A tour of CPython's bytecode compiler

Brandt Bucher (March 14th, 2023)

A tour of CPython's bytecode compiler

...and how you can speed up every Python process on the planet!

Brandt Bucher (March 14th, 2023)

Brandt Bucher (March 14th, 2023)

Brandt Bucher (Py Day, 2023)

Brandt Bucher

Brandt Bucher

- 2017: Started using Python.
- 2018: Contributed code to CPython.
- 2019: Joined Python's Triage Team.
- 2020: Joined Python's Core Development Team.
- 2021: Joined Microsoft's CPython Performance Engineering Team.
- 2022: Made CPython 3.11 25% faster!

Microsoft's CPython Performance Engineering Team

The "Faster CPython" Project

The "Faster CPython" Project

- Python's BDFL:
  - Guido van Rossum
- Four other Python core developers:
  - Mark Shannon
  - Irit Katriel
  - Eric Snow
  - Brandt Bucher
- One member of Python's triage team:
  - Michael Droettboom
- One Microsoft engineer:
  - L. A. F. Pereira

The "Faster CPython" Project

- Python's BDFL:
  - Guido van Rossum
- Four other Python core developers:
  - Mark Shannon
  - Irit Katriel
  - Eric Snow
  - Brandt Bucher
- One member of Python's triage team:
  - Michael Droettboom
- One Microsoft engineer:
  - L. A. F. Pereira

The "Faster CPython" Project

- Python's BDFL:
  - @gvanrossum
- Four other Python core developers:
  - @markshannon
  - @iritkatriel
  - @ericsnowcurrently
  - @brandtbucher
- One member of Python's triage team:
  - @mdboom
- One Microsoft engineer:
  - @lpereira

The "Faster CPython" Project

- faster-cpython
- faster-cpython/ideas
- faster-cpython/benchmarking-public

The "Faster CPython" Project

- California (Microsoft)
- Utah (Microsoft)
- Washington (Meta, Microsoft)
- South Dakota (Meta)
- Washington, D.C. (Microsoft)
- United Kingdom (Bloomberg, Microsoft)
- Singapore (National University of Singapore)

The "Faster CPython" Project

- California (Microsoft)
- Utah (Microsoft)
- Washington (Meta, Microsoft)
- South Dakota (Meta)
- Washington, D.C. (Microsoft)
- United Kingdom (Bloomberg, Microsoft)
- Singapore (National University of Singapore)

The "Faster CPython" Project

- California (Microsoft, UC Irvine?)
- Utah (Microsoft)
- Washington (Meta, Microsoft)
- South Dakota (Meta)
- Washington, D.C. (Microsoft)
- United Kingdom (Bloomberg, Microsoft)
- Singapore (National University of Singapore)

Python

Python

- 32 years old!
- *Very* high-level
- *Very* widely used
- Dynamic
- Object-oriented
- Interpreted
- Automatic memory management
- Deep introspection and metaprogramming

Python

- 32 years old!
- *Very* high-level
- *Very* widely used
- Dynamic
- Object-oriented
- Interpreted
- Automatic memory management
- Deep introspection and metaprogramming

Python

- 32 years old!
- *Very* high-level
- *Very* widely used
- Dynamic
- Object-oriented
- Interpreted
- Automatic memory management
- Deep introspection and metaprogramming

Python

```
def f():
    foo = 42
    test = False
    if test:
        print("🧟")

f()
```

Python

```
def f():
    foo = 42
    test = False
    if test:
        print("🧟")

f()
```

```
> zombie.py(4)f()
-> if test:
(Pdb)
```

Python

```
def f():
    foo = 42
    test = False
    if test:
        print("🧟")

f()
```

```
> zombie.py(4)f()
-> if test:
(Pdb) foo
42
(Pdb)
```

Python

```
def f():
    foo = 42
    test = False
    if test:
        print("🧟")

f()
```

```
> zombie.py(4)f()
-> if test:
(Pdb) foo
42
(Pdb) test = True
(Pdb)
```

Python

```
def f():
    foo = 42
    test = False
    if test:
        print("🧟")

f()
```

```
> zombie.py(4)f()
-> if test:
(Pdb) foo
42
(Pdb) test = True
(Pdb) continue
🧟
```

Python

```
class Zombie:
    def __del__(self):
        global resurrected
        resurrected = self
```

Python

- Most objects have arbitrary mappings of attributes: `instance.__dict__`.
- Bytecode is a runtime object: `function.__code__`.
- *Frames* are runtime objects: `sys._getframe()`.
- GC can run on *any* object allocation.
- Attribute/global name accesses and assignments can run arbitrary code.
- Even simple operators go through incredibly complex double-dispatching.

Python

- Most objects have arbitrary mappings of attributes: `instance.__dict__`.
- Bytecode is a runtime object: `function.__code__`.
- *Frames* are runtime objects: `sys._getframe()`.
- GC can run on *any* object allocation.
- Attribute/global name accesses and assignments can run arbitrary code.
- Even simple operators go through incredibly complex double-dispatching.

Python

- Most objects have arbitrary mappings of attributes: `instance.__dict__`.
- Bytecode is a runtime object: `function.__code__`.
- *Frames* are runtime objects: `sys._getframe()`.
- GC can run on *any* object allocation.
- Attribute/global name accesses and assignments can run arbitrary code.
- Even simple operators go through incredibly complex double-dispatching.

CPython

CPython

- Reference implementation of Python
- Used by ~100% of Python programmers
- Reference-counted (augmented with cyclic stop-the-world GC)
- Has an incredibly rich ecosystem of third-party C extensions
- Maintained by a few dozen active "core developers"
- Free and open-source
- python/cpython

CPython

CPython's Bytecode

CPython's Bytecode

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)
```

CPython's Bytecode

```
y = dy + self.y
cls = type(self)
```

CPython's Bytecode

```
y = dy + self.y
cls = type(self)
```

```
import dis
dis.dis(Point.shifted)
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
0x7C    0x02
0x7C    0x00
0x6A    0x02
0x7A    0x00
0x7D    0x04
0x74    0x05
0x7C    0x00
0xAB    0x01
0x7D    0x05
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y      LOAD_FAST    (dy)
cls = type(self)     LOAD_FAST    (self)
                     LOAD_ATTR    (y)
                     BINARY_OP    (+)
                     STORE_FAST   (y)
                     LOAD_GLOBAL  (type)
                     LOAD_FAST    (self)          stack
                     CALL         (1)
                     STORE_FAST   (cls)
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

stack

dy

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

stack
self
dy

CPython's Bytecode

dis.dis(Point.shifted)

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

stack
self.y
dy

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

<u>stack</u>

```
dy + self.y
```

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

```
y = dy + self.y
```

stack

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

```
y = dy + self.y
```

stack

type

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

```
y = dy + self.y
```

stack

self

type

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

```
y = dy + self.y
```

stack

type(self)

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

```
 y = dy + self.y
cls = type(self)
```

stack

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y      LOAD_FAST    (dy)
cls = type(self)     LOAD_FAST    (self)
                     LOAD_ATTR    (y)
                     BINARY_OP    (+)
                     STORE_FAST   (y)
                     LOAD_GLOBAL  (type)
                     LOAD_FAST    (self)          stack
                     CALL         (1)
                     STORE_FAST   (cls)
```

CPython's Tokenizer

CPython's Tokenizer

