Compiler Construction: Practical Introduction

Lecture 11
Case Study: the Python Virtual Machine

Eugene Zouev

Spring Semester 2023
Innopolis University

A Case Study:

The Python Virtual Machine

Is Python an Interpreted Language?

Is Python an Interpreted Language?

Although Python is not popularly regarded as a compiled language, **it actually is one**

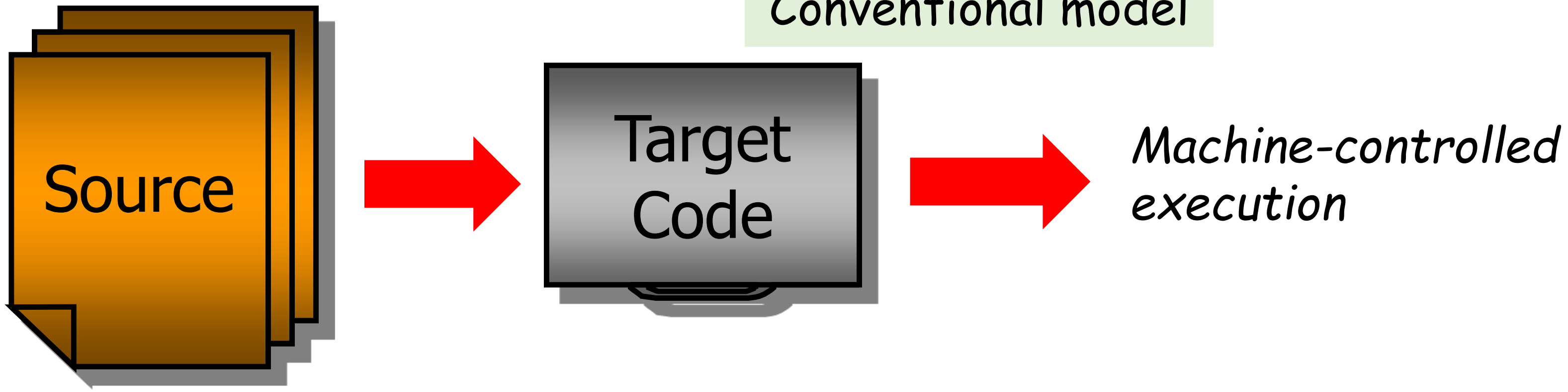
During compilation, some Python source code is transformed into bytecode that is executable **by the virtual machine.**

Is Python an Interpreted Language?

Although Python is not popularly regarded as a compiled language, **it actually is one**

During compilation, some Python source code is transformed into bytecode that is executable **by the virtual machine.**

Conventional model

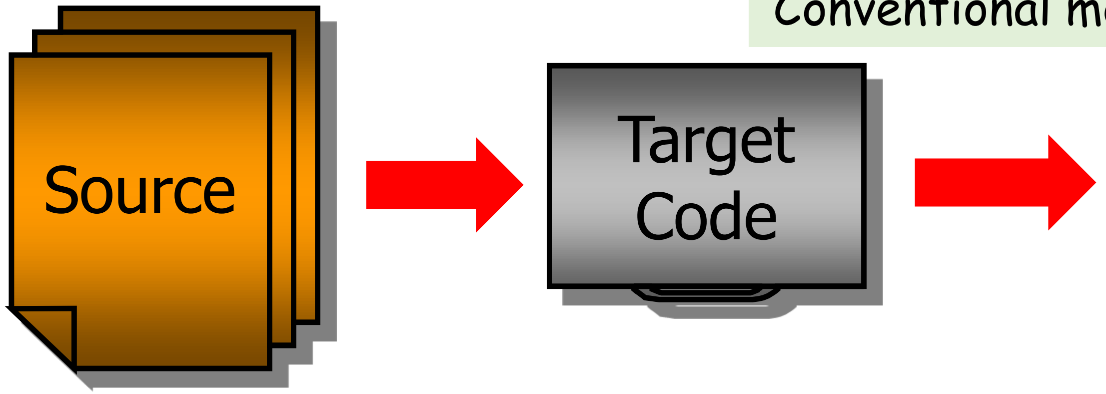


Is Python an Interpreted Language?

Although Python is not popularly regarded as a compiled language, **it actually is one**

During compilation, some Python source code is transformed into bytecode that is executable **by the virtual machine.**

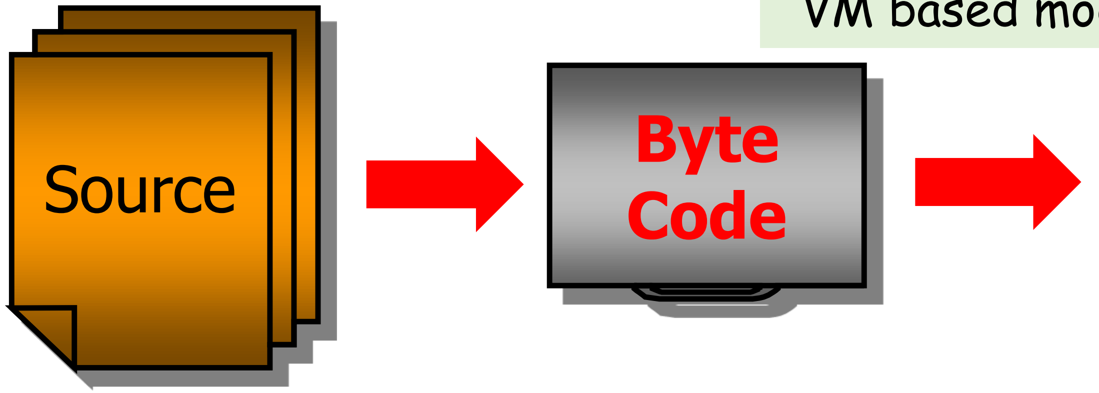
Conventional model



Machine-controlled execution

Which model does Java use??

