Quasiquotes

Release 0.2.1

February 15, 2016

Contents

Blocks of non-python code sprinkled in for extra seasoning.

CHAPTER 1

What is a `quasiquote`

An `quasiquote` is a new syntactical element that allows us to embed non python code into our existing python code. The basic structure is as follows:

```
# coding: quasiquotes

[$name|some code goes here|]
```

This desuagars to:

```
name.quote_expr("some code goes here", frame, col_offset)
```

where `frame` is the executing stack frame and `col_offset` is the column offset of the quasiquoter.

This allows us to use slightly nicer syntax for our code. The `# coding: quasiquotes` is needed to enable this extension. The syntax is chosen to match haskell's quasiquote syntax from GHC 6.12. We need to use the older syntax (with the `$`) because python's grammar would be ambiguous without it at the quote open step. To simplify the tokenizer, we chose to use slighly more verbose syntax.

We may also use statement syntax for quasiquotes in a modified with block:

```
# coding: quasiquotes

with $name:
    some code goes here
```

This desuagars to:

```
name.quote_stmt("    some code goes here", frame, col_offset)
```

CHAPTER 2

The `c` quasiquoter

The builtin `c` quasiquoter allows us to inline C code into our python. For example:

```
>>> from quasiquotes.c import c
>>> def f(a):
...     with $c:
...         printf("%ld\n", PyLong_AsLong(a));
...         a = Py_None;
...         Py_INCREF(a);
...     print(a)
...
>>> f(0)
0
None
>>> f(1)
1
None
```

Here we can see that the quasiquoter can read from and write to the local scope.

We can also quote C expressions with the quote expression syntax.

```
>>> def cell_new(n):
...     return [$c|PyCell_New(n);]
...
>>> cell_new(1)
<cell at 0x7f8dde6cd5e8: int object at 0x7f8ddf956780>
```

Here we can see that the `c` quasiquoter is really convenient as a python interface into the C API.

Warning: CPython uses a reference counting system to manage the lifetimes of objects. Code like:
`return [$|Py_None|]`
can cause a potential segfault when `None` because it will have 1 less reference than expected. Instead, be sure to remember to incref your expressions with:
`return [$|Py_INCREF(Py_None); Py_None|]`
You must also incref when reassigning names from the enclosing python scope. For more information, see the CPython docs.

CHAPTER 3

The `r` quasiquoter

The optional `r` quasiquoter allows us to inline R code into our python. For example:

```
>>> from quasiquotes.r import r
>>> def f(a):
...     with $r:
...         print(a)
...         a <- 1
...     print(a)
...
>>> f(0)
[1]
 0


array([ 1.])
>>> f(1)
[1]
 0


array([ 2.])
```

Here we can see that the quasiquoter can read from and write to the local scope.

Note: The return type is coerced to a numpy array of length one because there are no scalar types in R.

We can also quote R expressions with the quote expression syntax.

```
>>> def r_isna(df):
...     return [$r|is.na(df)|]
...
>>> df = pd.DataFrame({'a': [1, 2, None], 'b': [4, None, 6]})
>>> df
    a   b
0   1   4
1   2 NaN
2 NaN   6
>>> r_isna(df)
array([[0, 0],
       [0, 1],
       [1, 0]], dtype=int32)
```

Note: The `r` quasiquoter is installed with `pip install quasiquotes[r]` This will install rpy2 which is used to interface with R.

CHAPTER 4

IPython Integration

We can use the `c` quasiquoter in the IPython repl or notebook as a cell or line magic. When used as a line magic, it is quoted as an expression. When used as a cell magic, it is quoted as a statement.

```
In [1]: import quasiquotes.c

In [2]: a = 5

In [3]: %c PyObject *b = PyLong_FromLong(3); PyObject *ret = PyNumber_Add(a, b); Py_DECRE   F(b); ret
Out[3]: 8

In [4]: %%c
   ...: printf("%ld + %ld = %ld\n", 3, PyLong_AsLong(a), PyLong_AsLong(_3));
   ...: puts("reassigning 'a'");
   ...: a = Py_None;
   ...: Py_INCREF(a);
   ...:
3 + 5 = 8
reassigning 'a'

In [5]: a is None
Out[5]: True
```

Contents

4.1 Quasiquotes API

quasiquotes is designed to make it easy to extend python syntax with arbitrary parsing logic. To define a new syntax enhancement, create an instance of a subclass of *QuasiQuoter* that overrides the `quote_expr` or `quote_stmt` methods.

class `quasiquotes.quasiquoter.`**`QuasiQuoter`**

Custom parsing logic for python

static `locals_to_fast` (*frame*, ***, *_locals_to_fast=<_FuncPtr object>*, *_pyobject=<class 'ctypes.py_object'>*, *_true=c_int(1)*)

Write the `f_locals` of `frame` back into the fast local storage.