```
y = dy + self.y
cls = type(self)
```

CPython's Tokenizer

```
y = dy + self.y
```

```
cls = type(self)
```

CPython's Tokenizer

```
"y"
"="
"dy"
"+"
"self"
"."
"y"
"\n"
```

```
"cls"
"="
"type"
"("
"self"
")"
"\n"
```

CPython's Tokenizer

```
NAME       ("y")
EQUAL      ("=")
NAME       ("dy")
PLUS       ("+")
NAME       ("self")
DOT        (".")
NAME       ("y")
NEWLINE    ("\n")
```

```
NAME       ("cls")
EQUAL      ("=")
NAME       ("type")
LPAR       ("(")
NAME       ("self")
RPAR       (")")
NEWLINE    ("\n")
ENDMARKER  ("")
```

CPython's Tokenizer

```
NAME       ("y")
EQUAL      ("=")
NAME       ("dy")
PLUS       ("+")
NAME       ("self")
DOT        (".")
NAME       ("y")
NEWLINE    ("\n")
```

```
NAME       ("cls")
EQUAL      ("=")
NAME       ("type")
LPAR       ("(")
NAME       ("self")
RPAR       (")")
NEWLINE    ("\n")
ENDMARKER  ("")
```

CPython's Parser

CPython's Parser

```
y = dy + self.y
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a=compound_stmt { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
    | a[asdl_stmt_seq*]=simple_stmts { a }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
    | a[asdl_stmt_seq*]=';'.simple_stmt+ [';'] NEWLINE { a }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
    | e=star_expressions { _PyAST_Expr(e, EXTRA) }
    | &'return' return_stmt
    | &('import' | 'from') import_stmt
    | &'raise' raise_stmt
    | 'pass' { _PyAST_Pass(EXTRA) }
    | &'del' del_stmt
    | &'yield' yield_stmt
    | &'assert' assert_stmt
    | 'break' { _PyAST_Break(EXTRA) }
    | 'continue' { _PyAST_Continue(EXTRA) }
    | &'global' global_stmt
    | &'nonlocal' nonlocal_stmt
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a=NAME ':' b=expression c=['=' d=annotated_rhs { d }] {
        CHECK_VERSION(
            stmt_ty,
            6,
            "Variable annotation syntax is",
            _PyAST_AnnAssign(CHECK(expr_ty, _PyPegen_set_expr_context(p, a, Store)), b, c, 1, EXTRA)
        ) }
    | a=('(' b=single_target ')' { b }
         | single_subscript_attribute_target) ':' b=expression c=['=' d=annotated_rhs { d }] {
        CHECK_VERSION(stmt_ty, 6, "Variable annotations syntax is", _PyAST_AnnAssign(a, b, c, 0, EXTRA)) }
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
    | a=single_target b=augassign ~ c=(yield_expr | star_expressions) {
        _PyAST_AugAssign(a, b->kind, c, EXTRA) }
    | invalid_assignment
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
    | a=star_target b=(',' c=star_target { c })* [','] {
       _PyAST_Tuple(CHECK(asdl_expr_seq*, _PyPegen_seq_insert_in_front(p, a, b)), Store, EXTRA) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | '*' a=(!'*' star_target) {
        _PyAST_Starred(CHECK(expr_ty, _PyPegen_set_expr_context(p, a, Store)), Store, EXTRA) }
    | target_with_star_atom
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
target_with_star_atom[expr_ty] (memo):
    | a=t_primary '.' b=NAME !t_lookahead { _PyAST_Attribute(a, b->v.Name.id, Store, EXTRA) }
    | a=t_primary '[' b=slices ']' !t_lookahead { _PyAST_Subscript(a, b, Store, EXTRA) }
    | star_atom
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
target_with_star_atom[expr_ty] (memo):
    | star_atom
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
target_with_star_atom[expr_ty] (memo):
    | star_atom
star_atom[expr_ty]:
    | a=NAME { _PyPegen_set_expr_context(p, a, Store) }
    | '(' a=target_with_star_atom ')' { _PyPegen_set_expr_context(p, a, Store) }
    | '(' a=[star_targets_tuple_seq] ')' { _PyAST_Tuple(a, Store, EXTRA) }
    | '[' a=[star_targets_list_seq] ']' { _PyAST_List(a, Store, EXTRA) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
target_with_star_atom[expr_ty] (memo):
    | star_atom
star_atom[expr_ty]:
    | a=NAME { _PyPegen_set_expr_context(p, a, Store) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_targets[expr_ty]:
    | a=star_target !',' { a }
star_target[expr_ty] (memo):
    | target_with_star_atom
target_with_star_atom[expr_ty] (memo):
    | star_atom
star_atom[expr_ty]:
    | a=NAME { _PyPegen_set_expr_context(p, a, Store) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | a=star_expression b=(',' c=star_expression { c })+ [','] {
      _PyAST_Tuple(CHECK(asdl_expr_seq*, _PyPegen_seq_insert_in_front(p, a, b)), Load, EXTRA) }
    | a=star_expression ',' { _PyAST_Tuple(CHECK(asdl_expr_seq*, _PyPegen_singleton_seq(p, a)), Load, EXTRA) }
    | star_expression
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | '*' a=bitwise_or { _PyAST_Starred(a, Load, EXTRA) }
    | expression
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | invalid_expression
    | invalid_legacy_expression
    | a=disjunction 'if' b=disjunction 'else' c=expression { _PyAST_IfExp(b, a, c, EXTRA) }
    | disjunction
    | lambdef
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
    | a=sum '-' b=term { _PyAST_BinOp(a, Sub, b, EXTRA) }
    | term
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
...
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
...
primary[expr_ty]:
    | a=primary '.' b=NAME { _PyAST_Attribute(a, b->v.Name.id, Load, EXTRA) }
    | atom
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
...
primary[expr_ty]:
    | a=primary '.' b=NAME { _PyAST_Attribute(a, b->v.Name.id, Load, EXTRA) }
    | atom
atom[expr_ty]:
    | NAME
```

CPython's Parser

