fatoptimizer Documentation

Release 0.3

Victor Stinner

Jun 29, 2017

Contents

`fatoptimizer` is a static optimizer for Python 3.6 using function specialization with guards. It is implemented as an AST optimizer.

Optimized code requires the *fat module* at runtime if at least one function is specialized.

Links:

- fatoptimizer documentation (this documentation)
- fatoptimizer project at GitHub (code, bug tracker)
- fatoptimizer project at the Python Cheeseshop (PyPI) (download releases)
- FAT Python
- fatoptimizer tests running on the Travis-CI

The `fatoptimizer` module requires a Python 3.6 patched with PEP 510 "Specialize functions with guards" and PEP 511 "API for code transformers" patches.

CHAPTER 1

Table Of Contents

fatoptimizer module

fatoptimizer API

Warning: The API is not stable yet.

`fatoptimizer.__version__` is the module version string (ex: `'0.3'`).

optimize(*tree*, *filename*, *config*)
Optimize an AST tree. Return the optimized AST tree.

pretty_dump(*node*, *annotate_fields=True*, *include_attributes=False*, *lineno=False*, *indent=' '*)
Return a formatted dump of the tree in *node*. This is mainly useful for debugging purposes. The returned string will show the names and the values for fields. This makes the code impossible to evaluate, so if evaluation is wanted *annotate_fields* must be set to False. Attributes such as line numbers and column offsets are not dumped by default. If this is wanted, *include_attributes* can be set to True.

class Config
Configuration of the optimizer.

See *fatoptimizer configuration*.

class FATOptimizer(*config*)
Code transformers for `sys.set_code_transformers()`.

class OptimizerError
Exception raised on bugs in the optimizer.

Installation

The *fatoptimizer module* requires a Python 3.6 patched with PEP 510 "Specialize functions with guards" and PEP 511 "API for code transformers" patches.

Type:

```
pip install fatoptimizer
```

Manual installation:

```
python3.6 setup.py install
```

Optimized code requires the *fat module* at runtime if at least one function is specialized.

Configuration

It is possible to configure the AST optimizer per module by setting the `__fatoptimizer__` variable. Configuration keys:

- `enabled` (`bool`): set to `False` to disable all optimization (default: true)
- `constant_propagation` (`bool`): enable *constant propagation* optimization? (default: true)
- `constant_folding` (`bool`): enable *constant folding* optimization? (default: true)
- `copy_builtin_to_constant` (`bool`): enable *copy builtin functions to constants* optimization? (default: false)
- `inlining` (`bool`): enable *function inlining* optimization? (default: false)
- `remove_dead_code` (`bool`): enable *dead code elimination* optimization? (default: true)
- maximum size of constants:
  - `max_bytes_len`: Maximum number of bytes of a text string (default: 128)
  - `max_int_bits`: Maximum number of bits of an integer (default: 256)
  - `max_str_len`: Maximum number of characters of a text string (default: 128)
  - `max_seq_len`: Maximum length in number of items of a sequence like tuples (default: 32). It is only a preliminary check: `max_constant_size` still applies for sequences.
  - `max_constant_size`: Maximum size in bytes of other constants (default: 128 bytes), the size is computed with `len(marshal.dumps(obj))`
- `replace_builtin_constant` (`bool`): enable *replace builtin constants* optimization? (default: true)
- `simplify_iterable` (`bool`): enable *simplify iterable optimization*? (default: true)
- `unroll_loops`: Maximum number of loop iteration for loop unrolling (default: `16`). Set it to `0` to disable loop unrolling. See *loop unrolling* and *simplify comprehension* optimizations.

Example to disable all optimizations in a module:

```
__fatoptimizer__ = {'enabled': False}
```

Example to disable the constant folding optimization:

```
__fatoptimizer__ = {'constant_folding': False}
```

See the *Config* class.

Run tests

Type:

```
tox
```

You may need to install or update tox:

```
pip3 install -U tox
```

Run manually tests:

```
python3 test_fatoptimizer.py
```

There are also integration tests which requires a Python 3.6 with patches PEP 509, PEP 510 and PEP 511. Run integration tests:

```
python3.6 -X fat test_fat_config.py
python3.6 -X fat test_fat_size.py
```

fat module

The `fat` module is a Python extension module (written in C) implementing fast guards. The *fatoptimizer optimizer* uses `fat` guards to specialize functions. `fat` guards are used to verify assumptions used to specialize the code. If an assumption is no more true, the specialized code is not used.

The `fat` module is required to run code optimized by `fatoptimizer` if at least one function is specialized.

- fat project at GitHub
- fat project at the Python Cheeseshop (PyPI)

The `fat` module requires a Python 3.6 patched with PEP 509 "Add a private version to dict" and PEP 510 "Specialize functions with guards" patches.

fat module API

Warning: The API is not stable yet.

`fat.__version__` is the module version string (ex: `'0.3'`).

Functions

replace_consts(*code*, *mapping*)
Create a copy of the code object with replaced constants:

```
new_consts = tuple(mapping.get(const, const) for const in consts)
```

specialize(*func*, *code*, *guards*)
Specialize a Python function: add a specialized code with guards.

code must be a callable or code object, *guards* must be a non-empty sequence of guards.

get_specialized(*func*)
Get the list of specialized codes with guards. Return a list of `(code, guards)` tuples.

See the PEP 510 "Specialize functions with guards" for the API of `specialize()` and `get_specialized()`.

Guard types

class GuardArgType(*arg_index*, *arg_types*)
Check the type of the nth argument. *arg_types* must be a sequence of types.

The guard check fails temporarily (returns `1`) if the argument has a different type or if the guard is checked with less than *arg_index* parameters.

The guard does not support keyword parameters yet. If the guard is checked with keyword parameters, it fails temporarily (returns `1`).

Attributes:

arg_index
Index of the argument (`int`). Read-only attribute.

arg_types
List of accepted types for the argument: list of types. Read-only property.

Keep a strong reference to *arg_types* types.

class GuardBuiltins(**names*)
Subtype of *GuardDict*.

Watch for:

- globals of the current frame (`frame.f_globals`)
- `globals()[name]` for all *names*.
- builtins of the current frame (`frame.f_builtins`)
- `builtins.__dict__[name]` for all *names*

The guard initialization fails if `builtins.__dict__[name]` was replaced after `fat` was imported, or if `globals()[name]` already exists.

In addition to *GuardDict* checks and *GuardBuiltins.guard_globals* checks, the guard check always fails (returns `2`) if the frame builtins changed.

Attributes:

guard_globals
The *GuardGlobals* used to watch for the global variables. Read-only attribute.

Keep a strong references to the builtin namespace (`builtins.__dict__` dictionary), to the global namespace (`globals()` dictionary), to *names* and to existing builtin symbols called *names* (`builtins.__dict__[name]` for all *names*).

class GuardDict(*dict*, **keys*)
Watch for `dict[key]` for all *keys*.

The guard check always fails (returns `2`) if at least one key of *keys* was modified.

keys strings are interned: see *sys.intern*.

Attributes:

dict
Watched dictionary (`dict`). Read-only attribute.

keys
List of watched dictionary keys: list of `str`. Read-only property.

