Melbourne Bioinformatics
12 August 2022

How Python* works

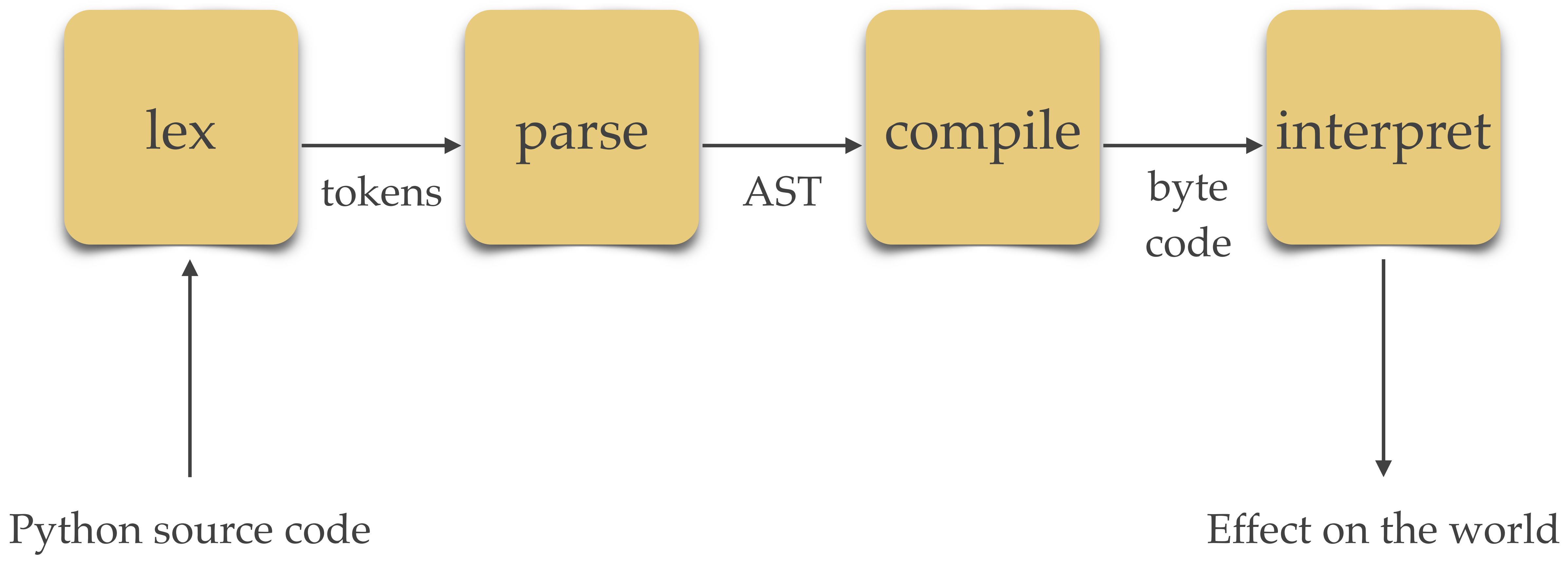
Bernie Pope, bjpope@unimelb.edu.au

*More specifically: How the reference implementation of Python, known as *CPython*, works

Outline

- Syntax analysis.
- Translation to bytecode.
- Execution.
- Other ways of implementing Python.

