COMP10001 Foundations of Computing

Semester 2 2014

Lecture 29 (advanced lecture, not examinable)

How Python works

Bernie Pope, bjpope@unimelb.edu.au

Outline

- What are programming languages?
- Syntax analysis.
- Translation to bytecode.
- Execution.
- Other ways of implementing programming languages.

What are programming languages?

- Programming languages are for **humans** …
    - … to describe computations …
    - ... which can be translated to run effectively on **machines**.
- The syntax of the language is a powerful and complex user interface.

What are programming languages?

- Programming languages are examples of **formal languages**.
    - They have an unambiguous grammar (syntax).
    - The have a precise meaning (semantics).
- Compare with natural languages (English, Chinese, Klingon, *etcetera*):
    - Ambiguous and incomplete grammar (usually defined *post hoc*).
    - Imprecise semantics (very hard to define, circular).

Incomplete and wrong history of programming languages

- Highlights:

- 1940s - Various "computers" are "programmed" using direct wiring and switches. Engineers do this in order to avoid the tabs versus spaces debate.

http://james-iry.blogspot.com.au/2009/05/brief-incomplete-and-mostly-wrong.html

Incomplete and wrong history of programming languages

- 1972 - Dennis Ritchie invents a powerful gun that shoots both forward and backward simultaneously. Not satisfied with the number of deaths and permanent maimings from that invention he invents C and Unix.

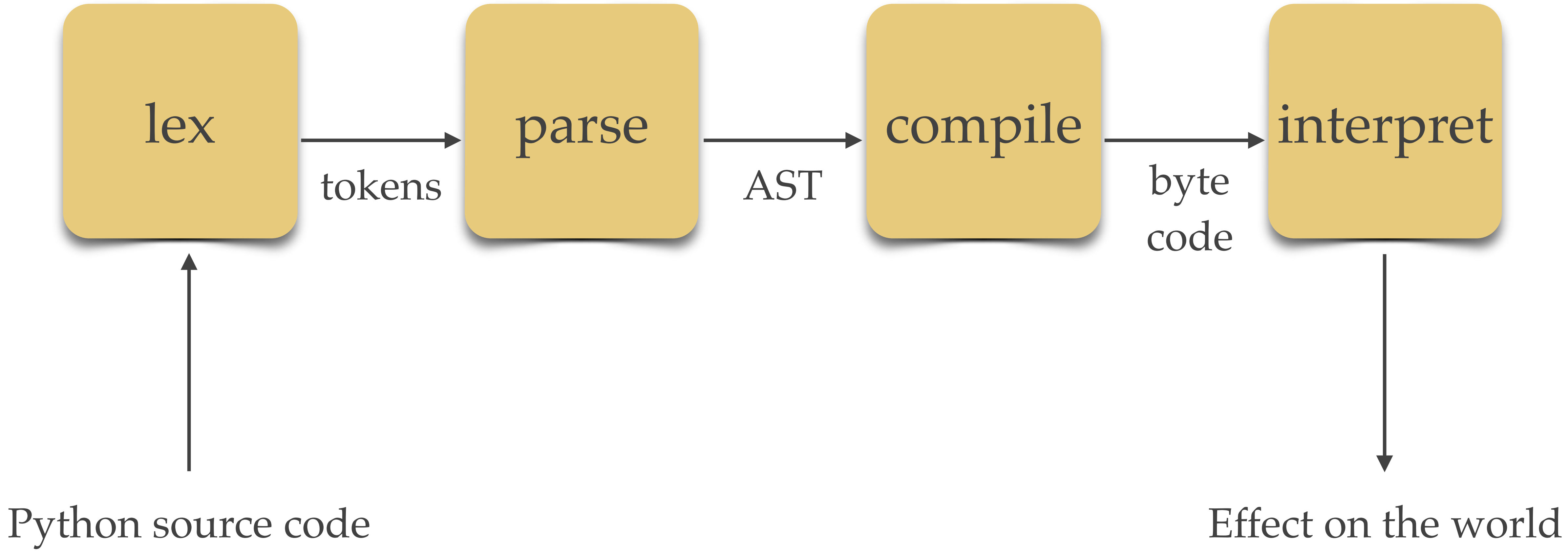
http://james-iry.blogspot.com.au/2009/05/brief-incomplete-and-mostly-wrong.html

Incomplete and wrong history of programming languages

- 1995 - Brendan Eich reads up on every mistake ever made in designing a programming language, invents a few more, and creates LiveScript. Later, in an effort to cash in on the popularity of Java the language is renamed JavaScript. Later still, in an effort to cash in on the popularity of skin diseases the language is renamed ECMAScript.

http://james-iry.blogspot.com.au/2009/05/brief-incomplete-and-mostly-wrong.html

Python's execution pipeline



Lexical analysis

- Recognises the tokens of the language (strings, variables, numbers, punctuation, comments etcetera).
- Input is a sequence of characters, output is a sequence of tokens.

Lexical analysis

```
>>> from StringIO import StringIO
>>> from tokenize import (generate_tokens, tok_name)
>>>
>>> stringIO = StringIO('x = y + 4')
>>> for t in generate_tokens(stringIO.readline):
...     print(tok_name[t[0]], repr(t[1]))
...
('NAME', "'x'")
('OP', "'='")
('NAME', "'y'")
('OP', "'+'")
('NUMBER', "'4'")
('ENDMARKER', "''")
```

Lexical analysis

```
>>> from StringIO import StringIO
>>> from tokenize import (generate_tokens, tok_name)
>>>
>>> stringIO = StringIO('x = y + 4')
>>> for t in generate_tokens(stringIO.readline):
...     print(tok_name[t[0]], repr(t[1]))
...
('NAME', "'x'")
('OP', "'='")
('NAME', "'y'")
('OP', "'+'")
('NUMBER', "'4'")
('ENDMARKER', "''")
```

We use `generate_tokens` in `utils.py` in Project 3.