```
statements[asdl_stmt_seq*]:
    | a=statement+ { (asdl_stmt_seq*)_PyPegen_seq_flatten(p, a) }
statement[asdl_stmt_seq*]:
    | a[asdl_stmt_seq*]=simple_stmts { a }
simple_stmts[asdl_stmt_seq*]:
    | a=simple_stmt !';' NEWLINE { (asdl_stmt_seq*)_PyPegen_singleton_seq(p, a) }
simple_stmt[stmt_ty] (memo):
    | assignment
assignment[stmt_ty]:
    | a[asdl_expr_seq*]=(z=star_targets '=' { z })+ b=(yield_expr | star_expressions) !'=' tc=[TYPE_COMMENT] {
        _PyAST_Assign(a, b, NEW_TYPE_COMMENT(p, tc), EXTRA) }
star_expressions[expr_ty]:
    | star_expression
star_expression[expr_ty] (memo):
    | expression
expression[expr_ty] (memo):
    | disjunction
...
sum[expr_ty]:
    | a=sum '+' b=term { _PyAST_BinOp(a, Add, b, EXTRA) }
...
primary[expr_ty]:
    | a=primary '.' b=NAME { _PyAST_Attribute(a, b->v.Name.id, Load, EXTRA) }
    | atom
atom[expr_ty]:
    | NAME
```

CPython's Parser

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

CPython's Parser

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

CPython's Symbol Table

CPython's Symbol Table

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

CPython's Symbol Table

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

CPython's Symbol Table

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

CPython's Symbol Table

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

CPython's Symbol Table

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

Optimization

Optimization

- Static single assignment form
- Common subexpression elimination
- Sparse conditional constant propagation
- Copy propagation
- Strength reduction
- Bounds-check elimination
- Loop-invariant code motion
- Loop fission/fusion/splitting/unrolling/unswitching

Optimization

- Build IR
- Type checking
- Optimization
- Compilation

Optimization

- Build IR
- Type checking
- Optimization
- Compilation

Optimization

- Build IR
- Profiling
- Optimization
- Compilation

Optimization

- Constant Folding
- Dead Code Elimination
- Jump Threading
- Liveness Analysis
- Peephole Optimizations

Constant Folding

AST Optimizer

AST Optimizer

```
-(1 << (128 - 1))
```

AST Optimizer

```
# -(1 << (128 - 1))

UnaryOp(
    op=USub(),
    operand=BinOp(
        left=Constant(value=1),
        op=LShift(),
        right=BinOp(
            left=Constant(value=128),
            op=Sub(),
            right=Constant(value=1),
        ),
    ),
)
```

AST Optimizer

```
# -(1 << (128 - 1))

UnaryOp(
    op=USub(),
    operand=BinOp(
        left=Constant(value=1),
        op=LShift(),
        right=BinOp(
            left=Constant(value=128),
            op=Sub(),
            right=Constant(value=1),
        ),
    ),
)
```

AST Optimizer

```
# -(1 << 127)

UnaryOp(
    op=USub(),
    operand=BinOp(
        left=Constant(value=1),
        op=LShift(),
        right=Constant(value=127),
    ),
)
```

AST Optimizer

```
# -(1 << 127)

UnaryOp(
    op=USub(),
    operand=BinOp(
        left=Constant(value=1),
        op=LShift(),
        right=Constant(value=127),
    ),
)
```

AST Optimizer

```
# -170141183460469231731687303715884105728

UnaryOp(
    op=USub(),
    operand=Constant(value=170141183460469231731687303715884105728),
)
```

AST Optimizer

```
# -170141183460469231731687303715884105728

UnaryOp(
    op=USub(),
    operand=Constant(value=170141183460469231731687303715884105728),
)
```

AST Optimizer

```
# -170141183460469231731687303715884105728

Constant(value=-170141183460469231731687303715884105728)
```

AST Optimizer

```
("Spam", "eggs")
```

AST Optimizer

```
# ("Spam", "eggs")

Tuple(
    elts=[
        Constant(value="Spam"),
        Constant(value="eggs"),
    ],
    ctx=Load(),
)
```

AST Optimizer

```
# ("Spam", "eggs")

Tuple(
    elts=[
        Constant(value="Spam"),
        Constant(value="eggs"),
    ],
    ctx=Load(),
)
```

AST Optimizer

```
# ("Spam", "eggs")

Constant(
    value=("Spam", "eggs")
)
```

CPython's Compiler

CPython's Compiler

- Instruction
  - Integer opcode
  - Integer oparg
  - Location information
- Compiler
  - Symbol table
  - Instruction sequence
  - Memory arena
  - Stack of exception handling blocks
- Control-flow graph
  - Singly-linked list of basic blocks
- Basic block
  - Instruction sequence
  - Metadata
- Assembler
  - Final instruction sequence
  - Exception table
  - Location table

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Assign_kind:
    VISIT(c, expr, s->v.Assign.value);
    VISIT(c, expr, asdl_seq_GET(s->v.Assign.targets, i));
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case BinOp_kind:
    VISIT(c, expr, e->v.BinOp.left);
    VISIT(c, expr, e->v.BinOp.right);
    ADDOP_BINARY(c, loc, e->v.BinOp.op);
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Attribute_kind:
    VISIT(c, expr, e->v.Attribute.value);
    ADDOP_NAME(c, loc, LOAD_ATTR, e->v.Attribute.attr);
```

```
LOAD_FAST    (dy)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Attribute_kind:
    VISIT(c, expr, e->v.Attribute.value);
    ADDOP_NAME(c, loc, LOAD_ATTR, e->v.Attribute.attr);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Attribute_kind:
    VISIT(c, expr, e->v.Attribute.value);
    ADDOP_NAME(c, loc, LOAD_ATTR, e->v.Attribute.attr);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case BinOp_kind:
    VISIT(c, expr, e->v.BinOp.left);
    VISIT(c, expr, e->v.BinOp.right);
    ADDOP_BINARY(c, loc, e->v.BinOp.op);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case BinOp_kind:
    VISIT(c, expr, e->v.BinOp.left);
    VISIT(c, expr, e->v.BinOp.right);
    ADDOP_BINARY(c, loc, e->v.BinOp.op);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
```

CPython's Compiler

```
Assign(
    targets=[Name("y", Store())],
    value=BinOp(
        left=Name("dy", Load()),
        op=Add(),
        right=Attribute(
            value=Name("self", Load()),
            attr="y",
            ctx=Load(),
        ),
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST   (dy)
LOAD_FAST   (self)
LOAD_ATTR   (y)
BINARY_OP   (+)
STORE_FAST  (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
LOAD_FAST   (dy)
LOAD_FAST   (self)
LOAD_ATTR   (y)
BINARY_OP   (+)
STORE_FAST  (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Assign_kind:
    VISIT(c, expr, s->v.Assign.value);
    VISIT(c, expr, asdl_seq_GET(s->v.Assign.targets, i));
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Call_kind:
    VISIT(c, expr, e->v.Call.func);
    VISIT_SEQ(c, expr, e->v.Call.args);
    ADDOP_I(c, loc, CALL, asdl_seq_LEN(e->v.Call.args));
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL (type)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL (type)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL (type)
LOAD_FAST    (self)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Call_kind:
    VISIT(c, expr, e->v.Call.func);
    VISIT_SEQ(c, expr, e->v.Call.args);
    ADDOP_I(c, loc, CALL, asdl_seq_LEN(e->v.Call.args));
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL (type)
LOAD_FAST    (self)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Call_kind:
    VISIT(c, expr, e->v.Call.func);
    VISIT_SEQ(c, expr, e->v.Call.args);
    ADDOP_I(c, loc, CALL, asdl_seq_LEN(e->v.Call.args));
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
```

CPython's Compiler

```
Assign(
    targets=[Name("cls", Store())],
    value=Call(
        func=Name("type", Load()),
        args=[Name("self", Load())],
        keywords=[],
    ),
)
```

```
case Name_kind:
    compiler_nameop(c, loc, e->v.Name.id, e->v.Name.ctx);
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL (type)
LOAD_FAST    (self)
CALL         (1)
```

