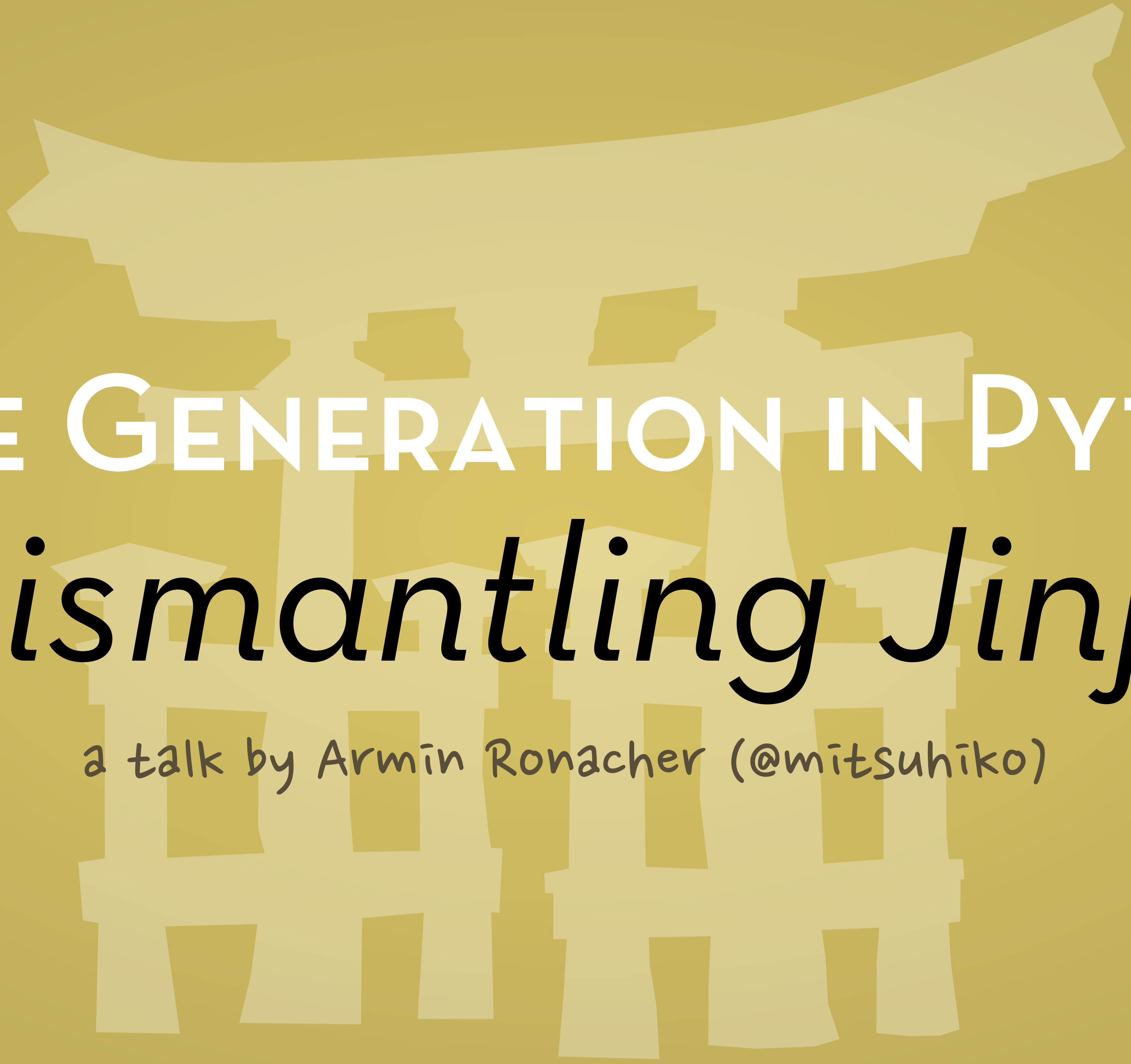


Code Generation in Python

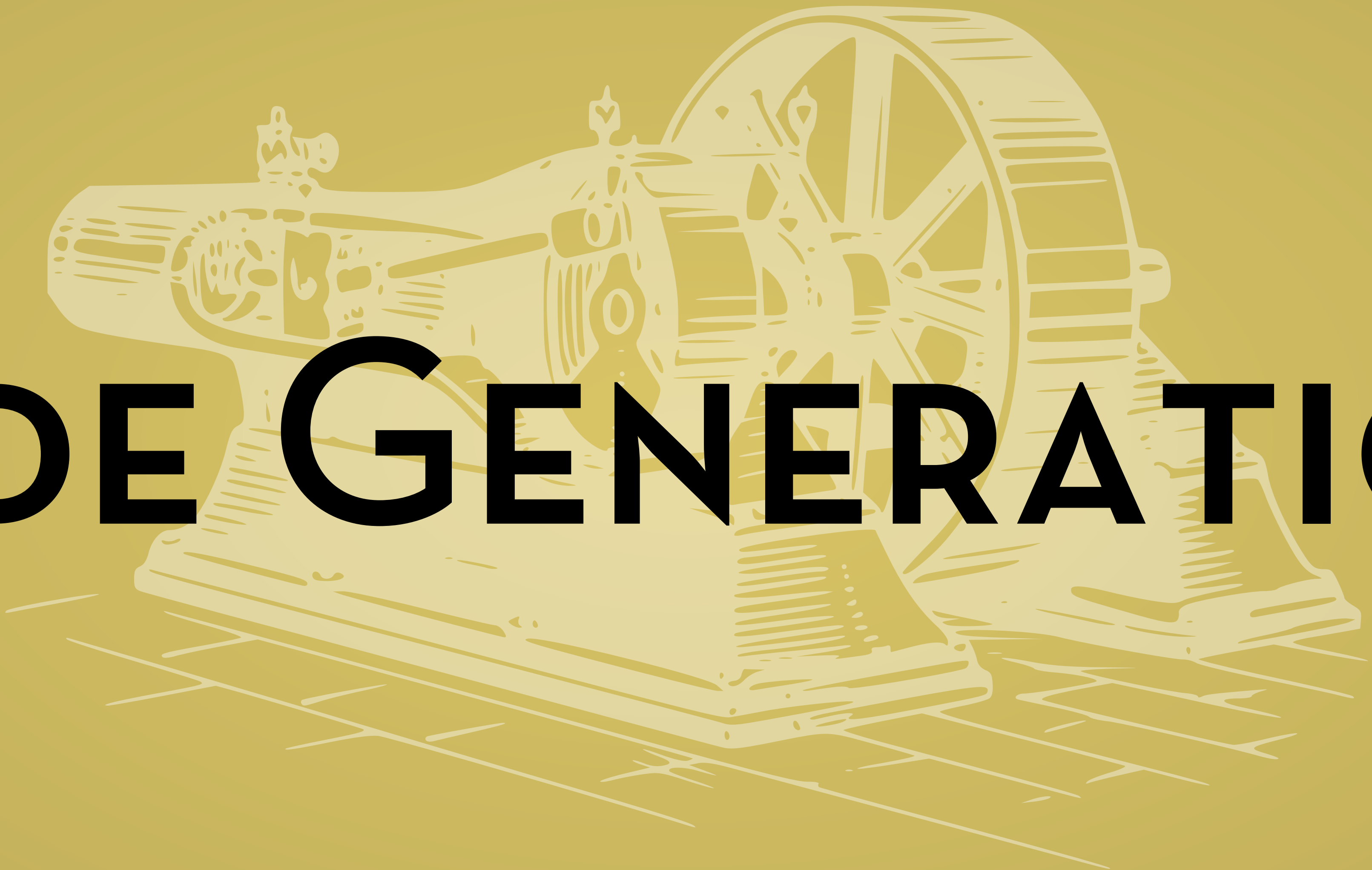
Dismantling Jinja

a talk by Armin Ronacher (@mitsuhiko)

bit.ly/codegeneration

Discuss this presentation, give feedback

Code Generation?



eval is evil

Or is it?