CPython'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 io 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'
NEWLINE ''
ENDMARKER ''
```

Python has a formal grammar

file: [statements] ENDMARKER

interactive: statement_newline

statements: statement+

statement: compound_stmt | simple_stmts

statement_newline:

| compound_stmt NEWLINE

| simple_stmts

| NEWLINE

| ENDMARKER

... *etcetera* …

see: https://docs.python.org/3/reference/grammar.html

Parsing produces an Abstract Syntax Tree

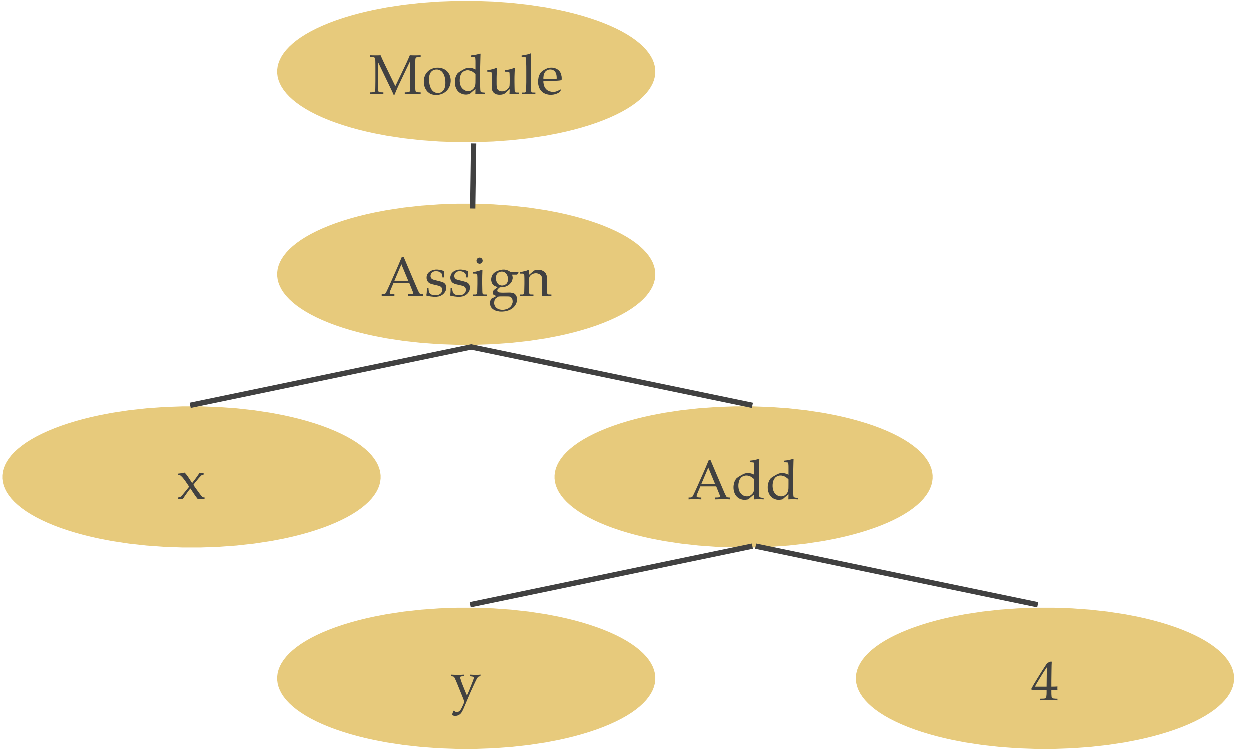
```
>>> from ast import (parse, dump)

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

>>>

>>> dump(tree, annotate_fields=False)

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



CPython 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*.

CPython Bytecode

An example bytecode instruction in binary:

```
01111100        0000000000000001
```

Opcode for the LOAD_FAST bytecode instruction

Operand (the integer 1)