CPython's Compiler

```
if test:
    x = "Spam"
else:
    x = "eggs"
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
    LOAD_FAST          ("test")
    POP_JUMP_IF_FALSE (orelse)
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
        POP_JUMP_IF_FALSE (orelse)
        LOAD_CONST         ("eggs")
        STORE_FAST         (x)
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
        POP_JUMP_IF_FALSE  (orelse)
        LOAD_CONST         ("eggs")
        STORE_FAST         (x)
        JUMP               (end)
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
        POP_JUMP_IF_FALSE (orelse)
        LOAD_CONST         ("eggs")
        STORE_FAST         (x)
        JUMP               (end)
orelse:
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
        POP_JUMP_IF_FALSE  (orelse)
        LOAD_CONST         ("eggs")
        STORE_FAST         (x)
        JUMP               (end)
orelse: LOAD_CONST         ("eggs")
        STORE_FAST         (x)
```

CPython's Compiler

```
# if test:
#     x = "Spam"
# else:
#     x = "eggs"

If(
    test=Name(id="test", ctx=Load()),
    body=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="Spam"),
        )
    ],
    orelse=[
        Assign(
            targets=[Name(id="x", ctx=Store())],
            value=Constant(value="eggs"),
        )
    ]
)
```

```
case If_kind:
    NEW_JUMP_TARGET_LABEL(c, orelse);
    NEW_JUMP_TARGET_LABEL(c, end);
    VISIT(c, expr, s->v.If.test);
    ADDOP_JUMP(c, LOC(e), POP_JUMP_IF_FALSE, orelse);
    VISIT_SEQ(c, stmt, s->v.If.body);
    ADDOP_JUMP(c, NO_LOCATION, JUMP, end);
    USE_LABEL(c, orelse);
    VISIT_SEQ(c, stmt, s->v.If.orelse);
    USE_LABEL(c, end);
```

```
        LOAD_FAST          ("test")
        POP_JUMP_IF_FALSE  (orelse)
        LOAD_CONST         ("eggs")
        STORE_FAST         (x)
        JUMP               (end)
orelse: LOAD_CONST         ("eggs")
        STORE_FAST         (x)
end:
```

Other Optimizations

Other Optimizations

- Peephole optimizations
  - More constant folding
    - `[LOAD_CONST("Spam"), LOAD_CONST("eggs"), BUILD_TUPLE(2)]`
    - `[NOP(), NOP(), LOAD_CONST(("Spam", "eggs"))]`
  - Basic dead code elimination
    - `[LOAD_CONST(False), POP_JUMP_IF_TRUE(...)]`
    - `[NOP(), NOP()]`
  - Limited jump threading
    - `JUMP_IF_TRUE_OR_POP(a) -> POP_JUMP_IF_TRUE(b)`
    - `POP_JUMP_IF_TRUE(b)`
  - Some instruction combining
    - `[LOAD_CONST(None), IS_OP(1), POP_JUMP_IF_TRUE(...)]`
    - `[NOP(), NOP(), POP_JUMP_IF_NOT_NONE(...)]`
    - `[LOAD_CONST(42), RETURN_VALUE()]`
    - `[NOP(), RETURN_CONST(42)]`
  - `SWAP`-timization
- Basic liveness analysis
- Move "statically cold" code out-of-line
- Small exit block inlining

CPython's Assembler

CPython's Assembler

- Fix up jumps
- Compute all jump offsets
- Emit all instructions

CPython's Bytecode

CPython's Bytecode

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST          (dy)
LOAD_FAST          (self)
LOAD_ATTR          (y)
BINARY_OP          (+)
STORE_FAST         (y)
LOAD_GLOBAL        (type)
LOAD_FAST          (self)
CALL               (1)
STORE_FAST         (cls)
```

Superinstructions

`dis.dis(Point.shifted)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST                (dy)
LOAD_FAST                (self)
LOAD_ATTR                (y)
BINARY_OP                (+)
STORE_FAST               (y)
LOAD_GLOBAL              (type)
LOAD_FAST                (self)
CALL                     (1)
STORE_FAST               (cls)
```

Superinstructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST                (dy)
LOAD_FAST                (self)
LOAD_ATTR                (y)
BINARY_OP                (+)
STORE_FAST               (y)
LOAD_GLOBAL              (type)
LOAD_FAST                (self)
CALL                     (1)
STORE_FAST               (cls)
```

Superinstructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST                (dy)
LOAD_FAST                (self)
LOAD_ATTR                (y)
BINARY_OP                (+)
STORE_FAST               (y)
LOAD_GLOBAL              (type)
LOAD_FAST                (self)
CALL                     (1)
STORE_FAST               (cls)
```

Superinstructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR                 (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Adaptive Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR                 (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Adaptive Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR                 (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR                 (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR                 (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

LOAD_ATTR_INSTANCE_VALUE

- Check if the class is the same as last time.
- Check if the object's `__dict__` keys are the same as last time.
- Reach directly into the `__dict__` values at the same offset as last time.
- Return the result.

LOAD_ATTR_INSTANCE_VALUE

- Check if the class is the same as last time.
- Check if the object's `__dict__` keys are the same as last time.
- Reach directly into the `__dict__` values at the same offset as last time.
- Return the result.

LOAD_ATTR_INSTANCE_VALUE

```
inst(LOAD_ATTR, (owner -- res2 if (oparg & 1), res)) {
    PyObject *name = GETITEM(names, oparg >> 1);
    if (oparg & 1) {
        PyObject* meth = NULL;
        if (_PyObject_GetMethod(owner, name, &meth)) {
            res2 = meth;
            res = owner;
        }
        else {
            Py_DECREF(owner);
            ERROR_IF(meth == NULL, error);
            res2 = NULL;
            res = meth;
        }
    }
    else {
        res = PyObject_GetAttr(owner, name);
        Py_DECREF(owner);
        ERROR_IF(res == NULL, error);
    }
}
```

LOAD_ATTR_INSTANCE_VALUE

```
inst(LOAD_ATTR_INSTANCE_VALUE, (owner -- res2 if (oparg & 1), res)) {
    PyTypeObject *tp = Py_TYPE(owner);
    DEOPT_IF(tp->tp_version_tag != type_version, LOAD_ATTR);
    PyDictOrValues dorv = *_PyObject_DictOrValuesPointer(owner);
    DEOPT_IF(!_PyDictOrValues_IsValues(dorv), LOAD_ATTR);
    res = _PyDictOrValues_GetValues(dorv)->values[index];
    DEOPT_IF(res == NULL, LOAD_ATTR);
    Py_INCREF(res);
    res2 = NULL;
    Py_DECREF(owner);
}
```

LOAD_ATTR_INSTANCE_VALUE

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

`BINARY_OP_ADD_FLOAT`

- Check if the left object's class is `float`.
- Check if the right object's class is `float`.
- Add the values together.
- Return the result.

`BINARY_OP_ADD_FLOAT`

- Check if the left object's class is `float`.
- Check if the right object's class is `float`.
- Add the values together.
- Return the result.