Python has a formal grammar

file_input: (NEWLINE | stmt)* ENDMARKER

stmt: simple_stmt | compound_stmt

simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE

small_stmt: expr_stmt | print_stmt | del_stmt | pass_stmt | flow_stmt | import_stmt | global_stmt | exec_stmt | assert_stmt

compound_stmt: if_stmt | while_stmt | for_stmt | try_stmt | with_stmt | funcdef | classdef | decorated

... *etcetera* …

see: https://docs.python.org/2.7/reference/grammar.html

Parsing produces an Abstract Syntax Tree

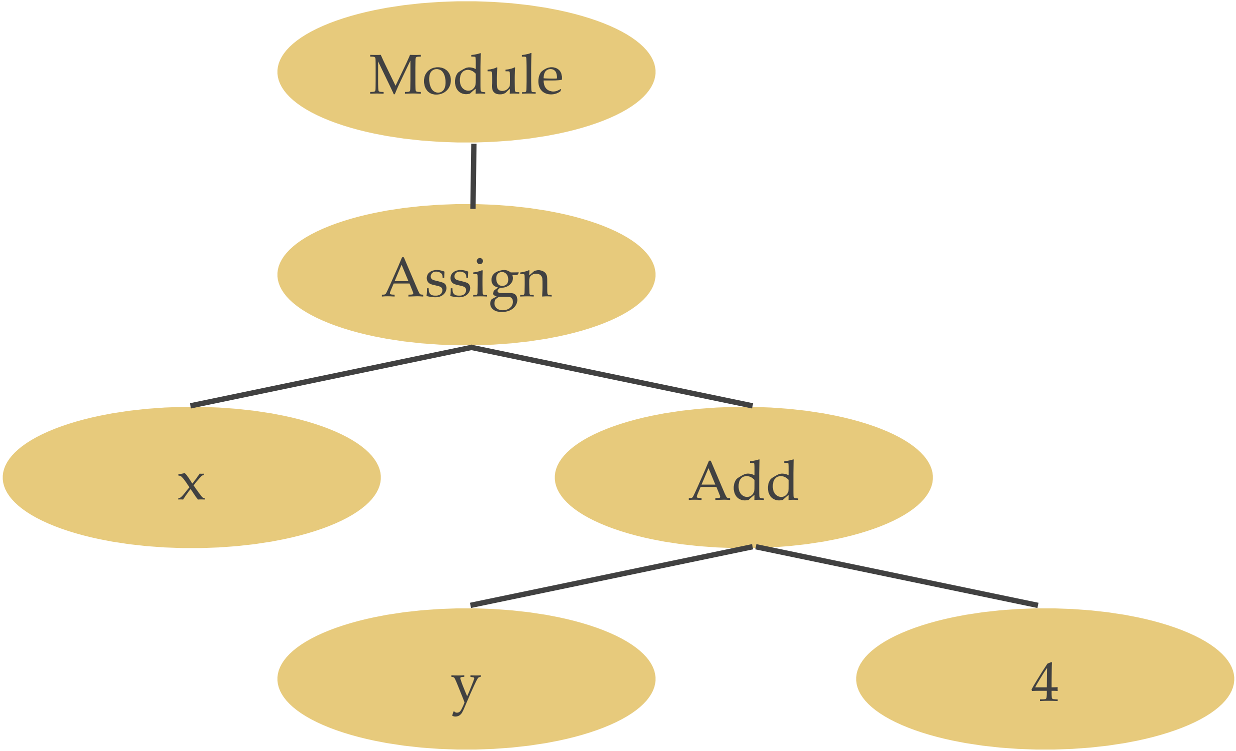
```
>>> from ast import (parse, dump)

>>> tree = parse('x = y + 4')

>>>

>>> dump(tree, annotate_fields=False)

"Module([Assign([Name('x', Store())], BinOp(Name('y', Load()), Add(), Num(4)))])"
```

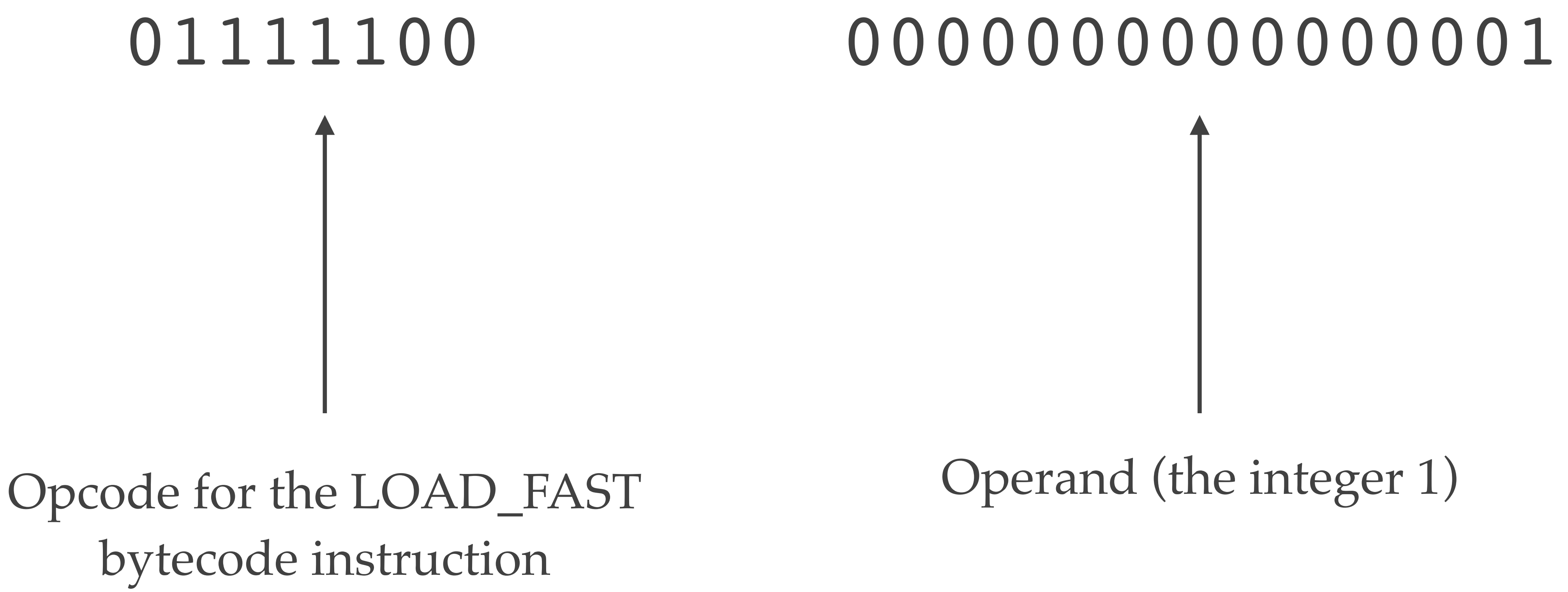


Bytecode

- The Abstract Syntax Tree is translated (compiled) into bytecode.
- Bytecode is a collection of roughly 150 instructions for a virtual machine.
- Each instruction consists of a single 8 bit (byte) *opcode* followed by an optional 16 bit *operand*.

Bytecode

An example bytecode instruction in binary:



01111100 0000000000000001

Opcode for the LOAD_FAST bytecode instruction

Operand (the integer 1)

Bytecode

```
>>> from dis import dis
>>> def f(y):
...     x = y + 4
...     return x
...
>>> dis(f)
  3           0 LOAD_FAST                0 (y)
              3 LOAD_CONST               1 (4)
              6 BINARY_ADD
              7 STORE_FAST               1 (x)

  5          10 LOAD_FAST                1 (x)
             13 RETURN_VALUE
```

Bytecode

```
>>> from dis import dis
>>> def f(y):
...     x = y + 4
...     return x
...
>>> dis(f)
  3                                      0 (y)
                                         1 (4)

                                         1 (x)

  5          10 LOAD_FAST                1 (x)
             13 RETURN_VALUE
```

Source code line numbers.

Bytecode

```
>>> from dis import dis
>>> def f(y):
...     x = y + 4
...     return x
...
>>> dis(f)
  3           0 LOAD_FAST
              3 LOAD_CONST
              6 BINARY_ADD
              7 STORE_FAST               1 (x)

  5          10 LOAD_FAST                1 (x)
             13 RETURN_VALUE
```

Bytecode instruction offsets.

Bytecode

```
>>> from dis import dis
>>> def f(y):
...     x = y + 4
...     return x
...
>>> dis(f)
  3           0 LOAD_FAST
              3 LOAD_CONST
              6 BINARY_ADD
              7 STORE_FAST

  5          10 LOAD_FAST                1 (x)
             13 RETURN_VALUE
```

Instruction Opcodes.

Bytecode

```
>>> from dis import dis
>>> def f(y):
...     x = y + 4
...     return x
...
>>> dis(f)
  3           0 LOAD_FAST                0 (y)
              3 LOAD_CONST               1 (4)
              6 BINARY_ADD
              7 STORE_FAST               1 (x)

  5          10 LOAD_FAST                1 (x)
             13 RETURN_VALUE
```

Instruction Operands.

Bytecode

- Most bytecode instructions fall into one of the following four categories:
    1. Control flow:
        - JUMP_ABSOLUTE, RETURN_VALUE, POP_JUMP_IF_FALSE …
    2. Variable manipulation:
        - LOAD_FAST, STORE_FAST, LOAD_GLOBAL, STORE_GLOBAL …
    3. Stack manipulation:
        - ROT_TWO, POP_TOP, DUP_TOP …
    4. Primitive operations
        - MAKE_FUNCTION, LOAD_ATTR, BUILD_LIST, BINARY_ADD…

Compilation

- Translates the Abstract Syntax Tree into bytecode instructions for the Python Virtual Machine.
    - Input is an Abstract Syntax Tree, output is a *code object*.
    - The code object might be loaded directly into the computer's memory and interpreted immediately, or it might be saved to file.
    - The `.pyc` files you see in IVLE are just serialised code objects.

Compilation

- Compilation converts the nested tree structure of the AST into a linear sequence of instructions.
- The linear sequence of instructions reflects the sequential nature of program execution.

Execution

- Compiled Python bytecode is executed by an interpreter which carries out the behaviour of the Virtual Machine.
- In addition to decoding and executing bytecode instructions, the interpreter provides the following functionality:
  - A **stack** for keeping track of local variables, intermediate values and control flow.
  - A **heap** for storing Python objects (pointed to by global variables and local variables on the stack).
  - Automatic memory management (called **garbage collection**).
  - **Input and output** via the operating system.

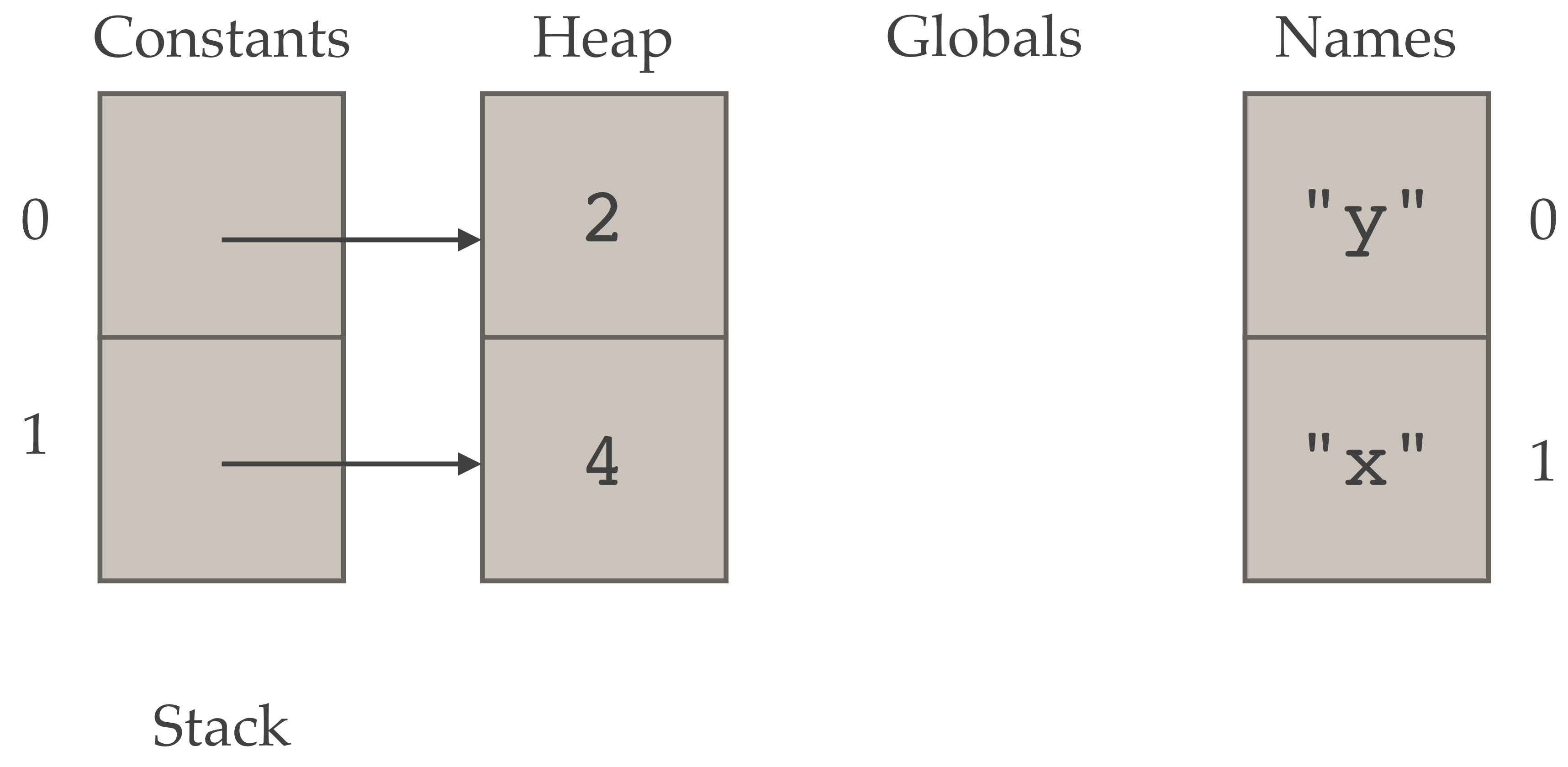
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



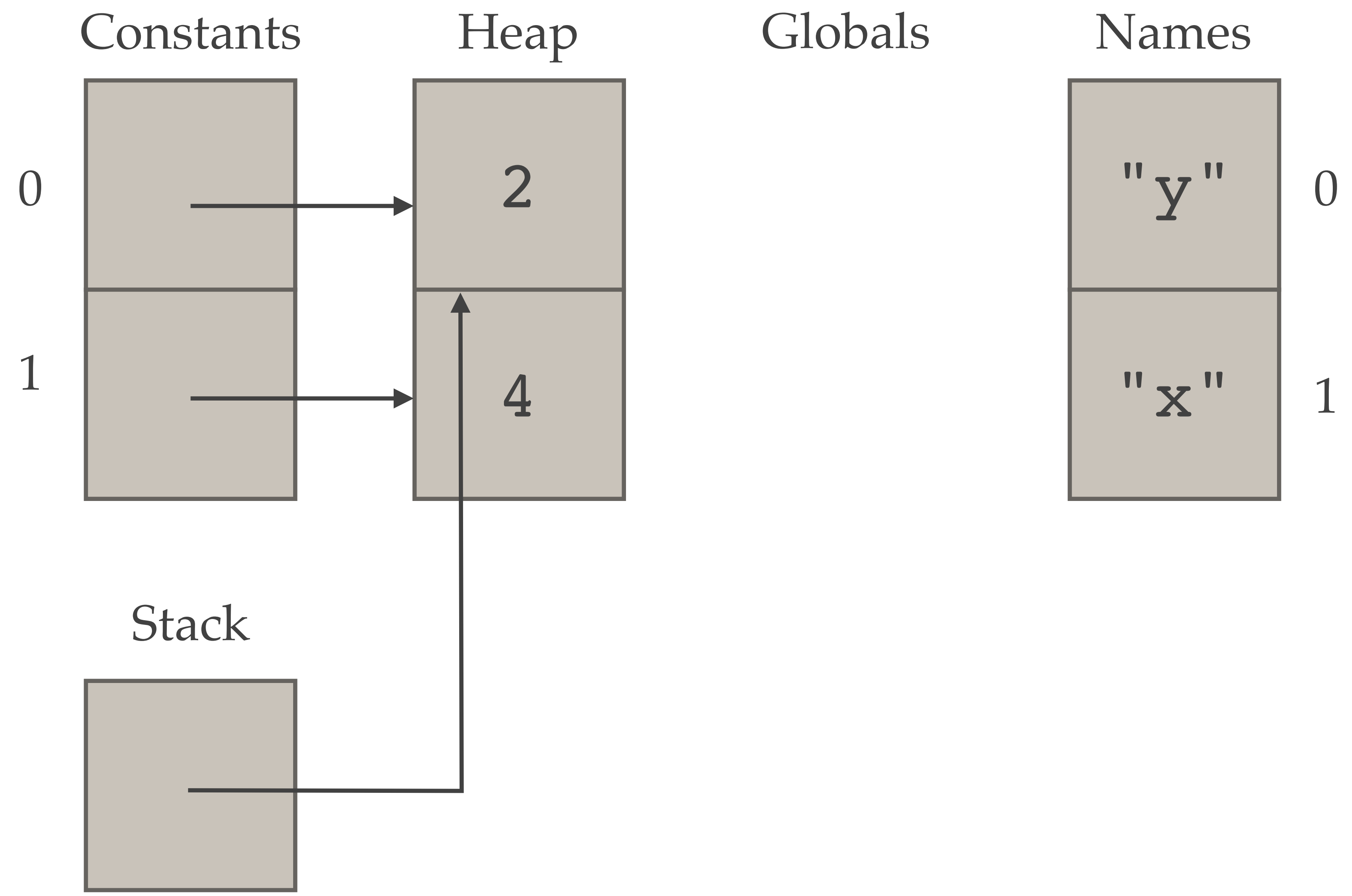
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



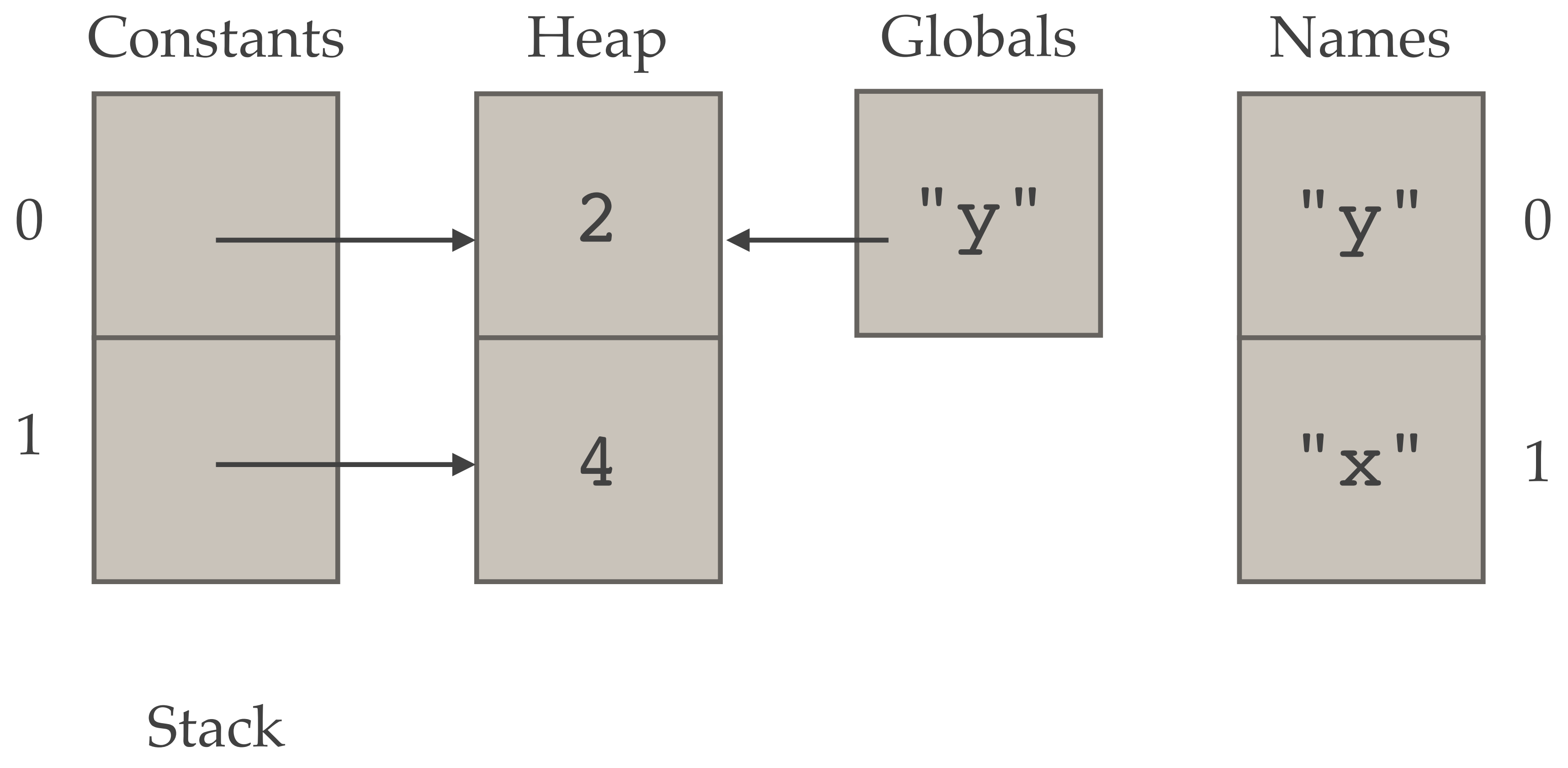
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



Constants

Heap

Globals

Names

0

1

2

4

"y"

"y"

"x"

0

1

Stack

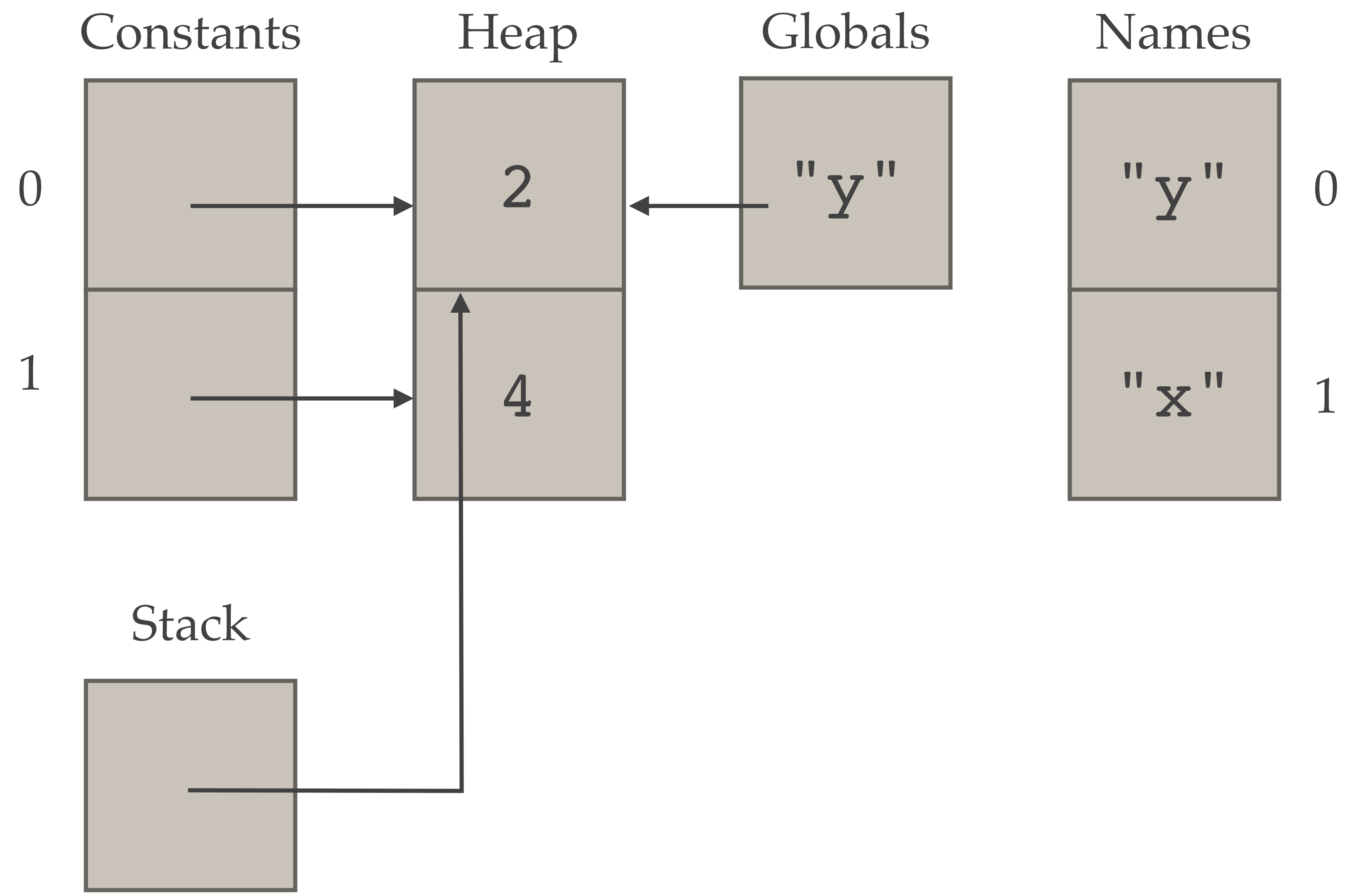
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



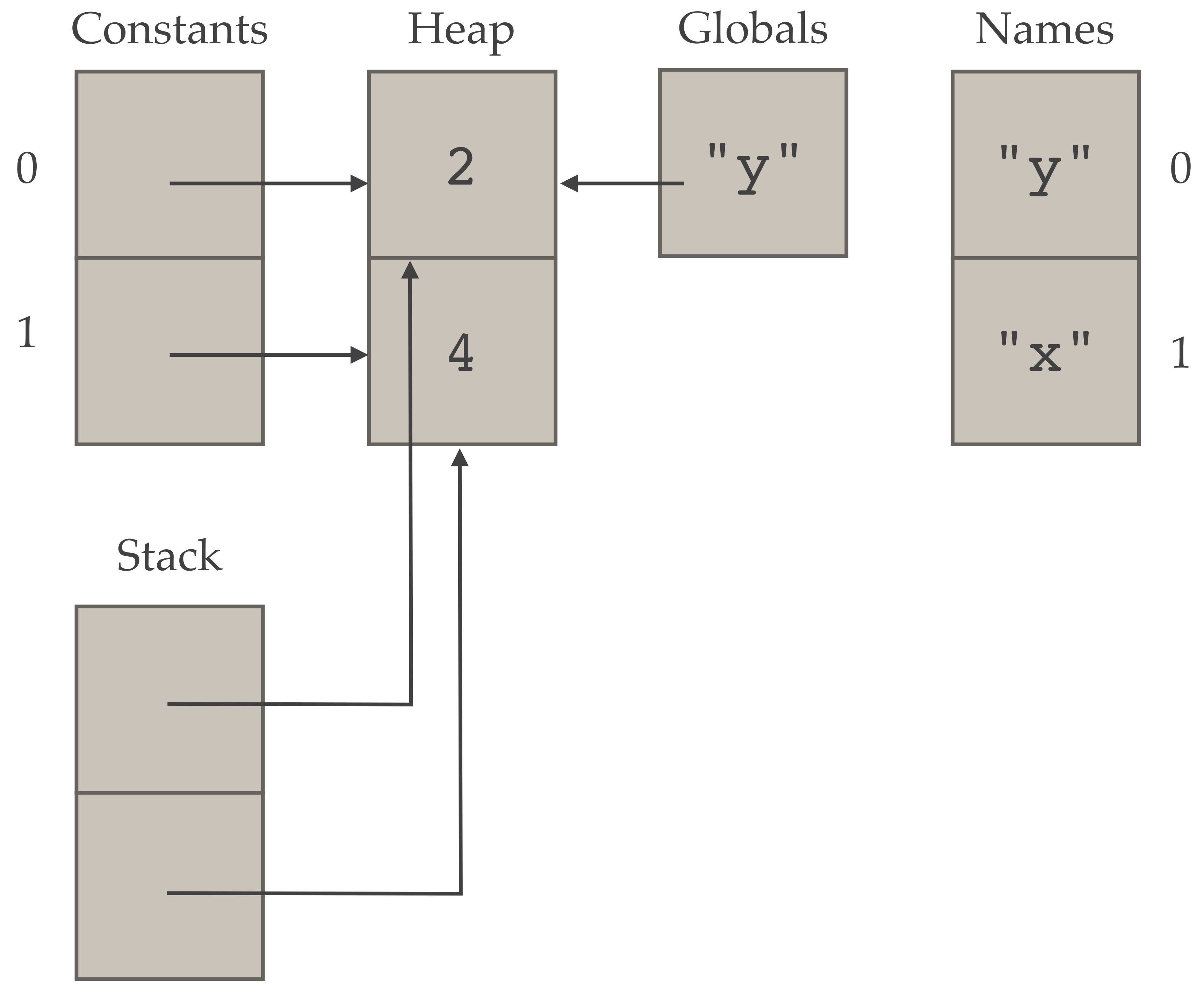
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



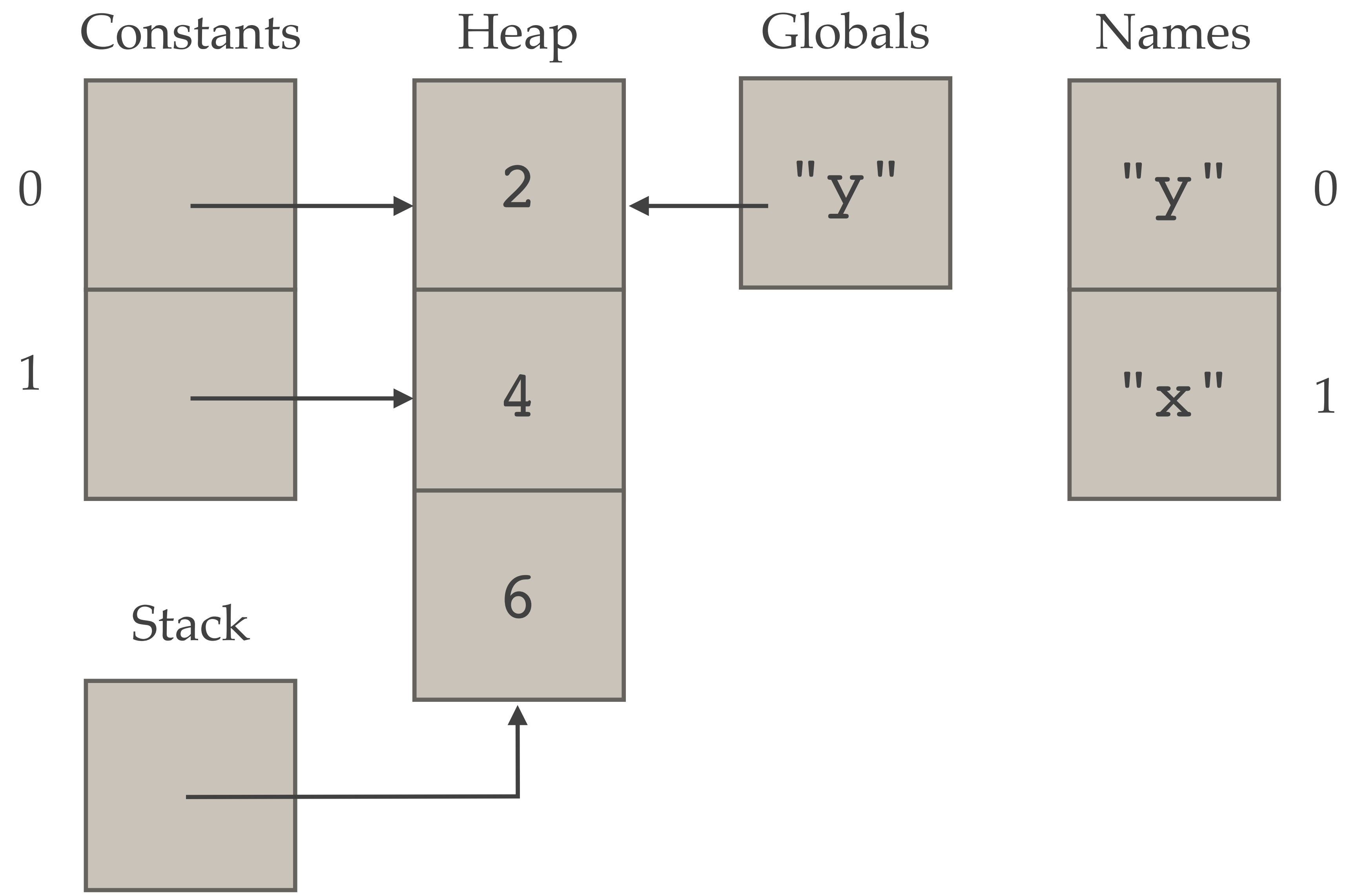
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



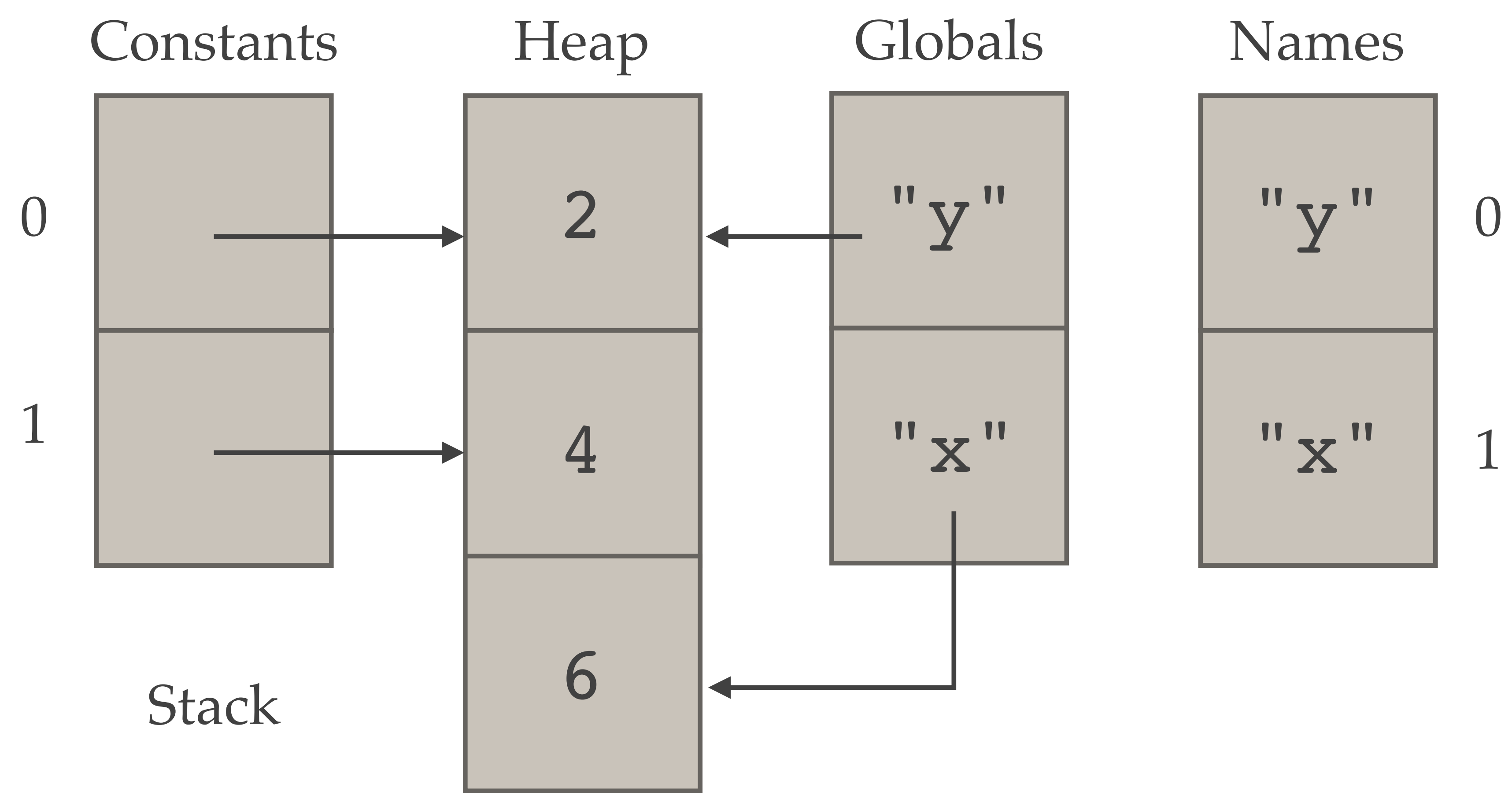
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