Why is **eval** evil?

Security & Performance

Security

Code Injection

Namespace pollution

Performance

No bytecode

Code makes code that code runs

So: Why?

No suitable alternatives

because of this:

use responsibly

EVAL

101

Compile

```
>>> code = compile('a = 1 + 2', '<string>', 'exec')
>>> code
<code object <module> at 0x1004d5120, file "<string>", line 1>
```

Eval

```
>>> ns = {}
>>> exec code in ns
>>> ns['a']
3
```

AST #1

```
>>> import ast
>>> ast.parse('a = 1 + 2')
<_ast.Module object at 0x1004fd250>
>>> code = compile(_, '<string>', 'exec')
```

AST #2

```
>>> n = ast.Module([
...     ast.Assign([ast.Name('a', ast.Store())],
...                ast.BinOp(ast.Num(1), ast.Add(),
...                          ast.Num(2)))]))
>>> ast.fix_missing_locations(n)
>>> code = compile(n, '<string>', 'exec')
```

Recap

No strings passed to eval()/exec

Explicit compilation to bytecode

Execution in explicit namespace

TEMPLATE ENGINE ARCHITECTURE

Jinja

Overview

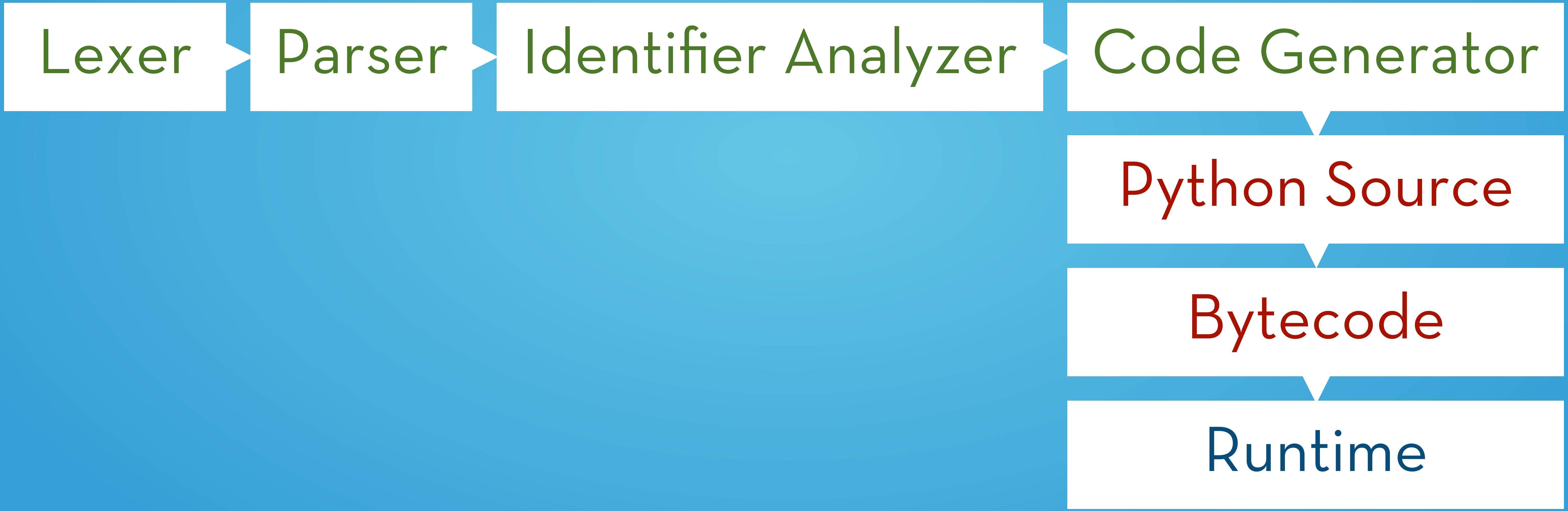
2nd Iteration

Generates Python Code

Python Semantics

Different Scoping

Pipeline



Lexer

Parser

Identifier Analyzer

Code Generator

Python Source

Bytecode

Runtime

Complexities

Different Scoping

WSGI & Generating

Debug-ability

Restricted Environments

Input

```
<ul>
{% for item in seq %}
    <li>{{ item }}</li>
{% endfor %}
</ul>
```

Behavior

```
print "<ul>"
for each item in the variable seq
  push the scope
  print "<li>"
  print the value of item and escape it as necessary
  print "</li>"
  pop the scope
print "</ul>"
```

Naive:

```
write(u'<ul>')
for _tmp in context['seq']:
    context.push({'item': _tmp})
    write(u'<li>')
    write(autoescape(context['item']))
    write(u'</li>')
    context.pop()
write(u'</ul>')
```

Actual:

```
l_seq = context.resolve('seq')
write(u'<ul>')
for l_item in l_seq:
    write(u'<li>')
    write(autoescape(l_item))
    write(u'</li>')
write(u'</ul>')
```

?

INTRODUCTION TO COMPILATION

The Art of Code Generation

Low Level
versus
High Level

Low Level

Code Generation

```
a = 1 + 2
```

```
  2           0 LOAD_CONST               1 (1)
              3 LOAD_CONST               2 (2)
              6 BINARY_ADD
              7 STORE_FAST               0 (a)
```

High Level

Code Generation

```
a = 1 + 2
```

```
Assign(targets=[Name(id='a', ctx=Store())],
       value=BinOp(left=Num(n=1),
                   op=Add(),
                   right=Num(n=2)))]
```

Building Blocks

Bytecode
Abstract Syntax Trees
Sourcecode

Bytecode

Undocumented

Does not work on GAE

Implementation Specific

AST

More Limited

Easier to Debug

Does not segfault the Interpreter

Source

Works always

Very Limited

Hard to Debug without Hacks

The Tale of Two Pieces of Code

(very similar pieces of code)

Fast

```python
def foo():
    a = 0
    for x in xrange(100):
        a += x
    print a

foo()
```

Slower

```
a = 0
for x in xrange(100):
    a += x
print a
```

?

Slower

```
  2           0 LOAD_CONST               0 (0)
              3 STORE_NAME               0 (a)

  3           6 SETUP_LOOP              30 (to 39)
              9 LOAD_NAME                1 (xrange)
             12 LOAD_CONST               1 (100)
             15 CALL_FUNCTION            1
             18 GET_ITER
        >>   19 FOR_ITER                16 (to 38)
             22 STORE_NAME               2 (x)

  4          25 LOAD_NAME                0 (a)
             28 LOAD_NAME                2 (x)
             31 INPLACE_ADD
             32 STORE_NAME               0 (a)
             35 JUMP_ABSOLUTE           19
        >>   38 POP_BLOCK

  5     >>   39 LOAD_NAME                0 (a)
             42 PRINT_ITEM
             43 PRINT_NEWLINE
```

Fast

```
2           0 LOAD_CONST               1 (0)
            3 STORE_FAST               0 (a)

3           6 SETUP_LOOP              30 (to 39)
            9 LOAD_GLOBAL              0 (xrange)
           12 LOAD_CONST               2 (100)
           15 CALL_FUNCTION            1
           18 GET_ITER
      >>   19 FOR_ITER                16 (to 38)
           22 STORE_FAST               1 (x)

4          25 LOAD_FAST                0 (a)
           28 LOAD_FAST                1 (x)
           31 INPLACE_ADD
           32 STORE_FAST               0 (a)
           35 JUMP_ABSOLUTE           19
      >>   38 POP_BLOCK

5     >>   39 LOAD_FAST                0 (a)
           42 PRINT_ITEM
           43 PRINT_NEWLINE
```

