

```
{
  "talk": {
    "title": "PyPy on cutting edge.",
    "event_id": "PiterPy #4",
  },
  "speaker": {
    "__qname__" : "Aleksandr Koshkin",
    "linkedin"  : "lnkfy.com/7Do",
    "github"    : "/magniff",
  }
}
```



https://bitbucket.org/pypy/pypy

Activities Br okt 17, 00:24

Bitbucket / pypy / pypy — Bitbucket

Atlassian, Inc. [US] | https://bitbucket.org/pypy/pypy

**PyPY**

- Overview
- Source
- Commits
- Branches
- Pull requests
- Pipelines
- Issues
- Wiki
- Downloads

PyPy / PyPy project / pypy

## Overview

HTTPS <https://bitbucket.org/pypy/pypy>

Last updated 4 hours ago	16 Open PRs	782 Watchers
Website <a href="http://pypy.org/">http://pypy.org/</a>		
Language Python	99+ Branches	503 Forks
Access level Read		

### PyPy: Python in Python Implementation

Welcome to PyPy!

PyPy is both an implementation of the Python programming language, and an extensive compiler framework for dynamic language implementations. You can build self-contained Python implementations which execute independently from CPython.

The home page is:

<http://pypy.org/>

If you want to help developing PyPy, this document might help you:

<http://doc.pypy.org/>

#### Recent activity

- 1 commit**  
Pushed to pypy/pypy  
[9f84023](#) Mark slots test as an implementa...  
Ronan Lamy · 4 hours ago
- 1 commit**  
Pushed to pypy/pypy  
[09772d3](#) Don't allow passing str to ctypes...  
Ronan Lamy · 5 hours ago
- 3 commits**  
Pushed to pypy/pypy  
[a5eb32f](#) merge cpyext-avoid-roundtrip ins...  
[0cd1a1f](#) \_Py\_NewReference did not initial...  
[718a914](#) add a comment about a potential...  
Antonio Cuni · 10 hours ago
- 1 commit**  
Pushed to pypy/pypy  
[c5011e4](#) Remove some entries from the T...  
Manuel Jacob · 18 hours ago
- 1 commit**



So, what is PyPy?



- Turbo awesome **dynamic language** designer framework.
- Boring tutorial that went too far.



Just a small example

```
def abs(value):  
    if value >= 0:  
        return value  
    else:  
        return -1 * value
```

```
for value in range(10**9):  
    abs(value)
```



```
$ time pypy3 abs.py  
real 1.082s
```

```
$ time python3 abs.py  
real 2m2.030s
```



**~120X** faster, Karl!



Cool, lets go check it!





:3



POSITIVE TECHNOLOGIES



# The aim of the talk

- review the basic idea behind PyPy JIT
- be as human friendly as possible
- avoid using JIT buzzword
- (I did it again, did I?)



So what's the scoop?




# Frame evaluator

```
PyObject *
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    co = f->f_code;
    for (;;) {
        switch (opcode) {
            TARGET(LOAD_FAST)    { /* implementation of LOAD_FAST */
            TARGET(LOAD_CONST)  { /* implementation of LOAD_CONST */
            TARGET(BINARY_ADD)  { /* implementation of BINARY_ADD */
            ...
        }
    }
```



## C API

```
TARGET (BINARY_ADD) {  
    PyObject *right = POP();  
    PyObject *left = TOP();  
    PyObject *sum;  
    sum = PyNumber_Add(left, right);  
    DISPATCH();  
}
```



```
PyObject *  
PyNumber_Add(PyObject *v, PyObject *w)  
{  
    PyObject *result = binary_op1(v, w, NB_SLOT(nb_add));  
    ...  
    return result;  
}
```



```
static PyObject *
binary_op1(PyObject *v, PyObject *w, const int op_slot)
{
    PyObject *x;
    slotv = NB_BINOP(v->ob_type->tp_as_number, op_slot);
    x = slotv(v, w);
    return x;
}
```



```
PyTypeObject PyLong_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "int",
    offsetof(PyLongObject, ob_digit),
    sizeof(digit),
    ...
    &long_as_number, ←
    ...
};
```







## Deadly sins of dynamic programming languages

late binding: `a.tp_as_number.number_add`

boxing: `PyObject, PyType_...`

vm overhead: `stack, program counter`





Late binding:

```
some_method = some_object.method  
for value in huge_list:  
    some_method(value)
```



Late binding:

```
def something(more=None):  
    for item in huge_list:  
        → more += func(20, 30)  
    return more
```



# Boxing:

```
00100000000100101000111100000000
00000000000000000000000000000000
10000000000100101000111100000000
00000000000000000000000000000000
00001111000000000000000000000000
00000000000000000000000000000000
11100000111001001000100100000000
00000000000000000000000000000000
00000001000000000000000000000000
00000000000000000000000000000000
01100100000000000000000000000000
```



So, how to get rid of VM overhead?