Keep a strong references to *dict*, to *keys* and to existing dictionary values (`dict[key]` for all keys).

class GuardFunc(*func*)
Watch for the code object (`func.__code__`) of a Python function.

The guard check always fails (returns `2`) if the function code was replaced.

`GuardFunc(func)` must not be used to specialize `func`. Replacing the code object of a function already removes its specialized code, no need to add a guard.

Attributes:

code
Watched code object. Read-only attribute.

func
Watched function. Read-only attribute.

Keep a strong references to *func* and to `func.__code__`.

class GuardGlobals(**names*)
Subtype of *GuardDict*.

In addition to *GuardDict* checks, the guard check always fails (returns `2`) if the frame globals changed.

Watch for:

- globals of the current frame (`frame.f_globals`)
- `globals()[name]` for all *names*.

Keep a strong references to the global namespace (`globals()` dictionary), to *names* and to existing global variables called *names* (`globals()[name]` for all *names*).

Guard helper functions

guard_type_dict(*type*, *attrs*)
Create `GuardDict(type.__dict__, attrs)` but access the real type dictionary, not `type.__dict` which is a read-only proxy.

Watch for `type.attr` (`type.__dict__[attr]`) for all *attrs*.

Installation

The `fat` module requires a Python 3.6 patched with PEP 509 "Add a private version to dict" and PEP 510 "Specialize functions with guards" patches.

Type:

```
pip install fat
```

Manual installation:

```
python3.6 setup.py install
```

Run tests

Type:

```
./runtests.sh
```

Changelog

- Version 0.3
- Change constructors:
  - `GuardDict(dict, keys)` becomes `GuardDict(dict, *keys)`
  - `GuardBuiltins(name)` becomes `GuardBuiltins(*names)`
  - `GuardGlobals(name)` becomes `GuardGlobals(*names)`
- `GuardFunc(func)` init function now raises a `ValueError` if it is used to specialize `func`.
- `GuardDict(keys)` now interns *keys* strings.
- 2016-01-22: Version 0.2
- *`GuardBuiltins`* now also checks the builtins and the globals of the current frame. In practice, the guard fails if it is created in a namespace and checked in a different namespace.
- Add a new *`GuardGlobals`* type which replaces the previous `guard_globals()` helper function (removed). The guard check checks if the frame globals changed or not.
- Guards are now tracked by the garbage collector to handle correctly a reference cycle with GuardGlobals which keeps a reference to the module namespace (`globals()`).
- Fix type of dictionary version for 32-bit platforms: `PY_UINT64_T`, not `size_t`.
- Fix *`GuardFunc`* traverse method: visit also the `code` attribute.
- Implement a traverse method to *`GuardBuiltins`* to detect correctly reference cycles.
- 2016-01-18: Version 0.1
  - GuardBuiltins check remembers if guard init failed
  - Rename *`GuardGlobals`* to `guard_globals()`
  - Rename `GuardTypeDict` to `guard_dict_type()`
- 2016-01-13: First public release, version 0.0.

Optimizations

Optimizations

Implementated optimizations:

- *Call pure builtins*
- *Loop unrolling*
- *Simplify comprehensions*
- *Constant propagation*

- *Constant folding*
- *Replace builtin constants*
- *Dead code elimination*
- *Copy builtin functions to constants*
- *Simplify iterable*
- *Function inlining*

Call pure builtins

Call pure builtin functions at compilation: replace the call with the result in the specialized bytecode, add guards on the called builtin functions.

The optimization is disabled when the builtin function is modified or if a variable with the same name is added to the global namespace of the function.

The optimization on the builtin `NAME` requires two guards:

- `NAME` key in builtin namespace
- `NAME` key in global namespace

Example:

Original	Specialized
`def func():` `    return len("abc")`	`def func():` `    return 3`

Loop unrolling

`for i in range(3): ...` and `for i in (1, 2, 3): ...` are unrolled. By default, only loops with 16 iterations or less are optimized.

Note: If `break` and/or `continue` instructions are used in the loop body, the loop is not unrolled.

Configuration option: `unroll_loops`.

See also:

Read the Wikipedia article on loop unrolling.

tuple example

Example with a tuple.

Original	Loop unrolled

`def func():`
`    for i in ("a", "b"):`
`        print(i)`
	`def func():`
	`    i = "a"`
	`    print(i)`
	
	`    i = "b"`
	`    print(i)`

No guard is required. The function has no specialized bytecode, the optimization is done directly on the function.

Original bytecode:

```
.      0 SETUP_LOOP              14 (to 17)
       3 LOAD_CONST               3 (('hello', 'world'))
       6 GET_ITER

>>     7 FOR_ITER                 6 (to 16)
      10 STORE_FAST               0 (i)

      13 JUMP_ABSOLUTE            7
>>    16 POP_BLOCK

>>    17 LOAD_CONST               0 (None)
      20 RETURN_VALUE
```

fatoptimizer bytecode:

```
LOAD_CONST    1 ("hello")
STORE_FAST    0 (i)

LOAD_CONST    2 ("world")
STORE_FAST    0 (i)

LOAD_CONST    0 (None)
RETURN_VALUE
```

range example

Example of a loop using `range()`.

Original	Loop unrolled

`def func():`
`    for i in range(2):`
`        print(i)`
	`def func():`
	`    i = 0`
	`    print(i)`
	
	`    i = 1`
	`    print(i)`

The specialized bytecode requires two *guards*:

- `range` builtin variable
- `range` global variable

Combined with *constant propagation*, the code becomes even more interesting:

```
def func():
    i = 0
    print(0)

    i = 1
    print(1)
```

Note: Since replacing `range()` requires a specialization with guard, the optimization is only implemented at function level.

Simplify comprehensions

Simplify list-comprehension, set-comprehension and dict-comprehension. Optimization similar to *Loop unrolling*, but applied to comprehensions.

Examples (combined with *Constant folding*):

Comprehension	Code	Simplified
List-comprehension	`[i for i in (1, 2, 3)]`	`[1, 2, 3]`
Set-comprehension	`{i*2 for i in "abc"}`	`{"aa", "bb", "cc"}`
Dict-comprehension	`{i : i * 2 for i in (1, 2, 3)}`	`{1: 2, 2: 4, 3: 6}`

Configuration option: `unroll_loops`.

Constant propagation

Propagate constant values of variables.

Original	Constant propagation

`def func()`
`    x = 1`
`    y = x`
`    return y`
	`def func()`
	`    x = 1`
	`    y = 1`
	`    return 1`

Configuration option: `constant_propagation`.

See also:

Read the Wikipedia article on copy propagation.

Constant folding

Compute simple operations at the compilation:

- arithmetic operations:

  - `a+b`, `a-b`, `a*b`, `a/b`: int, float, complex
  - `+x`, `-x`, `~x`: int, float, complex
  - `a//b`, `a%b`, `a**b`: int, float
  - `a<<b`, `a>>b`, `a&b`, `a|b`, `a^b`: int
- comparison, tests:
  - `a < b`, `a <= b`, `a >= b`, `a > b`
  - `a == b`, `a != b`: don't optimize bytes == str
  - `obj in seq`, `obj not in seq`: for bytes, str, tuple `seq`
  - `not x`: int
- str: `str + str`, `str * int`
- bytes: `bytes + bytes`, `bytes * int`
- tuple: `tuple + tuple`, `tuple * int`
- str, bytes, tuple, list: `obj[index]`, `obj[a:b:c]`
- dict: `obj[index]`
- replace `x in list` with `x in tuple` if list only contains constants
- replace `x in set` with `x in frozenset` if set only contains constants
- simplify tests:

Code	Constant folding
not(x is y)	x is not y
not(x is not y)	x is y
not(obj in seq)	obj not in seq
not(obj not in seq)	obj in seq

Note: `not (x == y)` is not replaced with `x != y` because `not x.__eq__(y)` can be different than `x.__ne__(y)` for deliberate reason Same rationale for not replacing `not(x < y)` with `x >= y`. For example, `math.nan` overrides comparison operators to always return `False`.