Parameters **frame** : frame

The frame whose `f_locals` and fast will be synced.

quote_expr (*expr*, *frame*, *col_offset*)
Quote an expression.

This is called in the oxford brackets case: [$qq|...|]

Parameters **expr** : str

The expression to quote.

frame : frame

The stack frame where this expression is being executed.

col_offset : int

The column offset for the quasiquoter.

Returns **v** : any

The value of the quoted expression.

quote_stmt (*stmt*, *frame*, *col_offset*)
Quote a statment.

This is called in the enhanced with block case: with $qq: ...

Parameters **stmt** : str

The statement to quote. This will have the unaltered indentation.

frame : frame

The stack frame where this statement is being executed.

col_offset : int

The column offset for the quasiquoter.

`quote_stmt` has no value. It is used to run normal imperitive code like you would normally put in the body of a context manager.

`quote_expr` has a value. It is used to create expressions that can be plugged into other expressions.

Both `quote_stmt` and `quote_expr` are passed 3 arguments:

1. String representing the body of either the expression or statement
2. Stackframe where this is being executed
3. Column offset of the quasiquoter

The string will be the pre-built string literal the we constructed at decode time. The stackframe will be the python stackframe where the quoted statement or expression is being used. Finally the column offset will be the pre-built integer constant that represents the offset of the quasiquote token.

Each quasiquoter is free to do whatever it wants with this information, including mutation of the calling frame's locals, compiling new code, or just ignoring the body.

A quasiquoter does not need to implement both `quote_stmt` and `quote_expr`. In some cases, it only makes sense to support one of these features. If a quote type is used syntactically; however, the runtime quasiquoter does not support this featere then a `quasiquotes.quasiquoter.QQNotImplementedError` exception will be raised.

4.2 Inline c

The most fully featured quasiquoter and the the reason that this project exists is the `c` quasiquoter. The c quasiquoter is designed to be a way to seamlessly use the CPython API while preserving code locality and avoiding boilerplate.

When optimizing python, we often find that very few functions are hotspots that require us to rewrite in c. Good practice says to start in python and then slowly port the slow functions into c one at a time. We don't just want to rewrite all of it because then we lose the maintainability of python for a trivial gain. The c quasiquoter gives us even more fine control over which parts of our program can be in c by allowing us to weave sections of c into our python functions. We can even do things like rewrite a single loop in a function in c.

One of the main benifits of this approach is that we can keep the optimized c code right next to the python that it is supporting. This is a huge benifit for maintainability.

4.2.1 Namespace Management

The c quasiquoter allows us to manipulate the python namespace of the enclosing scope. For example:

```
>>> a = 1
>>> b = 'test'
>>> with $c:
...     printf("%ld\n%s\n",
...            PyLong_AsLong(a),
...            PyUnicode_AsUTF8(b));
1
test
```

Here we can see that the variables from the enclosing scope have been passed into our function. All python values will have the standard type of `PyObject*` and can be used like normal.

We can also change the namespace just like a normal context manager.

```
>>> a = 1
... with $c:
...     printf("%ld\n", PyLong_AsLong(a));
...     a = Py_None;
...     Py_INCREF(a);
1
>>> a is None
True
```

Here we can see that the enhanced with block can reassign the names in scope. This even works for the locals of a function.

4.2.2 Quoted Expressions

The c quasiquoter also allows for quoted expressions. Just like the enhanced with statment, the quoted expression can use the names from the enclosing scope. For example:

```
>>> [$c|PyLong_FromLong(2)|] + 2
4
>>> a = 2
>>> [$c|PyLong_FromLong(PyLong_AsLong(a) + 2)|]
4
```

Quoted expressions are built on compound statements, a gnu extension to c. These look like:

```
int a = ({
    int b = 1;  /* This is a new block, new declarations are allowed
    int c = 2;
    b + c;  /* The final expression is the result of the block.
});
```

We need this because most quoted expressions that will return to python need to remember to incref the return. For example:

```
>>> [$c|Py_INCREF(Py_None); Py_None|] is None
True
```

We need to remember to call `Py_INCREF` or we will get a segfault somewhere in the garbage collector at interpreter shutdown.