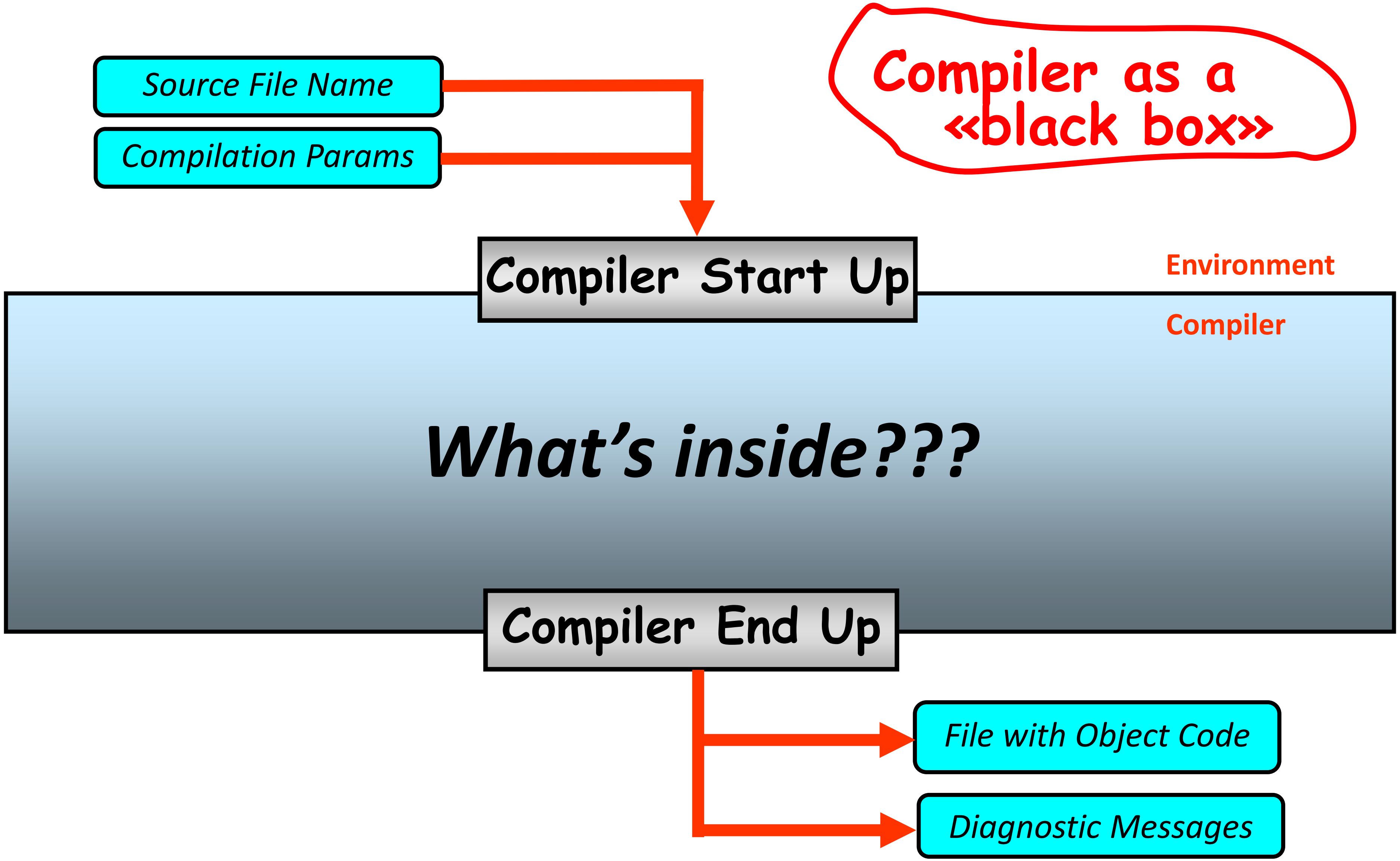
VM based model



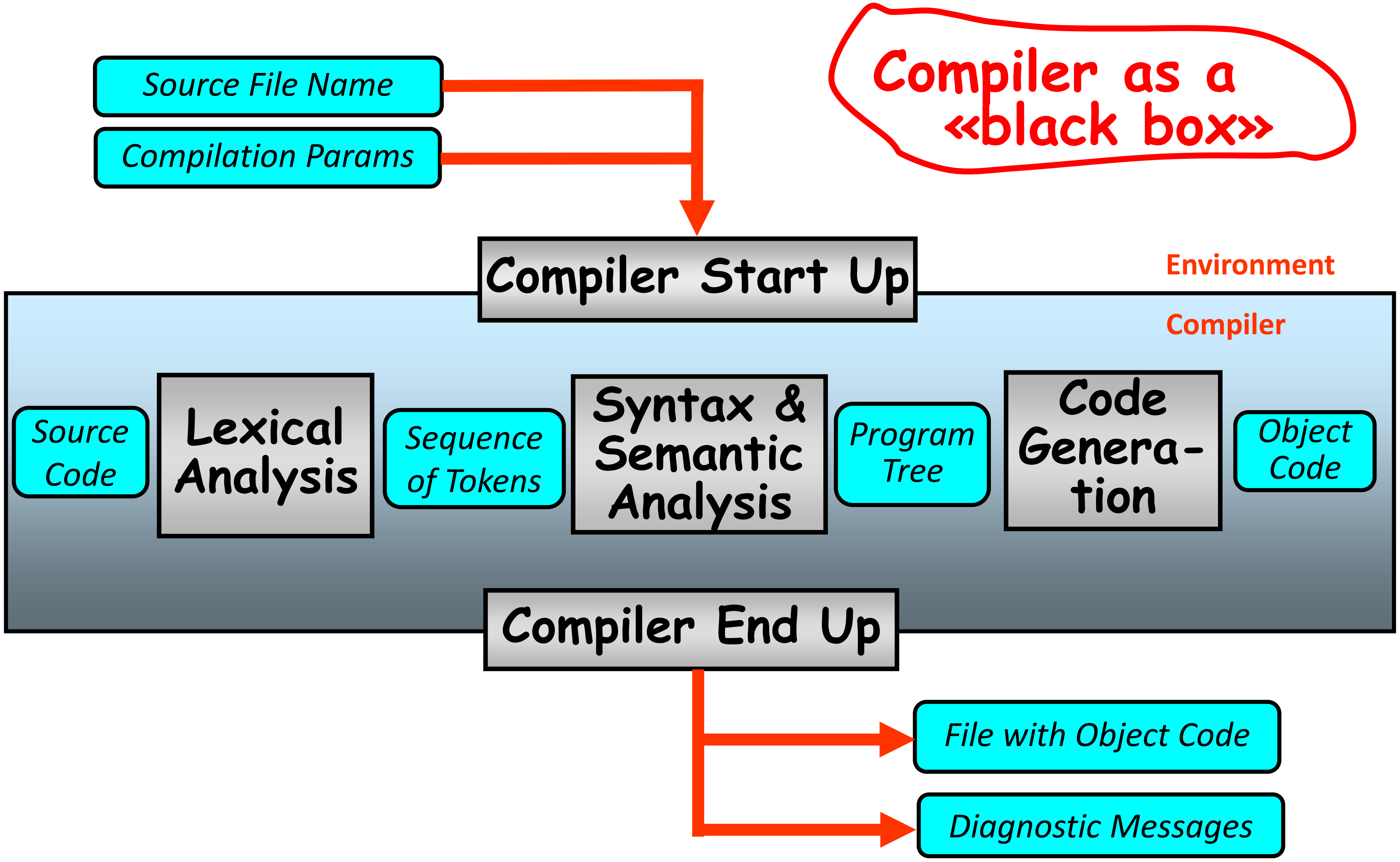
Execution by the virtual machine

Python as well!!!

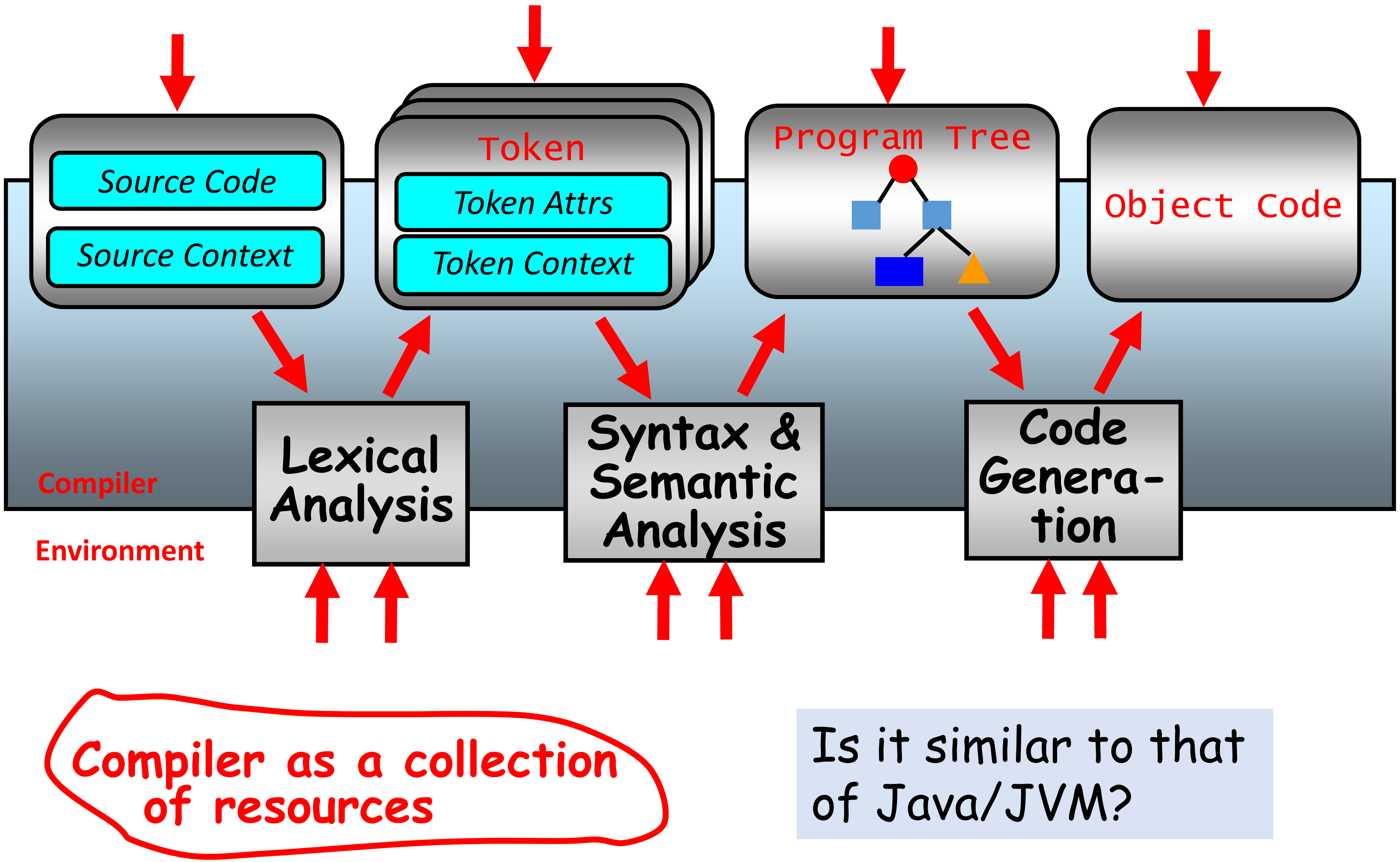
Compilation: Conventional Approach



Compilation: Conventional Approach



Compilation: Advanced Approach



Is it similar to that of Java/JVM?

Python Approach

So, as a conclusion:

- Python execution model assumes compilation to an intermediate code ("bytecode") for a specialized **virtual machine**.
- Python implementation allows **direct access** (APIs) to all compilation phases and intermediate data structures.

