Under the Hood of CPython

Simon Larsén

March 14, 2023

HiQ

Introduction

CPython is compiled

CPython is interpreted

Memory management

Closing words

Introduction

Please forgive me, here’s a logo

Who am I?

- Simon Larsén
- Software engineer at HiQ
- Open sourcerer
- Programming language enthusiast

Contact and social media

- Email: slarse@slar.se
- GitHub: https://github.com/slarse
  - https://github.com/slarse/talks
- LinkedIn (LINK)

Question for the audience

Is Python an interpreted or a compiled language?

Both and neither!

Python is a language specification

A language specification says what you are allowed to write (syntax) and what should happen if you run something syntactically correct (semantics).

CPython is the reference implementation of the Python language

So called because it is written in C!

But there are other implementations, such as PyPy and Jython [1].

CPython is compiled

Compiled? Really?

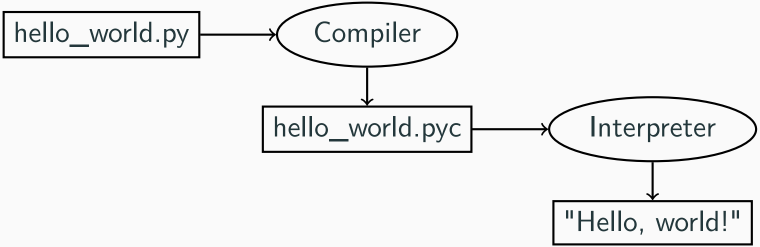
Running a program

When I run python hello_world.py, doesn't it just "interpret the text"?

No, that would be horrendously inefficient!

CPython compiles to bytecode before execution

And executes said bytecode in the interpreter (which we are getting to)



Python bytecode

- Each instruction consists of an opcode and an oparg
  - Generally instructions are two bytes (one for opcode, one for oparg)
- We can view bytecode with the dis module [2]
  - python −m dis <python_source_file>
  - Bytecode is an implementation detail of CPython and changes often
  - But the principle has remained the same since the dawn of CPython

This Python code

```python
def hello_world():
    greeting = "Hello,␣world!"
    print(greeting)
```

Compiles to this bytecode

```
 0 LOAD_CONST     1 ('Hello, world!')
 2 STORE_FAST     0 (greeting)

 4 LOAD_GLOBAL    0 (print)
 6 LOAD_FAST      0 (greeting)
 8 CALL_FUNCTION  1
10 POP_TOP
12 LOAD_CONST     0 (None)
14 RETURN_VALUE
```

CPython is interpreted

Bytecode evaluation

The interpreter loop

- Defined in Python/ceval.c [3]
- “Endless” for loop
- Runs until the main module returns

The interpreter is a stack machine!

- Computed values are pushed to the value stack
- Function arguments, return values etc are retrieved from the value stack

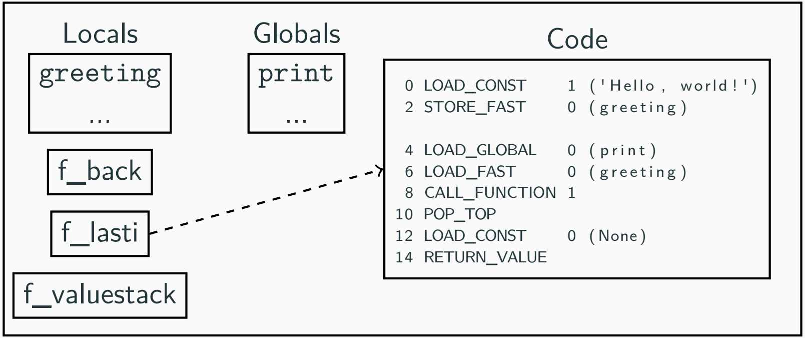
Part of the interpreter loop

```
main_loop:
    for (;;) {
        // [...]
        switch (opcode) {
        // [...]
        case TARGET(LOAD_CONST): {
            PREDICTED(LOAD_CONST);
            PyObject *value = GETITEM(consts, oparg);
            Py_INCREF(value);
            PUSH(value);
            FAST_DISPATCH();
        }
        // [...]
        case TARGET(POP_TOP): {
            PyObject *value = POP();
            Py_DECREF(value);
            FAST_DISPATCH();
        }
        // [...]
        }
    }
```

Frame objects

Python code is executed within a context called a *frame object*. When a function is called, a new frame object is created and entered. When it returns, the previous frame is entered and the returning function's frame is destroyed (typically).