Note: The last semicolon is optional in c quoted expression.

4.2.3 Type Conversion

Because one intended use case of the c quasiquoter is optimization, there is no implicit object conversion. All names passed from the outside scope will have type `PyObject*`. This matches the normal CPython API conventions. There are many type specific conversion functions, for example: `PyLong_AsLong` or `PyUnicode_AsUTF8`.

This is also true for the quoted expression return value. a *quasiquotes.c.CompilationError* will be raised if the final expression does not have type `PyObject*`.

4.2.4 Reference Counting

CPython uses a reference counting garbage collection strategy. This means that every `PyObject` has an `ob_refcnt` field (of type `Py_ssize_t`. This measures the number of objects that can refer to this object. Whenever an object is added to some container, the container will `Py_INCREF` the object, increasing the reference count by 1. When the object is removed from the container the container will `Py_DECREF` the object, reducing the reference count by 1. When an object with exactly 1 reference is `Py_DECREF`ed it will be destroyed immediatly by calling `((PyTypeObject*)Py_TYPE(ob))->tp_dealloc(ob)`. This will deallocate the object.

CPython documentation will also refer to the concept of borrowed references. A borrowed reference is a reference to an object that the current scope does not own. This means that the current scope is not responsible for calling `Py_DECREF` on this object. For example, when arguments are passed to a function, they are passed as a borrowed reference, if one wishes to hold onto the object, they must `Py_INCREF` it to take ownership. Some CPython API functions will return borrowed references.

Similar to the idea of borrowed reference is the idea of stealing references. This means that a function will not `Py_INCREF` the object but it will `Py_DECREF` it when it releases ownership. It is the job of the caller to ensure that they want to release ownership to the function.

quasiquotes does not help the programmer with reference counting. It is still the user's responsibility to manage the lifetimes on their objects.

4.2.5 Exceptions

When a function or quoted block raises an exception, the user should call `PyErr_SetString`, `PyErr_Format`, or one of the other functions used for setting the exception state. These will mark that a failure has occurred so that the interpreter knows which type of failure happened. This is very similar to the `raise` keyword in python.

When an exception has been set, the function should return `NULL` to show that an exception as occured. After calling most CPython API functions, the user should verify that the return is not `NULL`. Often the user should bubble the return of `NULL` up, making sure to `Py_DECREF` all of the values they had temporary ownership of.

4.2.6 Compilation Caching

Whenever a quoted statement or expression is compiled, it will create a shared object next to the python source of the file. The name of the shared object will start with `_qq_<kind>` where kind can be either `stmt` or `expr`. This marks the type of quasiquote that was used. Then it will have the name of the module it is in. After that is an md5 hash of the body of the quoted section. Finally, there is the ABI compat string, like `cpython-34m` that says that this was CPython major version 3 minor version 4 compiled with PyMalloc enabled.

The quasiquoter can also be configured to cache the generated c source code or to not cache the shared objects with the `keep_c` and `keep_so` keyword arguments to the *`c`* quasiquoter.

Every compiled chunk will be cached in memory after the quasiquote has been executed once.

Every so often you will want to cleanup stale compiled shared objects. This can be done with the *`quasiquotes.c.c.cleanup()`* method, or by executing: `python -m quasiquotes.c` Both of these accept two arguments: `path` and `recurse` defaulting to `.` and `True` respectivly. This marks where the search for cached c and shared objects should begin and if the search should recurse through subdirectories.

4.2.7 Compilation Options

The c quasiquoter accepts a keyword argument: `extra_compile_args` which should be a sequence of string to pass to `gcc`. This can be used to add include directories or link against other libraries.

