

FAT Python

New static optimizer for CPython 3.6



FOSDEM 2016, Bruxelles



redhat®

Victor Stinner
vstinner@redhat.com

Agenda



- (1) Python is slow
- (2) Guards, specialization & AST
- (3) Optimizations
- (4) Implementation
- (5) Coming next

Agenda



(1) Python is slow

(1) Python is slow



- CPython is slower than C, “compiled” language
- Slower than JavaScript and its fast JIT compilers

(1) Faster Python



- PyPy JIT
- Pyston JIT (LLVM)
- Pyjion JIT (CoreCLR)
- Numba JIT (LLVM), specific to numpy
- Cython static optimizer

(1) New optimizer?



- None replaced CPython yet
- PyPy is not always faster than Cpython
- CPython remains the reference implementation for new features
- Many libraries rely on CPython “implementation details” like the Python C API

(1) Simplified goal



```
def func():  
    return len("abc")
```



```
def func():  
    return 3
```

(1) Problem



Everything is mutable in Python:

- Builtin functions
- Function code
- Global variables
- etc.

(1) Problem



Replace builtin `len()` function:

```
builtins.len = lambda obj: "mock!"  
print(len("abc"))
```

Output:

mock!

(1) Constraints



- Respect the Python semantics
- Don't break applications
- Don't require to modify the application source code

Agenda



(2) Guards, specialization & AST

(2) Guards



- Efficient optimizations relying on assumptions
- Guards check these assumptions **at runtime**
- Example: was the builtin `len()` function modified?

(2) Namespace guards

Core feature of the Python language:

- Module: global variables
- Function: local variables
- Class: `type.method()`
- Instance: `obj.attr`
- etc.

(2) Namespace guards



- Namespaces are Python dict
- Technical challenge: make guard faster than dict lookups
- Solution: PEP 509, add a version to dict

(2) Specialize code



- Optimize the code with assumptions: “specialized” code
- Use guards to only call the specialized code if assumptions are still correct
- Example: specialize code if `x` and `y` parameters are `int`

(2) Specialize code



Pseudo code:

```
def call(func, args):  
    if check_guards(args):  
        # nothing changed  
        code = func.__specialized__  
    else:  
        # len() was replaced  
        code = func.__code__  
    execute(code, args)
```

(2) Peephole optimizer



Optimize bytecode:

- Constant folding
- Dead code elimination
- Optimize jumps
- Written in C, very limited

(2) AST



Abstract Syntax Tree:

.py file → tokens → **AST** → bytecode

AST of `len("abc")`:

```
Call(func=Name(id='len', ctx=Load()),  
      args=[Str(s='abc')])
```

(2) AST optimizer



```
import ast
```

```
class Optimizer(ast.NodeTransformer):  
    def visit_Call(self, node):  
        return ast.Num(n=3)
```

Agenda



(3) Optimizations

(3) Call builtin functions



`len('abc')` → 3

`int('123')` → 123

`pow(2, 8)` → 256

`frozenset('abc')` → `frozenset('abc')`
built at runtime constant

Need a guard on the called function

(3) Simplify iterables



for x in **range(3)** → for x in **(0, 1, 2)**

for x in **[7, 9]** → for x in **(7, 9)**

for x in **{}** → for x in **()**

Replacing `range(...)` requires a guard on the `range()` function

(3) Loop unrolling



```
x = 1  
print(x)
```

```
for x in (1, 2, 3):  
    print(x)
```

→

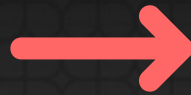
```
x = 2  
print(x)
```

```
x = 3  
print(x)
```


(3) Copy constants

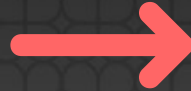


```
x = 1  
print(x)
```



```
x = 1  
print(1)
```

```
x = 2  
print(x)
```



```
x = 2  
print(2)
```

```
x = 3  
print(x)
```



```
x = 3  
print(3)
```