CPython 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
```

CPython 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.

CPython 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.

CPython 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.

CPython 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.

CPython 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 CPython 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 on your computer 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.
- Code objects (such as those stored in .pyc files) are not (supposed to be) portable across CPython versions.

Execution

- Compiled CPython bytecode is executed by an interpreter which carries out the behaviour of the Virtual Machine.
- In CPython, the bytecode interpreter is written in C (hence the name CPython).
- 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.

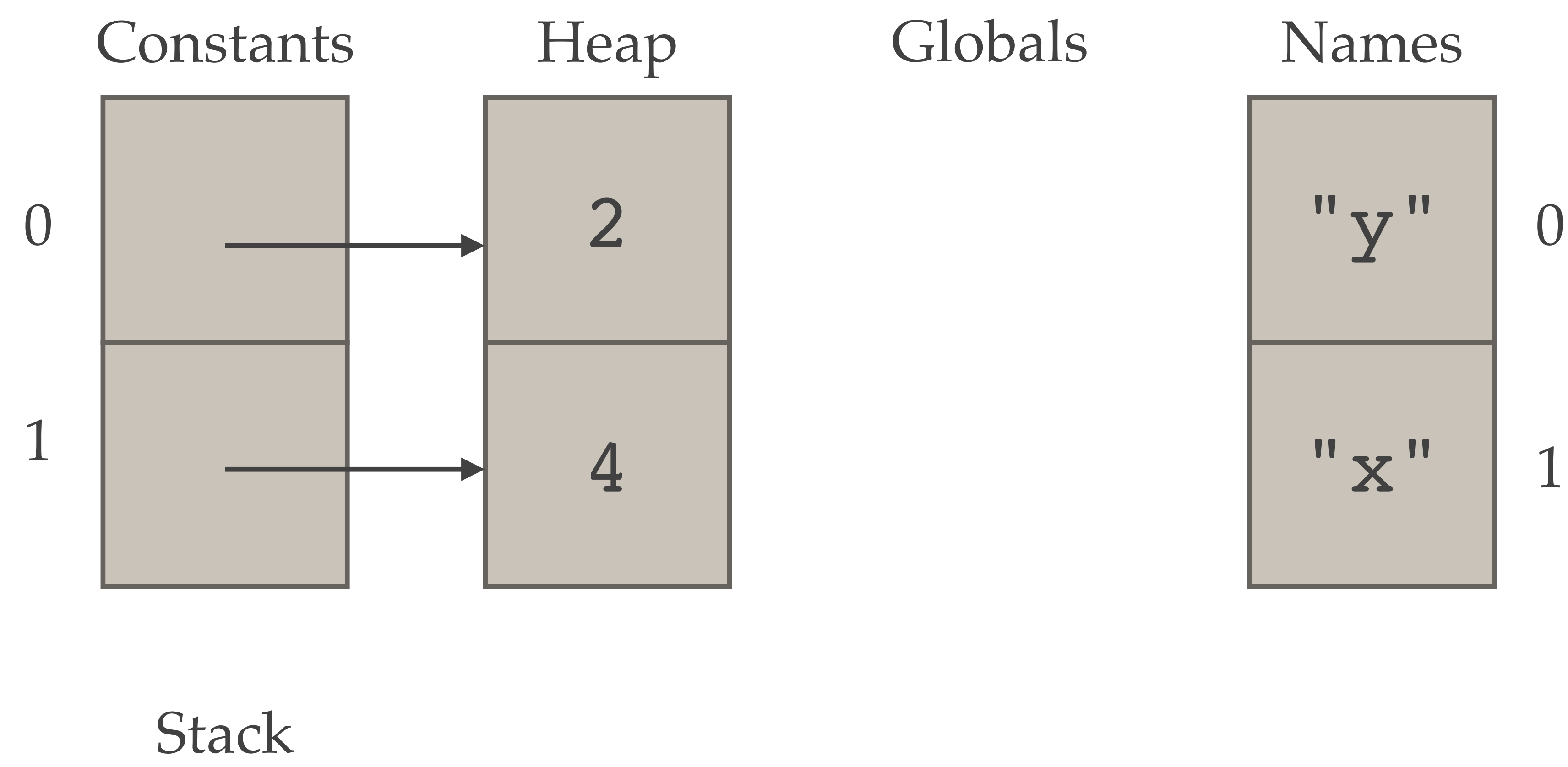
Example execution

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
```