See the prev slide

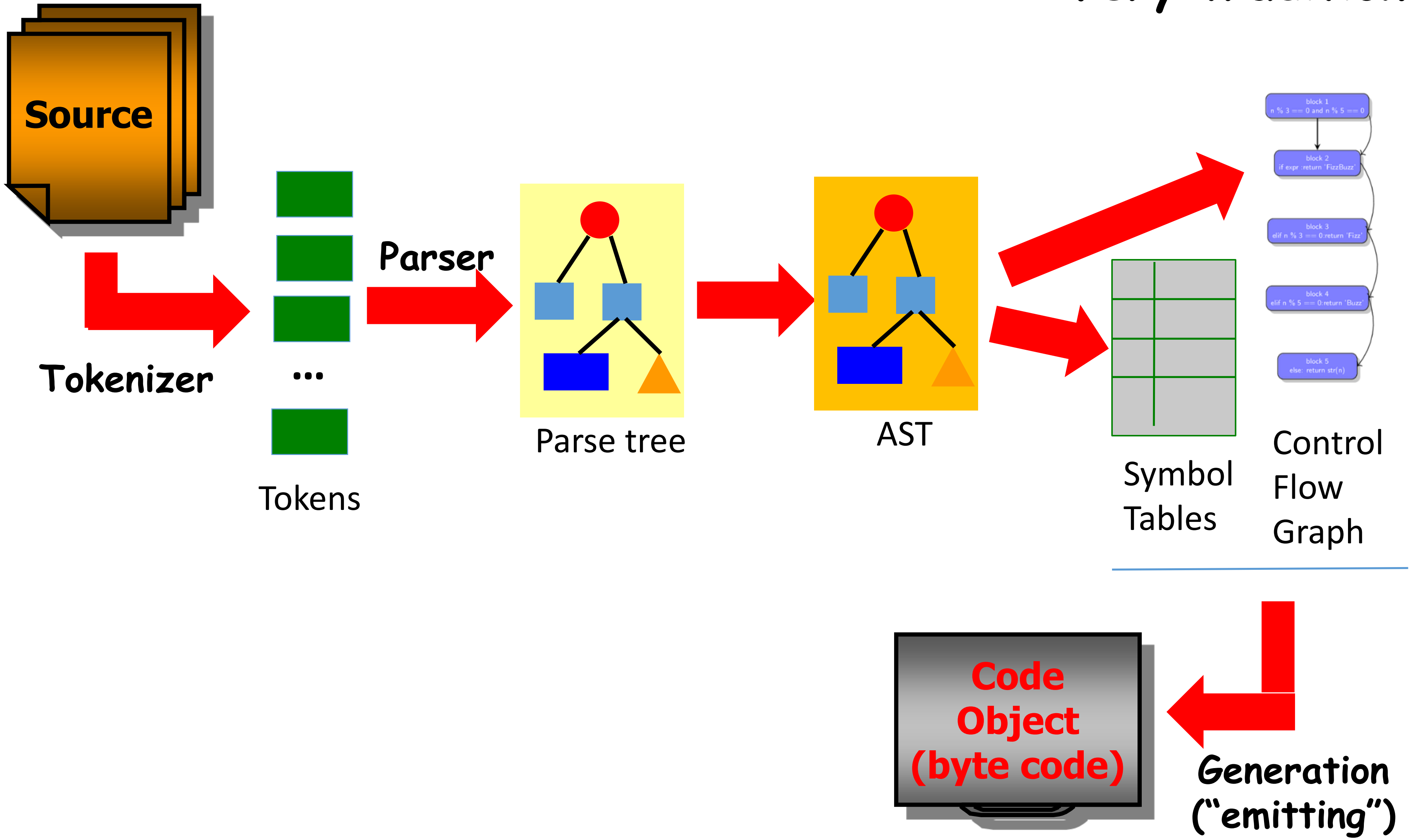
Python Compilation/Execution Model

- Parsing the Python source code into a parse tree.
- Transforming the parse tree into an abstract syntax tree (AST).
- Generation of the symbol table.
- Generation of the code object from the AST
  - Transforming the AST into a flow control graph.
  - Emitting a code object from the control flow graph.

- **Python code object gets executed under the control of the Python Virtual Machine.**

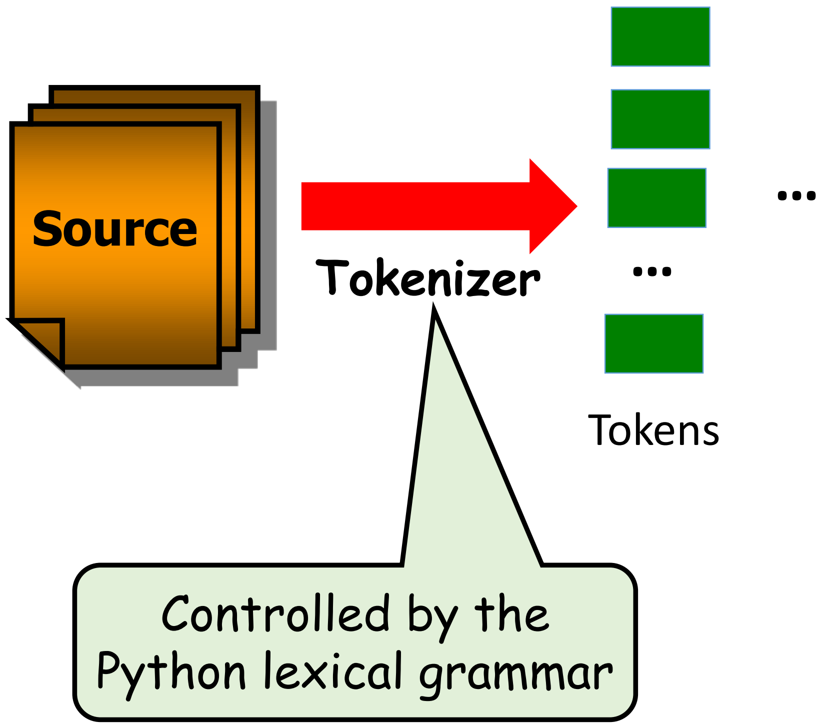
Python Compilation Process

Very traditional way!



1st Phase: Tokenization

The tokenization function breaks up the content of the module source into <u>legal python tokens</u> ("lexical grammar")



The tokens generated from the tokenizer are passed to the parser…

1st Phase: Tokenization

Python lexical grammar:
informal view

identifiers :

Names that defined by a programmer: function & variable names, class names etc. (The rules of identifiers are specified in the python documentation.)

operators

Special symbols: `+`, `*` that operate on data values and produce results.

delimiters:

Grouping expressions, provide punctuations and assignment: `(`, `)`, `{`, `}`, `=`, `*=` etc.

literals:

Symbols that provide a constant value of some type.

String and byte literals: `"Fred"`, `b"Fred"`, numeric literals: integer literals: `2`, floating point literals: `1e100` and imaginary literals: `10j`.

comments:

String literals that start with the hash symbol and end at the end of the physical line.

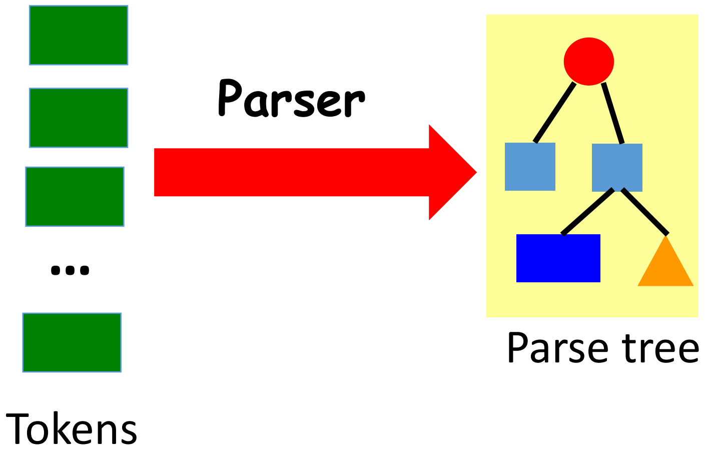
NEWLINE:

Special token that denotes the end of a logical line.

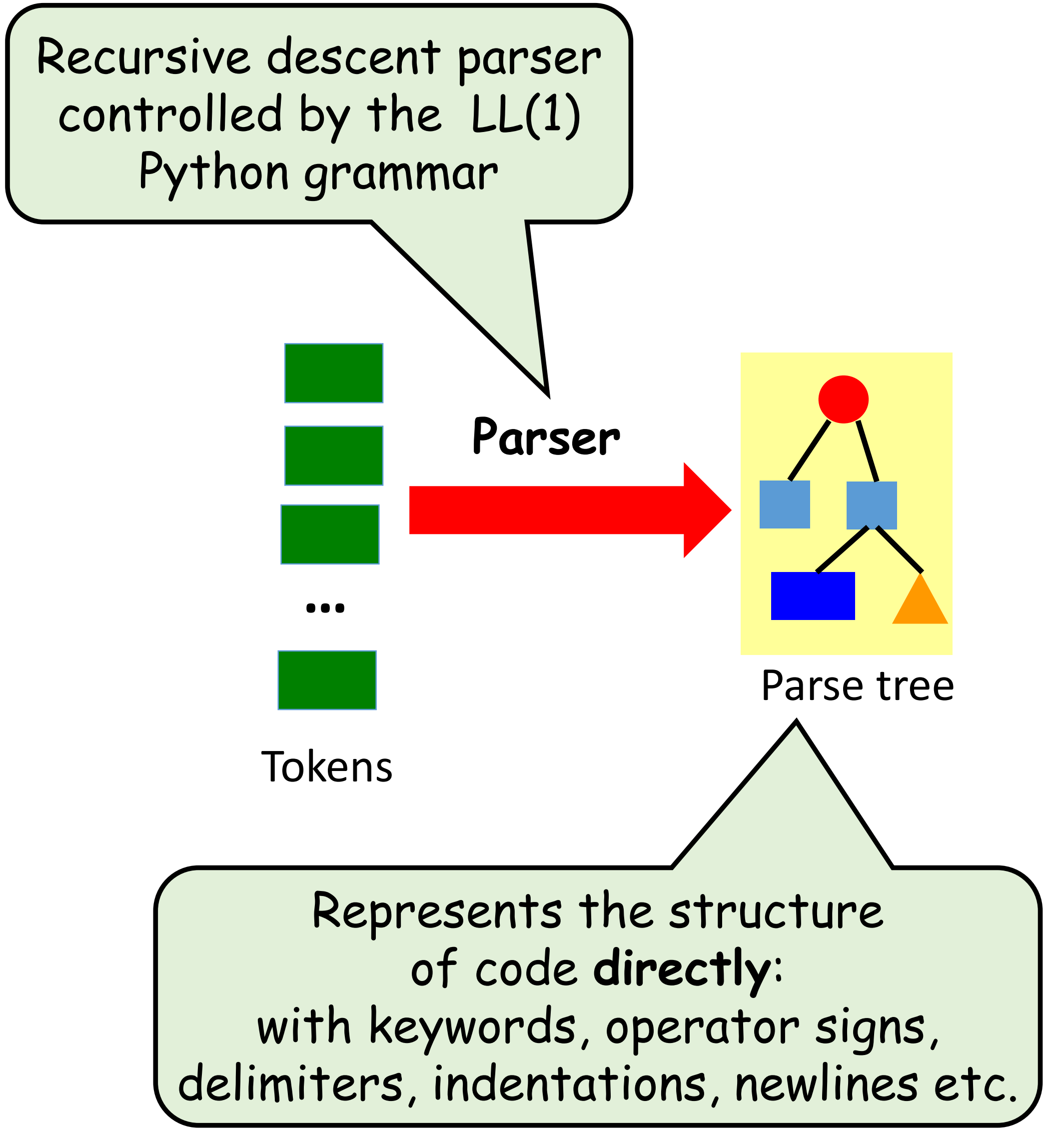
INDENT, **DEDENT**:

Tokens representing indentation levels which group compound statements.