(3) Constant folding



`+(5) → 5`

`x in [1, 2, 3] → x in (1, 2, 3)`

`(7,) * 3 → (7, 7, 7)`

`'python2.7[:-2] → 'python'`

`'P' in 'Python' → True`

`[5, 9, 20][1] → 9`

(3) Copy to constants



Python code:

```
def func(obj):  
    return len(obj)
```



Python code:

```
def func(obj):  
    return len(obj)
```

Bytecode:

```
LOAD_GLOBAL 'len'  
...
```




Bytecode:

```
LOAD_CONST 'len'  
...
```


Need a guard on
len() builtin

(3) Remove dead code



`if test:`  `if not test:`
 `pass` `else_block`
`else:`
 `else_block`

`if 0:`  `pass`
 `body_block`

`return result`  `return result`
`dead_code`

Agenda



(4) Implementation

(4) Merged changes



New AST node `ast.Constant` to simplify optimizers. Converted to `ast.Constant` by the optimizer:

- `ast.NameConstant`: `None`, `True`, `False`
- `ast.Num`: `int`, `float`, `complex`
- `ast.Str`: `str`
- `ast.Bytes`: `bytes`
- `ast.Tuple` (if items are constants): `tuple`

(4) Merged changes



Support negative line number delta:

```
for x in (1, 2, 3): # line 1
    print(x)         # line 2 (+1)
```



```
x = 1      # line 1
print(x)   # line 2 (+1)
x = 2      # line 1 (-1)
print(x)   # line 2 (+1)
```

(4) Merged changes



Support tuple and frozenset constants in the compiler:

obj in {1, 2, 3}



obj in **frozenset**({1, 2, 3})

(4) PEP 509: dict version

- Add a version to dicts
- Version is incremented at every change
- Version is unique for all dicts
- Guard compares the version: avoid dict lookup if nothing changed

(4) PEP 509: dict version



```
def check(self):
    version = dict_get_version(self.dict)
    if version == self.version:
        return True    # Fast-path: no lookup

    value = self.dict.get(self.key, UNSET)
    if value is self.value:
        self.version = version
        return True

    return False    # the key was modified
```

(4) PEP 510: Specialize



- Add `PyFunction_Specialize()` C function
- Specialized code can be a code object (bytecode) or any callable object
- Modify `Python/ceval.c` to check guards and use specialized code

(4) PEP 510: Specialize



Specialized code using:

- New AST optimizers: **fatoptimizer**
- Cython
- Pythran
- Numba
- etc.

(4) PEP 510: Specialize



```
def func():  
    return chr(65)
```

```
def fast_func():  
    return 'A'
```

```
fat.specialize(  
    func,  
    fast_func.__code__,  
    [fat.GuardBuiltins('chr')])
```

(4) PEP 511: Transformer



- Add `-O` command line option
- Add `sys.set_code_transformers()`
- A code transformer can modify the bytecode and/or the AST

Agenda

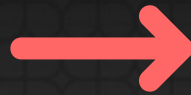


(5) Coming next

(5) Remove unused vars

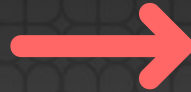


```
x = 1  
print(1)
```



```
print(1)
```

```
x = 2  
print(2)
```



```
print(2)
```

```
x = 3  
print(3)
```



```
print(3)
```

(5) Copy globals



```
KEYS = {2: 55}
```

```
KEYS = {2: 55}
```

```
def func():  
    return KEYS[2]
```



```
def func():  
    return 55
```

Need a guard on the KEYS global

(5) Function inlining



```
def inc(x):  
    return x+1
```

```
def inc(x):  
    return x+1
```

```
y = inc(3)
```



```
y = 3 + 1
```

Need a guard on the inc() function

(5) Profiling



- Run the application in a profiler
- Record types of function parameters
- Generate type annotations
- Use these types to specialize the code

Questions?



http://faster-cpython.rtfld.org/fat_python.html