FAT Python

New static optimizer for CPython 3.6

EuroPython 2016, Bilbao
Agenda

1. Python is slow
2. Guards, specialization & AST
3. Optimizations
4. Implementation
5. Coming next
(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
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
def func():
    return len("abc")

def func():
    return 3
Everything is mutable in Python:
- Built-in functions
- Function code
- Global variables
- etc.
(1) Problem

Replace builtin `len()` function:

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

Output:

`mock!`
1. My previous attempts

- astoptimizer: simple AST optimizer
- registervm: register-based bytecode

Bad feedback, both broke deeply the Python semantics, too many assumptions without using guards.
(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

Core feature of the Python language:
- Module: global variables
- Function: local variables
- Class: type.method()
- Instance: obj.attr
- etc.

Namespace
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 \( \text{int} \)
(2) Specialize code

Pseudo code:

```python
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
Abstract Syntax Tree:
.py file → tokens → AST → bytecode

AST of `len("abc")`:
```
Call(func=Name(id='len', ctx=Load()),
     args=[Str(s='abc')])
```
import ast

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

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

x = 1
print(x)
x = 2
print(x)
x = 3
print(x)
(3) Copy constants

<table>
<thead>
<tr>
<th>x = 1</th>
<th>print(x)</th>
<th>x = 1</th>
<th>print(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 2</td>
<td>print(x)</td>
<td>x = 2</td>
<td>print(2)</td>
</tr>
<tr>
<td>x = 3</td>
<td>print(x)</td>
<td>x = 3</td>
<td>print(3)</td>
</tr>
</tbody>
</table>
(3) Constant folding

\[(5) \rightarrow 5\]
\[x \text{ in } [1, 2, 3] \rightarrow x \text{ in } (1, 2, 3)\]
\[(7,) * 3 \rightarrow (7, 7, 7)\]
\['python2.7'[: -2] \rightarrow 'python'\]
\['P' \text{ in } 'Python' \rightarrow True\]
\[[5, 9, 20][1] \rightarrow 9\]
(3) **Copy to constants**

Python code:
```python
def func(obj):
    return len(obj)
```

Bytecode:
- `LOAD_GLOBAL` 'len'
- ...

Need a guard on `len()` builtin
(3) Remove dead code

```python
if test:
    pass
else:
    else_block

if 0:
    body_block

return result

dead_code

→

if not test:
    else_block

→

pass

→

return result
```
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 constant): tuple
Support negative line number delta:

```
for x in (50, 100):
    print(x)
```

```
x = 50
print(x)
x = 100
print(x)
```

Merged changes:

```
for x in (50, 100):
    # line 1
    print(x)  # line 2 (+1)

x = 50  # line 1
print(x)  # line 2 (+1)
x = 100  # line 1 (-1)
print(x)  # line 2 (+1)
```
(4) Merged changes

Support tuple and frozenset constants in the compiler:

```python
obj in {1, 2, 3}
```

```python
obj in frozenset({1, 2, 3})
```
Add a version to Python dict

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

Specialize 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')])
Add `-o` command line option

Add `sys.set_code_transformers()`

A code transformer can modify the bytecode and/or the AST
Good feedback on the 3 PEPs

Requirement: speedup on applications

Today only faster on microbenchmarks

Need 6 months to implement more optimizations
(5) Coming next
(5) Remove unused vars

```python
x = 1
print(1)

x = 2
print(2)

x = 3
print(3)
```

→

```python
print(1)
print(2)
print(3)
```
(5) Copy globals

```python
KEYS = {2: 55}
def func():
    return KEYS[2]
```

Need a guard on the KEYS global
(5) Function inlining

```python
def incr(x):
    return x + 1

y = incr(3)  # y = 3 + 1
```

Need a guard on the `incr()` function
(5) Profiling

- Run the application in a profiler
- Record types of function parameters
- Generate type annotations
- Use these types to specialize the code
Spawn multiple processes
Compute average and standard deviation
Store all individual run timings as JSON
Command line tool to display, compare and analyze data

http://perf.rtfd.io/
Three-year-old Cambodian boy Oeun Sambat hugs his best friend, a four-metre (13.1 feet) long female python named Chamreun or 'Lucky' in the village of Sit Tbow on May 18, 2003.
Questions?

http://faster-cpython.rtfd.org/fat_python.html
Sources, copyrights


- Red Hat and Python are registered trademarks.