Python Internals

A snake’s eye view

About Me

Ammar Askar

PhD student at Tech since 2019.

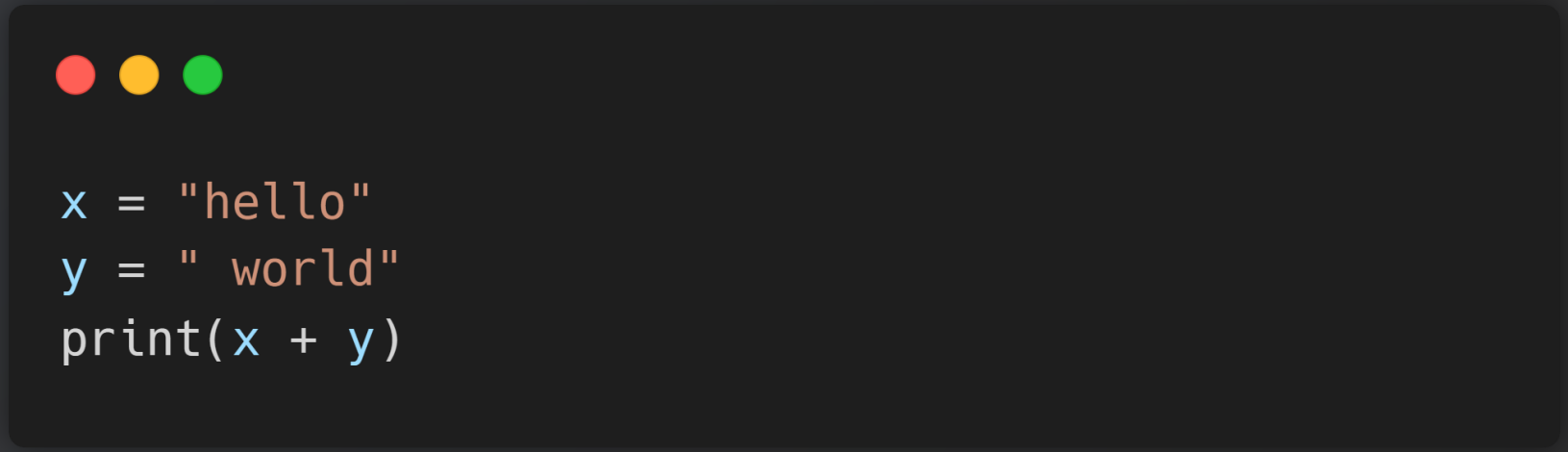
Researching security and programming languages.

Open-source enthusiast.

Python developer since 2016.

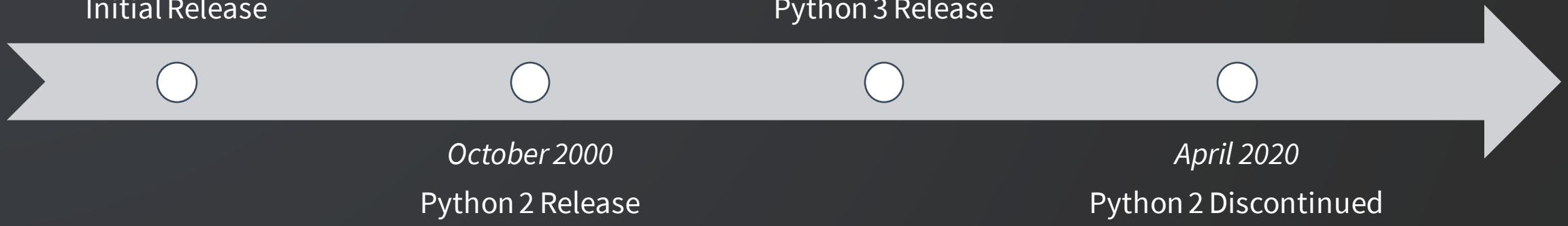
The Basics

- Interpreted language
- Fast prototyping
- Rich libraries

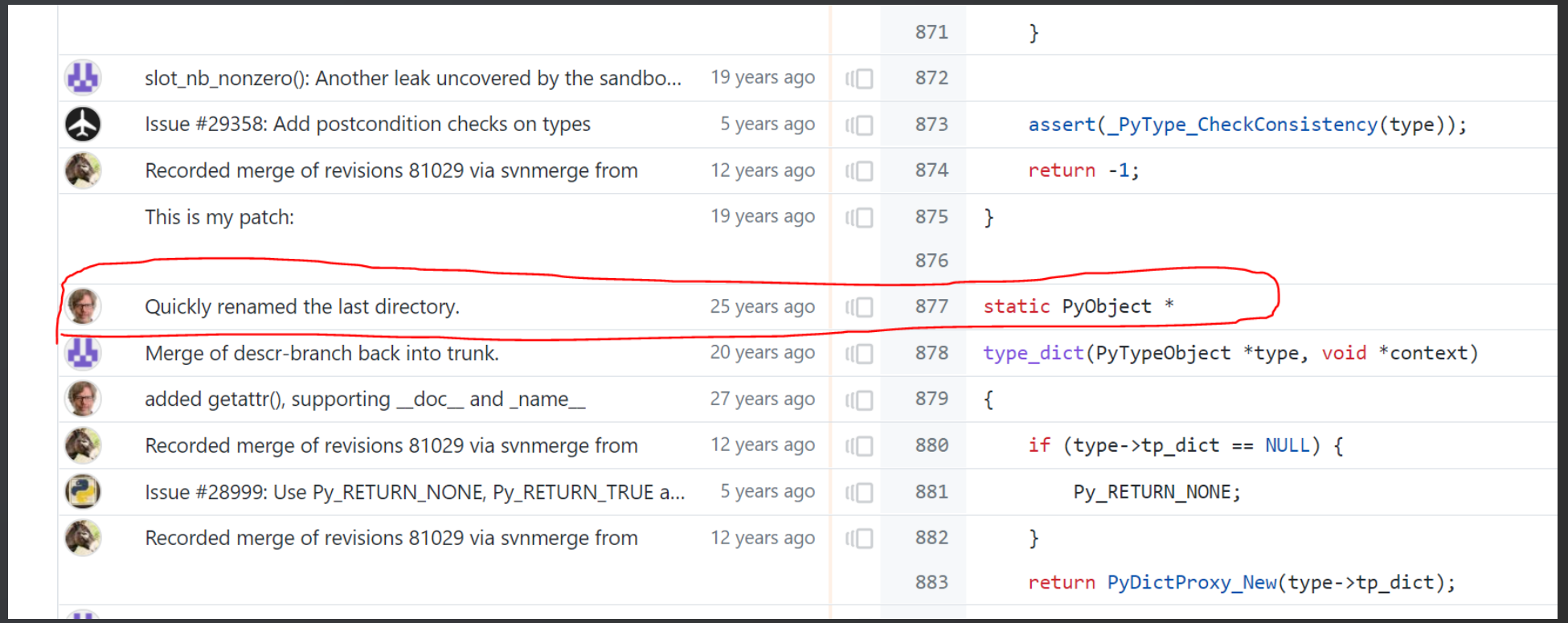


```
x = "hello"
y = " world"
print(x + y)
```

History


February 1991
Initial Release
October 2000
Python 2 Release
December 2008
Python 3 Release
April 2020
Python 2 Discontinued

History



		871 }
slot_nb_nonzero(): Another leak uncovered by the sandbo...	19 years ago	872
Issue #29358: Add postcondition checks on types	5 years ago	873 assert(_PyType_CheckConsistency(type));
Recorded merge of revisions 81029 via svnmerge from	12 years ago	874 return -1;
This is my patch:	19 years ago	875 }
		876
Quickly renamed the last directory.	25 years ago	877 static PyObject *
Merge of descr-branch back into trunk.	20 years ago	878 type_dict(PyTypeObject *type, void *context)
added getattr(), supporting __doc__ and _name__	27 years ago	879 {
Recorded merge of revisions 81029 via svnmerge from	12 years ago	880 if (type->tp_dict == NULL) {
Issue #28999: Use Py_RETURN_NONE, Py_RETURN_TRUE a...	5 years ago	881 Py_RETURN_NONE;
Recorded merge of revisions 81029 via svnmerge from	12 years ago	882 }
		883 return PyDictProxy_New(type->tp_dict);

Behind the Scenes

1. Compiler

Wait, *what*?

Why is there a **compiler** in an **interpreted** language?

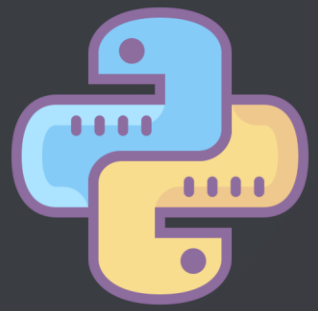
What are we interpreting?

Ruby

Ruby Virtual Machine (RVM) bytecode.

Java

Java Virtual Machine (JVM) bytecode.



Python

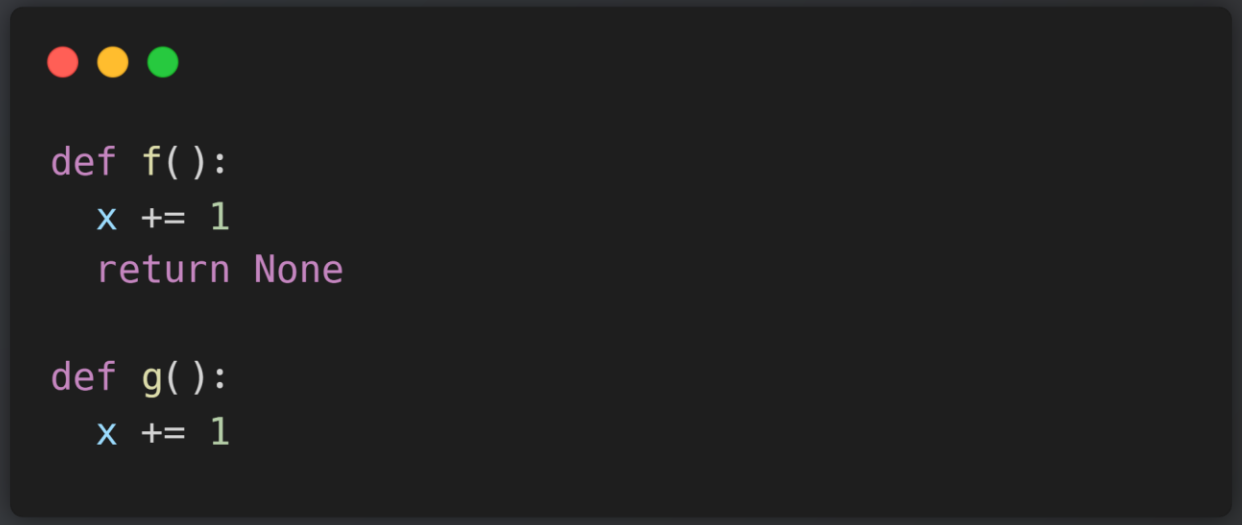
CPython interpreter bytecode.

- ~100 different opcodes
- Stack based machine, opcodes push and pop data to a stack.

```
case BINARY_ADD:
    w = POP();
    v = POP();
    u = add(ctx, v, w);
    PUSH(u);
    break;
```

Why add this bytecode layer?

Interpreting the AST directly gets harder as your language grows in complexity.



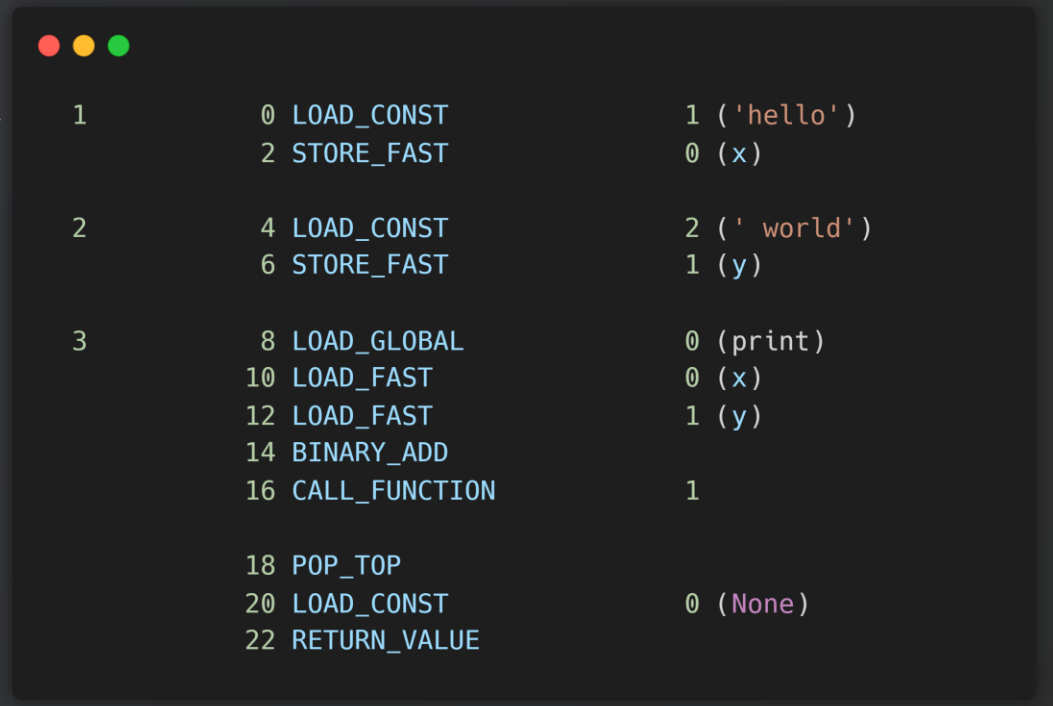
```
def f():
  x += 1
  return None

def g():
  x += 1
```

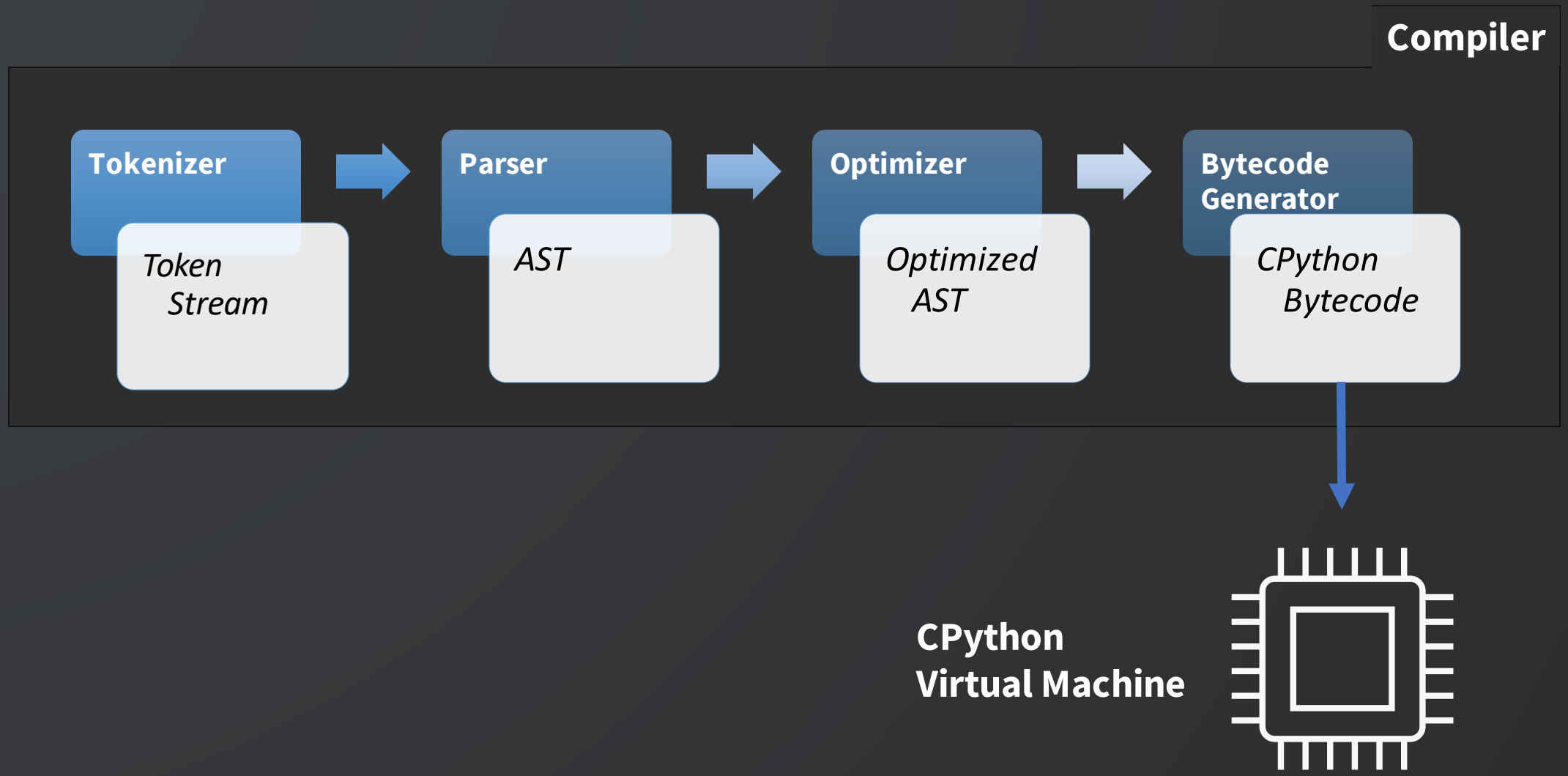
Bytecode is also much easier to optimize.

Compiler from Python to CPython Bytecode

```
1   x = "hello "
2   y = " world"
3   print(x + y)
```



(Actually) Behind the Scenes



Tokenizer

Converts Python source code to a stream of tokens.

```
1   x = "hello "
2   y = " world"
3   print(x + y)
```

```
1:          NAME        'x'
1:          OP          '='
1:          STRING      '"hello"'
1:          NEWLINE     '\r\n'

2:          NAME        'y'
2:          OP          '='
2:          STRING      '" world"'
2:          NEWLINE     '\r\n'

3:          NAME        'print'
3:          OP          '('
3:          NAME        'x'
3:          OP          '+'
3:          NAME        'y'
3:          OP          ')'
3:          NEWLINE     ''
```

Tokenizer

Implemented in hand-written C (tokenizer.c)

```c
char c = tok_nextc(tok);

/* Skip comment */
if (c == '#') {
  ...
}

/* Identifier (most frequent token!) */
if (is_potential_identifier_start(c)) {
  while (is_potential_identifier_char(c)) {
    c = tok_nextc(tok);
  }

  if (!verify_identifier(tok)) {
    return ERRORTOKEN;
  }

  return NAME;
}
```

Tokenizer

Examining its output

```
$ cat test.py
x = "hello"
y = " world"
print(x + y) f(500)

$ python -m tokenize test.py
0,0-0,0:            ENCODING       'utf-8'
1,0-1,1:            NAME           'x'
1,2-1,3:            OP             '='
1,4-1,11:           STRING         '"hello"'
1,11-1,13:          NEWLINE        '\r\n'
2,0-2,1:            NAME           'y'
2,2-2,3:            OP             '='
2,4-2,12:           STRING         '" world"'
2,12-2,14:          NEWLINE        '\r\n'
3,0-3,5:            NAME           'print'
3,5-3,6:            OP             '('
3,6-3,7:            NAME           'x'
3,8-3,9:            OP             '+'
3,10-3,11:          NAME           'y'
3,11-3,12:          OP             ')'
3,13-3,14:          NAME           'f'
3,14-3,15:          OP             '('
3,15-3,18:          NUMBER         '500'
3,18-3,19:          OP             ')'
3,19-3,20:          NEWLINE        ''
4,0-4,0:            ENDMARKER      ''
```

Exposed as a python module called **tokenize**.

Parser

Takes tokens and parses it into an Abstract Syntax Tree (AST) based on a grammar.

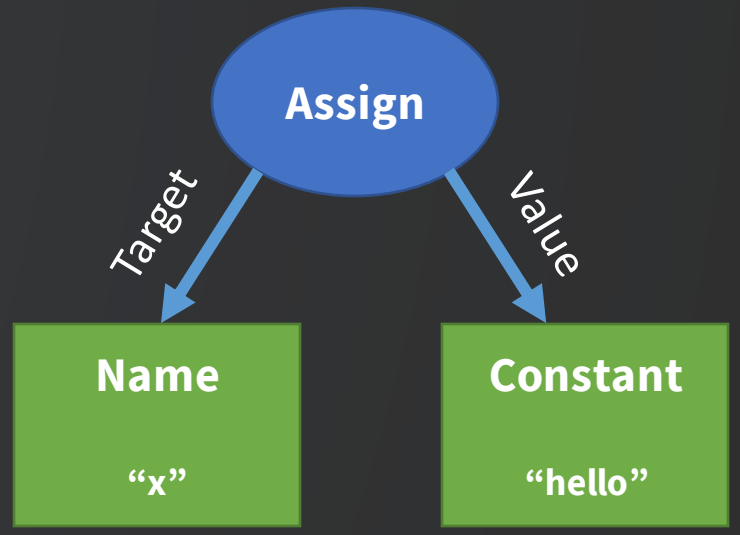
Python used to have an **LL(1) parser**, left-to-right-lookahead-1 parser but this required multiple hacks to support complex expressions.

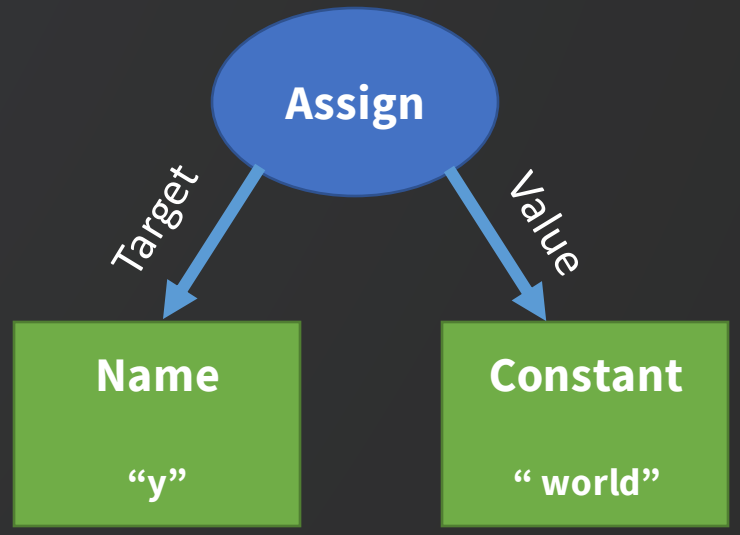
In 2020, the parser was replaced with a Parsing Expression Grammar (PEG) based parser.

Parser

Our parsed hello world example.

```
1   x = "hello "
2   y = " world"
3   print(x + y)
```

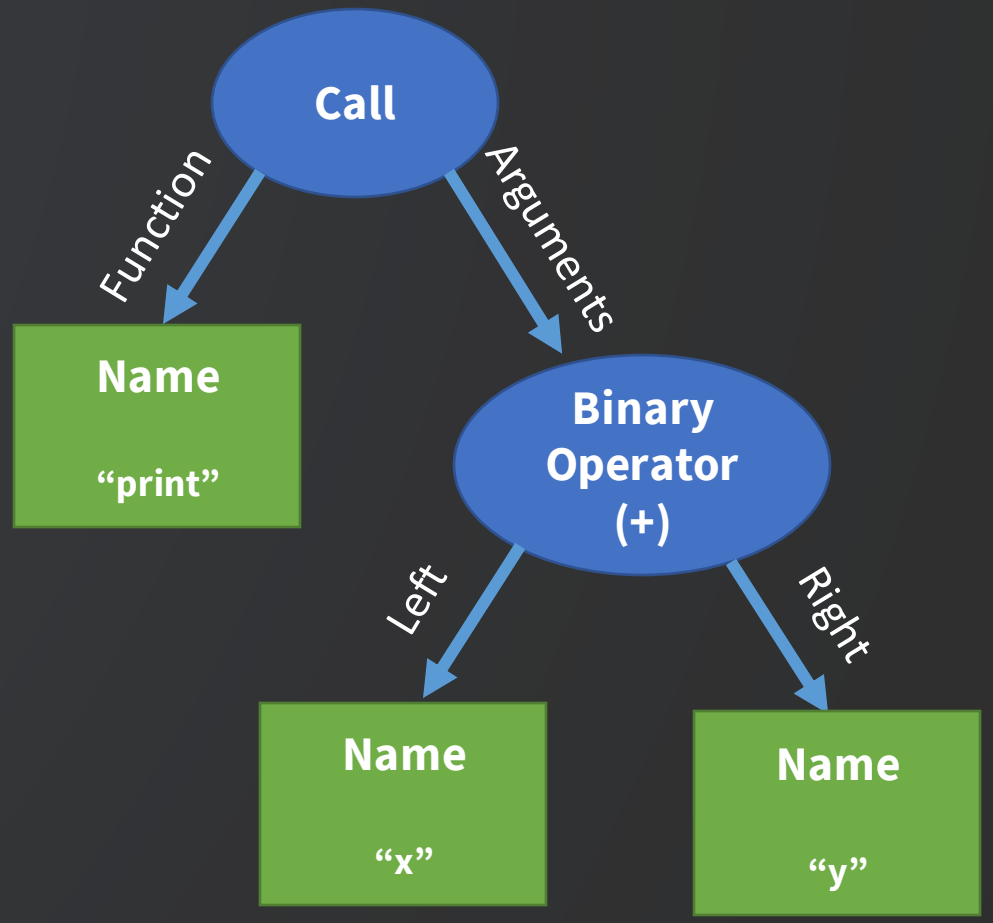




Parser

Our parsed hello world example.

```
1   x = "hello "
2   y = " world"
3   print(x + y)
```



Parser Grammar

Parser is generated from this Grammar file (python.gram)

```
assignment:
  | (NAME '=')+ expression
  | ...

expression:
  | NAME
  | 'True'
  | 'False'
  | list
  | ...
  | STRING


primary:
  | NAME '(' [arguments] ')'

arguments:
  | expression
  | expression ','
```

```
1  x = "hello "
2  y = " world"
3  print(x + y)
```

Parser

Examining its output

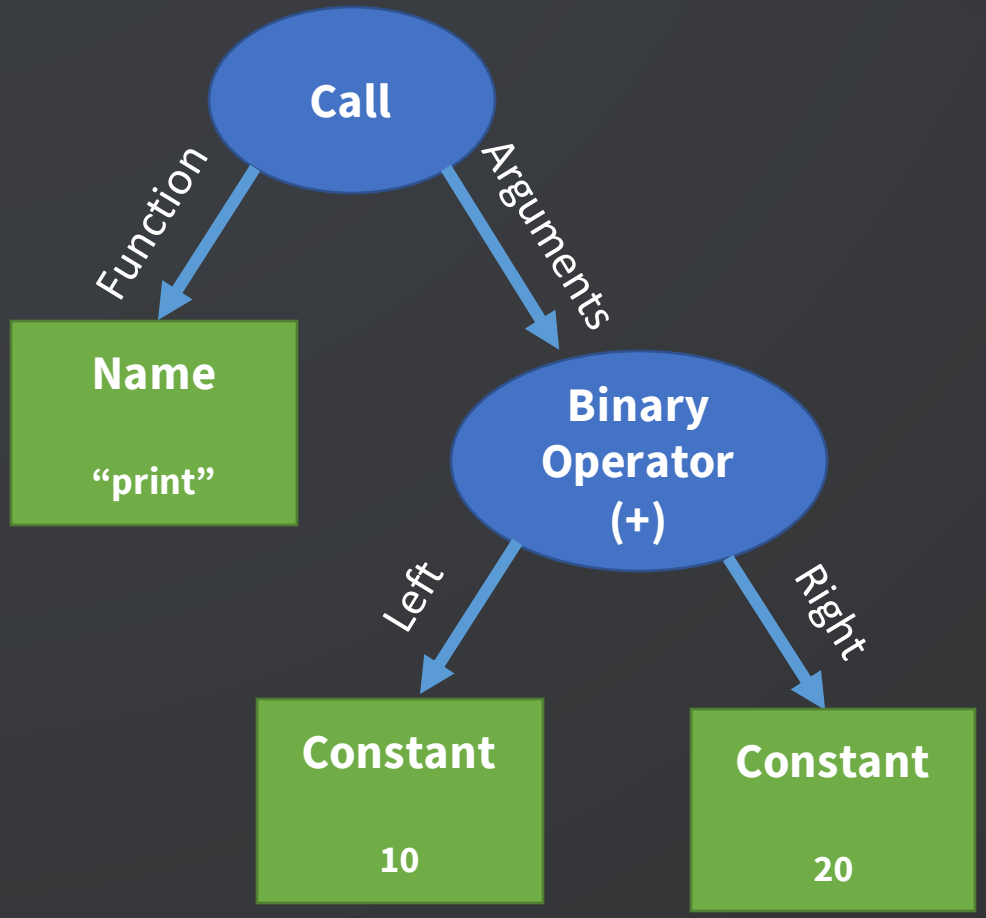
Exposed as a python module called **ast**.

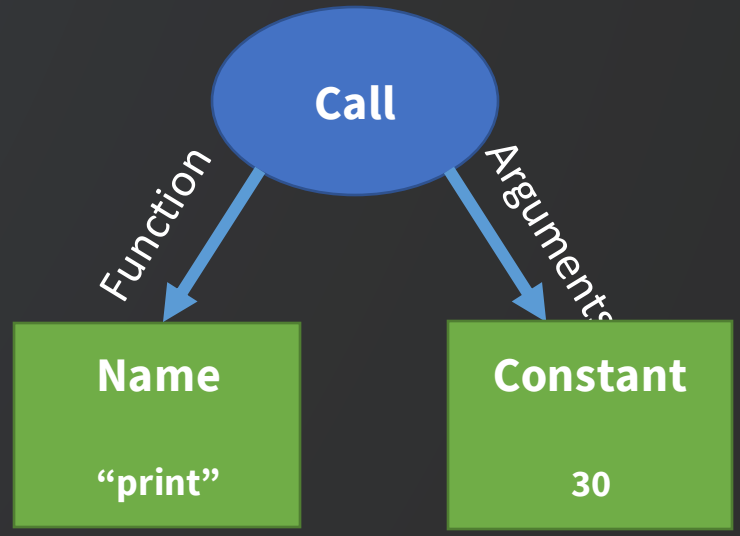
```
$ python -m ast test.py
Module(
   body=[
      Assign(
         targets=[
            Name(id='x', ctx=Store())],
         value=Constant(value='hello')),
      Assign(
         targets=[
            Name(id='y', ctx=Store())],
         value=Constant(value=' world')),
      Expr(
         value=Call(
            func=Name(id='print', ctx=Load()),
            args=[
               BinOp(
                  left=Name(id='x', ctx=Load()),
                  op=Add(),
                  right=Name(id='y', ctx=Load()))],
            keywords=[]))],
   type_ignores=[])
```

Optimizer

Simplifies the AST.

```
print(10 + 20)
```





Bytecode Generator

Takes the parsed and optimized AST and generates the bytecode (compile.c)

```c
static int
compiler_visit_stmt(struct compiler *c, stmt_ty s)
{
  ...

  case Assign_kind:
    VISIT(c, expr, s->v.Assign.value);
    VISIT(c, expr, s->v.Assign.target);
    break;
  case Constant_kind:
    ADDOP_LOAD_CONST(c, e->v.Constant.value);
    break;
  case Name_kind:
    compiler_nameop(c, e->v.Name.id, e->v.Name.ctx);
    break;
}

static int
compiler_nameop(struct compiler *c, identifier name)
{
  if (forbidden_name(c, name, ctx))
    return 0;

  ...
  return compiler_addop_i(c, LOAD_GLOBAL, arg);
}
```

Bytecode Generator

Examining its output

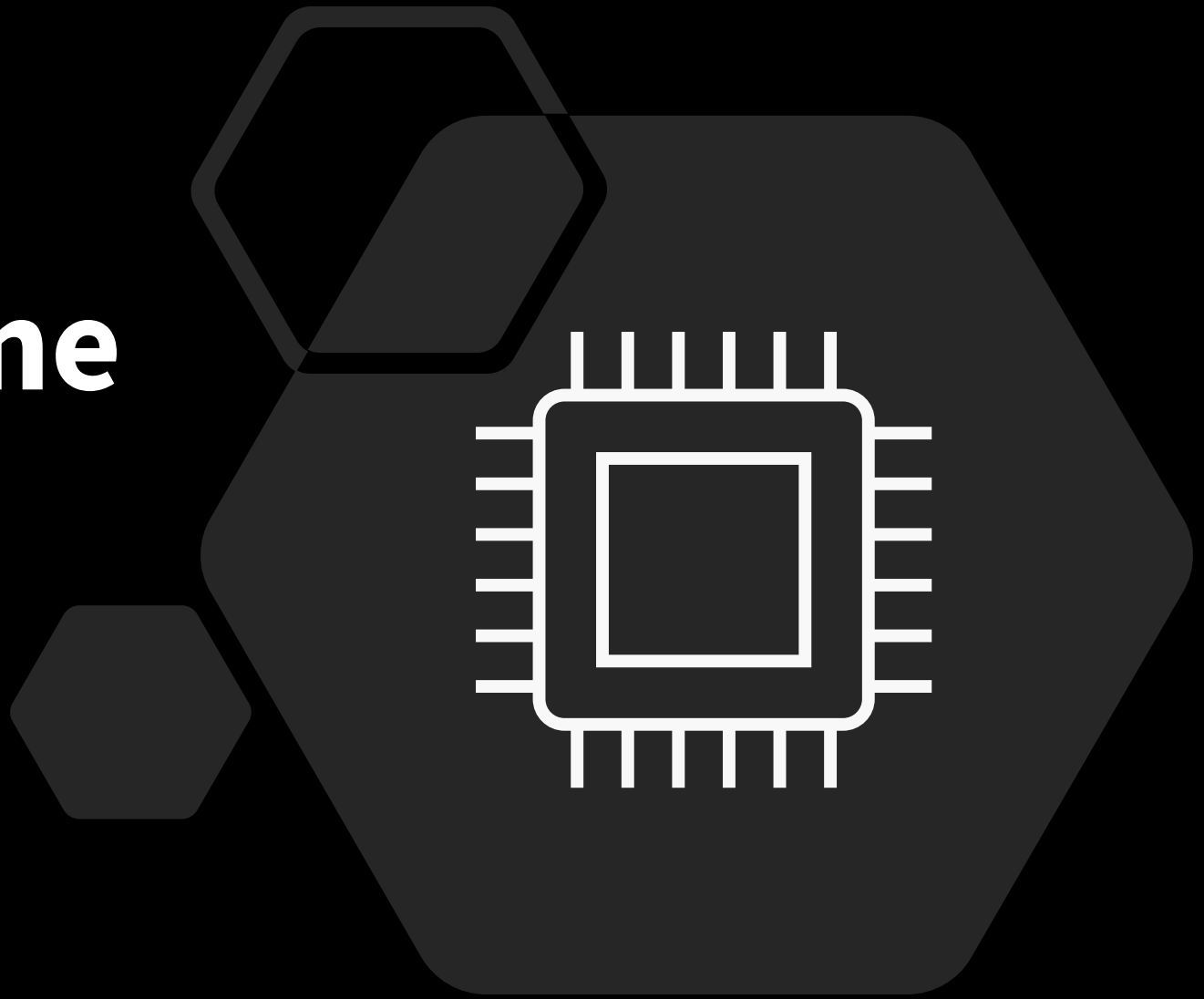
```
$ python -m dis test.py
  1           0 LOAD_CONST               0 ('hello')
              2 STORE_NAME               0 (x)

  2           4 LOAD_CONST               1 (' world')
              6 STORE_NAME               1 (y)

  3           8 LOAD_NAME                2 (print)
             10 LOAD_NAME                0 (x)
             12 LOAD_NAME                1 (y)
             14 BINARY_ADD
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               2 (None)
             22 RETURN_VALUE
```

Exposed as a python module called **dis**.

CPython Virtual Machine



The Evaluator

Essentially a giant switch case statement written in C that runs the bytecode produced by the compiler (ceval.c)

```
switch (opcode) {

  TARGET(NOP): {
    DISPATCH();
  }

  TARGET(POP_TOP): {
    PyObject *value = POP();
    Py_DECREF(value);
    DISPATCH();
  }

  ...
}
```

How does it run our example?

```
x = "hello"

---

LOAD_CONST    0 ('hello')
STORE_NAME    0 (x)
```

Locals Name	Value

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack

How does it run our example?

```
LOAD_CONST    0 ('hello')
STORE_NAME    0 (x)

---

TARGET(LOAD_CONST): {
  PyObject *value = GETITEM(consts, oparg);
  Py_INCREF(value);
  PUSH(value);
}

TARGET(STORE_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *v = POP();
  PyObject *ns = LOCALS();

  PyDict_SetItem(ns, name, v);
  Py_DECREF(v);
}
```

Locals Name	Value

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
“hello”

How does it run our example?

```
LOAD_CONST    0 ('hello')
STORE_NAME    0 (x)

---

TARGET(LOAD_CONST): {
  PyObject *value = GETITEM(consts, oparg);
  Py_INCREF(value);
  PUSH(value);
}

TARGET(STORE_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *v = POP();
  PyObject *ns = LOCALS();

  PyDict_SetItem(ns, name, v);
  Py_DECREF(v);
}
```

Locals Name	Value
x	“hello”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants	**Names**	**Stack**
“hello”	x	
“ world”	y	
	print	

How does it run our example?

```
y = " world"

---

LOAD_CONST    1 (' world')
STORE_NAME    1 (y)
```

Locals Name	Value
x	“hello”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack

How does it run our example?

```
LOAD_CONST   1 (' world')
STORE_NAME   1 (y)

---

TARGET(LOAD_CONST): {
  PyObject *value = GETITEM(consts, oparg);
  Py_INCREF(value);
  PUSH(value);
}

TARGET(STORE_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *v = POP();
  PyObject *ns = LOCALS();

  PyDict_SetItem(ns, name, v);
  Py_DECREF(v);
}
```

Locals Name	Value
x	“hello”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
“ world”

How does it run our example?

```
LOAD_CONST    1 (' world')
STORE_NAME    1 (y)

---

TARGET(LOAD_CONST): {
  PyObject *value = GETITEM(consts, oparg);
  Py_INCREF(value);
  PUSH(value);
}

TARGET(STORE_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *v = POP();
  PyObject *ns = LOCALS();

  PyDict_SetItem(ns, name, v);
  Py_DECREF(v);
}
```

Locals Name	Value
x	“hello”
y	“ world”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack

How does it run our example?

```
print(x + y)

---

LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1
```

Locals Name	Value
x	“hello”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack

How does it run our example?

```
LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1

---

TARGET(LOAD_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *locals = LOCALS();
  PyObject *v = PyDict_GetItem(locals, name);

  if (v == NULL) {
    v = PyDict_GetItemWithError(GLOBALS(), name);
  }

  PUSH(v);
}

TARGET(BINARY_ADD): {
  PyObject *right = POP();
  PyObject *left = POP();
  PyObject *sum = PyNumber_Add(left, right);
  PUSH(sum);
  Py_DECREF(left);
  Py_DECREF(right);
}
```

Locals Name	Value
x	“hello”
y	“ world”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
print

How does it run our example?

```
LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1

---

TARGET(LOAD_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *locals = LOCALS();
  PyObject *v = PyDict_GetItem(locals, name);

  if (v == NULL) {
    v = PyDict_GetItemWithError(GLOBALS(), name);
  }

  PUSH(v);
}

TARGET(BINARY_ADD): {
  PyObject *right = POP();
  PyObject *left = POP();
  PyObject *sum = PyNumber_Add(left, right);
  PUSH(sum);
  Py_DECREF(left);
  Py_DECREF(right);
}
```

Locals Name	Value
x	“hello”
y	“ world”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
“hello”
print

How does it run our example?

```
LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1

---

TARGET(LOAD_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *locals = LOCALS();
  PyObject *v = PyDict_GetItem(locals, name);

  if (v == NULL) {
    v = PyDict_GetItemWithError(GLOBALS(), name);
  }

  PUSH(v);
}

TARGET(BINARY_ADD): {
  PyObject *right = POP();
  PyObject *left = POP();
  PyObject *sum = PyNumber_Add(left, right);
  PUSH(sum);
  Py_DECREF(left);
  Py_DECREF(right);
}
```

Locals Name	Value
x	“hello”
y	“ world”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
“ world”
“hello”
print

How does it run our example?

```
LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1

---

TARGET(LOAD_NAME): {
  PyObject *name = GETITEM(names, oparg);
  PyObject *locals = LOCALS();
  PyObject *v = PyDict_GetItem(locals, name);

  if (v == NULL) {
    v = PyDict_GetItemWithError(GLOBALS(), name);
  }

  PUSH(v);
}

TARGET(BINARY_ADD): {
  PyObject *right = POP();
  PyObject *left = POP();
  PyObject *sum = PyNumber_Add(left, right);
  PUSH(sum);
  Py_DECREF(left);
  Py_DECREF(right);
}
```

Locals Name	Value
x	“hello”
y	“ world”

Globals Name	Value
print	Print function
open	Open function
__name__	Current module
…	…

Constants
“hello”
“ world”

Names
x
y
print

Stack
“hello world”
print

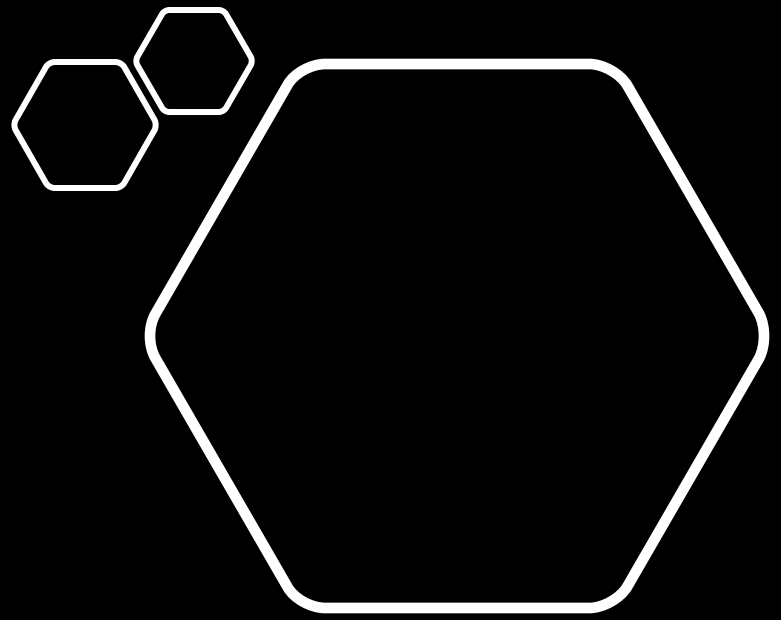
How does it run our example?

```
LOAD_NAME                2 (print)
LOAD_NAME                0 (x)
LOAD_NAME                1 (y)
BINARY_ADD
CALL_FUNCTION            1
```

hello world

Stack

Thank you!



Studying further:

- How functions like print and string concatenation are implemented in C (bltinmodule.c and unicodeobject.c)
- The PEG parser and how it is generated from the Grammar automatically https://devguide.python.org/parser/