Fast

```
2            0 LOAD_CONST               1 (0)
             3 STORE_FAST               0 (a)

3            6 SETUP_LOOP              30 (to 39)
             9 LOAD_GLOBAL              0 (xrange)
            12 LOAD_CONST               2 (100)
            15 CALL_FUNCTION            1
            18 GET_ITER
      >>    19 FOR_ITER                16 (to 38)
            22 STORE_FAST               1 (x)

4           25 LOAD_FAST                0 (a)
            28 LOAD_FAST                1 (x)
            31 INPLACE_ADD
            32 STORE_FAST               0 (a)
            35 JUMP_ABSOLUTE           19
      >>    38 POP_BLOCK

5     >>    39 LOAD_FAST                0 (a)
            42 PRINT_ITEM
            43 PRINT_NEWLINE
```

Example

```
>>> def foo():
...  a = 42
...  locals()['a'] = 23
...  return a
...
>>> foo()
42
```

A STORY ABOUT

SEMANTICS

Remember

```
print "<ul>"
for each item in the variable seq
  push the scope
  print "<li>"
  print the value of item and escape it as necessary
  print "</li>"
  pop the scope
print "</ul>"
```

That's not how Python works

… so how do you generate code for it?

Tracking

Keep tracks of identifiers
emulate desired semantics

Scopes

Context in Jinja2 is a Data Source

Context in Django is a Data Store

Source

```
<ul>
{% for item in seq %}
    {% include "item.html" %}
{% endfor %}
</ul>
```

Code

```
l_seq = context.resolve('seq')
write(u'<ul>')
for l_item in l_seq:
    t1 = env.get_template('other.html')
    for event in yield_from(t1, context, {'item': l_item})
        yield event
write(u'</ul>')
```

What happens in the include …

… stays in the include

Impossible

```
@contextfunction
def get_users_and_store(context, var='users'):
    context[var] = get_all_users()
    return u''
```

PRACTICAL EXAMPLES

Source

```
<ul class=navigation>
{% for item in sequence %}
  <li>{{ item }}</li>
{% endfor %}
</ul>
```

Generated

```
def root(context):
    l_sequence = context.resolve('sequence')
    yield u'\n<ul class=navigation>\n'
    l_item = missing
    for l_item in l_sequence:
        yield u'\n  <li>%s</li>' % (
            escape(l_item),
        )
    l_item = missing
    yield u'\n</ul>'
```

Source

```
<ul class=navigation>
{% for item in sequence %}
  <li>{{ loop.index }}: {{ item }}</li>
{% endfor %}
</ul>
```

Generated

```
def root(context):
    l_sequence = context.resolve('sequence')
    yield u'\n<ul class=navigation>\n'
    l_item = missing
    for l_item, l_loop in LoopContext(l_sequence):
        yield u'\n  <li>%s: %s</li>\n' % (
            escape(environment.getattr(l_loop, 'index')),
            escape(l_item),
        )
    l_item = missing
    yield u'\n</ul>'
```

Source

```
<ul class=navigation>
{% for item in sequence %}
  <li>{{ loop.index }}: {{ item }}</li>
{% endfor %}
</ul>
<p>Item: {{ item }}
```

Generated

```
def root(context):
    l_item = context.resolve('item')
    l_sequence = context.resolve('sequence')
    yield u'\n<ul class=navigation>\n'
    t_1 = l_item
    for l_item, l_loop in LoopContext(l_sequence):
        yield u'\n  <li>%s: %s</li>\n' % (
            escape(environment.getattr(l_loop, 'index')),
            escape(l_item),
        )
    l_item = t_1
    yield u'\n</ul>\n<p>Item: '
    yield escape(l_item)
```

Source

```
{% extends "layout.html" %}
{% block body %}
  <h1>Hello World!</h1>
{% endblock %}
```

Generated

```
def root(context):
    parent_template = environment.get_template('layout.html', None)
    for name, parent_block in parent_template.blocks.iteritems():
        context.blocks.setdefault(name, []).append(parent_block)
    for event in parent_template.root_render_func(context):
        yield event

def block_body(context):
    if 0: yield None
    yield u'\n  <h1>Hello World!</h1>\n'

blocks = {'body': block_body}
```

