Syntactic macros in Python

> *Colorless green ideas sleep furiously*

— Noam Chomsky, American linguistic

What are macros?

Traditionally, macros are **substitutions** of fragments of the source code by some transformation of themselves.

This substitution is called ***macro-expansion*** and its performed by the compiler in a previous pass before compiling the actual code.

There are several forms of macros, probably most famous are *text substitution macros* in which a preprocessor **search and replace** specific text sequences.

Basic C macro

```c
#define TRUE 1
int isamacro = TRUE; // becomes isamacro = 1
```

Parametrized C macro

```c
#define max(x, y) ((x) > (y) ? (x) : (y))
int maximum = max(5+1, v);
// becomes ((5+1) > (v) ? (5+1) : (v))
```

For-in iteration protocol

```
#define for_in(T, v, c) \
for (iter<T> v = c.iter(); v; v = v.next())

for_in(int, n, intarray) {
  printf("Double of %d is %d", *n, *n * 2);
}
/* becomes
for (iter<int> n = intarray.iter(); n; n = n.next()) {
  printf("Double of %d is %d", *n, *n * 2);
}
*/
```

Text preprocessors knows nothing about the structure of the source code is replacing.

But syntactic macros does...

What are syntactic macros?

They are transformations of the **syntactic tree**. The macro is actually a function taking an **AST as input** and returning another **AST as expansion**.

Basic LISP macro

```
(defmacro when (test exp . rest)
  `(if ,test
    (progn ,exp . ,rest)))

(when nil (display "Launching missiles!\n"))
;; Expand to
;; (if nil
;;   (progn (display "Launching missiles!\n")))
```

Proposal for *Python* `log` macro

```
log[people[7].name]
# Expands to print('people[7].name:', people[7].name)
```

With sintactic macros we can **abuse** the language syntax and provide new pragmatics. I.e. **create new meaning.**

Proposal for *Python* `customliterals` macro

```
# Runtime error: `AttributeError: __exit__`
with customliterals:
  tuple is point
  print((0,0).distance((1,1)))

'''Expands to:
print( point( (0,0) ).distance( point( (1,1) ) ) )
'''
```

But syntactic macros *per se* does not allow to extend the language.

The source code must be recognized as a valid AST before expansion.

The `d` (*dice roll*) operator

```
roll = 5 d 6
# would expand in (randint(1, 6+1) for n in range(5))
# Pre-runtime error: `SyntaxError: invalid syntax`
```

Would not be cool to **use Python to expand Python**?

macropy

lihaoyi/macropy

- A macro expander in ***import-time***.
- A complete library with lots of useful macros.
- An authoring framework for creating new macros.
- Works with **CPtyhon 2.7.2**, **PyPy 2.0**
- Partial support in Python 3.x

Install & basic setup

```
# pip2 install macropy
```

```
# run.py
import macropy.activate # important!
import myprogram.py

# myprogram.py
from mymacros import macros, ...
'''Do something with macros...'''

# mymacros.py
from macropy.core.macros import *
macros = Macros() # important!
'''Define macros here'''
```

```
# Or in the Python console, instead of `activate`
import macropy.console
```

The Case macro

```
from macropy.case_classes import macros, case

@case
class Point(x, y): pass

p = Point(1, 2)
print str(p) # Point(1, 2)
print p.x    # 1
print p.y    # 2
print Point(1, 2) == Point(1, 2) # True
x, y = p
print x, y   # 1 2
```

Advanced topics about case classes in the docs.

The Quick Lambda macro

```
from macropy.quick_lambda import macros, f, _

print map(f[_ + 1], [1, 2, 3])    # [2, 3, 4]
print reduce(f[_ * _], [1, 2, 3]) # 6
```

More about quick lambdas in the documentation.

The show_expanded macro

```
from macropy.case_classes import macros, case
from macropy.tracing import macros, show_expanded

with show_expanded:
  @case
  class Point(x, y): pass
```

More introspection utilities as show_expanded in the docs.

And tons of more features:

- Lazy, String Interpolation & Tracing macros.
- MacroPEG Parser Combinator.
- Experimental pattern matching & tail-call optimization.
- PINQ, SQL integration in Python.
- Pyxl & JS snippets.
- And even more…

Writing macros

The log macro

It's quite similar to write LISP macros.

```
character = { 'name': 'Iñigo Montoya' }
# We want this:
log[character['name']]
# ...to expand into:
print 'character[\'name\'] ->', character['name']
```

Mark the module as a macro container

```python
from macropy.core.macros import *
macros = Macros()
```

Use a decorator to specify what kind of use you want for your macro

```
from macropy.core.macros import *
macros = Macros()

@macros.expr
def log(tree, **kw):
  return tree
```

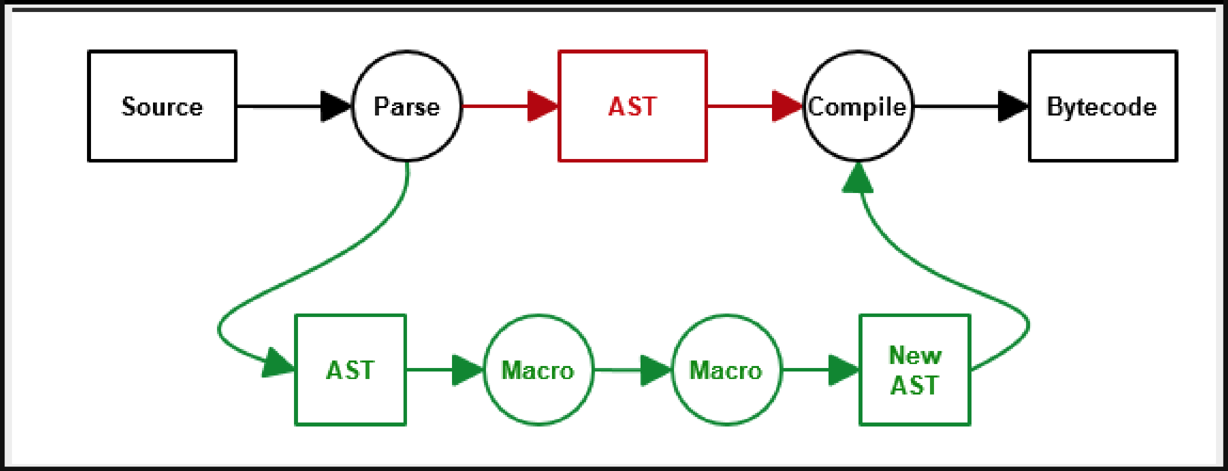
Use hygienic quasiquotes to build new ASTs avoiding the ugly AST API

```
from macropy.core.macros import *
from macropy.core.hquotes import macros, hq, ast, u
macros = Macros()

@macros.expr
def log(tree, **kw):
  label = unparse(tree) + ' ->'
  return hq[eprint(u[label], ast[tree])]

def eprint(label, target): print label, target
```

- `unparse(tree)` is a **function** returning the Python code for *tree*.
- `hq[tree]` is a **macro** that returns the AST for the Python code needed to build *tree* but preserving the macro context.
- `ast[tree]` is a **macro** used only inside *hq* to insert the AST in *tree* as part of the expression in which the *ast* macro is found.
- `u[tree]` is a **macro** used only inside *hq* to insert the AST of the result of evaluating *tree* in the macro context in the expression where the *u* macro is found. Only built-in types are supported.

More in the tutorials section.

How does it work?



macropy **intercepts the module when importing it**, expand the AST, and executes the new AST.

importing

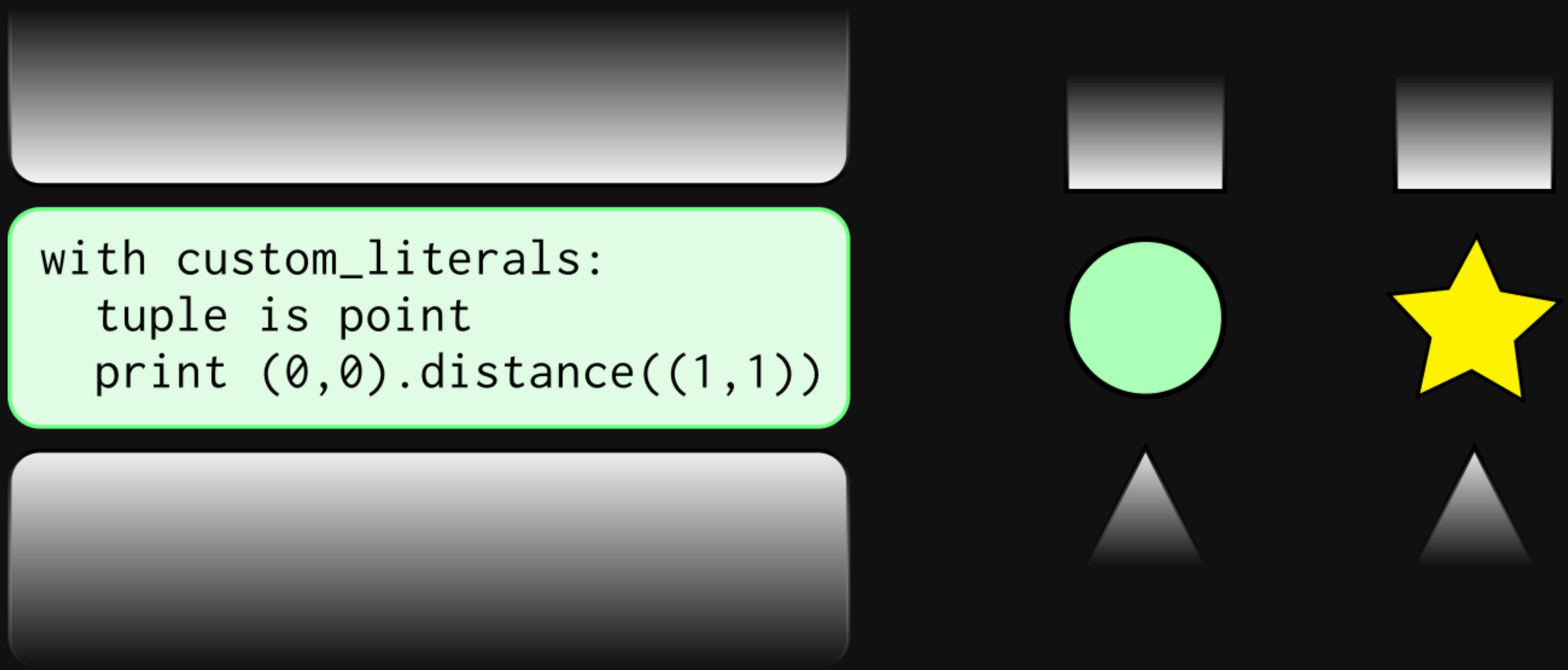
- **New Import Hooks** (PEP 0302) allows to customize *import system*.
- Import system relies on finders and loaders.
- A finder searches a module and return a loader for it.
- A loader reads and executes the module.

```
import macropy.activate
```

- That line adds a custom finder in charge of expanding the AST before executing it.

expansion I

ast.*parse()* function returns the AST for source code.

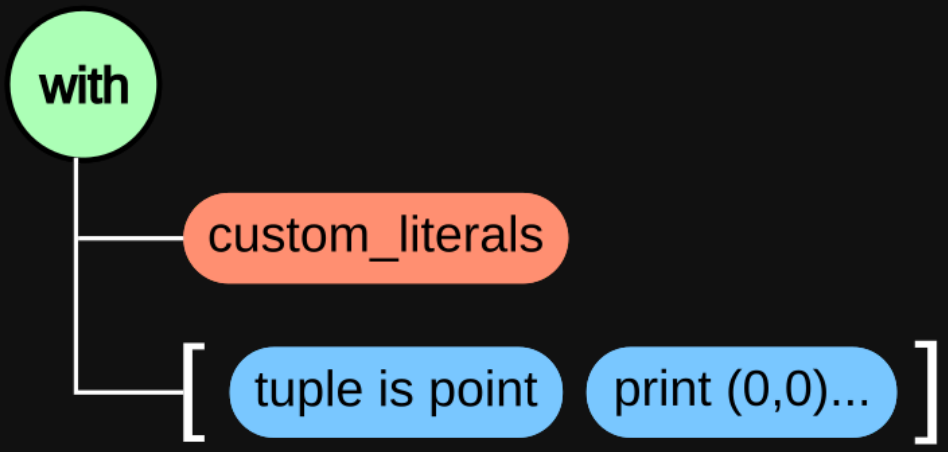


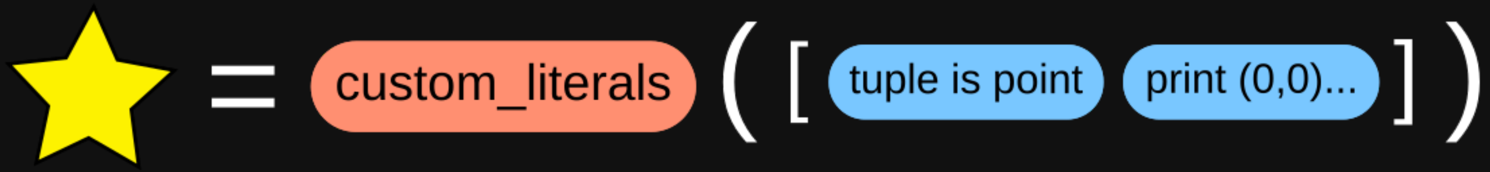
```
with custom_literals:
  tuple is point
  print (0,0).distance((1,1))
```

macropy looks for nodes representing macros.

expansion II

Found nodes are split into macro name and wrapped tree.





The macro function is executed passing the wrapped tree as parameter.

execution

Now the AST has been expanded, the custom loader executes the new AST in the module context.

mcpy

delapuente/mcpy

- Focus on **expanding** macros.
- Developed as an study case for **learning**.
- Very **small library** compared with macropy.
- No utilities for authoring.

Show me da code!

See also

- Wikipedia article about macros
- Macros: Defining Your Own
- The expansion code for macropy.

About me

me

Salvador de la Puente González

twitter

@salvadelapuente

My web sites

http://unoyunodiez.com

http://github.com/delapuente