BINARY_OP_ADD_FLOAT

```
inst(BINARY_OP, (lhs, rhs -- res)) {
    res = binary_ops[oparg](lhs, rhs);
    Py_DECREF(lhs);
    Py_DECREF(rhs);
    ERROR_IF(res == NULL, error);
}
```

BINARY_OP_ADD_FLOAT

```
inst(BINARY_OP_ADD_FLOAT, (left, right -- sum)) {
    DEOPT_IF(!PyFloat_CheckExact(left), BINARY_OP);
    DEOPT_IF(Py_TYPE(right) != Py_TYPE(left), BINARY_OP);
    double dsum = ((PyFloatObject *)left)->ob_fval + ((PyFloatObject *)right)->ob_fval;
    sum = PyFloat_FromDouble(dsum);
    _Py_DECREF_SPECIALIZED(right, _PyFloat_ExactDealloc);
    _Py_DECREF_SPECIALIZED(left, _PyFloat_ExactDealloc);
    ERROR_IF(sum == NULL, error);
}
```

BINARY_OP_ADD_FLOAT

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL               (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

LOAD_GLOBAL_BUILTIN

- Check if the globals dictionary keys are the same as last time.
- Check if the built-in dictionary keys are the same as last time.
- Reach directly into the built-in values at the same offset as last time.
- Return the result.

LOAD_GLOBAL_BUILTIN

- Check if the globals dictionary keys are the same as last time.
- Check if the built-in dictionary keys are the same as last time.
- Reach directly into the built-in values at the same offset as last time.
- Return the result.

LOAD_GLOBAL_BUILTIN

```
inst(LOAD_GLOBAL, ( -- null if (oparg & 1), v)) {
    PyObject *name = GETITEM(names, oparg>>1);
    if (PyDict_CheckExact(GLOBALS()) && PyDict_CheckExact(BUILTINS())) {
        v = _PyDict_LoadGlobal((PyDictObject *)GLOBALS(), (PyDictObject *)BUILTINS(), name);
        if (v == NULL) {
            if (!_PyErr_Occurred(tstate)) {
                format_exc_check_arg(tstate, PyExc_NameError, NAME_ERROR_MSG, name);
            }
            ERROR_IF(true, error);
        }
        Py_INCREF(v);
    }
    else {
        v = PyObject_GetItem(GLOBALS(), name);
        if (v == NULL) {
            ERROR_IF(!_PyErr_ExceptionMatches(tstate, PyExc_KeyError), error);
            _PyErr_Clear(tstate);
            v = PyObject_GetItem(BUILTINS(), name);
            if (v == NULL) {
                if (_PyErr_ExceptionMatches(tstate, PyExc_KeyError)) {
                    format_exc_check_arg(tstate, PyExc_NameError, NAME_ERROR_MSG, name);
                }
                ERROR_IF(true, error);
            }
        }
    }
    null = NULL;
}
```

LOAD_GLOBAL_BUILTIN

```
inst(LOAD_GLOBAL_BUILTIN, ( -- null if (oparg & 1), res)) {
    DEOPT_IF(!PyDict_CheckExact(GLOBALS()), LOAD_GLOBAL);
    DEOPT_IF(!PyDict_CheckExact(BUILTINS()), LOAD_GLOBAL);
    PyDictObject *mdict = (PyDictObject *)GLOBALS();
    PyDictObject *bdict = (PyDictObject *)BUILTINS();
    DEOPT_IF(mdict->ma_keys->dk_version != mod_version, LOAD_GLOBAL);
    DEOPT_IF(bdict->ma_keys->dk_version != bltn_version, LOAD_GLOBAL);
    PyDictUnicodeEntry *entries = DK_UNICODE_ENTRIES(bdict->ma_keys);
    res = entries[index].me_value;
    DEOPT_IF(res == NULL, LOAD_GLOBAL);
    Py_INCREF(res);
    null = NULL;
}
```

LOAD_GLOBAL_BUILTIN

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL                      (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

`CALL_NO_KW_TYPE_1`

- Check if the object we're calling is still `type`.
- Get the argument's class.
- Return the result.

`CALL_NO_KW_TYPE_1`

- Check if the object we're calling is still `type`.
- Get the argument's class.
- Return the result.

CALL_NO_KW_TYPE_1

```
inst(CALL, (method, callable, args[oparg] -- res)) {
    int is_meth = method != NULL;
    int total_args = oparg;
    if (is_meth) {
        callable = method;
        args--;
        total_args++;
    }
    if (!is_meth && Py_TYPE(callable) == &PyMethod_Type) {
        is_meth = 1;
        args--;
        total_args++;
        PyObject *self = ((PyMethodObject *)callable)->im_self;
        args[0] = Py_NewRef(self);
        method = ((PyMethodObject *)callable)->im_func;
        args[-1] = Py_NewRef(method);
        Py_DECREF(callable);
        callable = method;
    }
    int positional_args = total_args - KWNAMES_LEN();
    if (Py_TYPE(callable) == &PyFunction_Type && tstate->interp->eval_frame == NULL && ((PyFunctionObject *)callable)->vectorcall == _PyFunction_Vectorcall) {
        int code_flags = ((PyCodeObject*)PyFunction_GET_CODE(callable))->co_flags;
        PyObject *locals = code_flags & CO_OPTIMIZED ? NULL : Py_NewRef(PyFunction_GET_GLOBALS(callable));
        _PyInterpreterFrame *new_frame = _PyEvalFramePushAndInit(tstate, (PyFunctionObject *)callable, locals, args, positional_args, kwnames);
        kwnames = NULL;
        STACK_SHRINK(oparg + 2);
        if (new_frame == NULL) {
            goto error;
        }
        JUMPBY(INLINE_CACHE_ENTRIES_CALL);
        DISPATCH_INLINED(new_frame);
    }
    if (cframe.use_tracing) {
        res = trace_call_function(tstate, callable, args, positional_args, kwnames);
    }
    else {
        res = PyObject_Vectorcall(callable, args, positional_args | PY_VECTORCALL_ARGUMENTS_OFFSET, kwnames);
    }
    kwnames = NULL;
    Py_DECREF(callable);
    for (int i = 0; i < total_args; i++) {
        Py_DECREF(args[i]);
    }
    ERROR_IF(res == NULL, error);
    CHECK_EVAL_BREAKER();
}
```

CALL_NO_KW_TYPE_1

```
inst(CALL_NO_KW_TYPE_1, (null, callable, args[oparg] -- res)) {
    DEOPT_IF(null != NULL, CALL);
    PyObject *obj = args[0];
    DEOPT_IF(callable != (PyObject *)&PyType_Type, CALL);
    res = Py_NewRef(Py_TYPE(obj));
    Py_DECREF(obj);
    Py_DECREF(&PyType_Type);
}
```

CALL_NO_KW_TYPE_1

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP                 (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Specialized Instructions

`dis.dis(Point.shifted, adaptive=True)`

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

dis.dis(Point.shifted, adaptive=True)