Examples of optimizations:

Code	Constant folding
-(5)	-5
+5	5
x in [1, 2, 3]	x in (1, 2, 3)
x in {1, 2, 3}	x in frozenset({1, 2, 3})
'Python' * 2	'PythonPython'
3 * (5,)	(5, 5, 5)
'python2.7'[:-2]	'python2'
'P' in 'Python'	True
9 not in (1, 2, 3)	True
[5, 9, 20][1]	9

Configuration option: `constant_folding`.

See also:

Read the Wikipedia article on constant folding.

Replace builtin constants

Replace `__debug__` constant with its value.

Configuration option: `replace_builtin_constant`.

Dead code elimination

Remove the dead code.

Examples:

Code	Dead code removed

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

`if 1:` `    body_block`	`body_block`
`if 0:` `    body_block`	`pass`

`if False:`
`    body_block`
`else:`
`    else_block`
	`else_block`

`while 0:` `    body_block`	`pass`

`while 0:`
`    body_block`
`else:`
`    else_block`
	`else_block`

`...`
`return ...`
`dead_code_block`
	`...`
	`return ...`

`...`
`raise ...`
`dead_code_block`
	`...`
	`raise ...`

`try:`
`    pass`
`except ...:`
`    ...`
	`pass`

`try:`
`    pass`
`except ...:`
`    ...`
`else:`
`    else_block`
	`else_block`

`try:`
`    pass`
`except ...:`
`    ...`
`else:`
`    else_block`
`finally:`
`    final_block`
	`try:`
	`   else_block`
	`finally:`
	`   final_block`

Note: If a code block contains `continue`, `global`, `nonlocal`, `yield` or `yield from`, it is not removed.

Configuration option: `remove_dead_code`.

See also:

Read the Wikipedia article on Dead code elimination.

Copy builtin functions to constants

Opt-in optimization (disabled by default) to copy builtin functions to constants.

Example with a function simple:

```
def log(message):
    print(message)
```

Bytecode	Specialized bytecode

`LOAD_GLOBAL 0 (print)`
`LOAD_FAST 0 (message)`
`CALL_FUNCTION 1 (1 positional, 0 keyword pair)`
`POP_TOP`
`LOAD_CONST 0 (None)`
`RETURN_VALUE`
	`LOAD_CONST 1 (<built-in function print>)`
	`LOAD_FAST 0 (message)`
	`CALL_FUNCTION 1 (1 positional, 0 keyword pair)`
	`POP_TOP`
	`LOAD_CONST 0 (None)`
	`RETURN_VALUE`

The first `LOAD_GLOBAL` instruction is replaced with `LOAD_CONST`. `LOAD_GLOBAL` requires to lookup in the global namespace and then in the builtin namespaces, two dictionary lookups. `LOAD_CONST` gets the value from a C array, O(1) lookup.

The specialized bytecode requires two *guards*:

- `print` builtin variable
- `print` global variable

The `print()` function is injected in the constants with the `func.patch_constants()` method.

The optimization on the builtin `NAME` requires two guards:

- `NAME` key in builtin namespace
- `NAME` key in global namespace

This optimization is disabled by default because it changes the *Python semantics*: if the copied builtin function is replaced in the middle of the function, the specialized bytecode still uses the old builtin function. To use the optimization on a project, you may have to add the following *configuration* at the top of the file:

```
__fatoptimizer__ = {'copy_builtin_to_constant': False}
```

Configuration option: `copy_builtin_to_constant`.

See also:

- codetransformer: `@asconstants(len=len)` decorator replaces lookups to the `len` name with the builtin `len()` function

- Thread on python-ideas mailing list: Specifying constants for functions by Serhiy Storchaka, propose to add `const len=len` (or alternatives) to declare a constant (and indirectly copy a builtin functions to constants)

Simplify iterable

Try to replace literals built at runtime with constants. Replace also range(start, stop, step) with a tuple if the range fits in the *configuration*.

When `range(n)` is replaced, two guards are required on `range` in builtin and global namespaces and the function is specialized.

This optimization helps *loop unrolling*.

Examples:

Code	Simplified iterable
`for x in range(3): ...`	`for x in (0, 1, 2): ...`
`for x in {}: ...`	`for x in (): ...`
`for x in [4, 5. 6]: ...`	`for x in (4, 5, 6): ...`

Configuration option: `simplify_iterable`.

Function inlining

Replace a function call site with the body of the called function.

Note: The implementation is currently experimental and so disabled by default.

Original code	Function inlining

`def g():`
`  return 42`
 	
`def f():`
`  return g(x) + 3`
	`def g():`
	`  return 42`
	
	`def f():`
	`  return 42 + 3`

Configuration option: `inlining`.

See also:

Read the Wikipedia article on Inline expansion.

Comparison with the peephole optimizer

The CPython peephole optimizer only implements a few optimizations: *constant folding*, *dead code elimination* and optimizations of jumps. fatoptimizer implements more *optimizations*.

The peephole optimizer doesn't support *constant propagation*. Example:

```
def f():
    x = 333
    return x
```

Regular bytecode		fatoptimizer bytecode	
LOAD_CONST	1 (1)	LOAD_CONST	1 (333)
STORE_FAST	0 (x)	STORE_FAST	0 (x)
LOAD_FAST	0 (x)	LOAD_CONST	1 (333)
RETURN_VALUE		RETURN_VALUE	

The *constant folding optimization* of the peephole optimizer keeps original constants. For example, `"x" + "y"` is replaced with `"xy"` but `"x"` and `"y"` are kept. Example:

```
def f():
    return "x" + "y"
```

Regular constants	fatoptimizer constants
`(None, 'x', 'y', 'xy')`: 4 constants	`(None, 'xy')`: 2 constants

The peephole optimizer has a similar limitation even when building tuple constants. The compiler produces AST nodes of type `ast.Tuple`, the tuple items are kept in code constants.

Python semantics and Limitations

fatoptimizer bets that the Python code is not modified when modules are loaded, but only later, when functions and classes are executed. If this assumption is wrong, fatoptimizer changes the semantics of Python.

Python semantics

It is very hard, to not say impossible, to implementation and keep the exact behaviour of regular CPython. CPython implementation is used as the Python "standard". Since CPython is the most popular implementation, a Python implementation must do its best to mimic CPython behaviour. We will call it the Python semantics.

fatoptimizer should not change the Python semantics with the default configuration. Optimizations modifting the Python semantics must be disabled by default: opt-in options.