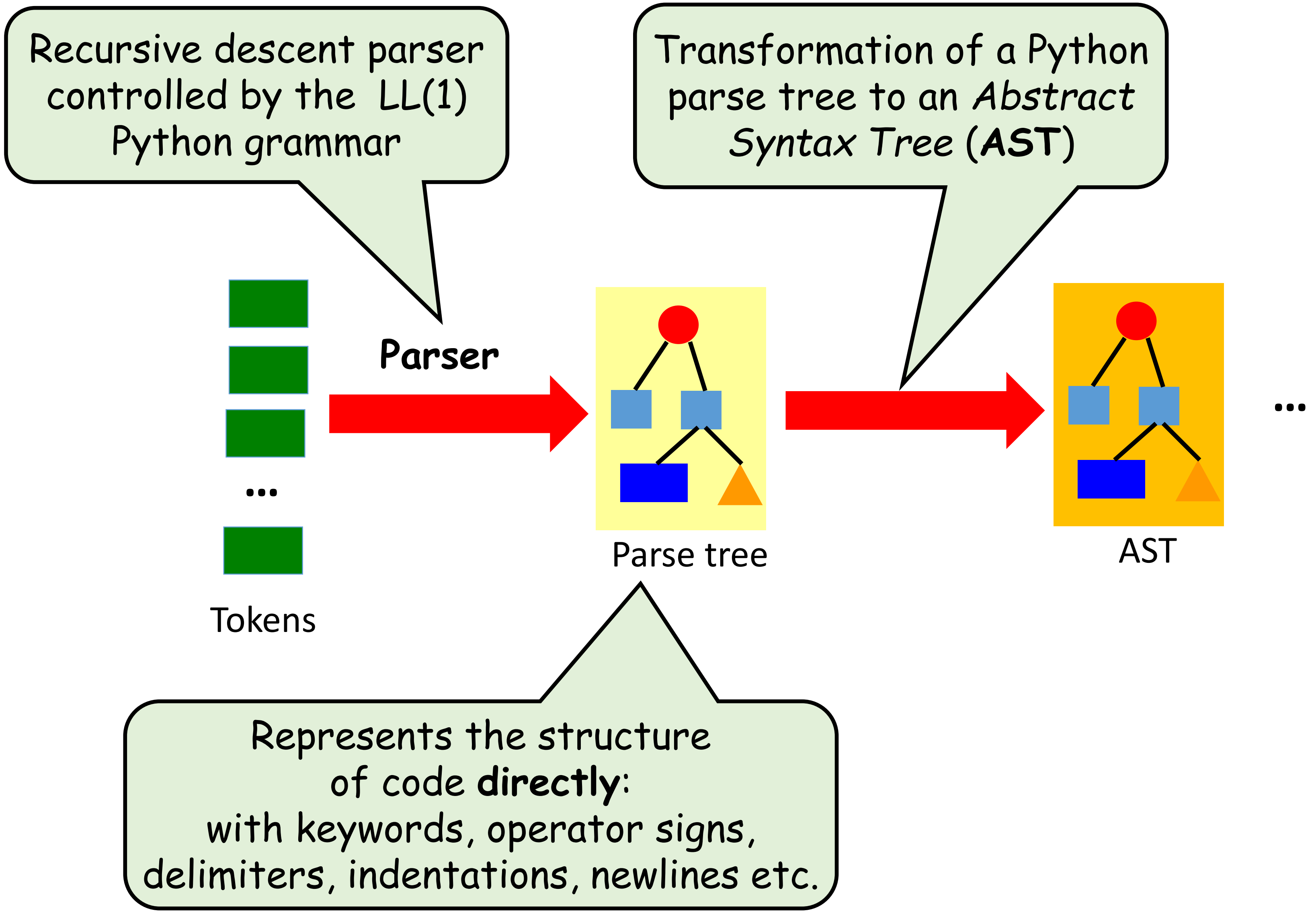
2nd Phase: Building Program Tree(s)



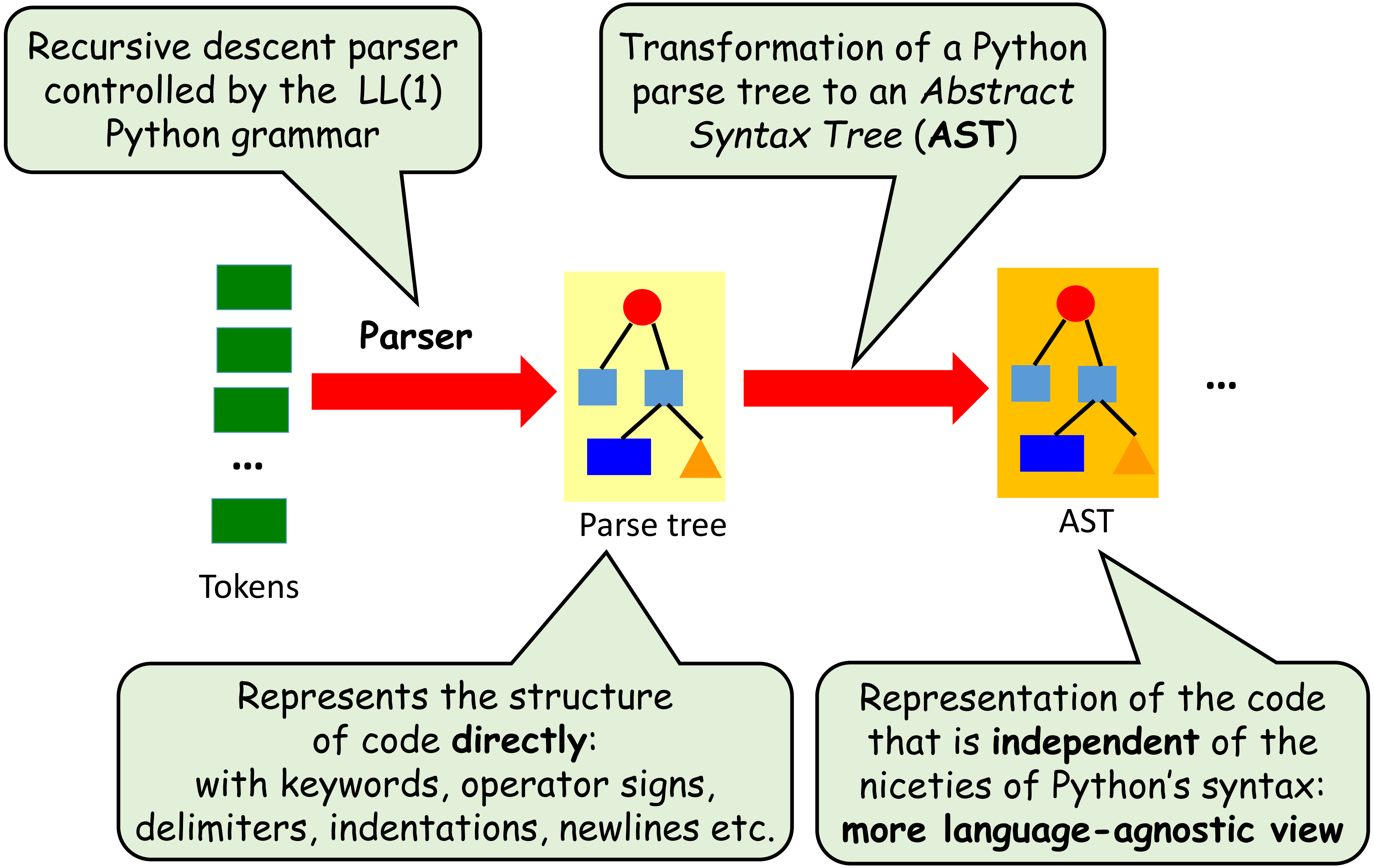
2nd Phase: Building Program Tree(s)



2^{nd} Phase: Building Program Tree(s)



2nd Phase: Building Program Tree(s)



Python Grammar

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: expr_stmt | del_stmt | pass_stmt | flow_stmt | import_stmt
           | global_stmt | nonlocal_stmt | assert_stmt
expr_stmt: testlist_star_expr ( augassign (yield_expr|testlist) |
                               ( '=' (yield_expr|testlist_star_expr))* )
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: '+='|'-='|'*='|'@='|'/='|'%='|'&='|'|='|'^='|'<<='|'>>='|'**='|'//='
del_stmt: 'del' exprlist
pass_stmt: 'pass'
flow_stmt: break_stmt | continue_stmt | return_stmt | raise_stmt | yield_stmt
break_stmt: 'break'
continue_stmt: 'continue'
return_stmt: 'return' [testlist]
yield_stmt: yield_expr
raise_stmt: 'raise' [test ['from' test]]
import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
import_from: ('from' (('.'|'...')* dotted_name | ('.'|'...')+)
              'import' ('*' | '(' import_as_names ')' | import_as_names))
import_as_name: NAME ['as' NAME]
dotted_as_name: dotted_name ['as' NAME]
import_as_names: import_as_name (',' import_as_name)* [',']
dotted_as_names: dotted_as_name (',' dotted_as_name)*
dotted_name: NAME ('.' NAME)*
global_stmt: 'global' NAME (',' NAME)*
nonlocal_stmt: 'nonlocal' NAME (',' NAME)*
assert_stmt: 'assert' test [',' test]
```

Extended Backus-Naur Form (*EBNF*) grammar specification for Python: Grammar/Grammar module

Python Syntax Tree

The parser Python module provides limited access to the **parse tree** of a block of Python code

```
import parser
from pprint import pprint
source = "def quad(a): return a*a\n"
st = parser.suite(source)
pprint(parser.st2list(st))
```

Python Syntax Tree

```
[268,
 [269,
  [295,
   [263,
    [1, 'def'],
    [1, 'quad'],
    [264, [7, '('], [265, [266, [1, 'a']]], [8, ')']],
    [11, ':'],
    [304,
     [270,
      [271,
       [278,
        [281,
         [1, 'return'],
         [331,
          [305,
           [309,
            [310,
             [311,
              [312,
               [315,
                [316,
                 [317,
                  [318,
                   [319,
                    [320,
                     [321, [322, [323, [324, [1, 'a']]]]],
                     [16, '*'],
                     [321, [322, [323, [324, [1, 'a']]]]]]]]]]]]]]]]]]]],
      [4, '']]]]]],
 [4, ''],
 [0, '']]
```

The parser Python module provides limited access to the **parse tree** of a block of Python code

```
import parser
from pprint import pprint
source = "def quad(a): return a*a\n"
st = parser.suite(source)
pprint(parser.st2list(st))
```

Python Syntax Tree