4.3 fromfile

`quasiquotes.fromfile` is designed to take an existing quasiquoter and return a new quasiquoter that reads its input from a file. For example, let's write an "identity" quasiquoter that executes the body as python code.

```
from textwrap import dedent

from quasiquotes import QuasiQuoter
from quasiquotes.utils.instance import instance

@instance
class py(QuasiQuoter):
    def quote_stmt(self, code, frame, col_offset):
        exec(dedent(code), frame.f_globals, frame.f_locals)
        self.locals_to_fast(frame)

    def quote_expr(self, code, frame, col_offset):
        return eval(code, frame.f_globals, frame.f_locals)
```

We can use this silly quasiquoter as expected:

```
>>> a = 2
>>> with $py:
...     print(a + 2)
4
>>> print([$py|a + 2|])
4
```

We can now use this to inline python from another file in our function. For example, let's imagine that `other_file.py` looks like:

```
print(a + 2)
```

We can then use this in our files like:

```
>>> inlinepy = fromfile(py)  # remember, we need to bind this before use.
>>> a = 2
>>> with $inlinepy:
...     other_file.py
4
>>> [$inlinepy|other_file.py|] is None
4
True
```

4.4 Implementation

4.4.1 Tokens

quasiquotes works by hooking into the file encoding logic. Every file is marked with an encoding type, defaulting to utf-8. This is shown with the `# coding: <encoding>` coments at the top of some files. This encoding defines the functions needed to convert the raw bytes that come in from the filesystem into python `str` objects. Users are also able to register their own encoding types by providing their own conversion functions. quasiquotes sits on top of the utf-8 encoding functions; however, it tokenizes the files coming in so that it can rewrite certian patterns.

Let's look at some source code and the tokens that come out of it:

```
with $qq:
    this should not parse
    but it will
```

```
NAME('with')
ERROR(' ')
ERROR('$')
NAME('qq')
OP(':')
NEWLINE('\n')
<body>
DEDENT
```

This says we have the string 'with' followed by 2 errors. These tokens appear as `ERROR` because this would normally be an invalid token in python. The next part is the actual name of the quasiquoter you would want to use. Finally we have the colon and newline. The body is whatever sequence of tokens make up the indented region in the quasiquoter, and then we have the `DEDENT` token marking the end of the body.

By manipulating the tokens, we can change this into something that looks like:

```
cc._quote_stmt(0,'    this should not parse\n    but it will')
```

Here the `0` is the column offset of this quoted expression, and the string is the body of the context manager. The lack of space after the comma accuratly reflects the column offsets of the tokens that the quasiquotes tokenizer emits.

Note: The original indentation is preserved.

We can do this because we still have access to the raw text that makes up each line between the NEWLINE and the DEDENT.

Let's also look at the quoted expressions:

```
[$qq|this is also invalid|]
```

```
  OP('[')
  ERROR('$')
  NAME('qq')
  OP('|')
  <body>
  OP('|')
  OP(']')


Just like with quoted statements, we can rewrite this to look more like:


.. code-block:: python

   qq._quote_expr(0,'    this is also invalid')
```

Note: Indentation is also preserved in a quoted expression.

4.4.2 Runtime Lookups

An important thing to notice about the implementation is that it builds source that has method calls of a dynamic object. While we are doing static work to make the parser see the quoted block as valid python, we do *not* load the quasiquoter until the function is being executed and we have a running frame. This means that the current value for the name of the quasiquoter will be used.

4.4.3 Expressions as `QuasiQuoters`

`QuasiQuoters` are instances, so one might think that they should be able to do:

```
with $MyQQ(some_arg=some_value):
    ...
```

Unfortunately, this changes the token stream. We no longer have an `OP(':')`, `NEWLINE('\n')` following the name of the quoter. Currently, we do not detect this case and the normal python syntax error will be thrown. This is also true for quoted expressions.

4.5 Appendix

4.5.1 c

`quasiquotes.c.`**c = <quasiquotes.c.c object>**
quasiquoter for inlining c.

Parameters **keep_c** : bool, optional

Keep the generated .c files. Defaults to False.

keep_so : bool, optional

Keep the compiled .so files. Defaults to True.

extra_compile_args : iterable[str or Flag]

Extra command line arguments to pass to gcc.

Notes

You cannot pass arguments in the quasiquote syntax. You must construct a new instance of *c* and then use that as the quasiquoter. For example:

```
with $c(keep_so=False):
    Py_None;
```

is a syntax error. Instead, you must do:

```
c_no_keep_so = c(keep_so=False)
with $c_no_keep_so:
    Py_None;
```

This is because of the way the quasiquotes lexer identifies quasiquote sections.

Methods

`quote_stmt`
`quote_expr`

`c.`**`cleanup`**(*path='.'*, *recurse=True*)

Remove cached shared objects and c code generated by the c quasiquoter.

Parameters **path** : str, optional

The path to the directory that will be searched.

recurse : bool, optional

Should the search recurse through subdirectories of `path`.