As written above, it's really hard to mimic exactly CPython behaviour. For example, in CPython, it's technically possible to modify local variables of a function from anywhere, a function can modify its caller, or a thread B can modify a thread A (just for fun). See Everything in Python is mutable for more information. It's also hard to support all introspections features like `locals()` (`vars()`, `dir()`), `globals()` and `sys._getframe()`.

Builtin functions replaced in the middle of a function

fatoptimizer uses *guards* to disable specialized function when assumptions made to optimize the function are no more true. The problem is that guard are only called at the entry of a function. For example, if a specialized function ensures that the builtin function `chr()` was not modified, but `chr()` is modified during the call of the function, the specialized function will continue to call the old `chr()` function.

The *copy builtin functions to constants* optimization changes the Python semantics. If a builtin function is replaced while the specialized function is optimized, the specialized function will continue to use the old builtin function. For this reason, the optimization is disabled by default.

Example:

```
def func(arg):
    x = chr(arg)
```

```
    with unittest.mock.patch('builtins.chr', result='mock'):
        y = chr(arg)

    return (x == y)
```

If the *copy builtin functions to constants* optimization is used on this function, the specialized function returns `True`, whereas the original function returns `False`.

It is possible to work around this limitation by adding the following *configuration* at the top of the file:

```
__fatoptimizer__ = {'copy_builtin_to_constant': False}
```

But the following use cases works as expected in FAT mode:

```
import unittest.mock

def func():
    return chr(65)

def test():
    print(func())
    with unittest.mock.patch('builtins.chr', return_value="mock"):
        print(func())
```

Output:

```
A
mock
```

The `test()` function doesn't use the builtin `chr()` function. The `func()` function checks its guard on the builtin `chr()` function only when it's called, so it doesn't use the specialized function when `chr()` is mocked.

Guards on builtin functions

When a function is specialized, the specialization is ignored if a builtin function was replaced after the end of the Python initialization. Typically, the end of the Python initialization occurs just after the execution of the `site` module. It means that if a builtin is replaced during Python initialization, a function will be specialized even if the builtin is not the expected builtin function.

Example:

```
import builtins

builtins.chr = lambda: mock

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

In this example, the `func()` is optimized, but the function is *not* specialize. The internal call to `func.specialize()` is ignored because the `chr()` function was replaced after the end of the Python initialization.

Guards on type dictionary and global namespace

For other guards on dictionaries (type dictionary, global namespace), the guard uses the current value of the mapping. It doesn't check if the dictionary value was "modified".

Tracing and profiling

Tracing and profiling works in FAT mode, but the exact control flow and traces are different in regular and FAT mode. For example, *loop unrolling* removes the call to `range(n)`.

See `sys.settrace()` and `sys.setprofiling()` functions.

Expected limitations

Function inlining optimization makes debugging more complex:

- sys.getframe()
- locals()
- pdb
- etc.
- don't work as expected anymore

Bugs, shit happens:

- Missing guard: specialized function is called even if the "environment" was modified

FAT python! Memory vs CPU, fight!

- Memory footprint: loading two versions of a function is memory uses more memory
- Disk usage: .pyc will be more larger

Possible worse performance:

- guards adds an overhead higher than the optimization of the specialized code
- specialized code may be slower than the original bytecode

Benchmarks

fatoptimizer is not ready for macro benchmarks. Important optimizations like function inlining are still missing. See the *fatoptimizer TODO list*.

See *Microbenchmarks*.

The Grand Unified Python Benchmark Suite

Project hosted at https://hg.python.org/benchmarks

2016-01-22, don't specialized nested functions anymore:

```
$ time python3 ../benchmarks/perf.py ../default/python ../fatpython/python 2>&1|tee␣
↪log
INFO:root:Automatically selected timer: perf_counter
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3`
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3` 1 time
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3`
```

```
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3` 1 time
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_chameleon_v2.py -
↪n 50 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_chameleon_v2.py -n␣
↪50 --timer perf_counter`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_django_v3.py -n␣
↪50 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_django_v3.py -n 50 -
↪-timer perf_counter`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle pickle`
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle pickle`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle unpickle`
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle unpickle`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json_v2.py -n 50 -
↪-timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_json_v2.py -n 50 --
↪timer perf_counter`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json.py -n 50 --
↪timer perf_counter json_load`
INFO:root:Running `../default/python ../benchmarks/performance/bm_json.py -n 50 --
↪timer perf_counter json_load`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_nbody.py -n 50 --
↪timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_nbody.py -n 50 --
↪timer perf_counter`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_regex_v8.py -n 50␣
↪--timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_regex_v8.py -n 50 --
↪timer perf_counter`
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_tornado_http.py -
↪n 100 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_tornado_http.py -n␣
↪100 --timer perf_counter`
[ 1/10] 2to3...
[ 2/10] chameleon_v2...
[ 3/10] django_v3...
[ 4/10] fastpickle...
[ 5/10] fastunpickle...
[ 6/10] json_dump_v2...
[ 7/10] json_load...
[ 8/10] nbody...
[ 9/10] regex_v8...
[10/10] tornado_http...

Report on Linux smithers 4.2.8-300.fc23.x86_64 #1 SMP Tue Dec 15 16:49:06 UTC 2015␣
↪x86_64 x86_64
Total CPU cores: 8

### 2to3 ###
7.232935 -> 7.078553: 1.02x faster

### chameleon_v2 ###
Min: 5.740738 -> 5.642322: 1.02x faster
```

```
Avg: 5.805132 -> 5.669008: 1.02x faster
Significant (t=5.61)
Stddev: 0.17073 -> 0.01766: 9.6699x smaller

### fastpickle ###
Min: 0.448408 -> 0.454956: 1.01x slower
Avg: 0.450220 -> 0.469483: 1.04x slower
Significant (t=-8.28)
Stddev: 0.00364 -> 0.01605: 4.4102x larger

### fastunpickle ###
Min: 0.546227 -> 0.582611: 1.07x slower
Avg: 0.554405 -> 0.602790: 1.09x slower
Significant (t=-6.05)
Stddev: 0.01496 -> 0.05453: 3.6461x larger

### regex_v8 ###
Min: 0.042338 -> 0.043736: 1.03x slower
Avg: 0.042663 -> 0.044073: 1.03x slower
Significant (t=-4.09)
Stddev: 0.00173 -> 0.00172: 1.0021x smaller

### tornado_http ###
Min: 0.260895 -> 0.274085: 1.05x slower
Avg: 0.265663 -> 0.277511: 1.04x slower
Significant (t=-11.26)
Stddev: 0.00988 -> 0.00360: 2.7464x smaller

The following not significant results are hidden, use -v to show them:
django_v3, json_dump_v2, json_load, nbody.

real        20m7.994s
user        19m44.016s
sys 0m22.894s
```

2016-01-21:

```
$ time python3 ../benchmarks/perf.py ../default/python ../fatpython/python
INFO:root:Automatically selected timer: perf_counter
[ 1/10] 2to3...
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3`
INFO:root:Running `../fatpython/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3` 1 time
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3`
INFO:root:Running `../default/python ../benchmarks/lib3/2to3/2to3 -f all ../
↪benchmarks/lib/2to3` 1 time
[ 2/10] chameleon_v2...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_chameleon_v2.py -
↪n 50 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_chameleon_v2.py -n␣
↪50 --timer perf_counter`
[ 3/10] django_v3...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_django_v3.py -n␣
↪50 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_django_v3.py -n 50 -
↪-timer perf_counter`
```