```
[268,
 [269,
  [295,
   [263,
    [1, 'def'],
    [1, 'quad'],
    [264, [7, '('], [265, [266, [1, 'a']]], [8, ')']],
    [11, ':'],
    [304,
     [270,
      [271,
       [278,
        [281,
         [1, 'return'],
         [331,
          [305,
           [309,
            [310,
             [311,
              [312,
               [315,
                [316,
                 [317,
                  [318,
                   [319,
                    [320,
                     [321, [322, [323, [324, [1, 'a']]]]],
                     [16, '*'],
                     [321, [322, [323, [324, [1, 'a']]]]]]]]]]]]]]]]]]],
      [4, '']]]]]],
 [4, ''],
 [0, '']]
```

The `parser` Python module provides limited access to the **parse tree** of a block of Python code

```python
import parser
from pprint import pprint
source = "def quad(a): return a*a\n"
st = parser.suite(source)
pprint(parser.st2list(st))
```

Sic! Access to Python parse tree from within a Python program!

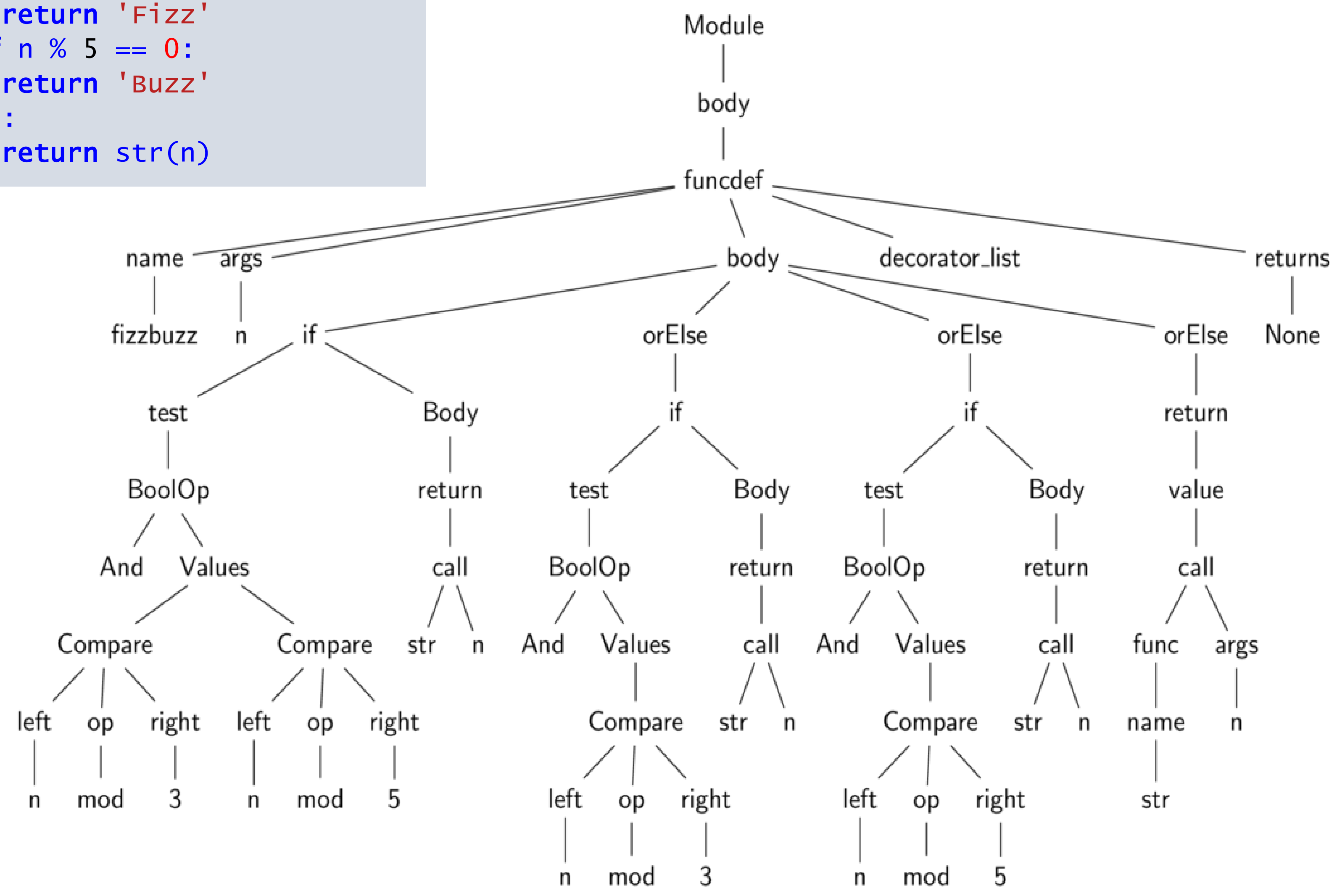
Python Syntax Tree

C implementation of the parse tree node

```
typedef struct _node
{
   short  n_type;           // node type
   char*  n_str;            // node name
   int    n_lineno;         // line number
   int    n_col_offset;     // offset within the line
   int    n_nchildren;      // number of children nodes
   struct _node* n_child; // the list of children
} node;
```

Python AST

```python
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```



Python Abstract Syntax Tree

Example: the same source as before

```
import ast
source = "def quad(a): return a*a\n"
node = ast.parse(source,mode="exec")
ast.dump(node,
         annotate_fields=True,
         include_attributes=True)
```

Sic! Access to Python AST from within a Python program!

The result of dump:

```
"Module(body=[FunctionDef(name='quad',
args=arguments(args=[arg(arg='a', annotation=None)],
vararg=None, kwonlyargs=[], kw_defaults=[], kwarg=None,
defaults=[]), body=[Return(value=BinOp(left=Name(id='a',
ctx=Load()), op=Mult(), right=Name(id='a', ctx=Load())))],
decorator_list=[], returns=None)])"
```

Python Abstract Syntax Tree

```
"Module (
    body = [
      FunctionDef (
          name = 'quad',
          args = arguments(
                    args = [
                       arg(arg='a',annotation=None,lineno=1, col_offset=9)
                    ],
                    vararg = None, kwonlyargs=[], kw_defaults=[], kwarg=None,
                    defaults = []
                 ),
          body = [
              Return(
                  value = BinOp(
                      left = Name(id='a',ctx=Load(),lineno=1,col_offset=20),
                      op = Mult(),
                      right = Name(id='a',ctx=Load(),lineno=1,col_offset=22),
                      lineno = 1, col_offset = 20
                  ),
                  lineno = 1, col_offset = 13
              )
          ],
          decorator_list=[], returns=None, lineno=1, col_offset=0
      )
    ]
)"
```

From the prev. slide:
the formatted version

Python Abstract Syntax Tree

```
"Module (
    body = [
      FunctionDef (
          name = 'quad',
          args = arguments(
                    args = [
                       arg(arg='a',annotation=None,lineno=1, col_offset=9)
                    ],
                    vararg = None, kwonlyargs=[], kw_defaults=[], kwarg=None,
                    defaults = []
                 ),
          body = [
              Return(
                  value = BinOp(
                      left = Name(id='a',ctx=Load(),lineno=1,col_offset=20),
                      op = Mult(),
                      right = Name(id='a',ctx=Load(),lineno=1,col_offset=22),
                      lineno = 1, col_offset = 20
                  ),
                  lineno = 1, col_offset = 13
              )
          ],
          decorator_list=[], returns=None, lineno=1, col_offset=0
      )
    ]
)"
```

From the prev. slide:
the formatted version

Look carefully: does it look familiar? ☺

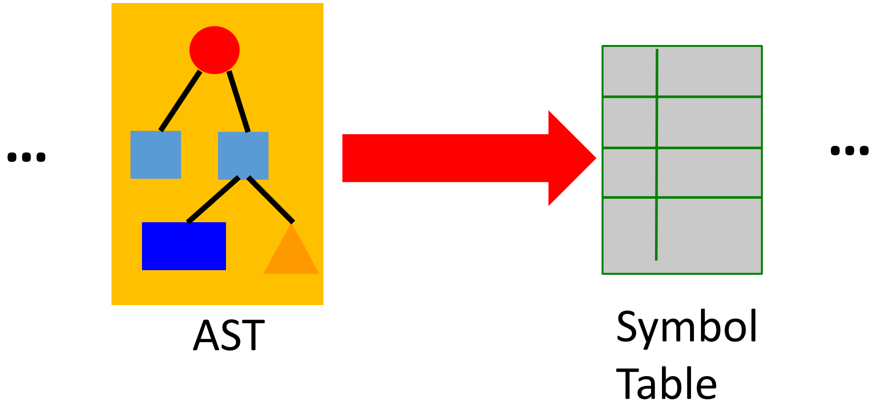
Python Abstract Syntax Tree

The C implementation of the AST node (example for statements)

```c
struct _stmt {
    enum _stmt_kind kind;
    union {
        struct {
            identifier name;
            arguments_ty args;
            asdl_seq* body;
            asdl_seq* decorator_list;
            expr_ty returns;
        } FunctionDef;
        ...
        struct {
            identifier name;
            asdl_seq* bases;
            asdl_seq* keywords;
            asdl_seq* body;
            asdl_seq* decorator_list;
        } ClassDef;
        ...
    } v;
    int lineno;
    int col_offset
}
```

3rd Phase: Building Symbol Tables

Symbol Table:
A collection of the names within a code block
and the context in which names are used



A **code block** is a piece of program code that is executed as a single unit.