Source

```
<!doctype html>
{% block body %}{% endblock %}
```

Generated

```
def root(context):
    yield u'<!doctype html>\n'
    for event in context.blocks['body'][0](context):
        yield event

def block_body(context):
    if 0: yield None

blocks = {'body': block_body}
```

Source

```
{% extends "layout.html" %}
{% block title %}Hello | {{ super() }}{% endblock %}
```

Generated

```
def root(context):
    parent_template = environment.get_template('layout.html', None)
    for name, parent_block in parent_template.blocks.iteritems():
        context.blocks.setdefault(name, []).append(parent_block)
    for event in parent_template.root_render_func(context):
        yield event

def block_title(context):
    l_super = context.super('title', block_title)
    yield u'Hello | '
    yield escape(context.call(l_super))

blocks = {'title': block_title}
```

WHY DOES JINJA DO

why
… manual code generation?

Originally the only option

AST compilation was new in 2.6

GAE traditionally did not allow it

why … generators instead of buffer.append()

Required for WSGI streaming

unless greenlets are in use

Downside: StopIteration :-(

why
... map "var_x" to "l_var_x"

Reversible to debugging purposes

Does not clash with internals

see templatetk for better approach

HOW DOES

JINJA DO

how
… does automatic escaping work

Markup object

Operator overloading

Compile-time and Runtime

Const

```
<h1>{{ "<strong>Hello World!</strong>" }}</h1>
```

```
def root(context):
    yield u'<h1>&lt;strong&gt;Hello World!&lt;/strong&gt;</h1>'
```

Runtime

```
<h1>{{ variable }}</h1>
```

```
def root(context):
    l_variable = context.resolve('variable')
    yield u'<h1>%s</h1>' % (
        escape(l_variable),
    )
```

Control #1

```
{% autoescape false %}<h1>{{ variable }}</h1>{% endautoescape %}
```

```
def root(context):
    l_variable = context.resolve('variable')
    t_1 = context.eval_ctx.save()
    context.eval_ctx.autoescape = False
    yield u'<h1>%s</h1>' % (
        l_variable,
    )
    context.eval_ctx.revert(t_1)
```

Control #2

```
{% autoescape flag %}<h1>{{ variable }}</h1>{% endautoescape %}
```

```
def root(context):
    l_variable = context.resolve('variable')
    l_flag = context.resolve('flag')
    t_1 = context.eval_ctx.save()
    context.eval_ctx.autoescape = l_flag
    yield u'%s%s%s' % (
        (context.eval_ctx.autoescape and escape or to_string)((context.eval_ctx.autoescape and Markup or identity)(u'<h1>')),
        (context.eval_ctx.autoescape and escape or to_string)(l_variable),
        (context.eval_ctx.autoescape and escape or to_string)((context.eval_ctx.autoescape and Markup or identity)(u'</h1>')),
    )
    context.eval_ctx.revert(t_1)
```

how … far does the Markup object go?

All operators are overloaded

All string operations are safe

necessary due to operator support

Example

```
>>> from markupsafe import Markup
>>> Markup('<em>%s</em>') % '<insecure>'
Markup(u'<em>&lt;insecure&gt;</em>')
>>> Markup('<em>') + '<insecure>' + Markup('</em>')
Markup(u'<em>&lt;insecure&gt;</em>')
>>> Markup('<em>Complex&nbsp;value</em>').striptags()
u'Complex\xa0value'
```

how … do undefined values work

Configurable

Replaced by special object

By default one level of silence

Example

```
>>> from jinja2 import Undefined
>>> unicode(Undefined(name='missing_var'))
u''
>>> unicode(Undefined(name='missing_var').attribute)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
UndefinedError: 'missing_var' is undefined
```

Q&A

@mitsuhiko

http://lucumr.pocoo.org/

armin.ronacher@active-4.com

FIRETEAM™

Oh hai. We're hiring

http://fireteam.net/careers