```
[ 4/10] fastpickle...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle pickle`
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle pickle`
[ 5/10] fastunpickle...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle unpickle`
INFO:root:Running `../default/python ../benchmarks/performance/bm_pickle.py -n 50 --
↪timer perf_counter --use_cpickle unpickle`
[ 6/10] json_dump_v2...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json_v2.py -n 50 -
↪-timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_json_v2.py -n 50 --
↪timer perf_counter`
[ 7/10] json_load...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_json.py -n 50 --
↪timer perf_counter json_load`
INFO:root:Running `../default/python ../benchmarks/performance/bm_json.py -n 50 --
↪timer perf_counter json_load`
[ 8/10] nbody...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_nbody.py -n 50 --
↪timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_nbody.py -n 50 --
↪timer perf_counter`
[ 9/10] regex_v8...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_regex_v8.py -n 50␣
↪--timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_regex_v8.py -n 50 --
↪timer perf_counter`
[10/10] tornado_http...
INFO:root:Running `../fatpython/python ../benchmarks/performance/bm_tornado_http.py -
↪n 100 --timer perf_counter`
INFO:root:Running `../default/python ../benchmarks/performance/bm_tornado_http.py -n␣
↪100 --timer perf_counter`

Report on Linux smithers 4.2.8-300.fc23.x86_64 #1 SMP Tue Dec 15 16:49:06 UTC 2015␣
↪x86_64 x86_64
Total CPU cores: 8

### 2to3 ###
6.969972 -> 7.362033: 1.06x slower

### chameleon_v2 ###
Min: 5.686547 -> 5.945011: 1.05x slower
Avg: 5.731851 -> 5.976754: 1.04x slower
Significant (t=-21.46)
Stddev: 0.06645 -> 0.04580: 1.4511x smaller

### fastpickle ###
Min: 0.489443 -> 0.448850: 1.09x faster
Avg: 0.518914 -> 0.458638: 1.13x faster
Significant (t=6.48)
Stddev: 0.05688 -> 0.03304: 1.7218x smaller

### fastunpickle ###
Min: 0.598339 -> 0.559612: 1.07x faster
Avg: 0.604129 -> 0.564821: 1.07x faster
```

```
Significant (t=13.55)
Stddev: 0.01493 -> 0.01408: 1.0601x smaller

### json_dump_v2 ###
Min: 2.794058 -> 4.456882: 1.60x slower
Avg: 2.806195 -> 4.467750: 1.59x slower
Significant (t=-801.42)
Stddev: 0.00722 -> 0.01276: 1.7678x larger

### regex_v8 ###
Min: 0.041685 -> 0.050890: 1.22x slower
Avg: 0.042082 -> 0.051579: 1.23x slower
Significant (t=-26.94)
Stddev: 0.00177 -> 0.00175: 1.0105x smaller

### tornado_http ###
Min: 0.258212 -> 0.272552: 1.06x slower
Avg: 0.263689 -> 0.280610: 1.06x slower
Significant (t=-8.59)
Stddev: 0.01614 -> 0.01130: 1.4282x smaller

The following not significant results are hidden, use -v to show them:
django_v3, json_load, nbody.

real        21m53.511s
user        21m29.279s
sys 0m23.055s
```

Microbenchmarks

REMINDER: on a microbenchmark, even a significant speedup doesn't mean that you will get a significant speedup on your application.

The `benchmarks/` directory contains microbenchmarks used to test `fat` and `fatoptimizer` performances.

Function inlining and specialization using the parameter type

Optimize:

```
def _get_sep(path):
    if isinstance(path, bytes):
        return b'/'
    else:
        return '/'

def isabs(s):
    """Test whether a path is absolute"""
    sep = _get_sep(s)
    return s.startswith(sep)
```

to:

```
def isabs(s):
    return s.startswith('/')
```

but only if *s* parameter is a string.

2016-01-21:

```
original isabs() bytecode: 488 ns
_get_sep() inlined in isabs(): 268 ns (-220 ns, -45.0%, 1.8x faster :-))
```

2015-10-21:

```
$ ./python -m timeit 'import posixpath; isabs = posixpath.isabs' 'isabs("/root")'
1000000 loops, best of 3: 0.939 usec per loop
$ ./python -F -m timeit 'import posixpath; isabs = posixpath.isabs' 'isabs("/root")'
1000000 loops, best of 3: 0.755 usec per loop
```

Script: `benchmarks/bench_posixpath.py`.

Move invariant out of loop (list.append)

Optimize:

```
def func(obj, data):
    for item in data:
        obj.append(item)
```

to:

```
def func(obj, data):
    append = obj.append
    for item in data:
        append(item)
```

2016-01-21:

```
range(10 ** 0)
- original bytecode: 297 ns
- append=obj.append with guards: 310 ns (+13 ns, +4.4%, 1.0x slower :-()
- append=obj.append: 306 ns (+9 ns, +3.1%, 1.0x slower :-()
range(10 ** 1)
- original bytecode: 972 ns
- append=obj.append with guards: 703 ns (-268 ns, -27.6%, 1.4x faster :-))
- append=obj.append: 701 ns (-271 ns, -27.9%, 1.4x faster :-))
range(10 ** 3)
- original bytecode: 72.2 us
- append=obj.append with guards: 43.8 us (-28.4 us, -39.4%, 1.6x faster :-))
- append=obj.append: 43.7 us (-28.6 us, -39.5%, 1.7x faster :-))
range(10 ** 5)
- original bytecode: 8131.5 us
- append=obj.append with guards: 5289.5 us (-2842.0 us, -35.0%, 1.5x faster :-))
- append=obj.append: 5294.2 us (-2837.4 us, -34.9%, 1.5x faster :-))
```

2015-10-21:

```
$ ./python bench.py
regular python: range(1)-> 502 ns
regular python: range(10)-> 1.7 us
regular python: range(10**3)-> 122.0 us
regular python: range(10**5)-> 8.5 ms
```

```
$ ./python -F bench.py
fat python: range(1)-> 479 ns (-5%)
fat python: range(10)-> 1.1 us (-35%)
fat python: range(10**3)-> 65.2 us (-47%)
fat python: range(10**5)-> 5.3 ms (-38%)
```

Script: `benchmarks/bench_list_append.py`.

Call builtin

Optimize:

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

to:

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

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (call len)	116 ns
return 3 with guard on builtins	90 ns
return 3	79 ns

GuardBuiltins has a cost of 11 ns.

Script: `benchmarks/bench_len_abc.py`.

Copy builtin function to constant

Optimize:

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

to:

```
def func(obj):
    return 'LEN'(obj)
func.__code__ = fat.replace_consts(func.__code__, {'LEN': len})
```

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (LOAD_GLOBAL)	121 ns
LOAD_CONST with guard on builtins	116 ns
LOAD_CONST	105 ns

GuardBuiltins has a cost of 11 ns.

Script: `benchmarks/bench_copy_builtin_to_cst.py`.

Copy global function to constant

Optimize:

```
mylen = len

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

to:

```
mylen = len

def func(obj):
    return 'MYLEN'(obj)
func.__code__ = fat.replace_consts(func.__code__, {'MYLEN': len})
```

2015-01-21 (best timing of 5 runs):

Test	Perf
Original bytecode (LOAD_GLOBAL)	115 ns
LOAD_CONST with guard on globals	112 ns
LOAD_CONST	105 ns

GuardGlobals has a cost of 7 ns.