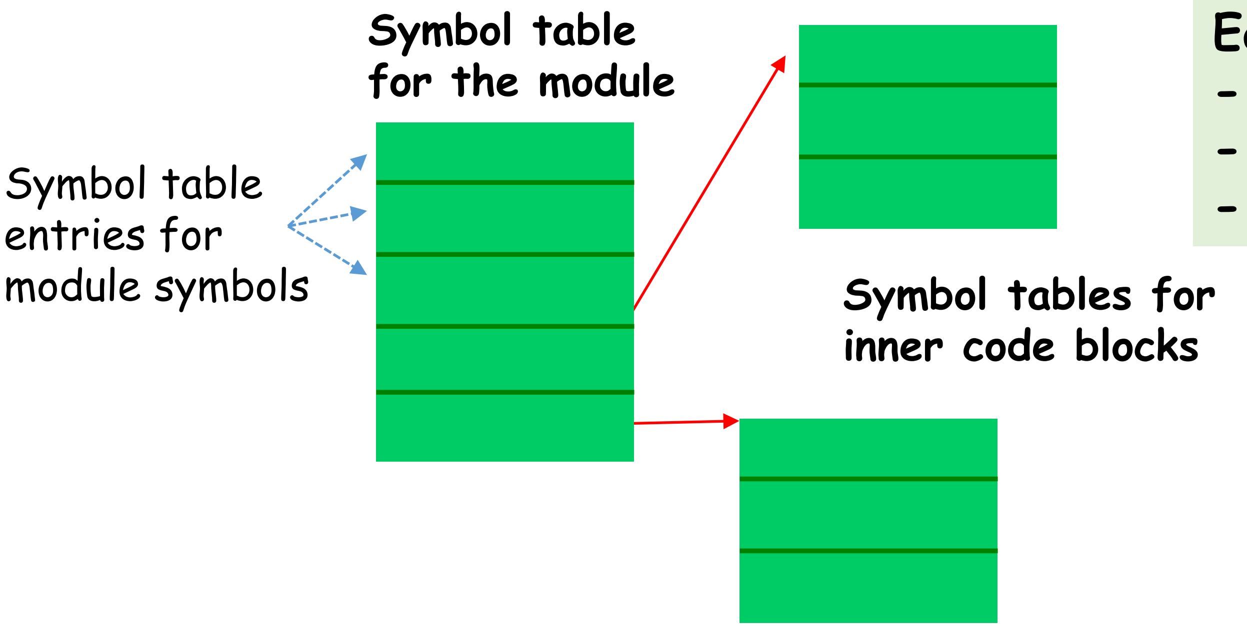
Examples of code blocks: **modules**, **functions** and **classes**.

More examples: **commands** typed in interactively at the REPL

A code block has a number of **namespaces** associated with it.

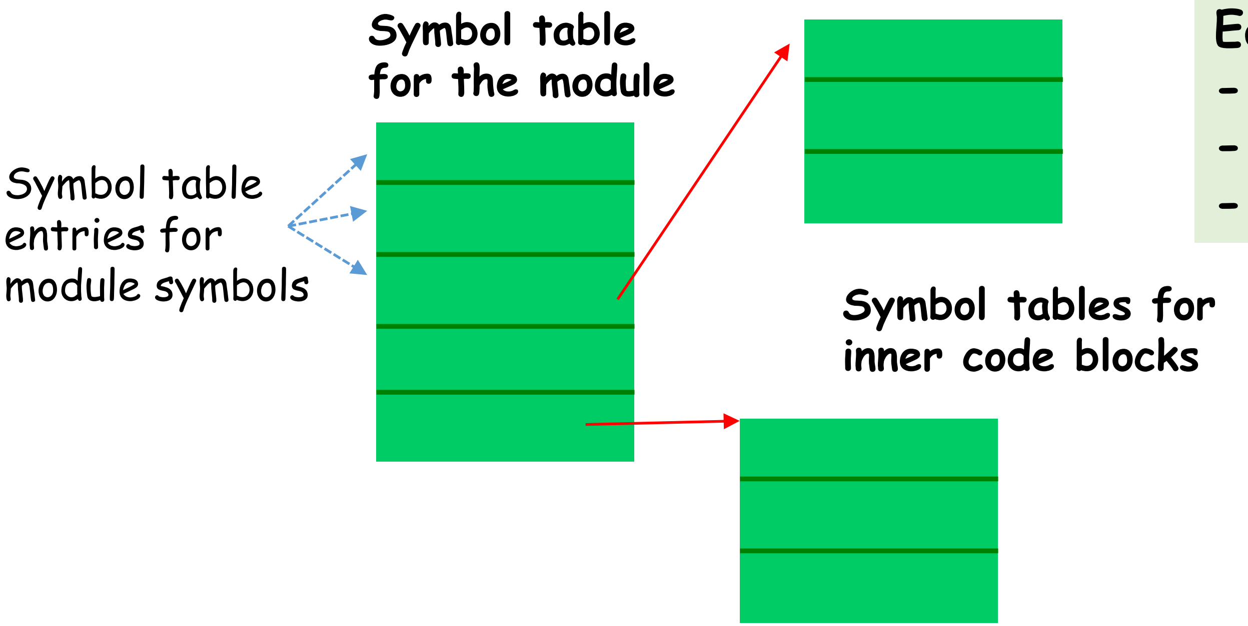
Module code block: has access to the global namespace
Function code block: has access to the local as well as the global namespace.

The Structure of Symbol Tables



Each symbol table contains:
- Info about own names
- Info about "child" symbol tables
- Info about the context

The Structure of Symbol Tables



Each symbol table contains:
- Info about own names
- Info about "child" symbol tables
- Info about the context

The structure of a single symbol table

```
struct symtable {
    PyObject* st_filename;  // name of file being compiled
    struct _symtable_entry* st_top;
                            // symbols declared within the module
    PyObject* st_blocks;    // symbol tables for inner code blocks
     ...  // many other fields
};
```

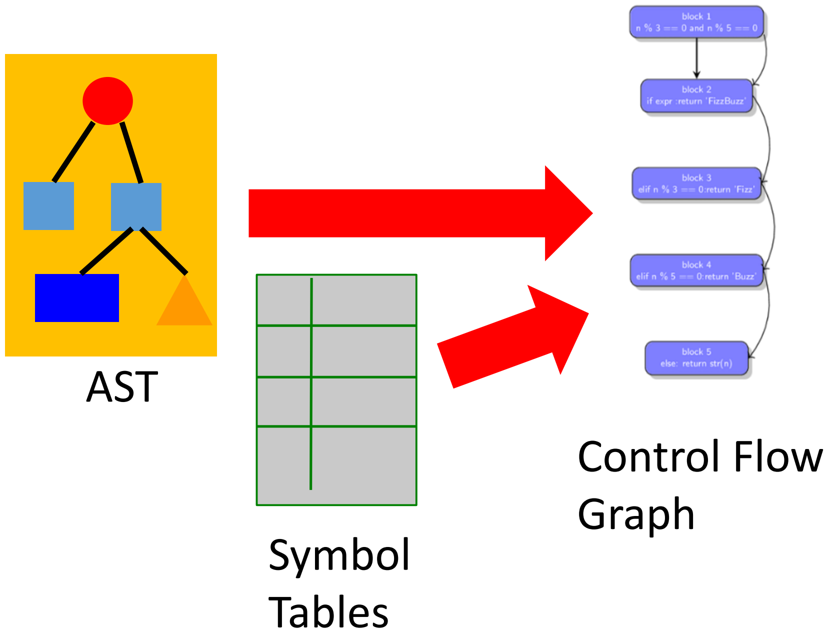
The Structure of Symbol Tables

The structure of a single symbol table entry

```
typedef struct _symtable_entry {
    ...
   PyObject* ste_name;          // string: name of current block
   PyObject* ste_varnames;      // list of function parameters
   PyObject* ste_children;      // list of child blocks
   PyObject* ste_directives;    // locations of global and
                                // nonlocal statements
   _Py_block_ty ste_type;       // module, class, or function
   int ste_nested;              // true if block is nested
    ...
} PySTEntryObject;
```

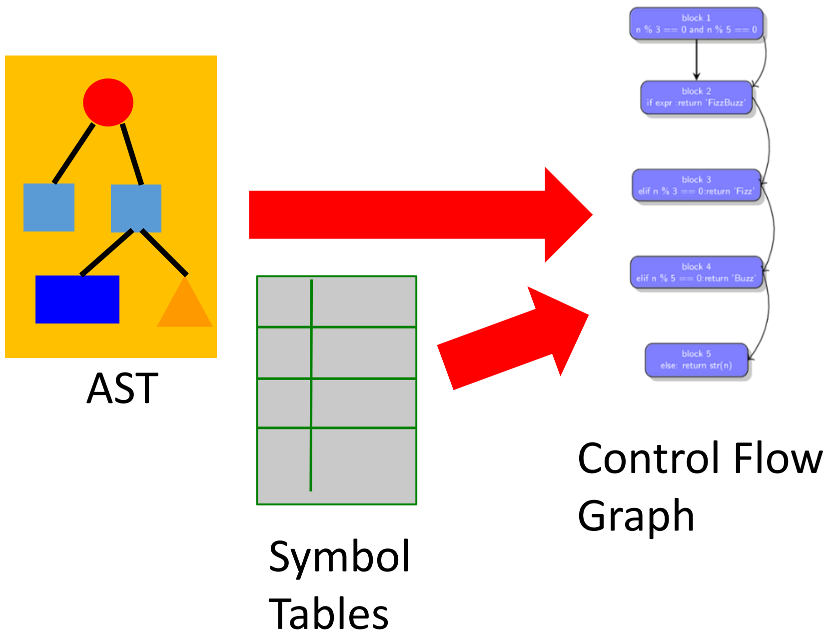
4th Phase: Building CFG & Bytecode

The next step for the compiler is to generate code objects from the AST incorporating information contained in the symbol table



4^{th} Phase: Building CFG & Bytecode

The next step for the compiler is to generate code objects from the AST incorporating information contained in the symbol table



<u>Step 1</u>

The AST is converted into **basic blocks** of Python byte code instructions. The result is the Control Flow Graph (CFG).

<u>Step2</u>

The generated control flow graph is flattened using a post-order depth first search transversal.

