return values.

Unlike the base NodeVisitor in this module, this subclass returns an iterable from the visit method which contains all values returned by individual visit methods.

visit()
Visit the nodes.

This method returns an iterable of values returned by visitor methods.

If multiple values are returned by a visitor they will all be included in the resulting iterable.

Module contents

Utilities for working with Python AST nodes.

pycc.astwrappers package

Submodules

pycc.astwrappers.name module

Wrappers for ast.Name nodes.

class pycc.astwrappers.name.**Name**(*node*)
Bases: object

Wrapper for an ast.Name node for ease of use.

assignments
Get an iterable of all assignments to a name.

The scoping rules for this method are identical to that which produces an iterable of name uses.

constant
True if name is assigned to only once within its lexical scope.

declaration
Get the first declaration of the Name.

node
Get the raw ast.Name node.

source
Get the asttools.name.NAME_SOURCE of the Name.

token
Get the string which represents the Name.

uses
Get an iterable of all uses of the name.

If the source is asttools.name.NAME_SOURCE.BUILTIN this iterable will contain all uses of the name in the module. Otherwise only uses within the lexical scope of the declaration are contained within the iterable.

class pycc.astwrappers.name.**NameGenerator**(*node*)
Bases: pycc.asttools.visitor.NodeVisitorIter, pycc.asttools.name.NameVisitorMixin

Visitor which produces Name objects.

visit()
Produce Name objects from a NameVisitorMixin.

Module contents

Wrappers for AST nodes.

pycc.cli package

Subpackages

pycc.cli.extensions package

Submodules

pycc.cli.extensions.constants module Core extension for constant in-lining.

class pycc.cli.extensions.constants.**ConstantInlineExtension**
Bases: pycc.cli.extensions.interfaces.CliExtension

A CLI extension which enables constant in-lining.

arguments = ()

description = u'Inline constant values.'

name = u'constants'

static optimize (*node*)
In-line all constant values.

pycc.cli.extensions.interfaces module Standardized interfaces for extensions.

class pycc.cli.extensions.interfaces.**Arg**
Bases: tuple

Arg(name, type, description)

description
Alias for field number 2

name
Alias for field number 0

type
Alias for field number 1

class pycc.cli.extensions.interfaces.**CliExtension**
Bases: object

This object represents the interface that extensions must implement.

arguments = (Arg(name=u'num-tries', type=<type 'int'>, description=u'Number of times to try something.'),)

description = u'A CLI Extension.'

name = u'extension'

static optimize (*node*, **args*, ***kwargs*)
Run the optimizer for the given node.

Arguments given on the CLI will be passed in as keyword arguments.

pycc.cli.extensions.utils module Utilities for getting and using extensions.

pycc.cli.extensions.utils.**execute**(*args*, *node*)
Run the enabled extensions.

pycc.cli.extensions.utils.**iter_extensions**()
Get an iterable of extensions.

pycc.cli.extensions.utils.**register_extensions**(*parser*)
Register the CLI args for all extensions using the given parser.

Module contents Utilities for building CLI extensions and the core extensions.

Submodules

pycc.cli.common module

Common utilities for CLI modules.

class pycc.cli.common.**PySource**
Bases: tuple

PySource(node, path)

node
Alias for field number 0

path
Alias for field number 1

pycc.cli.common.**abspath**(*path*)
Resolve a path to an absolute path.

pycc.cli.common.**load_dir**(*path*)
Get an iterable of PySource from the given directory.

pycc.cli.common.**load_path**(*path*)
Get an iterable of PySource from the given path.

pycc.cli.common.**register_arguments**(*parser*)
Add arguments required for all CLI tools.

pycc.cli.common.**run_optimizers**(*args*)
Optimize each of an iterable of PySource objects.

pycc.cli.common.**write_result**(*body*, *path*, *path_override=None*)
Write the body into a file in path or path_override.

pycc.cli.compile module

pycc.cli.transform module

Module contents

Modules related to the command line interface of the project.

pycc.optimizers package

Submodules

pycc.optimizers.constant module

Optimizer for inlining constant values.

class pycc.optimizers.constant.**ConstantInliner**(**args*, ***kwargs*)
Bases: pycc.asttools.visitor.NodeTransformer

NodeTransformer which places constant values in-line.

visit_BinOp(*node*)
Perform binary ops if all values are constant.

visit_Name(*node*)
Replace ast.Name with a value if it is a constant reference.

pycc.optimizers.constant.**optimize**(*node*)
Optimize an AST by in-lining constant values.

Module contents

Python code optimizers which rewrite the AST.

4.1.2 Submodules

4.1.3 pycc.enum module

Enum implementation.

class pycc.enum.**Enum**(*names*)
Bases: object

An object which acts like an Enum.

Each enum attribute has a unique object value which makes it possible to use the 'is' operator for comparisons.

class pycc.enum.**EnumValue**(*name*)
Bases: object

An object which represents a value in an Enum.

4.1.4 pycc.pycompat module

Compatibility helpers for Py2 and Py3.

class pycc.pycompat.**VERSION**
Bases: object

Stand in for sys.version_info.

The values from sys only have named parameters starting in PY27. This allows us to use named parameters for all versions of Python.

major = 2

micro = 6

minor = 7

releaselevel = 'final'

serial = 0

4.1.5 Module contents

A Python optimizing compiler and AST toolkit.

CHAPTER 5

Basic Example

Symbol table (variable) lookups don't seem expensive at first.

```
# awesome_module.py

MAGIC_NUMBER = 7

for x in xrange(10000000):

    MAGIC_NUMBER * MAGIC_NUMBER
```

Now let's make a crude benchmark.

```
# Generate bytecode file to skip compilation at runtime.
python -m compileall awesome_module.py
# Now get a simple timer.
time python awesome_module.pyc

# real    0m0.923s
# user    0m0.920s
# sys     0m0.004s
```

What does PyCC have to say about it?

```
pycc-transform awesome_module.py --constants

MAGIC_NUMBER = 7
for x in xrange(10000000):
    (7 * 7)
```

Neat, but what good does that do?

```
pycc-compile awesome_module.py -- constants
time python awesome_module.pyc

# real    0m0.473s
# user    0m0.469s
# sys     0m0.004s
```

CHAPTER 6

How To Get It

```
pip install pycc
```

CHAPTER 7

Source?

If you want to file a bug, request an optimization, contribute a patch, or just read through the source then head over to the GitHub page.

CHAPTER 8

License

The project is licensed under the Apache 2 license.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*

p

A

C

D

E

G

I

L

M

N

O

P

R

S

T

U

V

W