Frame object (well, a partial one) for call to hello_world()



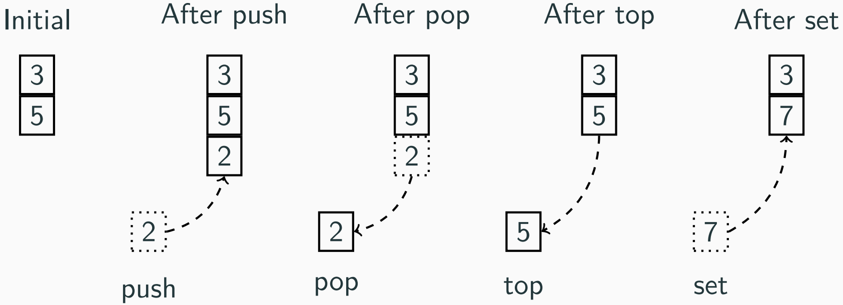
```
 0 LOAD_CONST      1 ('Hello, world!')
 2 STORE_FAST      0 (greeting)

 4 LOAD_GLOBAL     0 (print)
 6 LOAD_FAST       0 (greeting)
 8 CALL_FUNCTION   1
10 POP_TOP
12 LOAD_CONST      0 (None)
14 RETURN_VALUE
```

What is a stack?

A last-in-first-out (LIFO) data structure. We need to know about four operations:

- Push: Put something on top of the stack.
- Pop: Remove and return the topmost value from the stack.
- Peek/top: Look at the topmost value on the stack without removing it.
- Set: Overwrite the top of the stack

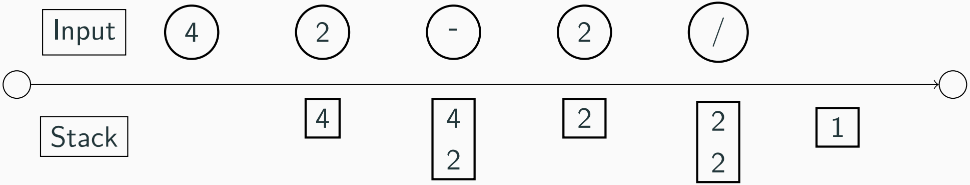


Stack machine example: Postfix (Reverse Polish) notation

- Use a stack to compute expressions without the need for parentheses
- Evaluate expression from left to right, and when we encounter an:
  - Operand: We push it to the stack
  - Operator: We pop two operands from the stack, apply the operator and push the result back on the stack

Example: (4 – 2) / 2

- In postfix notation it's: 4 2 – 2 /
- Note: The stack grows downward



Input

4 2 - 2 /

Stack

4

4
2

2

2
2

1

Python arithmetic expression

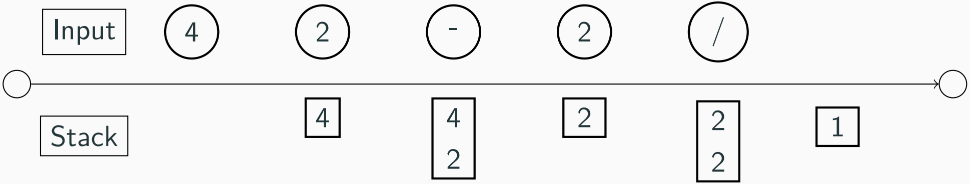
The expression (4 − 2) / 2 is evaluated the same way in Python!

Source code

```python
def example_expr(two, four):
    return (four - two) / two
```

Bytecode

```
 0 LOAD_FAST                1 (four)
 2 LOAD_FAST                0 (two)
 4 BINARY_SUBTRACT
 6 LOAD_FAST                0 (two)
 8 BINARY_TRUE_DIVIDE
10 RETURN_VALUE
```



BINARY_SUBTRACT evaluation

```
case TARGET(BINARY_SUBTRACT): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *diff = PyNumber_Subtract(left, right);
    Py_DECREF(right);
    Py_DECREF(left);
    SET_TOP(diff);
    if (diff == NULL)
        goto error;
    DISPATCH();
}
```

LOAD_FAST evaluation

```
case TARGET(LOAD_FAST): {
    PyObject *value = GETLOCAL(oparg);
    if (value == NULL) {
        // [...]
        goto error;
    }
    Py_INCREF(value);
    PUSH(value);
    FAST_DISPATCH();
}
```

BINARY_TRUE_DIVIDE evaluation

```c
case TARGET(BINARY_TRUE_DIVIDE): {
    PyObject *divisor = POP();
    PyObject *dividend = TOP();
    PyObject *quotient = PyNumber_TrueDivide(dividend, divisor);
    Py_DECREF(dividend);
    Py_DECREF(divisor);
    SET_TOP(quotient);
    if (quotient == NULL)
        goto error;
    DISPATCH();
}
```

Memory management

Question for the audience

How large is a Python ‘bool‘?

Everything is a PyObject

Every single value you can interact with in Python is wrapped in a PyObject struct, which is typically 28 bytes or larger.

```
# running Python 3.9
>>> sys.getsizeof(None)
16
>>> sys.getsizeof(True)
28
>>> sys.getsizeof(dict())
64
```

Takeaway: Python objects have *huge* memory footprints, so we need efficient disposal of unused objects.