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

`dis.dis(Point.shifted, adaptive=True, show_caches=True)`

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

`dis.dis(Point.shifted, adaptive=True, show_caches=True)`

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
```

```
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

`dis.dis(Point.shifted, adaptive=True, show_caches=True)`

```
LOAD_FAST__LOAD_FAST      (dy, self)
CACHE                     (oparg)
LOAD_ATTR_INSTANCE_VALUE  (y)
CACHE                     (counter)
CACHE                     (version)
CACHE                     (version)
CACHE                     (version)
CACHE                     (version)
CACHE                     (pointer)
CACHE                     (pointer)
CACHE                     (pointer)
CACHE                     (pointer)
BINARY_OP_ADD_INT         (+)
CACHE                     (counter)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
CACHE                     (counter)
CACHE                     (index)
CACHE                     (version)
CACHE                     (version)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
CACHE                     (counter)
CACHE                     (version)
CACHE                     (version)
CACHE                     (args)
STORE_FAST                (cls)
```

Inline Caches

`dis.dis(Point.shifted, adaptive=True, show_caches=True)`

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
```

```
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

dis.dis(Point.shifted, adaptive=True)

```
LOAD_FAST__LOAD_FAST      (dy, self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_INT         (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
```

Inline Caches

`dis.dis(Point.shifted)`

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

Bytecode

`dis.dis(Point.shifted, adaptive=True)`

```
RESUME                          
LOAD_FAST__LOAD_FAST      (dx)  
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (x)   
BINARY_OP_ADD_FLOAT       (+)   
STORE_FAST__LOAD_FAST     (x)   
LOAD_FAST__LOAD_FAST      (dy)  
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (y)   
BINARY_OP_ADD_FLOAT       (+)   
STORE_FAST                (y)   
```

```
LOAD_GLOBAL_BUILTIN   (type)
LOAD_FAST             (self)
CALL_NO_KW_TYPE_1     (1)
STORE_FAST            (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST  (cls)
LOAD_FAST__LOAD_FAST  (x)
LOAD_FAST             (y)
CALL                  (2)
RETURN_VALUE
```

Bytecode

`dis.dis(Point.shifted, adaptive=True)`

```
RESUME
LOAD_FAST__LOAD_FAST      (dx)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (x)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST__LOAD_FAST     (x)
LOAD_FAST__LOAD_FAST      (dy)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
```

```
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST      (cls)
LOAD_FAST__LOAD_FAST      (x)
LOAD_FAST                 (y)
CALL                      (2)
RETURN_VALUE
```

Bytecode

`dis.dis(Point.shifted, adaptive=True)`

```
RESUME
LOAD_FAST__LOAD_FAST      (dx)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (x)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST__LOAD_FAST     (x)
LOAD_FAST__LOAD_FAST      (dy)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
LOAD_GLOBAL_BUILTIN       (type)
LOAD_FAST                 (self)
CALL_NO_KW_TYPE_1         (1)
STORE_FAST                (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST      (cls)
LOAD_FAST__LOAD_FAST      (x)
LOAD_FAST                 (y)
CALL                      (2)
RETURN_VALUE
```

Source Code

```
RESUME
LOAD_FAST__LOAD_FAST      (dx)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (x)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST__LOAD_FAST     (x)
LOAD_FAST__LOAD_FAST      (dy)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
```

```
LOAD_GLOBAL_BUILTIN   (type)
LOAD_FAST             (self)
CALL_NO_KW_TYPE_1     (1)
STORE_FAST            (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST  (cls)
LOAD_FAST__LOAD_FAST  (x)
LOAD_FAST             (y)
CALL                  (2)
RETURN_VALUE
```

Source Code

```
x = dx + self.x
```

```
LOAD_FAST__LOAD_FAST      (dy)
LOAD_FAST                 (self)
LOAD_ATTR_INSTANCE_VALUE  (y)
BINARY_OP_ADD_FLOAT       (+)
STORE_FAST                (y)
```

```
LOAD_GLOBAL_BUILTIN   (type)
LOAD_FAST             (self)
CALL_NO_KW_TYPE_1     (1)
STORE_FAST            (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST  (cls)
LOAD_FAST__LOAD_FAST  (x)
LOAD_FAST             (y)
CALL                  (2)
RETURN_VALUE
```

Source Code

```
x = dx + self.x

y = dy + self.y
```

```
LOAD_GLOBAL_BUILTIN   (type)
LOAD_FAST             (self)
CALL_NO_KW_TYPE_1     (1)
STORE_FAST            (cls)
PUSH_NULL
LOAD_FAST__LOAD_FAST  (cls)
LOAD_FAST__LOAD_FAST  (x)
LOAD_FAST             (y)
CALL                  (2)
RETURN_VALUE
```

Source Code

```
x = dx + self.x
```

```
y = dy + self.y
```

```
cls = type(self)
```

```
PUSH_NULL
LOAD_FAST__LOAD_FAST (cls)
LOAD_FAST__LOAD_FAST (x)
LOAD_FAST            (y)
CALL                 (2)
RETURN_VALUE
```

Source Code

```
x = dx + self.x
```

```
y = dy + self.y
```

```
cls = type(self)
```

```
return cls(x, y)
```

Source Code

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)
```

Source Code

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)
```

Specialist

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)
```

Specialist

$ specialist --blue point.py

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)
```

Specialist

$ specialist --blue point.py

```
import typing


class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def shifted(self, dx, dy):
        x = dx + self.x
        y = dy + self.y
        cls = type(self)
        return cls(x, y)


def actually_run_something():
    p = Point(3.14, 2.72)
    for i in range(100):
        p = p.shifted(19.95, 12.06)


if __name__ == "__main__":
    actually_run_something()
```

Specialist

$ specialist --blue point.py

```
def actually_run_something():
    p = Point(3.14, 2.72)
    for i in range(100):
        p = p.shifted(19.95, 12.06)


if __name__ == "__main__":
    actually_run_something()
```

Specialist

brandtbucher/specialist

Specialist

Specialist uses fine-grained location information to create visual representations of exactly *where* and *how* CPython 3.11's new specializing, adaptive interpreter optimizes your code.

```
def encode_decode(key: str, text: str) -> str:
    out = []
    for i, t in enumerate(text):
        k = key[i % len(key)]
        out.append(chr(ord(t) ^ ord(k)))
    return "".join(out)
```

Installation

Specialist supports CPython 3.11+ on all platforms.

To install, just run:

```
$ pip install specialist
```

Future Work

Register VM

Register VM

- Zhang, Qiang, Lei Xu, and Baowen Xu. "RegCPython: A Register-based Python Interpreter for Better Performance." *ACM Transactions on Architecture and Code Optimization* 20, no. 1 (2022): 1-25.

Register VM

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

Register VM

```
y = dy + self.y
cls = type(self)
```

```
LOAD_FAST    (dy)
LOAD_FAST    (self)
LOAD_ATTR    (y)
BINARY_OP    (+)
STORE_FAST   (y)
LOAD_GLOBAL  (type)
LOAD_FAST    (self)
CALL         (1)
STORE_FAST   (cls)
```

Register VM

```
y = dy + self.y
cls = type(self)
```

```
LOAD_ATTR    (%0, self, y)
BINARY_OP    (y, +, dy, %0)

LOAD_GLOBAL (%0, type)

CALL         (cls, %0, 1, self)
```

Register VM

```
y = dy + self.y
cls = type(self)
```

```
LOAD_ATTR    (%0, self, y)
BINARY_OP    (y, +, dy, %0)
LOAD_GLOBAL  (%0, type)
CALL         (cls, %0, 1, self)
```

Register VM

- Work was based on 3.10...
- Unsolved issues:
  - Keeping objects alive too long (perhaps indefinitely)
  - Clobber analysis

Lazy Basic Block Versioning

Lazy Basic Block Versioning

- Chevalier-Boisvert, Maxime, and Marc Feeley. "Simple and effective type check removal through lazy basic block versioning." *arXiv preprint arXiv:1411.0352* (2014).
- Chevalier-Boisvert, Maxime, Noah Gibbs, Jean Boussier, Si Xing Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. "YJIT: a basic block versioning JIT compiler for CRuby." In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*, pp. 25-32. 2021.

Lazy Basic Block Versioning

```
x = dx + self.x
y = dy + self.y
cls = type(self)
return cls(x, y)
```

Lazy Basic Block Versioning

```
x = dx + self.x
y = dy + self.y
cls = type(self)
return cls(x, y)
```

```
RESUME
LOAD_FAST__LOAD_FAST       (dx, self)
LOAD_ATTR_INSTANCE_VALUE   (x)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST__LOAD_FAST      (x, dy)
LOAD_FAST                  (self)
LOAD_ATTR_INSTANCE_VALUE   (y)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST                 (y)
LOAD_GLOBAL_BUILTIN        (type)
LOAD_FAST                  (self)
CALL_NO_KW_TYPE_1          (1)
STORE_FAST__LOAD_FAST      (cls, cls)
LOAD_FAST__LOAD_FAST       (x, y)
CALL_NO_KW_ALLOC_AND_INIT  (1)
RETURN_VALUE
```

Lazy Basic Block Versioning

```
RESUME
LOAD_FAST__LOAD_FAST       (dx, self)
LOAD_ATTR_INSTANCE_VALUE   (x)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST__LOAD_FAST      (x, dy)
LOAD_FAST                  (self)
LOAD_ATTR_INSTANCE_VALUE   (y)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST                 (y)
LOAD_GLOBAL_BUILTIN        (type)
LOAD_FAST                  (self)
CALL_NO_KW_TYPE_1          (1)
STORE_FAST__LOAD_FAST      (cls, cls)
LOAD_FAST__LOAD_FAST       (x, y)
CALL_NO_KW_ALLOC_AND_INIT (1)
RETURN_VALUE
```

Lazy Basic Block Versioning

```
RESUME
LOAD_FAST__LOAD_FAST       (dx, self)
LOAD_ATTR_INSTANCE_VALUE   (x)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST__LOAD_FAST      (x, dy)
LOAD_FAST                  (self)
LOAD_ATTR_INSTANCE_VALUE   (y)
BINARY_OP_ADD_FLOAT        (+)
STORE_FAST                 (y)
LOAD_GLOBAL_BUILTIN        (type)
LOAD_FAST                  (self)
CALL_NO_KW_TYPE_1          (1)
STORE_FAST__LOAD_FAST      (cls, cls)
LOAD_FAST__LOAD_FAST       (x, y)
CALL_NO_KW_ALLOC_AND_INIT (1)
RETURN_VALUE
```

Lazy Basic Block Versioning

```
BB-0A: RESUME
       LOAD_FAST__LOAD_FAST       (dx, self)
BB-1A: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)
BB-3A: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST                 (y)
BB-5A: LOAD_GLOBAL_BUILTIN        (type)
       LOAD_FAST                  (self)
BB-6A: CALL_NO_KW_TYPE_1          (1)
       STORE_FAST__LOAD_FAST      (cls, cls)
       LOAD_FAST__LOAD_FAST       (x, y)
BB-7A: CALL_NO_KW_ALLOC_AND_INIT  (1)
       RETURN_VALUE
```

Lazy Basic Block Versioning

```
BB-0A: RESUME
       LOAD_FAST__LOAD_FAST       (dx, self)
BB-1A: LOAD_ATTR_INSTANCE_VALUE   (x)        BB-1B: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)
BB-3A: LOAD_ATTR_INSTANCE_VALUE   (y)        BB-3B: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST                 (y)
BB-5A: LOAD_GLOBAL_BUILTIN        (type)
       LOAD_FAST                  (self)
BB-6A: CALL_NO_KW_TYPE_1          (1)
       STORE_FAST__LOAD_FAST      (cls, cls)
       LOAD_FAST__LOAD_FAST       (x, y)
BB-7A: CALL_NO_KW_ALLOC_AND_INIT  (1)        BB-7B: CALL_NO_KW_ALLOC_AND_INIT  (1)
       RETURN_VALUE                                 RETURN_VALUE
```

Lazy Basic Block Versioning

```
BB-0A: RESUME
       LOAD_FAST__LOAD_FAST       (dx, self)
BB-1A: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)
BB-3A: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST                 (y)
BB-5A: LOAD_GLOBAL_BUILTIN        (type)
       LOAD_FAST                  (self)
BB-6A: CALL_NO_KW_TYPE_1          (1)
       STORE_FAST__LOAD_FAST      (cls, cls)
       LOAD_FAST__LOAD_FAST       (x, y)
BB-7A: CALL_NO_KW_ALLOC_AND_INIT  (1)
       RETURN_VALUE
```

Lazy Basic Block Versioning

```
BB-0A: RESUME
       LOAD_FAST__LOAD_FAST       (dx, self)
BB-1A: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2A: BINARY_OP_ADD_FLOAT        (+)          BB-2B: BINARY_OP_ADD_INT          (+)
       STORE_FAST__LOAD_FAST      (x, dy)             STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)              LOAD_FAST                  (self)
BB-3A: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4A: BINARY_OP_ADD_FLOAT        (+)          BB-4B: BINARY_OP_ADD_INT          (+)
       STORE_FAST                 (y)                 STORE_FAST                 (y)
BB-5A: LOAD_GLOBAL_BUILTIN        (type)
       LOAD_FAST                  (self)
BB-6A: CALL_NO_KW_TYPE_1          (1)
       STORE_FAST__LOAD_FAST      (cls, cls)
       LOAD_FAST__LOAD_FAST       (x, y)
BB-7A: CALL_NO_KW_ALLOC_AND_INIT  (1)
       RETURN_VALUE
```

Lazy Basic Block Versioning

```
BB-0A: RESUME
       LOAD_FAST__LOAD_FAST       (dx, self)
BB-1A: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)
BB-3A: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4A: BINARY_OP_ADD_FLOAT        (+)
       STORE_FAST                 (y)
BB-5A: LOAD_GLOBAL_BUILTIN        (type)
       LOAD_FAST                  (self)
BB-6A: CALL_NO_KW_TYPE_1          (1)
       STORE_FAST__LOAD_FAST      (cls, cls)
       LOAD_FAST__LOAD_FAST       (x, y)
BB-7A: CALL_NO_KW_ALLOC_AND_INIT  (1)
       RETURN_VALUE
```

```
BB-1B: LOAD_ATTR_INSTANCE_VALUE   (x)
BB-2B: BINARY_OP_ADD_INT          (+)
       STORE_FAST__LOAD_FAST      (x, dy)
       LOAD_FAST                  (self)
BB-3B: LOAD_ATTR_INSTANCE_VALUE   (y)
BB-4B: BINARY_OP_ADD_INT          (+)
       STORE_FAST                 (y)
```

```
BB-7B: CALL_NO_KW_ALLOC_AND_INIT (1)
       RETURN_VALUE
```

Copy-and-Patch Compilation

Copy-and-Patch Compilation

- Xu, Haoran, and Fredrik Kjolstad. "Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode." *Proceedings of the ACM on Programming Languages* 5, no. OOPSLA (2021): 1-30.