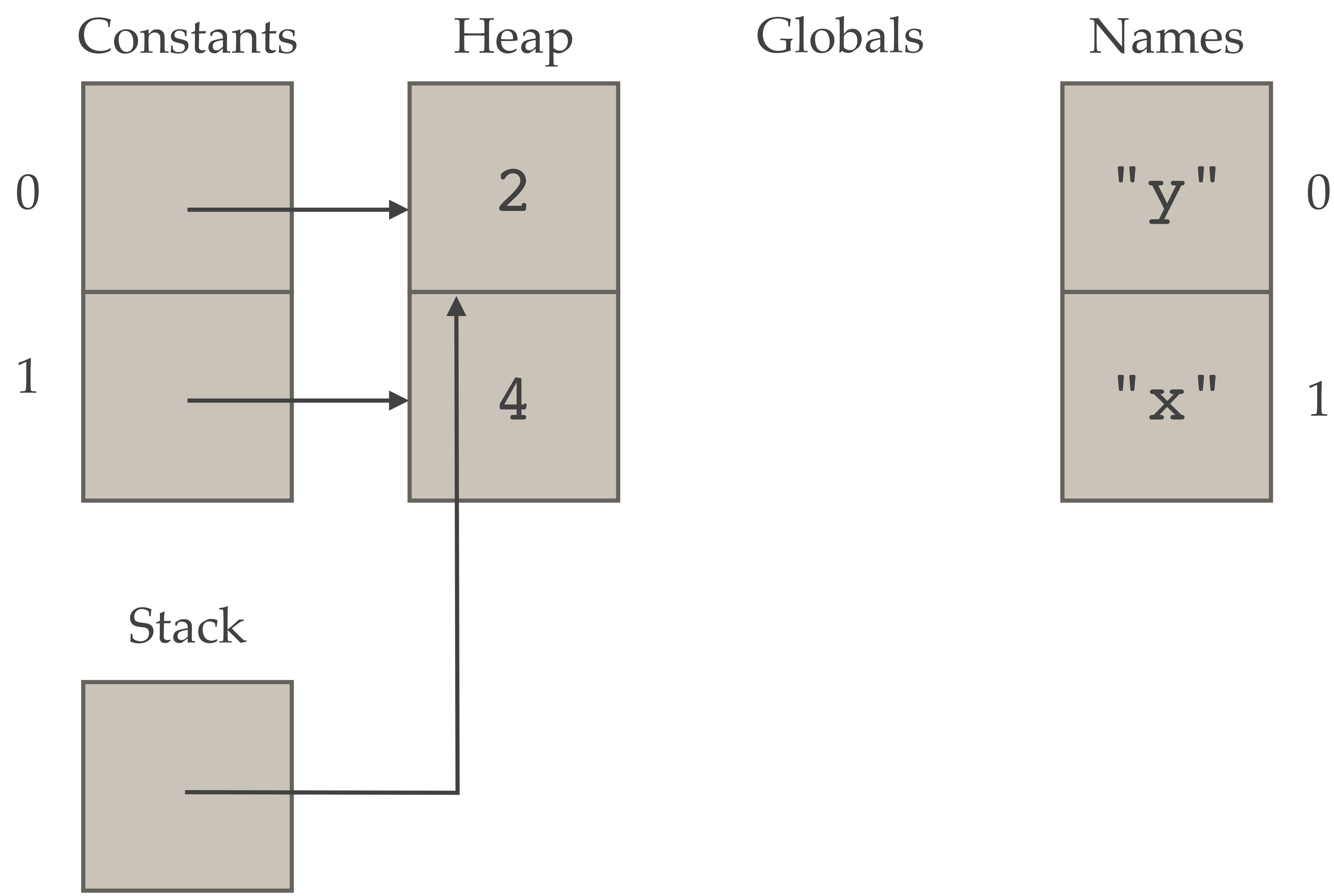
Example execution

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
```