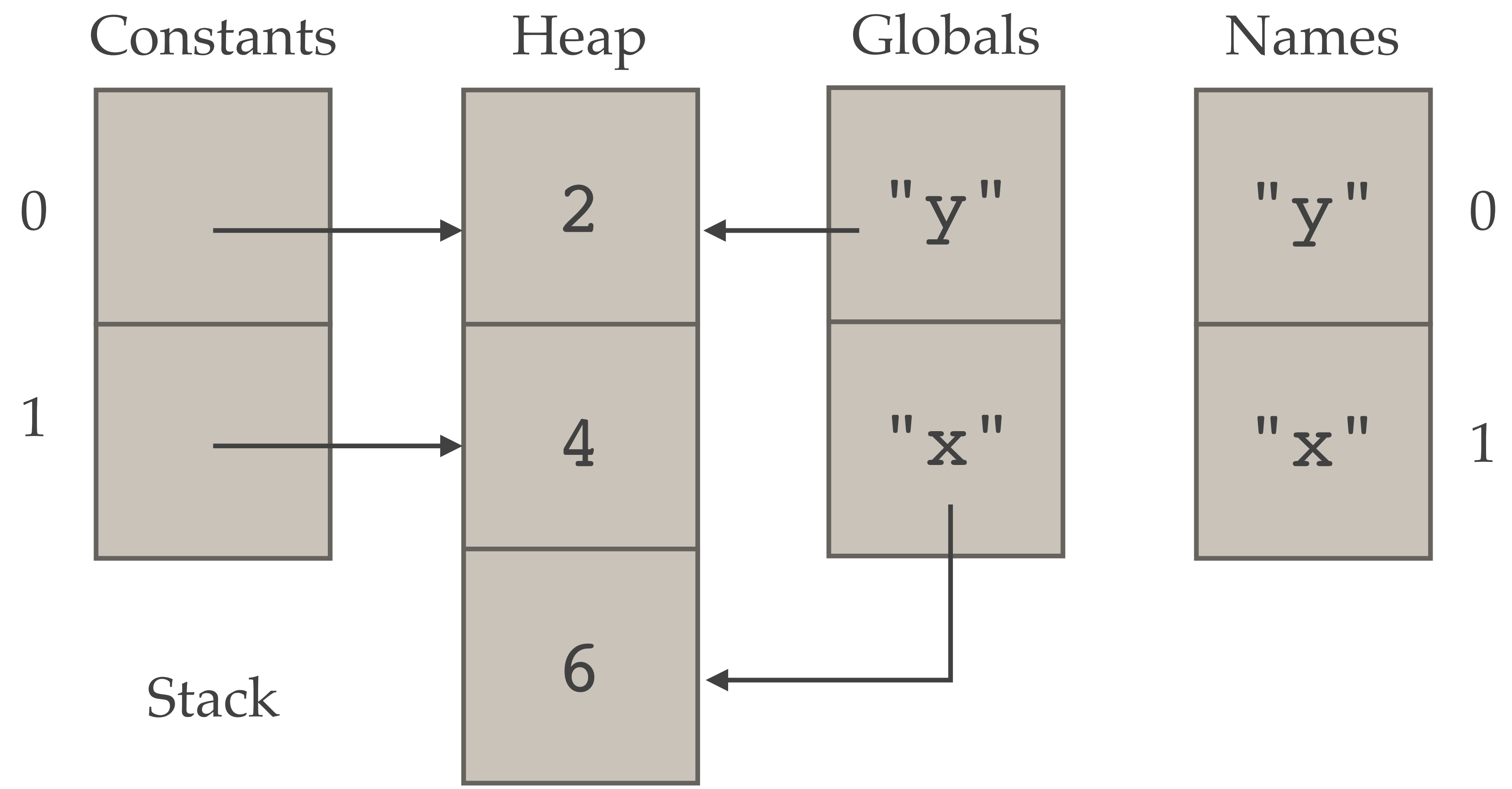
Example execution

Python source

```
y = 2
x = y + 4
```

Bytecode

```
LOAD_CONST 0
STORE_NAME 0
LOAD_NAME 0
LOAD_CONST 1
BINARY_ADD
STORE_NAME 1
```



Other ways of implementing languages

- In today's lecture we have described how the standard implementation of Python works (CPython). This is the one you use on IVLE.
- CPython is written in the C programming language.
- There are other alternative implementations of Python, such as:
    - Jython (compiles to Java bytecode)
    - PyPy (just-in-time compilation to machine code)
    - IronPython (implemented in C#, runs on .NET)

Other ways of implementing languages

- Some language implementations compile (translate) directly to machine language.
- We tend to say such implementations are **compiled**, whereas CPython is **interpreted**, but the distinction is blurry.
- Curiously most C compilers are written themselves in the C language, and they compile themselves! This is called bootstrapping.

Homework

- If modern C compilers are written in C (and compile themselves), where did the first C compilers come from?
- Would it be possible to implement Python in Python?