Script: `benchmarks/bench_copy_global_to_cst.py`.

Cost of guards

Cost of GuardDict guard.

2016-01-21:

```
no guard: 81 ns
with 1000 guards on globals: 3749 ns
cost of 1000 guards: 3667 ns (4503.4%)
average cost of 1 guard: 4 ns (4.5%)

no guard: 82 ns
with 100 guards on globals: 419 ns
cost of 100 guards: 338 ns (414.6%)
average cost of 1 guard: 3 ns (4.1%)

no guard: 81 ns
with 10 guards on globals: 117 ns
cost of 10 guards: 36 ns (43.9%)
average cost of 1 guard: 4 ns (4.4%)

no guard: 82 ns
with 1 guards on globals: 87 ns
cost of 1 guards: 5 ns (6.5%)
average cost of 1 guard: 5 ns (6.5%)
```

2016-01-06:

```
no guard: 431 ns
with 1000 guards on globals: 7974 ns
```

```
cost of 1000 guards: 7542 ns (1748.1%)
average cost of 1 guard: 8 ns (1.7%)

no guard: 429 ns
with 100 guards on globals: 1197 ns
cost of 100 guards: 768 ns (179.0%)
average cost of 1 guard: 8 ns (1.8%)

no guard: 426 ns
with 10 guards on globals: 515 ns
cost of 10 guards: 89 ns (20.8%)
average cost of 1 guard: 9 ns (2.1%)

no guard: 430 ns
with 1 guards on globals: 449 ns
cost of 1 guards: 19 ns (4.5%)
average cost of 1 guard: 19 ns (4.5%)
```

Script: `benchmarks/bench_guards.py`.

Changelog

fatoptimizer changelog

- Version 0.3
  - Experimental implementation of function inlining, implemented by David Malcolm.
  - New optimization: call pure methods of builtin types. For example, replace `"abc".encode()` with `b'abc'`.
  - Update for fat API version 0.3, GuardBuiltins constructor changed.
  - Basic "loop unrolling" on list-comprehension, set-comprehension and dict-comprehension. Only if there is a single comprehension using a constant iterable without if.
- 2016-01-23: Version 0.2
  - Fix the function optimizer: don't specialized nested function. The specialization is more expensive than the speedup of optimizations.
  - Fix Config.replace(): copy logger attribute
  - get_literal() now also returns tuple literals when items are not constants
  - Adjust usage of get_literal()
  - SimplifyIterable also replaces empty dict (created a runtime) with an empty tuple (constant)
  - Update benchmark scripts and benchmark results in the documentation
- 2016-01-18: Version 0.1
  - Add `fatoptimizer.pretty_dump()`
  - Add Sphinx documentation: `doc/` directory
  - Add benchmark scripts: `benchmarks/` directory
  - Update `fatoptimizer._register()` for the new version of the PEP 511 (`sys.set_code_transformers()`)

- 2016-01-14: First public release, version 0.0.

fatoptimizer TODO list

Easy issues, for new contributors

- Complete fatoptimizer/methods.py to support more pure methods.
- Complete fatoptimizer/builtins.py to support more pure builtin functions.

Goal

To get visible performance gain, the following optimizations must be implemented:

- Function inlining
- Detect and call pure functions
- Elimination of unused variables (set but never read): the constant propagation and loop unrolling create many of them. For example, replace "def f(): x=1; return x" with "def f(): return 1"
- Copy constant global variable to function globals
- Specialization for argument types: move invariant out of loops. Ex: create a bounded method "obj_append = obj.append" out of the loop.

Even if many optimizations can be implemented with a static optimizers, it's still not a JIT compiler. A JIT compiler is required to implement even more optimizations.

Known Bugs

- `import *` is ignored
- Usage of locals() or vars() must disable optimization. Maybe only when the optimizer produces new variables?

Search ideas of new optimizations

- python.org wiki: PythonSpeed/PerformanceTips
- Open issues of type Performance
- Closed issues of type Performance
- Unladen Swallow ProjectPlan
- Ideas from PyPy, Pyston, Numba, etc.

More optimizations

MUST HAVE

More complex to implement (without breaking Python semantics).

- Remove useless temporary variables. Example:

  Code:

```
def func():
    res = 1
    return res
```

  Constant propagation:

```
def func():
    res = 1
    return 1
```

  Remove *res* local variable:

```
def func():
    return 1
```

  Maybe only for simple types (int, str). It changes object lifetime: https://bugs.python.org/issue2181#msg63090

- Function inlining: see Issue #10399, AST Optimization: inlining of function calls
- Inline calls to all functions, short or not? Need guards on these functions and the global namespace. Example: posixpath._get_sep().
- Call pure functions of math, struct and string modules. Example: replace math.log(32) / math.log(2) with 5.0.

Pure functions

- Compute if a function is pure. See pythran.analysis.PureFunctions of pythran project, depend on ArgumentEffects and GlobalEffects analysys

Random

Easy to implement.

- [Python-ideas] (FAT Python) Convert keyword arguments to positional? https://mail.python.org/pipermail/python-ideas/2016-January/037874.html
- Loop unrolling: support multiple targets:

```
for x, y in ((1, 2), (3, 4)):
    print(x, y)
```

- Tests:
  - `if a: if b: code` => `if a and b: code`
- Optimize `str%args` and `bytes%args`
- Constant folding:
  - replace get_constant() with get_literal()?
    - list + list
    - frozenset | frozenset
    - set | set

  - 2.0j ** 3.0
  - 1 < 2 < 3
  - `if x and True: pass` => `if x: pass` http://bugs.python.org/issue7682
  - replace '(a and b) and c' (2 op) with 'a and b and c' (1 op), same for "or" operator
- Specialize also AsyncFunctionDef (run stage 2, not only stage 1)

Can be done later

Unknown speedup, easy to medium to implement.

- Replace dict(...) with {...} (dict literal): https://doughellmann.com/blog/2012/11/12/the-performance-impact-of-using-dict-instead-of-in-cpython-2-7-2/
- Use SimplifyIterable on dict/frozenset argument
- print(): convert arguments to strings
- Remove dead code: remove "pass; pass"
- Simplify iteratable:
  - for x in set("abc"): ... => for x in frozenset("abc"): ... Need a guard on set builtin
  - for x in "abc": ... => for x in ("a", "b", "c"): ... Is it faster? Does it use less memory?
  - at least, loop unrolling must work on "for x in 'abc': ..."

Can be done later and are complex

Unknown speedup, complex to implement.

- Remove "if 0: yield" but tag FunctionDef as a generator?
- Implement CALL_METHOD bytecode, but execute the following code correctly (output must be 1, 2 and not 1, 1):

```
class C(object):
    def foo(self):
        return 1
c = c()
print c.foo()
c.foo = lambda: 2
print c.foo()
```

  Need a guard on C.foo?

  See https://bugs.python.org/issue6033#msg95707

  Is it really possible? FAT Python doesn't support guards on the instance dict, it's more designed to use guards on the type dict.