Copy-and-Patch Compilation

```
#include "Python.h"

PyObject *MAGICALLY_GET_THE_CONSTANT(void);
int MAGICALLY_CONTINUE_EXECUTION(PyObject *);

int
binary_op_add_fast_const(PyObject *lhs)
{
    PyObject *rhs = MAGICALLY_GET_THE_CONSTANT();
    PyObject *sum = PyNumber_Add(lhs, rhs);
    if (sum == NULL) {
        return -1;
    }
    return MAGICALLY_CONTINUE_EXECUTION(sum);
}
```

Copy-and-Patch Compilation

```
#include "Python.h"

PyObject *MAGICALLY_GET_THE_CONSTANT(void);
int MAGICALLY_CONTINUE_EXECUTION(PyObject *);

int
binary_op_add_fast_const(PyObject *lhs)
{
    PyObject *rhs = MAGICALLY_GET_THE_CONSTANT();
    PyObject *sum = PyNumber_Add(lhs, rhs);
    if (sum == NULL) {
        return -1;
    }
    return MAGICALLY_CONTINUE_EXECUTION(sum);
}
```

Copy-and-Patch Compilation

```c
#include "Python.h"

extern PyObject _PATCHED_CONSTANT;
extern int _PATCHED_CONTINUATION(PyObject *);

int
binary_op_add_fast_const(PyObject *lhs)
{
    PyObject *rhs = &_PATCHED_CONSTANT;
    PyObject *sum = PyNumber_Add(lhs, rhs);
    if (sum == NULL) {
        return -1;
    }
    __attribute__((musttail))
    return _PATCHED_CONTINUATION(sum);
}
```

Copy-and-Patch Compilation

```
Disassembly of section __TEXT,__text:

0000000000000000 <_binary_op_add_fast_const>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 48 be 00 00 00 00 00 00 00 00 movabsq $0, %rsi
                0000000000000006:  X86_64_RELOC_UNSIGNED        __PATCHED_CONSTANT
       e: 48 b8 00 00 00 00 00 00 00 00 movabsq $0, %rax
                0000000000000010:  X86_64_RELOC_UNSIGNED        _PyNumber_Add
      18: ff d0                         callq   *%rax
      1a: 48 85 c0                      testq   %rax, %rax
      1d: 74 10                         je      0x2f <_binary_op_add_fast_const+0x2f>
      1f: 48 b9 00 00 00 00 00 00 00 00 movabsq $0, %rcx
                0000000000000021:  X86_64_RELOC_UNSIGNED        __PATCHED_CONTINUATION
      29: 48 89 c7                      movq    %rax, %rdi
      2c: 5d                            popq    %rbp
      2d: ff e1                         jmpq    *%rcx
      2f: b8 ff ff ff ff                movl    $4294967295, %eax       ## imm = 0xFFFFFFFF
      34: 5d                            popq    %rbp
      35: c3                            retq
```

Copy-and-Patch Compilation

```
Disassembly of section __TEXT,__text:

0000000000000000 <_binary_op_add_fast_const>:
       0: 55                                pushq   %rbp
       1: 48 89 e5                          movq    %rsp, %rbp
       4: 48 be 00 00 00 00 00 00 00 00     movabsq $0, %rsi
                0000000000000006:  X86_64_RELOC_UNSIGNED        __PATCHED_CONSTANT
       e: 48 b8 00 00 00 00 00 00 00 00     movabsq $0, %rax
                0000000000000010:  X86_64_RELOC_UNSIGNED        _PyNumber_Add
      18: ff d0                             callq   *%rax
      1a: 48 85 c0                          testq   %rax, %rax
      1d: 74 10                             je      0x2f <_binary_op_add_fast_const+0x2f>
      1f: 48 b9 00 00 00 00 00 00 00 00     movabsq $0, %rcx
                0000000000000021:  X86_64_RELOC_UNSIGNED        __PATCHED_CONTINUATION
      29: 48 89 c7                          movq    %rax, %rdi
      2c: 5d                                popq    %rbp
      2d: ff e1                             jmpq    *%rcx
      2f: b8 ff ff ff ff                    movl    $4294967295, %eax       ## imm = 0xFFFFFFFF
      34: 5d                                popq    %rbp
      35: c3                                retq
```

Copy-and-Patch Compilation

```
Disassembly of section __TEXT,__text:

0000000000000000 <_binary_op_add_fast_const>:
       0: 55                                pushq   %rbp
       1: 48 89 e5                          movq    %rsp, %rbp
       4: 48 be 00 00 00 00 00 00 00 00     movabsq $0, %rsi
                0000000000000006:  X86_64_RELOC_UNSIGNED        __PATCHED_CONSTANT
       e: 48 b8 00 00 00 00 00 00 00 00     movabsq $0, %rax
                0000000000000010:  X86_64_RELOC_UNSIGNED        _PyNumber_Add
      18: ff d0                             callq   *%rax
      1a: 48 85 c0                          testq   %rax, %rax
      1d: 74 10                             je      0x2f <_binary_op_add_fast_const+0x2f>
      1f: 48 b9 00 00 00 00 00 00 00 00     movabsq $0, %rcx
                0000000000000021:  X86_64_RELOC_UNSIGNED        __PATCHED_CONTINUATION
      29: 48 89 c7                          movq    %rax, %rdi
      2c: 5d                                popq    %rbp
      2d: ff e1                             jmpq    *%rcx
      2f: b8 ff ff ff ff                    movl    $4294967295, %eax       ## imm = 0xFFFFFFFF
      34: 5d                                popq    %rbp
      35: c3                                retq
```

Other Ideas

Other Ideas

- Hybrid stack/register VM using variable-length instructions.
- Deferred reference counting on the stack using tagged pointers.
- Better benchmarks, with more emphasis on modern idioms.
- ...?

faster-cpython/ideas

Python Developer’s Guide

Python Developer's Guide

- devguide.python.org
- devguide.python.org/internals
- devguide.python.org/internals/parser
- devguide.python.org/internals/compiler
- devguide.python.org/internals/interpreter
- devguide.python.org/internals/garbage-collector

Thank you!

@brandtbucher

Thank you!

@brandtbucher | brandt@python.org