Returns **removed** : list[str]

The paths to the files that were removed.

exception `quasiquotes.c.`**`CompilationError`**

An exception that indicates that gcc failed to compile the given C code.

exception `quasiquotes.c.`**`CompilationWarning`**

A warningthat indicates that gcc warned when compiling the given C code.

4.5.2 fromfile

class `quasiquotes.quasiquoter.`**`fromfile`**(*qq*)

Create a `QuasiQuoter` from an existing one that reads the body from a filename.

Parameters **qq** : QuasiQuoter

The QuasiQuoter to wrap.

Examples

```
>>> from quasiquotes.quasiquoter import fromfile
>>> from quasiquotes.c import c
>>> include_c = fromfile(c)
>>> # quote_expr on the contents of the file
>>> [$include_c|mycode.c|]
>>> # quote_stmt on the contents of the file
>>> with $include_c:
...     mycode.c
```

4.5.3 Codec

class `quasiquotes.codec.tokenizer.`**`FuzzyTokenInfo`**
A token info object that check equality only on `type` and `string`.

Parameters **type** : int

The enum for the token type.

string : str

The string represnting the token.

start, end, line : any, optional

Ignored.

class `quasiquotes.codec.tokenizer.`**`PeekableIterator`**(*stream*)
An iterator that can peek at the next `n` elements without consuming them.

Parameters **stream** : iterator

The underlying iterator to pull from.

Notes

Peeking at `n` items will pull that many values into memory until they have been consumed with `next`.

The underlying iterator should not be consumed while the `PeekableIterator` is in use.

`lookahead_iter`()
Return an iterator that yields the next element and then consumes it.

This is particularly useful for `takewhile` style functions where you want to break when some predicate is matched but not consume the element that failed the predicate.

Examples

```
>>> it = PeekableIterator(iter((1, 2, 3)))
>>> for n in it.lookahead_iter():
...     if n == 2:
...         break
>>> next(it)
2
```

peek(*n=1*)
Return the next `n` elements of the iterator without consuming them.

Parameters **n** : int

Returns **peeked** : tuple

The next `elements`

Examples

```
>>> it = PeekableIterator(iter((1, 2, 3, 4)))
>>> it.peek(2)
(1, 2)
>>> next(it)
1
>>> it.peek(1)
(2,)
>>> next(it)
2
>>> next(it)
3
```

`quasiquotes.codec.tokenizer.`**quote_expr_tokenizer**(*name*, *start*, *tok_stream*)
Tokenizer for quote_expr.

Parameters **name** : str

The name of the quasiquoter.

start : TokenInfo

The starting token.

tok_stream : iterator of TokenInfo

The token stream to pull from.

`quasiquotes.codec.tokenizer.`**quote_stmt_tokenizer**(*name*, *start*, *tok_stream*)
Tokenizer for quote_stmt.

Parameters **name** : str

The name of the quasiquoter.

start : TokenInfo

The starting token.

tok_stream : iterator of TokenInfo

The token stream to pull from.

`quasiquotes.codec.tokenizer.`**tokenize**(*readline*)
Tokenizer for the quasiquotes language extension.

Parameters **readline** : callable

A callable that returns the next line to tokenize.

`quasiquotes.codec.tokenizer.`**tokenize_bytes**(*bs*)
Tokenize a bytes object.

Parameters **bs** : bytes

The bytes to tokenize.

quasiquotes.codec.tokenizer.**tokenize_string**(*cs*)
Tokenize a str object.

Parameters **cs** : str

The string to tokenize.

quasiquotes.codec.tokenizer.**transform_bytes**(*bs*)
Run bytes through the tokenizer and emit the pure python representation.

Parameters **bs** : bytes

The bytes to transform.

Returns **transformed** : bytes

The pure python representation of bs.

quasiquotes.codec.tokenizer.**transform_string**(*cs*)
Run a str through the tokenizer and emit the pure python representation.

Parameters **cs** : str

The string to transform.

Returns **transformed** : bytes

The pure python representation of cs.

4.5.4 Utilities

class quasiquotes.utils.shell.**Executable**(*name*)
An executable from the shell.

quasiquotes.utils._traceback.**new_tb**(*frame*)
Create a traceback object starting at the given stackframe.

Parameters **frame** : frame

The frame to start the traceback from.

Returns **tb** : traceback

The new traceback object.

Notes

This function creates a new traceback object through the C-API. Use at your own risk.

q

C

E

F

L

N

P

Q

T