- Optimize 'lambda: chr(65)'. Lambda are functions, but defined as expressions. It's not easy to inject the func.specialize() call, func.__code__.replace_consts() call, etc. Maybe only optimize in some specific cases?

  Specialization of nested function was disabled because the cost to specialize the function can be higher than the speedup if the function is called once and then destroyed.

- Enable copy builtins to constants when we know that builtins and globals are not modified. Need to ensure that the function is pure and only calls pure functions.

- Move invariant out of loops using guards on argument types:
  - Merge duplicate LOAD_ATTR, when we can make sure that the attribute will not be modified
  - list.append: only for list type
- Loop unrolling:
  - support break and continue
  - support raise used outside try/except
- Constant propagation, copy accross namespaces:
  - list-comprehension has its own separated namespace:

```
n = 100
seq = [randrange(n) for i in range(n)]
```

  - copy globals to locals: need a guard on globals
- Convert naive loop to list/dict/set comprehension. Replace "x=[]; for item in data: x.append(item.upper())" with "x=[item.upper() for item in data]". Same for x=set() and x={}.
- Call more builtin functions:
  - all(), any()
  - enumerate(iterable), zip()
  - format()
  - filter(pred, iterable), map(pred, iterable), reversed()
- operator module:
  - need to add an import, need to ensure that operator name is not used
  - lambda x: x[1] => operator.itemgetter(1)
  - lambda x: x.a => operator.attrgetter('a')
  - lambda x: x.f('a', b=1) => operator.methodcaller('f', 'a', b=1)
- map, itertools.map, filter:
  - [f(x) for x in a] => map(f, a) / list(map(f, a))
  - (f(x) for x in a) => itertools.map(f, a) / map(f, a) ? scope ?
  - (x for x in a if f(x)) => filter(f, a)
  - (x for x in a if not f(x)) => __builtin_filternot__(f, a) ?
  - (2 * x for x in a) => map((2).__mul__, a)
  - (x for x in a if x in 'abc') => filter('abc'.__contains__, a)

Profiling

- implement code to detect the exact type of function parameters and function locals and save it into an annotation file
- implement profiling directed optimization: benchmark guards at runtime to decide if it's worth to use a specialized function. Measure maybe also the memory footprint using tracemalloc?

- implement basic stategy to decide if specialized function must be emitted or not using raw estimation, like the size of the bytecode in bytes

Later

- efficient optimizations on objects, not only simple functions
- handle python modules and python imports
  - checksum of the .py content?
  - how to handle C extensions? checksum of the .so file?
  - how to handle .pyc files?
- find an efficient way to specialize nested functions
- configuration to manually help the optimizer:
  - give a whitelist of "constants": app.DEBUG, app.enum.BLUE, ...
  - type hint with strict types: x is Python int in range [3; 10]
  - expect platform values to be constant: sys.version_info, sys.maxunicode, os.name, sys.platform, os.linesep, etc.
  - declare pure functions
  - see fatoptimizer for more ideas
- Restrict the number of guards, number of specialized bytecode, number of arg_type types with fatoptimizer.Config
- fatoptimizer.VariableVisitor: support complex assignments like 'type(mock)._mock_check_sig = checksig'
- Support specialized CFunction_Type, not only specialized bytecode?
- Add an opt-in option to skip some guards if the user knows that the application will never modify function __code__, override builtin methods, modify a constant, etc.
- Optimize real objects, not only simple functions. For example, inline a method.
- Function parameter: support more complex guard to complex types like list of integers?
- handle default argument values for argument type guards?
- Support locals()[key], vars()[key], globals()[key]?
- Support decorators
- Copy super() builtin to constants doesn't work. Calling the builtin super() function creates a free variable, whereas calling the constant doesn't create a free variable.
- Tail-call recursion?

  def factorial(n):

  if n > 1: return n * factorial(n-1)

  else: return 1

Support decorator

weakref.py:

```
@property
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit

@atexit.setter
def atexit(self, value):
    info = self._registry.get(self)
    if info:
        info.atexit = bool(value)
```

It's not possible to replace it with:

```
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit
atexit = property(atexit)

def atexit(self, value):
    info = self._registry.get(self)
    if info:
        info.atexit = bool(value)
atexit = atexit.setter(atexit)
```

The last line 'atexit = atexit.setter(atexit)' because 'atexit' is now the second function, not more the first decorated function (the property).

Define the second atexit under a different name? No! It changes the code name, which is wrong.

Maybe we can replace it with:

```
def atexit(self):
    """Whether finalizer should be called at exit"""
    info = self._registry.get(self)
    return bool(info) and info.atexit
atexit = property(atexit)

_old_atexit = atexit
def atexit(self, value):
    info = self._registry.get(self)
    if info:
        info.atexit = bool(value)
atexit = _old_atexit.setter(atexit)
```

But for this, we need to track the namespace during the optimization. The VariableVisitor in run *before* the optimizer, it doesn't track the namespace at the same time.

Possible optimizations

Short term:

- Function func2() calls func1() if func1() is pure: inline func1() into func2()

- Call builtin pure functions during compilation. Example: replace len("abc") with 3 or range(3) with (0, 1, 2).
- Constant folding: replace a variable with its value. We may do that for optimal parameters with default value if these parameters are not set. Example: replace app.DEBUG with False.

Using types:

- Detect the exact type of parameters and function local variables
- Specialized code relying on the types. For example, move invariant out of loops (ex: obj.append for list).
- x + 0 gives a TypeError for str, but can be replaced with x for int and float. Same optimization for x*0.
- See astoptimizer for more ideas.

Longer term:

- Compile to machine code using Cython, Numba, PyPy, etc. Maybe only for numeric types at the beginning? Release the GIL if possible, but check "sometimes" if we got UNIX signals.

Google Summer of Code

Google Summer of Code and the PSF

The Google Summer of Code (GSoC) "is a global program focused on bringing more student developers into open source software development. Students work with an open source organization on a 3 month programming project during their break from school."

The Python Software Foundation is part of the Google Summer of Code (GSoC) program in 2016, as previous years. See:

- PSF GSoC projects 2016
- Python core projects of PSF GSoC 2016: this page mentions the GSoC project on FAT Python (but this page is more complete).

FAT Python

FAT Python is a new static optimizer for Python 3.6, it specializes functions and use guards to decide if specialized code can be called or not. See FAT Python homepage and the slides of my talk at FOSDEM 2016 for more information.

The design is *inspired* by JIT compilers, but is simpler. FAT Python has been designed to be able to merge changes required to use FAT Python into CPython 3.6. The expected use case is to compile modules and applications ahead-of-time, so the performance of the optimizer itself don't matter much.

FAT Python is made of different parts:

- CPython 3.6
- fat module: fast guards implemented in C
- fatoptimizer module: the static optimizer implemented as an AST optimizer. It produces specialized functions of functions using guards.
- PEP 509 (dict version): patch required by the fat module to implement fast guards on Python namespaces
- PEP 510 (function specialization): private C API to add specialized code to Python functions
- PEP 511 (API for AST optimizers): new Python API to register an AST optimizer

Status at March 2016:

- First patches to implement AST optimizers have already been merged in CPython 3.6
- fat and fatoptimizer have been implemented, are fully functional and have unit tests
- early benchmarks don't show major speedup on "The Grand Unified Python Benchmark Suite"
- fatoptimizer is quite slow
- PEP 509 need to be modified to make the dictionary versions globally unique. PEP 509 is required by the promising Speedup method calls 1.2x change written by Yury Selivanov. This change can help to get this PEP accepted.
- PEP 511 is still a work-in-progress, it's even unclear if the whole PEP is required. It only makes the usage of FAT Python more practical. It avoids conflicts on .pyc files using the `-o` command line option proposed in the PEP.
- PEP 509, 510 and 511 are basically blocked by an expectation on concrete speedup

FAT Python GSoC Roadmap

GSoC takes 4 months, the exact planning is not defined yet.

Goal

The overall goal is to enhance FAT Python to get concrete speedup on the benchmark suite and on applications.

Requirements

- All requirements of the GSoC program! (like being available during the 4 months of the GSoC program)
- Able to read and write technical english
- Better if already the student worked remotely on a free software before
- Good knowledge of the Python programming language
- (Optional?) Basic knowledge of how compilers are implemented
- (Optional?) Basic knowledge of static optimizations like constant folding

Milestone 0 to select the student

fatoptimizer:

- Download fatoptimizer and run tests:

```
git clone https://github.com/haypo/fatoptimizer
cd fatoptimizer
python3 test_fatoptimizer.py
```

- Read the fatoptimizer documentation
- Pick a simple task in the *fatoptimizer TODO list* and send a pull request
- MANDATORY: Submit a final PDF proposal: see https://wiki.python.org/moin/SummerOfCode/2016 for a template

Optional:

- Download and compile FAT Python: https://faster-cpython.readthedocs.io/fat_python.html#getting-started
- Run Python test suite of FAT Python

Milestone 1

Discover FAT Python.

- Select a set of benchmarks
- Run benchmarks to have a reference for performances (better: write a script for that)
- Implement the most easy optimizations of the *TODO list* (like remaining constant folding optimizations)
- Run the full Python test suite with FAT Python
- Run real applications like Django with FAT Python to identify bugs
- Propose fixes or workaround for bugs

Goal: have at least one new optimization merged into fatoptimizer.

Milestone 2

Function inlining.

- Test the existing (basic) implementation of function inlining
- Fix function inlining
- Enhance function inlining to use it in more cases
- Wider tests of the new features
- Fix bugs

Goal: make function inlining usable with the default config without breaking the Python test suite, even if it's only a subset of the feature.

Milestone 3

Remove useless variables. For example, remove `x` in `def func(): x = 1; return 2`.

- Add configuration option to enable this optimization
- Write an unit test for the expected behaviour
- Implement algorithm to compute where and when a variable is alive or not
- Use this algorithm to find dead variables and then remove them
- Wider tests of the new features
- Fix bugs

Goal: remove useless variables with the default config without breaking the Python test suite, even if it's only a subset of the feature.

Milestone 4 (a)

Detect pure function, first subpart: implement it manually.

- Add an option to __fatoptimizer__ module configuration to explicitly declare constants
- Write a patch to declare some constants in the Python standard library
- Add an option to __fatoptimizer__ module configuration to explicitly declare pure functions
- Write a patch to declare some pure functions in the Python standard library, ex: os.path._getsep().

Goal: annotate a few constants and pure functions in the Python standard library and ensure that they are optimized.

Milestone 4 (b)

Detect pure function, second and last subpart: implement automatic detection.

- Write a safe heuristic to detect pure functions using a small whitelist of instructions which are known to be pure
- Wider tests of the new features
- Fix bugs
- Extend the whitelist, add more and more instructions
- Run tests
- Fix bugs
- Iterate until the whitelist is considered big enough?
- Maybe design a better algorithm than a white list?

See also pythran which already implemented this feature :-)

Goal: detect that os.path._getsep() is pure.

Goal 2, optional: inline os.path._getsep() in isabs().

More milestones?

The exact planning will be adapted depending on the speed of the student, the availability of mentors, etc.

Misc

Implementation

Steps and stages

The optimizer is splitted into multiple steps. Each optimization has its own step: fatoptimizer.const_fold.ConstantFolding implements for example constant folding.

The function optimizer is splitted into two stages:

- stage 1: run steps which don't require function specialization
- stage 2: run steps which can add guard and specialize the function

Main classes:

- ModuleOptimizer: Optimizer for ast.Module nodes. It starts by looking for *__fatoptimizer__ configuration*.
- FunctionOptimizer: Optimizer for ast.FunctionDef nodes. It starts by running FunctionOptimizerStage1.
- Optimizer: Optimizer for other AST nodes.

Steps used by ModuleOptimizer, Optimizer and FunctionOptimizerStage1:

- NamespaceStep: populate a Namespace object which tracks the local variables, used by ConstantPropagation
- ReplaceBuiltinConstant: replace builtin optimization
- ConstantPropagation: constant propagation optimization
- ConstantFolding: constant folding optimization
- RemoveDeadCode: dead code elimitation optimization

Steps used by FunctionOptimizer:

- NamespaceStep: populate a Namespace object which tracks the local variables
- UnrollStep: loop unrolling optimization
- CallPureBuiltin: call builtin optimization
- CopyBuiltinToConstantStep: copy builtins to constants optimization

Some optimizations produce a new AST tree which must be optimized again. For example, loop unrolling produces new nodes like "i = 0" and duplicates the loop body which uses "i". We need to rerun the optimizer on this new AST tree to run optimizations like constant propagation or constant folding.

Pure functions

A "pure" function is a function with no side effect.

Example of pure operators:

- x+y, x-y, x*y, x/y, x//y, x**y for types int, float, complex, bytes, str, and also tuple and list for x+y

Example of instructions with side effect:

- "global var"

Example of pure function:

```
def mysum(x, y):
    return x + y
```

Example of function with side effect:

```
global _last_sum

def mysum(x, y):
    global _last_sum
    s = x + y
    _last_sum = s
    return s
```

Constants

FAT Python introduced a new AST type: `ast.Constant`. The optimizer starts by converting `ast.NameConstant`, `ast.Num`, `ast.Str`, `ast.Bytes` and `ast.Tuple` to `ast.Constant`. Later, it can create constant of other types. For example, `frozenset('abc')` creates a `frozenset` constant.

Supported constants:

- `None` singleton
- `bool`: `True` and `False`
- numbers: `int`, `float`, `complex`
- strings: `bytes`, `str`
- containers: `tuple`, `frozenset`

Literals

Literals are a superset of constants.

Supported literal types:

- (all constant types)
- containers: `list`, `dict`, `set`

FunctionOptimizer

`FunctionOptimizer` handles `ast.FunctionDef` and emits a specialized function if a call to a builtin function can be replaced with its result.

For example, this simple function:

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

is optimized to:

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

_ast_optimized = func

def func():
    return "A"
_ast_optimized.specialize(func,
                          [{'guard_type': 'builtins', 'names': ('chr',)}])

func = _ast_optimized
del _ast_optimized
```

Detection of free variables

VariableVisitor detects local and global variables of an `ast.FunctionDef` node. It is used by the `FunctionOptimizer` to detect free variables.

Corner cases

Calling the `super()` function requires a cell variables.

Index