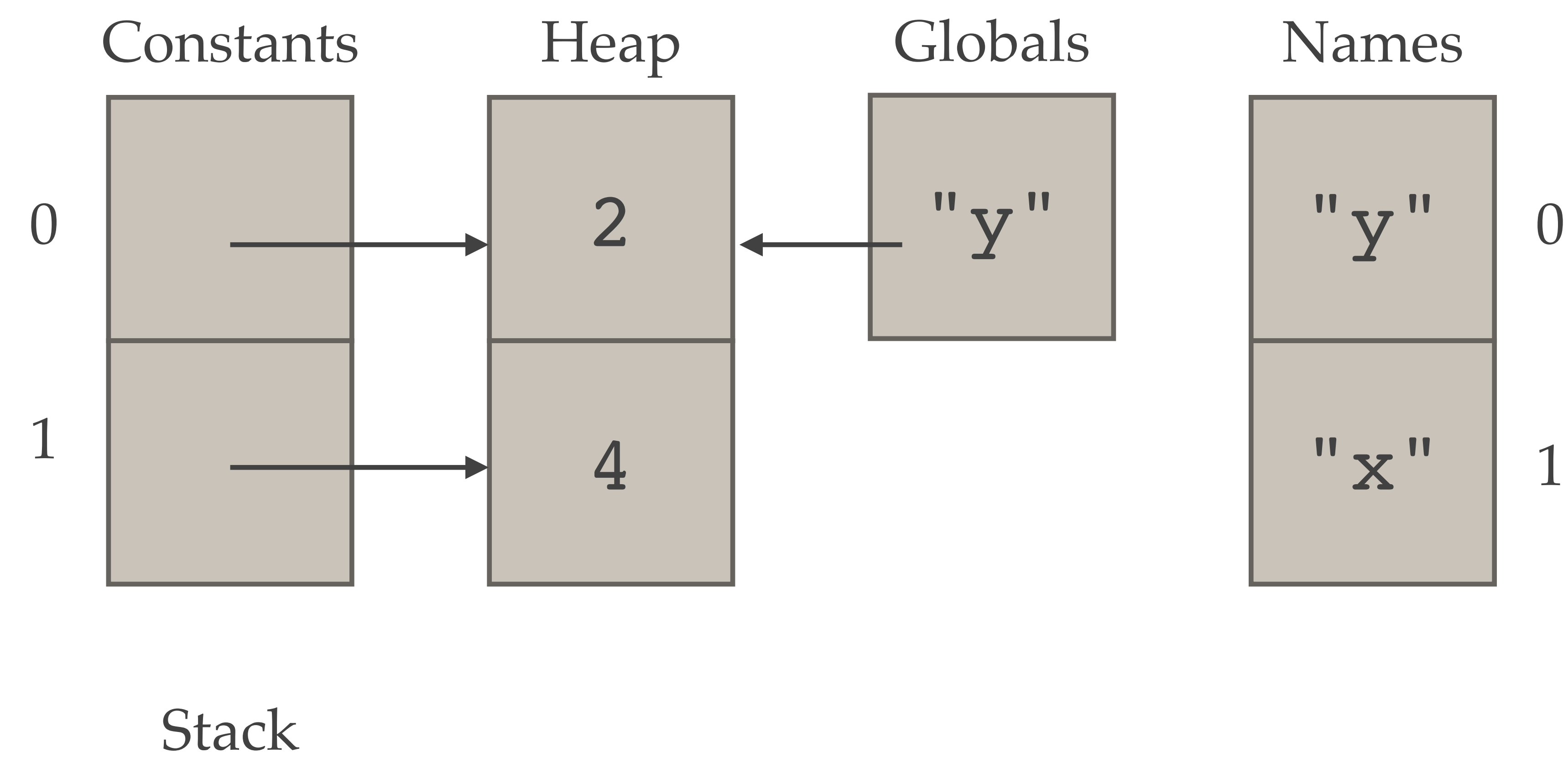
Example execution

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
```



