DEBUG IS THE NEW RELEASE
THE UNEXPECTED BENEFITS OF SLOW LANGUAGES
ARMIN @MITSUHIKO RONACHER / FLASK, WERKZEUG, JINJA, CLICK, ... / DIRECTOR OF ENGINEERING AT SENTRY / LOVES OSS
1. developer experience matters
2. the ability to debug matters
3. debugging does not stop when shipping a release
>>> import sys
>>> a_variable = 'Hello World!'
>>> sys._getframe().f_locals['a_variable']
'Hello World!'
RUNTIME INTROSPECTION IS POWERFUL
RUNTIME INTRUSION IS SLOW
RUNTIME INTROSPECTION IS IMPORTANT
“WORKS ON MY MACHINE”
MILLIONS OF BROWSER SESSIONS
HUNDREDS OF COUNTRIES
IOT DEVICES
MOBILE PHONES
you need basic debugging and introspection in production
PROD VS DEBUG PERF
let's talk about runtimes ...
Simple Interpreter
JIT compiled
AOT compiled
• these are examples
• not scientific
• not entirely comparable
A Simple Interpreter: CPython
>>> import dis

>>> def add_numbers(a, b):
...     return a + b
...

>>> dis.dis(add_numbers)

   2           0 LOAD_FAST                0 (a)
   2 LOAD_FAST                1 (b)
   4 BINARY_ADD
   6 RETURN_VALUE
while ... {
    switch (...) {
        case TARGET(LOAD_FAST): {
            PyObject *value = GETLOCAL(oparg);
            if (value == NULL) {
                format_exc_check_arg(tstate, PyExc_UnboundLocalError,
                UNBOUNDLOCAL_ERROR_MSG,
                PyTuple_GetItem(co->co_varnames, oparg));

                goto error;
            }
            Py_INCREF(value);
            PUSH(value);
            PUSH(value);
            FAST_DISPATCH();
        }
    }
}
case TARGET(BINARY_ADD): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *sum;
    if (PyUnicode_CheckExact(left) &&
        PyUnicode_CheckExact(right)) {
        sum = unicode_concatenate(tstate, left, right, f, next_instr);
    } else {
        sum = PyNumber_Add(left, right);
        Py_DECREF(left);
    }
    Py_DECREF(right);
    SET_TOP(sum);
    if (sum == NULL)
        goto error;
    DISPATCH();
}
there is a lot of compiled code executing every instruction
import sys

def failing_func():
    raise Exception('Oh noes')

def catching_func():
    try:
        failing_func()
    except Exception:
        pass

def stacktrace_making_func():
    try:
        failing_func()
    except Exception:
        sys.exc_info()
mituhiko at argus in /tmp
$ python -mtimeit -s 'from test import catching_func as x' 'x()
1000000 loops, best of 3: 1.34 usec per loop

mituhiko at argus in /tmp
$ python -mtimeit -s 'from test import stacktrace_making_func as x' 'x()
1000000 loops, best of 3: 1.44 usec per loop

7% SLOWER
JIT Compiled Interpreter: V8
function throwingFunc() {
    throw new Error('Oh noes');
}

function catchingFunc() {
    try {
        throwingFunc();
    } catch (err) {
    }
}

function stacktraceMakingFunc() {
    try {
        throwingFunc();
    } catch (err) {
        return err.stack;
    }
}
catching \times 160,895 \ text{ops/sec} \ \pm2.30\% \ (60 \ runs \ sampled)

stacktrace making \times 26,495 \ text{ops/sec} \ \pm1.98\% \ (86 \ runs \ sampled)

83\% \ \texttt{SLOWER}
Native Code: clang
well what's a stack trace anyways?
STACK WALKING
there is a little DWARF in your computer
**stack unwinding:** go to where a function would return to
okay it's “fast”, but it's also pretty bad
because better would be much slower
• unwinding on device
• deferred symbolication
• pain, suffering and disappointment
want stack traces? need to capture when exceptions are thrown
1. debuggability incurs runtime cost
2. JIT/AOT optimizations break down
3. If you want debug functionality in production, percentage performance loss matters
Sentry exists, because cheap in-production debugging is amazing and not much slower in Python
BUT ARMIN, STACK TRACES ARE FAST!!!
but we expect more
value what you have, Python developers
WHAT DO WE HAVE?
>>> import sys
>>> sys._current_frames()
{4656870848: <frame at 0x109177d50, file '<stdin>', line 1, code <module>>}
>>> import sys
>>> sys._getframe().f_locals
{'__annotations__': {},
 '.__builtins__': <module 'builtins' (built-in)>,
 '.__cached__': None,
 '.__doc__': None,
 '.__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x1090d55d0>,
 '.__name__': '__main__',
 '.__package__': None,
 '.__spec__': None,
 'sys': <module 'sys' (built-in)>>}
```python
>>> try:
...   1/0
... except Exception as e:
...   e.__traceback__
...
<traceback object at 0x1093559b0>
```
from threading import Thread
old_start = Thread.start
Thread.start = make_new_start(old_start)
you can also attach a debugger, run some code and start a reverse python shell on a running process
& Python 3.7 has execution contexts (context vars)
WHAT WILL THE FUTURE BRING
ASK YOUR QUESTIONS — I DON'T BITE