Metaprogramming

CS 242
November 29, 2017

Today's goals

- Seeing the diversity of tools for generating code
- Understanding the use cases for metaprogramming
- Formalizing a structured taxonomy of metaprogramming

CODE

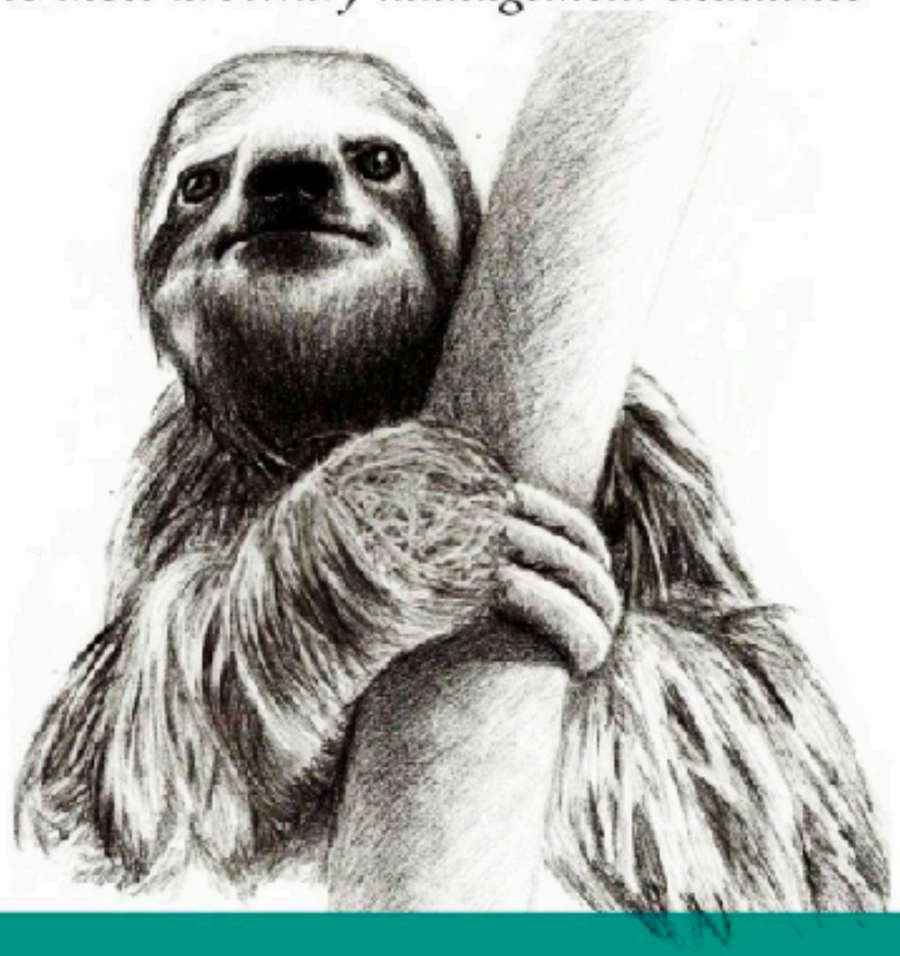
IS

DATA

"One man's program is another program's data."

Olivier Danvy

Cutting corners to meet arbitrary management deadlines



Essential

Copying and Pasting from Stack Overflow

O'REILLY®

The Practical Developer
@ThePracticalDev

C preprocessor: programs as strings

- Programmatic, iterative find + replace
  - Composable! Macros within macros

- Expressiveness: add constructs to the language
  - "Polymorphic" functions without void*
  - Foreach loops

- Performance: inline everything
  - Don't leave it up to the compiler

- Correctness: easy to break (not *hygienic*)
  - Variable names clash
  - Incorrect precedence

C++ templates: sugaring "polymorphism"

- Templates provide illusion of polymorphism
  - Thinly veiled mechanism for static dispatch
  - Not type safe
  - Sugar is more convenient than C

- Waaaay more to templates than meets the eye
  - Who put a Turing-complete language runtime in my compiler?

Rust: many kinds of metaprogramming

- Macros: find+replace with hygiene
  - Principled pattern matching: expressions vs. statements
  - No accidental variable use (partially thanks to lexical scoping)

- Custom derive: Rust code introspecting struct fields
  - Function : struct –> trait impl
  - First example of staging: running Rust code at compile time
  - Specifically generates trait implementation of a struct

- Procedural macros: Rust code doing anything
  - Function : tokens –> Rust code
  - Highly expressive
  - Not composable

Staging

Finite levels of evaluation

Scripting languages close the loop

- **Fused compiler/interpreter = runtime metaprogramming**
  - **eval/dostring/loadfile/etc.**
  - **Varying support for quotations**

- **"Infinitely" staged (no limit on theoretical recursion)**

- **Reflection is commonplace, but still meta programming**
  - **Getting the type of a variable**
  - **Inspecting the fields of a class**
  - **Generating new classes at runtime**

Higher order functions = macros?

```
let add_one = List.map ~f:(fun x -> x + 1)
```

OCaml

Functions = macros?

```
def map(f):
    return lambda l: [f(x) for x in l]

@map
def add_one(n):
    return n + 1

print(add_one([1, 2, 3]))  # [2, 3, 4]
```

Python

Chapter 9. Metaprogramming

One of the most important mantras of software development is "don't repeat yourself." That is, any time you are faced with a problem of creating highly repetitive code (or cutting or pasting source code), it often pays to look for a more elegant solution. In Python, such problems are often solved under the category of "metaprogramming." In a nutshell, metaprogramming is about creating functions and classes whose main goal is to manipulate code (e.g., modifying, generating, or wrapping existing code). The main features for this include decorators, class decorators, and metaclasses. However, a variety of other useful topics—including signature objects, execution of code with exec(), and inspecting the internals of functions and classes—enter the picture. The main purpose of this chapter is to explore various metaprogramming techniques and to give examples of how they can be used to customize the behavior of Python to your own whims.

Putting a Wrapper Around a Function

Problem

You want to put a wrapper layer around a function that adds extra processing (e.g., logging, timing, etc.).

Solution

If you ever need to wrap a function with extra code, define a decorator function. For example:

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
    '''
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper
```

Decorators vs. the Decorator Pattern

First, you need to understand that the word "decorator" was used with some trepidation in Python, because there was concern that it would be completely confused with the *Decorator* pattern from the Design Patterns book. At one point other terms were considered for the feature, but "decorator" seems to be the one that sticks.

Indeed, you can use Python decorators to implement the *Decorator* pattern, but that's an extremely limited use of it. Python decorators, I think, are best equated to macros.

History of Macros

The macro has a long history, but most people will probably have had experience with C preprocessor macros. The problems with C macros were (1) they were in a different language (not C) and (2) the behavior was sometimes bizarre, and often inconsistent with the behavior of the rest of C.

Both Java and C# have added *annotations*, which allow you to do some things to elements of the language. Both of these have the problems that (1) to do what you want, you sometimes have to jump through some enormous and untenable hoops, which follows from (2) these annotation features have their hands tied by the bondage-and-discipline (or as Martin Fowler gently puts it: "Directing") nature of those languages.

In a slightly different vein, many C++ programmers (myself included) have noted the generative abilities of C++ templates and have used that feature in a macro- like fashion.

Many other languages have incorporated macros, but without knowing much about it I will go out on a limb and say that Python decorators are similar to Lisp macros in power and possibility.

Programming

Python decorators: metaprogramming with style

DECORATORS FUNCTIONAL PROGRAMMING METAPROGRAMMING OOP PYTHON

By Leonardo Giordani • Published on 23/04/2015

Homogenous metaprogramming

Language generates code in the same language

Terra: metaprogrammable C in Lua

- Research project out of Pat Hanrahan's group at Stanford
- Code generation important for perf in scripting
  - Generating the host language isn't sufficient
  - Lower level targets are hard to generate/interoperate
- Terra makes it easy to:
  - Generate C-ish code without using strings
  - Mix Lua values into C-ish
  - Compile and run generated C-ish code

Taxonomy of metaprogramming

- Goal: generation vs. analysis
- Representation: strings vs. syntax trees vs. quotes
- Execution mode: staged vs. interpreted
- Output: homogeneous vs. heterogeneous

Additional topics

- Lisps: Common Lisp, Emacs Lisp, Racket, Clojure, Scheme
  - Originator of macros… in the 1960s!
  - Homoiconicity: the concrete syntax = the abstract syntax

- Typed metaprogramming
  - Rust's types are just syntactic ("this the syntax for a function")
  - What about "this is function syntax that returns type int"?
  - Originated with MetaML, still active area of research
  - Lightweight Modular Staging (LMS) in Scala

Summary

- Metaprogramming used for performance or expressiveness
  - "Abstraction without regret" – compile away general APIs for perf
  - Paper over missing features like closures or polymorphism
  - Drawback is often usability (debugging, error messages, no formalisms), hard to ensure correctness in macro definition
- Macro systems generate code in a multitude of ways
  - C preprocessor/C++ templates/Rust macros: find+replace in templates
  - Rust custom derive/procedural macros: staged Rust code
- Scripting languages get most metaprogramming for free
  - Main problem is generating/interoperating with efficient code