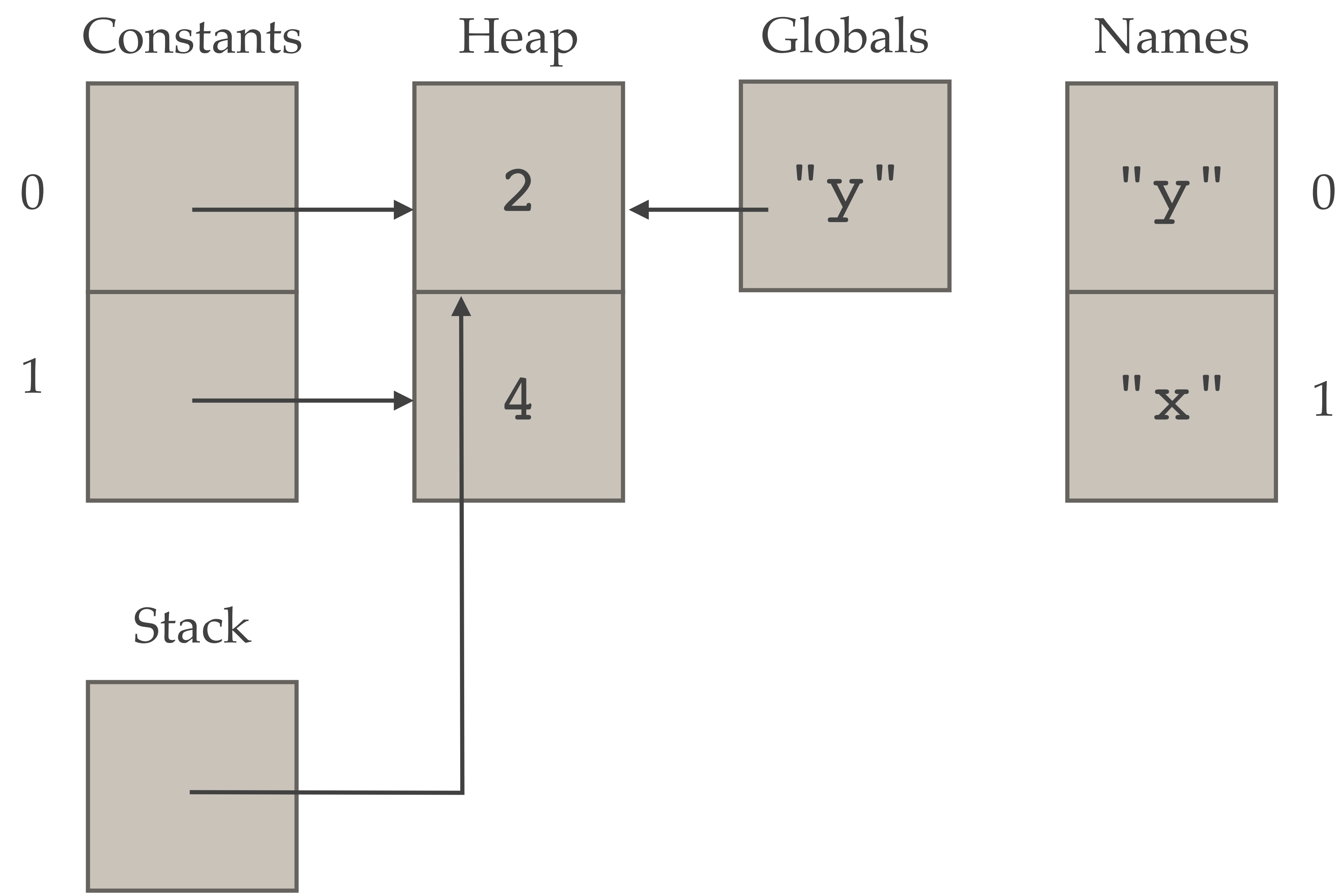
Example execution

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
```



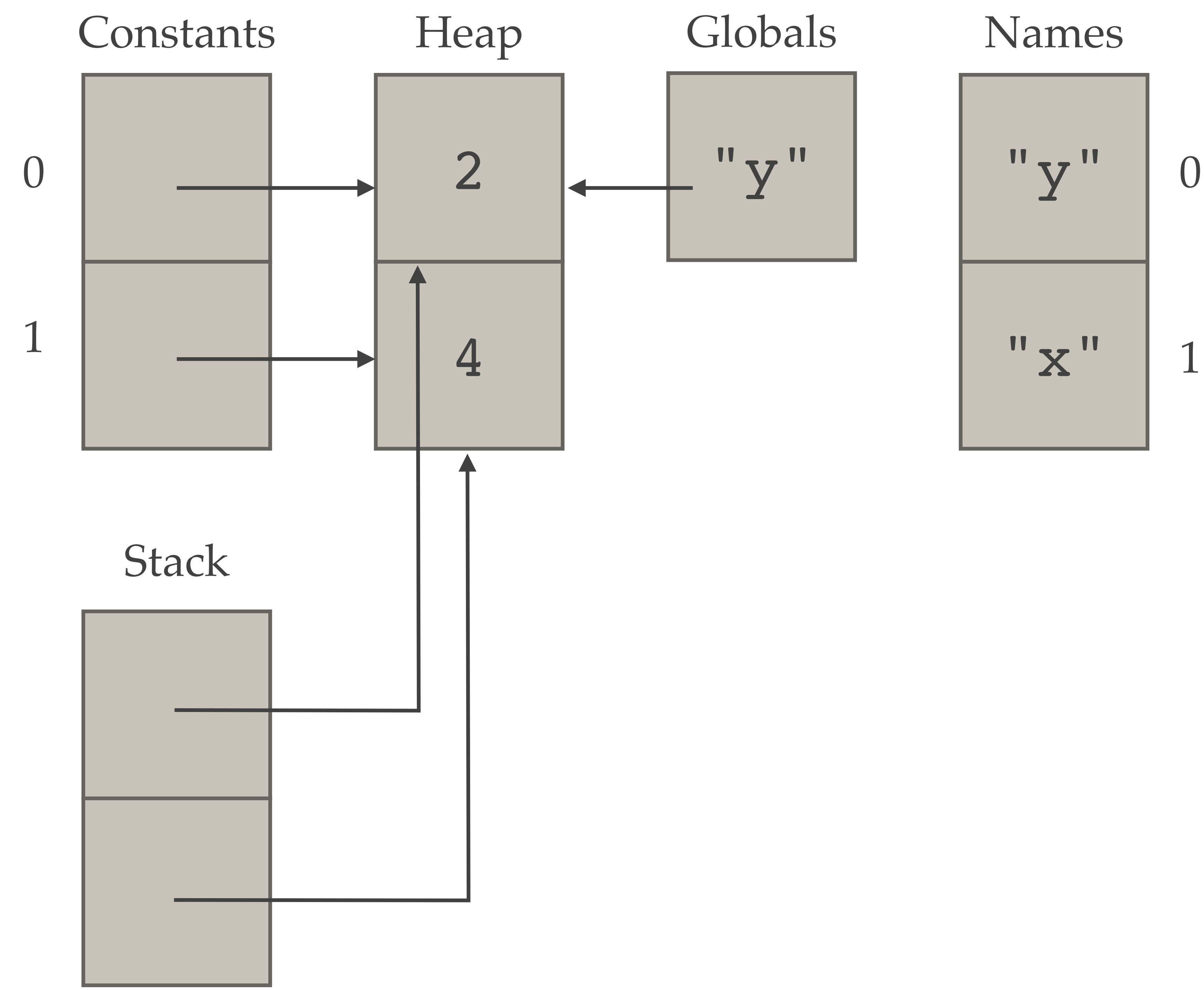
Example execution

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
```



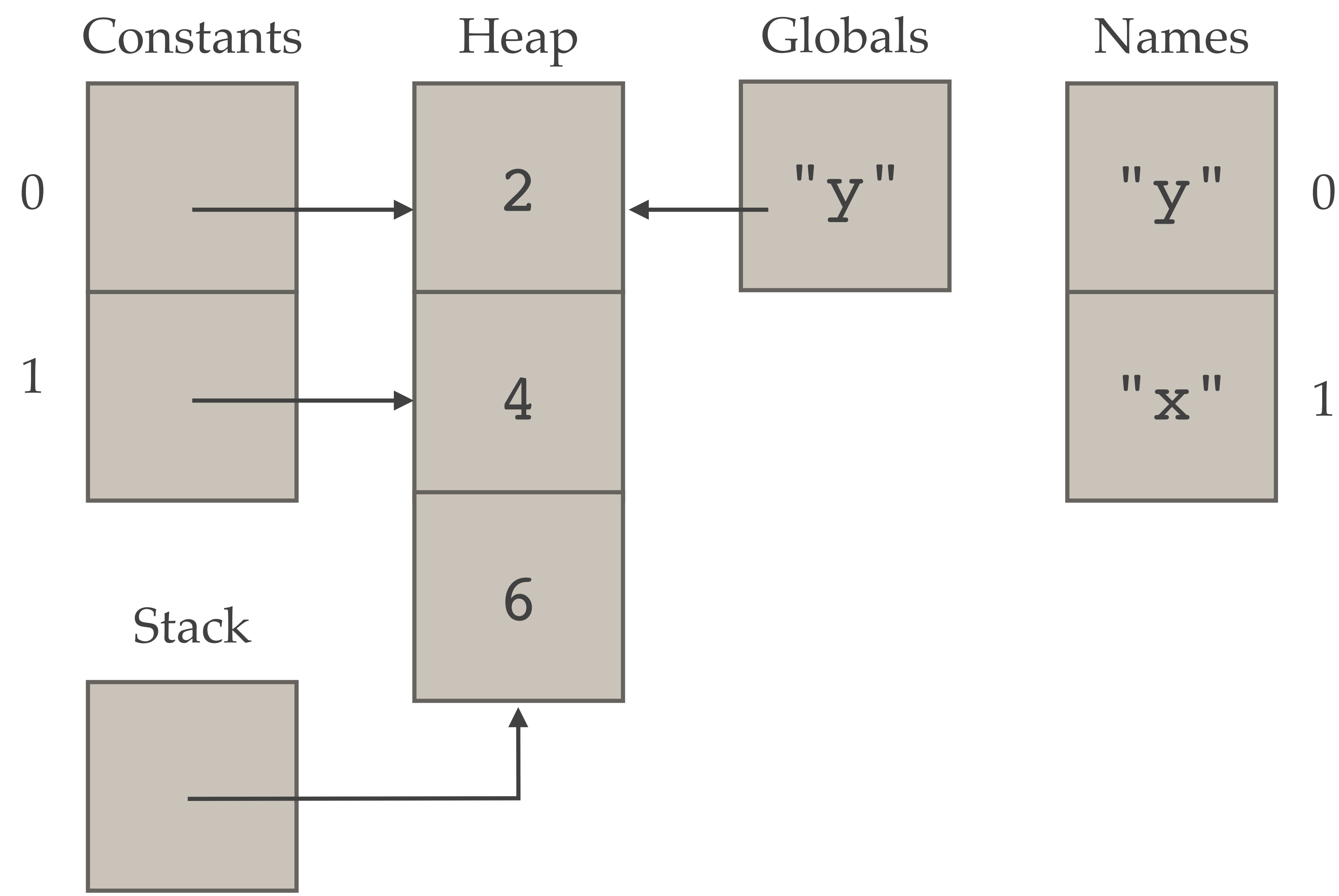
Example execution

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
```



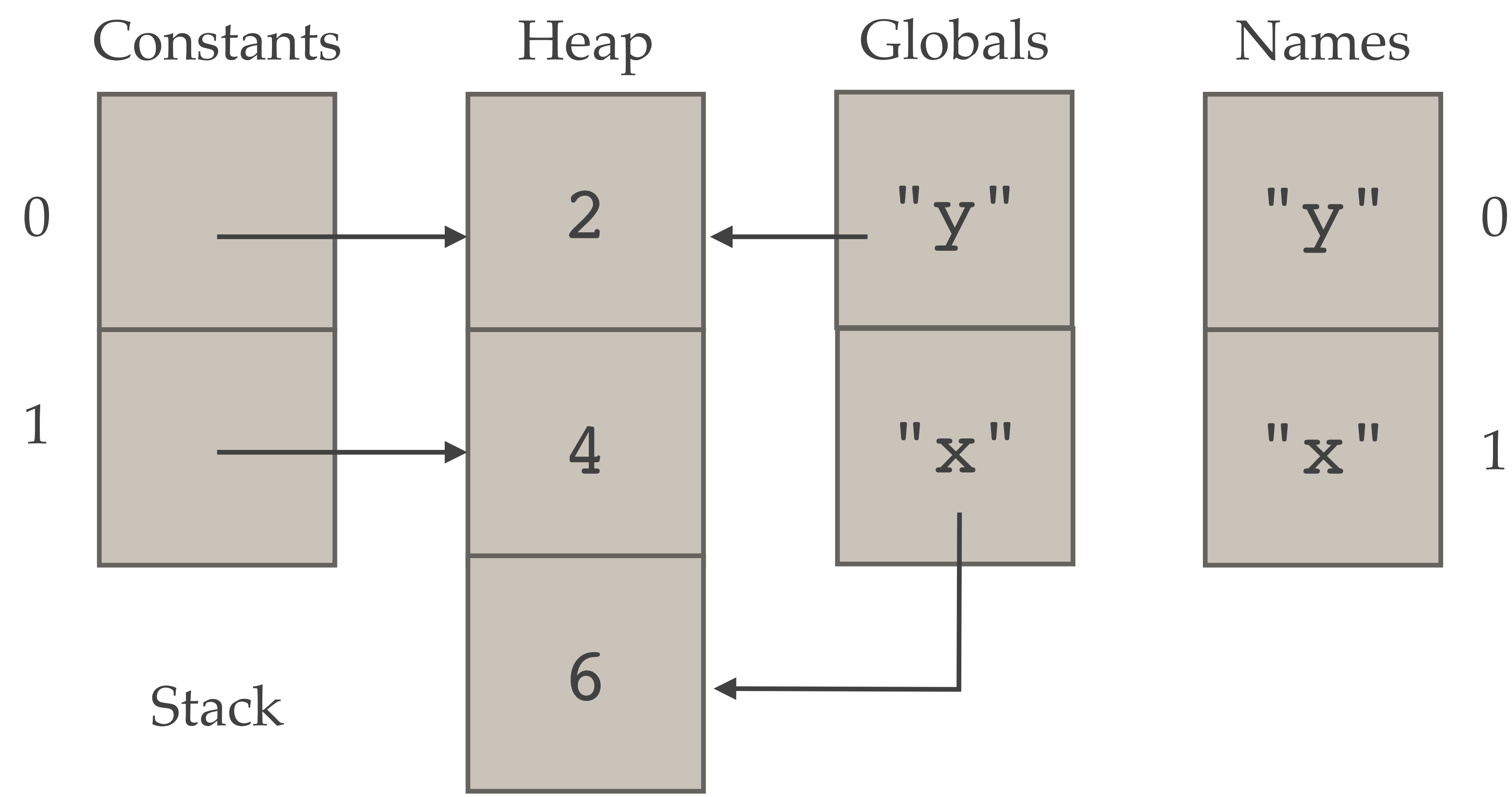
Example execution

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
```



Garbage collection

- Garbage collection (GC) identifies data in the interpreter heap that is no longer reachable by the running program.
- Memory used by unreachable heap data is reclaimed by GC for reuse.
- Without GC, heap usage would grow proportionally to program running time and eventually exhaust available virtual memory.

Garbage collection

- There have been lots of GC algorithms proposed for programming languages.
- CPython uses a very simple approach called *reference counting*.
- Every heap object contains a reference counter.
- The counter is incremented whenever a new pointer refers to the object, and decremented when a pointer no longer refers to the object.
- If the reference count reaches 0 then there are no longer any live pointers to the object and it becomes garbage. Its heap memory can be freed immediately.

Garbage collection

- Pros of reference counting:
  - simple to implement
  - easy to work with data from foreign code
  - memory is reclaimed immediately when an object becomes garbage
- Cons of reference counting:
  - Each object requires a counter. For small objects this is proportionally quite a large overhead in space.
  - Counter increments/decrements must be atomic operations to remain safe in a multi-threaded computation.
    - Atomic operations are relatively expensive on modern CPUs.
    - This overhead would be paid even in sequential code!

The Global Interpreter Lock