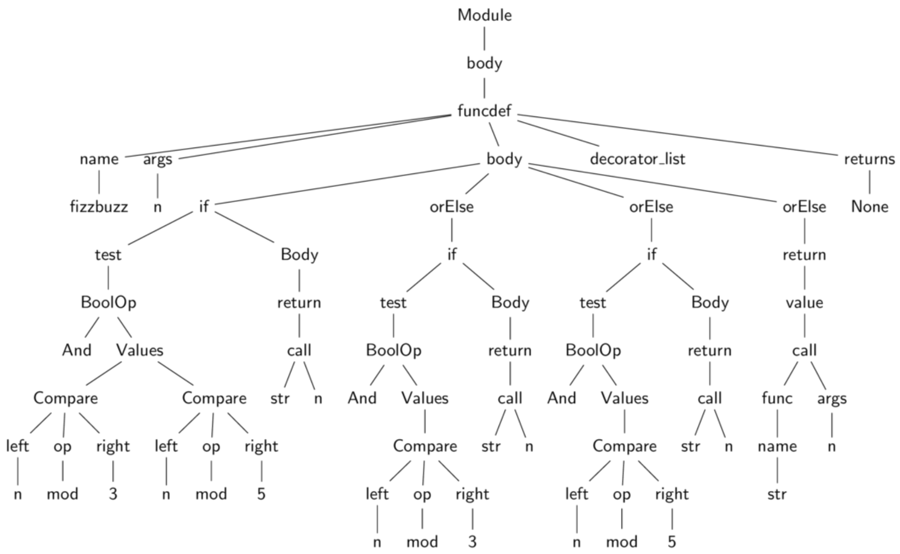
Control Flow Graph

- Maximal portion of language constructs that are to be executed sequentially is called **basic block**.
- The basic blocks have a single entry point but can have multiple exits.
- The basic blocks and paths between them implicitly represent a graph - the **control flow graph**.
- Therefore, the CFG is basically composed of basic blocks and connections between these basic blocks.

Control Flow Graph

Example

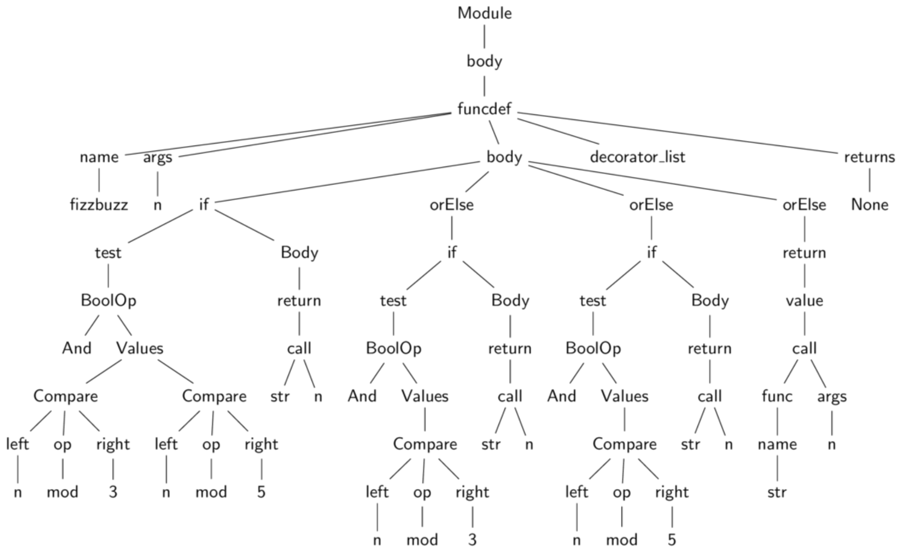
```python
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```



Control Flow Graph

Example

```python
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```



Block 1
`n%3 == 0 and n%5 == 0`

Block 2
`return 'FizzBuzz'`

Block 3
`n % 3 == 0`

Block 4
`return 'Fizz'`

Block 5
`n % 5 == 0`

Block 6
`return 'Buzz'`

Block 7
`return str(n)`

Code Object Structure

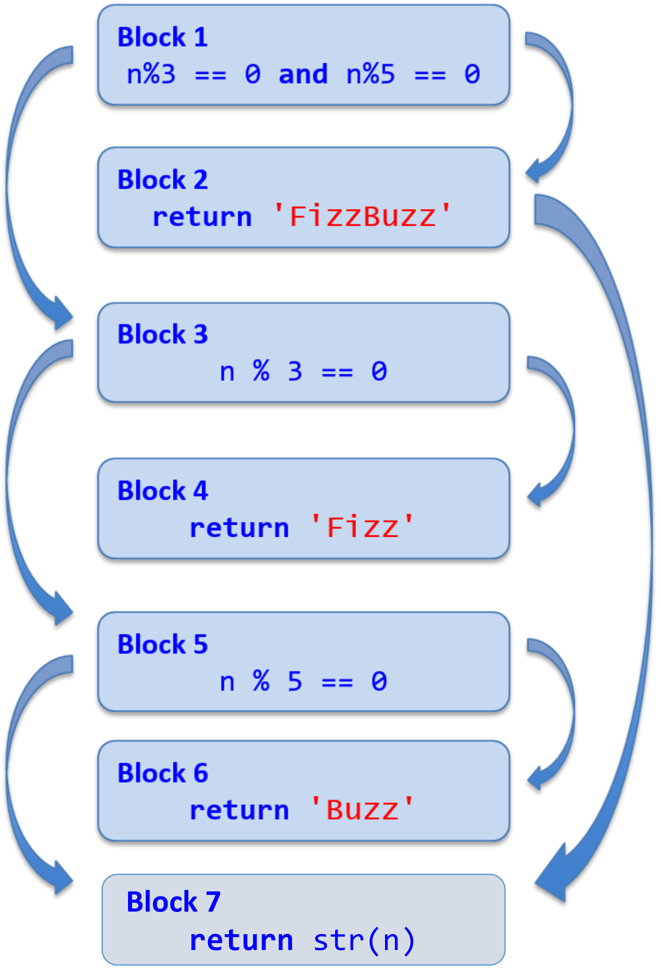
```
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

Code Object Structure

```python
def fizzbuzz(n):
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return str(n)
```

```
co_argcount = 1   // number of arguments to a code block
co_cellvars = ()
co_code = <sequence of bytecode instructions>
     // list of constants: string literals and numeric values
co_consts = (None, 3, 0, 5, 'FizzBuzz', 'Fizz', 'Buzz')
co_filename = <full path to the source file fizzbuzz.py>
co_firstlineno = 6
co_flags = 67
co_freevars = ()
co_kwonlyargcount = 0
co_lnotab = <maps from opcodes to line numbers>
co_name = fizzbuzz
co_names = ('str',) // collection of non-local names
co_nlocals = 1
co_stacksize = 2
co_varnames = ('n',)  // names defined locally
```

From CFG to Code Object



From CFG to Code Object

Block 1

```
 0 LOAD_FAST    0 (n)
 2 LOAD_CONST   1 (3)
 4 BINARY_MODULO
 6 LOAD_CONST   2 (0)
 8 COMPARE_OP   2 (==)
10 JUMP_IF_FALSE_OR_POP 28
12 LOAD_FAST    0 (n)
14 LOAD_CONST   3 (5)
16 BINARY_MODULO
18 LOAD_CONST   2 (0)
20 COMPARE_OP   2 (==)
22 JUMP_IF_FALSE_OR_POP 28
```

Block 2

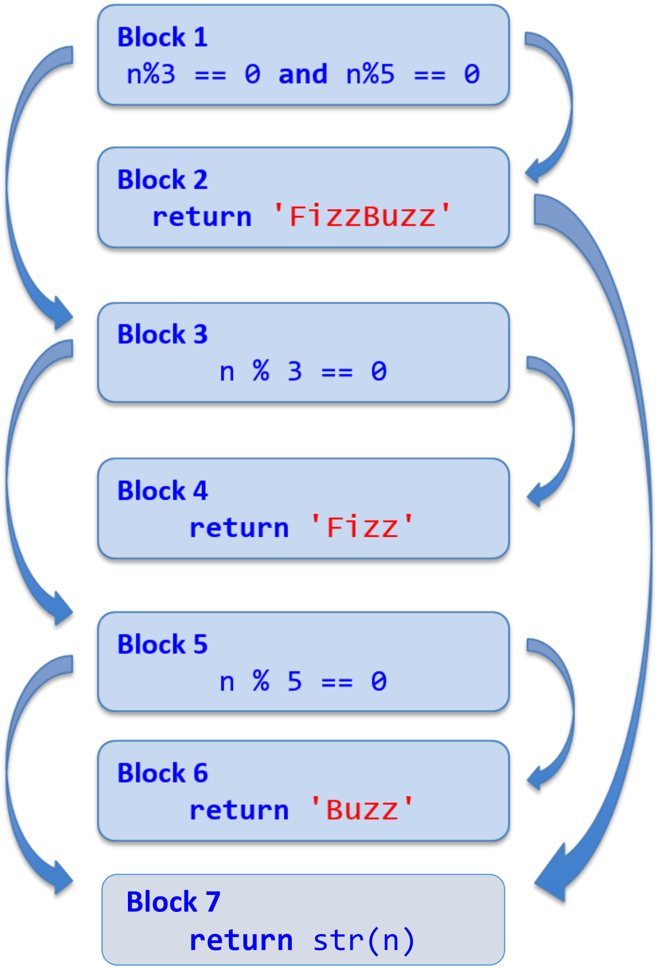
```
24 LOAD_CONST 4 ('FizzBuzz')
26 RETURN_VALUE
```

Block 3

```
28 LOAD_FAST      0 (n)
30 LOAD_CONST     1 (3)
32 BINARY_MODULO
34 LOAD_CONST     2 (0)
36 COMPARE_OP     2 (==)
38 POP_JUMP_IF_FALSE 44
```

Block 4

```
40 LOAD_CONST     5 ('Fizz')
42 RETURN_VALUE
```



Block 5

```
44 LOAD_FAST    0 (n)
46 LOAD_CONST   3 (5)
48 BINARY_MODULO
50 LOAD_CONST   2 (0)
52 COMPARE_OP   2 (==)
54 POP_JUMP_IF_FALSE 60
```

Block 6

```
56 LOAD_CONST   6 ('Buzz')
68 RETURN_VALUE
```

Block 7

```
60 LOAD_GLOBAL    0 (str)
62 LOAD_FAST      0 (n)
64 CALL_FUNCTION 1
66 RETURN_VALUE
68 LOAD_CONST     0 (None)
70 RETURN_VALUE
```

Instructions in More Details