Reference counting

PyObjects have an internal count of the amount of references to that object, which is:

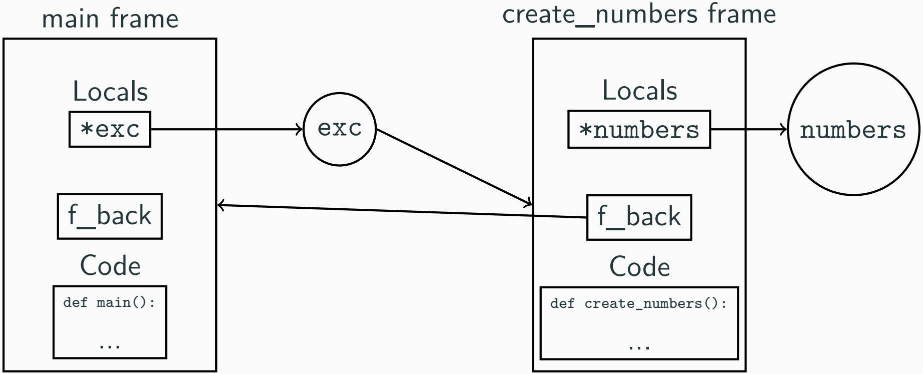
- Incremented when new references are created
- Decremented when references go out of scope
  - Out of scope $\sim$ frame object is destroyed
  - When the count reaches 0, the object is destroyed

Count increases with new references and decreases when references go out of scope

```python
my_obj = object()    # ob_refcnt = 1
also_obj = my_obj    # ob_refcnt = 2
print(my_obj)        # ob_refcnt = 3, reference is stored in print's frame (locals)
                     # ob_refcnt = 2 after print returns, its frame being destroyed
```

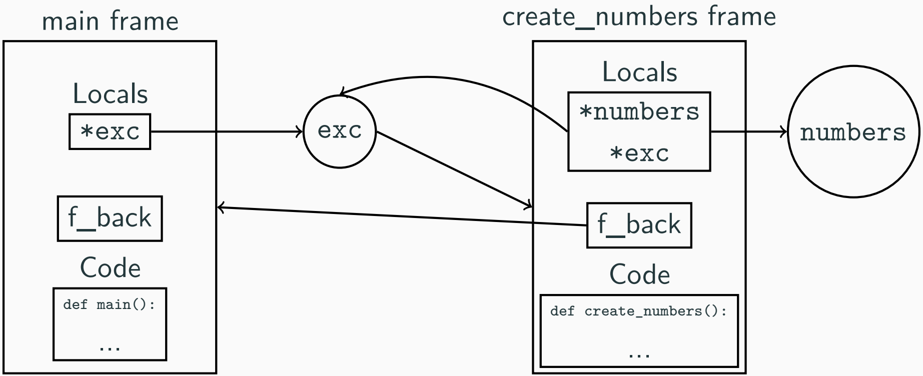
Cyclical references from exception handling

If you catch an exception, you get a cyclical reference!



Cyclical references from exception handling

If you catch an exception, you get a cyclical reference!



Garbage collection

Reference counting does not work if there are cyclical references.

```python
def create_cyclical_reference():
    my_list = []
    my_list.append(my_list)
```

It is the job of the garbage collector (GC) to find and dispose of objects that are *unreachable* from the running program.

Closing words

Recap

- Python is compiled and interpreted
  - Compiling a program before hand makes startup faster
  - Although to be completely honest I've never, ever, ever done that :)
- Python's interpreter is a stack machine
  - Actually that's not all that useful to know unless you work on the interpreter
  - But great inspiration if you want to create a programming language of your own!
- Python's memory management: reference counting and garbage collection
  - Make sure variables with large amounts of data go out of scope ASAP
  - Cyclical references requires garbage collection (runs automatically)

Want to learn more?

- I learned most of this from the book *CPython Internals* [4]
  - Great book, highly recommend it
  - The author also made a blog post with a (much) shorter overview of the same thing [5]
- The official developer guide contains a lot of good info on how to work with CPython [6]

References i

[1] “Python alternate implementations,” 2022. [Online]. Available: https://docs.python.org/3/reference/introduction.html#alternate-implementations.

[2] “Python dis module,” 2022. [Online]. Available: https://docs.python.org/3/library/dis.html.

[3] “CPython source code,” 2022. [Online]. Available: https://github.com/python/cpython.

[4] A. Shaw, “CPython internals,” 2019. [Online]. Available: https://realpython.com/products/cpython-internals-book/.

[5] A. Shaw, “Your guide to the CPython source code,” 2019. [Online]. Available: https://realpython.com/cpython-source-code-guide/.

References ii

[6] “Python developer’s guide,” 2022. [Online]. Available: https://devguide.python.org/.