- The CPython bytecode interpreter is protected by a Global Interpreter Lock (GIL).
- This prevents more than one OS thread from executing the interpreter at any point in time in a given process.
- This allows the interpreter to use non-atomic reference count increment/decrements.
- However, the GIL does not apply to foreign code called from the bytecode interpreter: e.g. calls into C/Fortran/whatever libraries that don't call back into the interpreter.

The Global Interpreter Lock

- A "workaround" for the GIL is provided by the multiprocessing library.
- Each parallel instance is a separate OS process (multiple independent CPython instances running at once).
- Communication between processes is done by serialising/deserialising data.

The Global Interpreter Lock

- There have been many attempts to remove the GIL, but they have usually penalised the performance of single-threaded code. This has been considered untenable by the CPython maintainers.
- However very recent work from Sam Gross (at Facebook) called "nogil" shows *very* promising results.

Other ways of implementing Python

- 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)
    - Shameless plug: blip (implemented in Haskell)
        - https://github.com/bjpop/blip

Closing remarks on Python performance

- Python is a pleasant language in many ways but it was not designed with performance in mind.
- The dynamic nature of Python (i.e. no static types) means many operations are *extremely* slow compared to what is possible in other languages.
- Python can achieve good performance, but mostly by calling into foreign code, e.g. C/Fortran libraries, as is done in numpy for example.