```
...
if n % 3 == 0 and n % 5 == 0:
...
```

Instructions in More Details

```
...
if n % 3 == 0 and n % 5 == 0:
...
```

```
 0 LOAD_FAST              0 (n)
 2 LOAD_CONST             1 (3)
 4 BINARY_MODULO
 6 LOAD_CONST             2 (0)
 8 COMPARE_OP             2 (==)
10 JUMP_IF_FALSE_OR_POP   28
12 LOAD_FAST              0 (n)
14 LOAD_CONST             3 (5)
16 BINARY_MODULO
18 LOAD_CONST             2 (0)
20 COMPARE_OP             2 (==)
22 JUMP_IF_FALSE_OR_POP   28
```

Instructions in More Details

```
...
if n % 3 == 0 and n % 5 == 0:
...
```

Offset of the given instruction from the start of the bytecode sequence

Human readable instruction opcode

the index into the `co_varnames` array

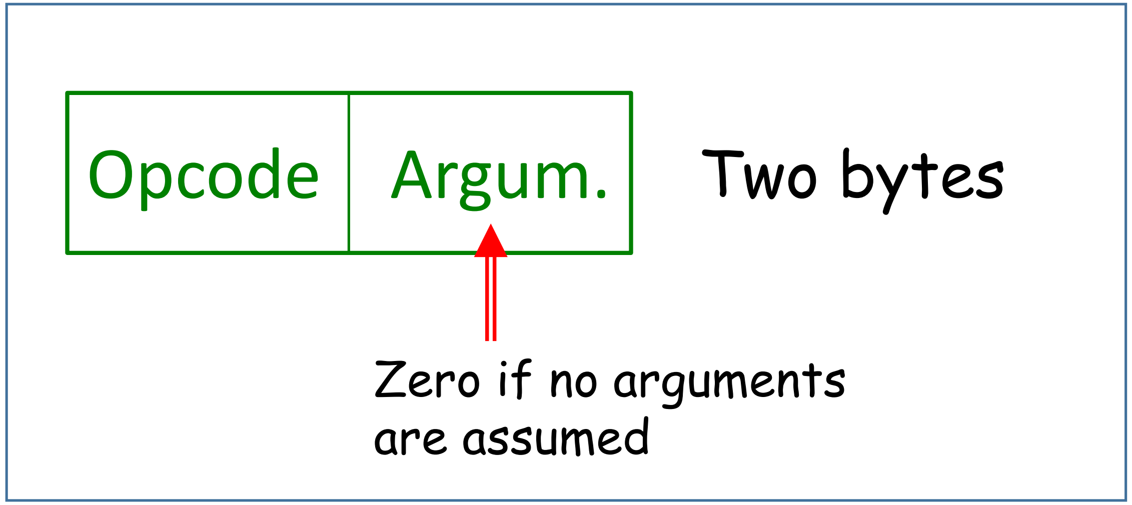
```
 0 LOAD_FAST               0 (n)
 2 LOAD_CONST              1 (3)
 4 BINARY_MODULO
 6 LOAD_CONST              2 (0)
 8 COMPARE_OP              2 (==)
10 JUMP_IF_FALSE_OR_POP   28
12 LOAD_FAST               0 (n)
14 LOAD_CONST              3 (5)
16 BINARY_MODULO
18 LOAD_CONST              2 (0)
20 COMPARE_OP              2 (==)
22 JUMP_IF_FALSE_OR_POP   28
```

the index into the `co_const` array

The argument to the instruction

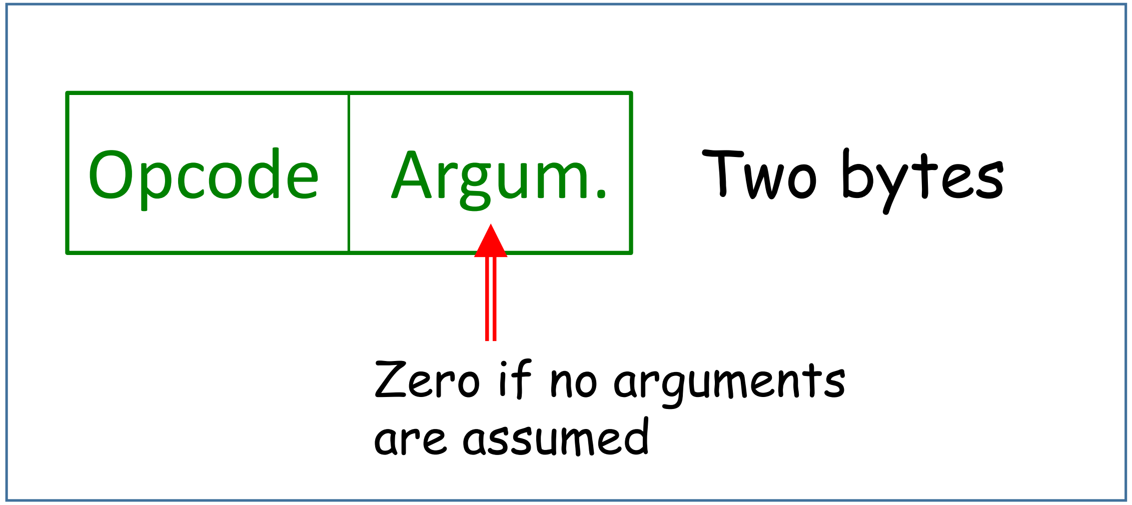
The Single Instruction Format

…is quite simple ☺



The Single Instruction Format

…is quite simple ☺



How to obtain the bytecode:

```python
def cube(a):
   return a*a*a

from dis import dis
dis(cube)
```

```
2     0 LOAD_FAST           0 (a)
      2 LOAD_FAST           0 (a)
      4 BINARY_MULTIPLY
      6 LOAD_FAST           0 (a)
      8 BINARY_MULTIPLY
     10 RETURN_VALUE
```

Extra Slides

Representation of Python Objects

Python Objects

C#

```
class C { ... }

var c = new C()
```

What's the type of c?

Python Objects

C#

```
class C { ... }

var c = new C()
```

What's the type of `c`?

`c` is the **object** of the reference type. The type of `c` is `class C`

Python Objects

C#

```
class C { ... }

var c = new C()
```

What's the type of c?

c is the **object** of the reference type. The type of c is class C

Python

```
class C:
    ...
c = C()
```

What's the type of c?

Python Objects

C#

```
class C { ... }

var c = new C()
```

What's the type of c?

c is the **object** of the reference type. The type of c is `class C`

Python

```
class C:
    ...
c = C()
```

What's the type of c?

c is the **name** (reference) to an object. c doesn't have a type. It just refers to an object. **The object** has a type.

Python: Names & Binding

From the book **Inside the Python Virtual Machine**

In Python, objects are referenced by *names*. Names are analogous to variables in C++ and Java (*but not exactly*).

```
>>> x = 5
```

Here, x is a name that references the object, 5. The process of *assigning* a reference to 5 to x is called *binding*. A binding causes a name to be associated with an object in the innermost scope of the currently executing program. Bindings may occur during a number of instances such as during variable assignment or function or method call when the supplied parameter is bound to the argument.

It is important to note that names are just symbols and they have no *type* associated with them; **names are just references to objects that actually have types**.

How Objects Are Implemented?

```
typedef struct _object
{
  _PyObject_HEAD_EXTRA
  Py_ssize_t ob_refcnt;
  struct _typeobject* ob_type;
} PyObject;
```

How Objects Are Implemented?

Support for memory management

```
typedef struct _object
{
  _PyObject_HEAD_EXTRA
  Py_ssize_t ob_refcnt;
  struct _typeobject* ob_type;
} PyObject;
```

The number of references to the object

The reference to the information about the object type (!)

How Objects Are Implemented?

`aName = Expression`

An object

`PyObject* aName`

To the prev. object

To the next object

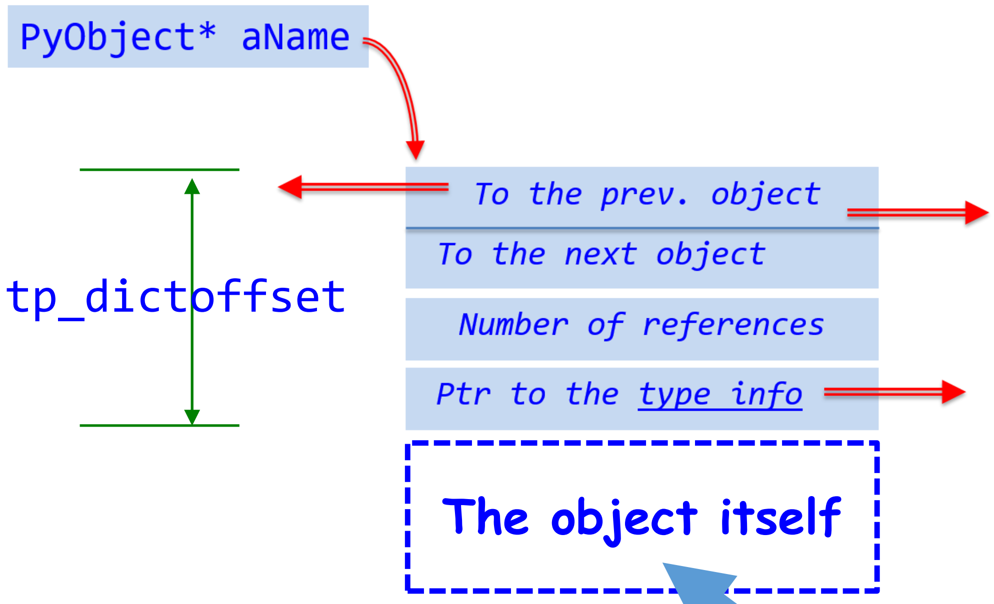
Number of references

Ptr to the type info

But where is the value??

How Objects Are Implemented?

For instances of user-defined types:



Example: tuple type

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject* ob_item[1];
} PyTupleObject;
```

```
tuple =
   (PyTupleObject*)(aName+tp_dictoffset)
```

How Objects Are Implemented?

For instances of class types: