Lab 4: Macro

March 25, 2015

1 Get the code

The base code used in the labs is in the directory */home/TDDA69/Labs/Lab4/src*. To get the code and start working on it, in your home directory:

```
cp -r /home/TDDA69/Labs/Lab4/src $HOME/Lab4
```

This will copy the skeleton for the *Lab4* assignments, you can now find them in the directory *$HOME/Lab4*. In the rest of the document and in other lab, we will refer to this directory as *dir_to_lab4*.

2 Decorator

2.1 Bound checking decorator

In the lecture, a bound checking decorator for a single argument function was presented:

```python
def bound_checking_decorator(min, max):
  def make_decorator(func):
    def decorator(x):
      if(x < min or x > max):
        raise Exception()
      return func(x)
    return decorator
  return make_decorator

@bound_checking_decorator(0, float('inf'))
def fib(n):
  return n if n < 2 else fib(n-2) + fib(n-1)
```

The goal is to extend it for functions with multiple arguments, such as:

```python
@bound_checking_decorator(-1, 1, -2, 2, -3, 3)
def func(a, b, c)
  return a + b + c
```

2.2 Run the test

```
tdda69_lab4_tests dir_to_lab4 bound_checking_decorator
```

2.3 Timing and logging

Create a decorator that log function call and timings:

```python
@logtiming
def fib(n):
  return n if n < 2 else fib(n-2) + fib(n-1)
```

You can access the function name with the *__name__* member:

```python
print(fib.__name__) # prints "fib"
```

2.4 Run the test

```
tdda69_lab4_tests dir_to_lab4 log_timing_decorator
```

3 Template

For this assignment, your goal is to develop a more general templating system than the one presented in the lecture:

```python
def apply_template(template):
  def t(f):
    f_ast            = ast.parse(inspect.getsource(f)).body[0]
    body_node    = f_ast.body[0]
    template_ast = ast.parse(inspect.getsource(template))
    template_ast.body[0].args = f_ast.args
    class T(ast.NodeTransformer):
      def visit_Expr(self, node):
        if(node.value.id == '__body__'):
          return body_node
        else:
          return node
    exec(compile(T().visit(template_ast), __file__, mode='exec'))
    return locals()[template_ast.body[0].name]

  return t


def my_template():
  for x in range(1,10):
    __body__
  return v

@apply_template(my_template)
def func(v):
    v = v * x
```

The idea is to define a template as function (for instance *func1* or *func2* below). Then the *apply_template* function will be able to replace part of that template.

```python
def func_body(v):
    v = v * x

def func_return():
  return v

@apply_template("__body__", func_body, "__return__", func_return)
def func1(v):
  for x in range(1,10):
    __body__
  __return__

@apply_template("__body__", func_body, "__return__", func_return)
def func2(v):
  x = 2
  __body__
  __return__
```

After applying the template, the functions *func1* and *func2* should look like the following:

```python
def func1(v):
  for x in range(1,10):
    v = v * x
  return v

def func2(v):
  x = 2
  v = v * x
  return v
```

3.1 Run the test

```
tdda69_lab4_tests dir_to_lab4 apply_template
```