Can we merge eval function and actual code somehow?



```
def abs(value):  
    if value >= 0:  
        return value  
    else:  
        return -1 * value
```





```
>>> dis.dis(abs)
```

```
2          0 LOAD_FAST          0 (value)
          3 LOAD_CONST          1 (0)
          6 COMPARE_OP          5 (>=)
          9 POP_JUMP_IF_FALSE    16
```

```
3          12 LOAD_FAST         0 (value)
          15 RETURN_VALUE
```

```
5          >> 16 LOAD_CONST         3 (-1)
          19 LOAD_FAST          0 (value)
          22 BINARY_MULTIPLY
          23 RETURN_VALUE
```



```
def eval_frame(frame):
    stack = []
    code_counter = 0

    while code_counter < len(frame.code.co_code):
        opcode = frame.code.co_code[code_counter]
        if opcode == "LOAD_FAST":
            stack.append(frame.lookup_fast(argument))
            code_counter += 3
        elif opcode == "LOAD_CONST":
            stack.append(frame.code_object.get_const(argument))
            code_counter += 3
        elif opcode == "COMPARE_OP":
            value1 = stack.pop()
            value0 = stack.pop()
            comparator = get_comparator(argument)
            result = comparator(value0, value1)
            stack.append(result)
            code_counter += 2
        elif opcode == "BINARY_MULTIPLY":
            value1 = stack.pop()
            value0 = stack.pop()
            result = do_multiplication(value0, value1)
            stack.append(result)
            code_counter += 2
        ...
```



```
class BoxedValue:  
    def __init__(self, value):  
        self.low_level_value = value
```

```
class LongObject(BoxedObject):  
    pass
```

```
class BoolObject(BoxedObject):  
    pass
```



Direct manipulation is impossible:

```
result = LongObject(10) + LongObject(20)
```

Use object layer API instead:

```
result = add_value(  
    LongObject(10), LongObject(20)  
)
```



Can we merge eval function and actual code somehow?



```
def eval_frame_abs(frame):
    stack = []
    stack.append(frame.get_fast(0))
    stack.append(code.get_const(1))
    value0 = stack.pop()
    value1 = stack.pop()
    comparator = get_comparator(5)
    stack.append(comparator(value0, value1))
    if is_true(stack.pop()).low_level_value:
        stack.append(frame.get_fast(0))
        return stack.pop()
    else:
        stack.append(code.get_const(3))
        stack.append(frame.get_fast(0))
        value0 = stack.pop()
        value1 = stack.pop()
        stack.append(do_multiplication(value0, value1))
        return stack.pop()
```



```
def eval_frame_abs(frame):
    stack = []
    stack.append(frame.get_fast(0))
    stack.append(code.get_const(1))
    value0 = stack.pop()
    value1 = stack.pop()
    comparator = get_comparator(5)
    stack.append(comparator(value0, value1))
    if is_true(stack.pop()).low_level_value:
        stack.append(frame.get_fast(0))
        return stack.pop()
    else:
        stack.append(code.get_const(3))
        stack.append(frame.get_fast(0))
        value0 = stack.pop()
        value1 = stack.pop()
        stack.append(do_multiplication(value0, value1))
        return stack.pop()
```



```
def eval_frame_abs(frame):
    comparator = get_comparator(5)
    compare_result = comparator(
        frame.get_fast(0), frame.get_const(1)
    )
    if is_true(compare_result).low_level_value:
        return frame.get_fast(0)
    else:
        return do_multiplication(
            code.get_const(3),
            frame.get_fast(0)
        )
```





```
def eval_frame_abs(frame):
    comparator = get_comparator(5)
    compare_result = comparator(
        frame.get_fast(0), frame.get_const(1)
    )
    if is_true(compare_result).low_level_value:
        return frame.get_fast(0)
    else:
        return do_multiplication(
            code.get_const(3),
            frame.get_fast(0)
        )
```



```
def eval_frame_abs(frame):
    comparator = get_comparator(5)
    compare_result = comparator(
        frame.get_fast(0), frame.get_const(1)
    )
    if compare_result.low_level_value:
        return frame.get_fast(0)
    else:
        return do_multiplication(
            code.get_const(3),
            frame.get_fast(0)
        )
```



```
def eval_frame_abs(frame):
    comparator = get_comparator(5)
    compare_result = comparator(
        frame.get_fast(0), frame.get_const(1)
    )
    if compare_result.low_level_value:
        return frame.get_fast(0)
    else:
        return do_multiplication(
            code.get_const(3),
            frame.get_fast(0)
        )
```



```
def eval_frame_abs(frame):
    compare_result = gt_eq(
        frame.fast_variables[0], LongObject(0)
    )
    if compare_result.low_level_value:
        return frame.fast_variables[0]
    else:
        return do_multiplication(
            LongObject(-1),
            frame.fast_variables[0]
        )
```



```
def eval_frame_abs(frame):
    compare_result = gt_eq(
        frame.fast_variables[0], LongObject(0)
    )
    if compare_result.low_level_value:
        return frame.fast_variables[0]
    else:
        return do_multiplication(
            LongObject(-1),
            frame.fast_variables[0]
        )
```



```
def eval_frame_abs(frame):
    compare_result = gt_eq(
        frame.fast_variables[0], LongObject(0)
    )
    if compare_result.low_level_value:
        return frame.fast_variables[0]
    else:
        return do_multiplication(
            LongObject(-1),
            frame.fast_variables[0]
        )
```



```
def do_multiplication(value0, value1):  
    if value0.type == value1.type == LongType:  
        result = (  
            value0.low_level_value * value1.low_level_value  
        )  
        return LongObject(result)  
    elif ...
```

```
def gt_eq(value0, value1):  
    if value0.type == value1.type == LongType:  
        result = (  
            value0.low_level_value >= value1.low_level_value  
        )  
        return BoolObject(result)  
    elif ...
```



In this case types define control flow inside the object layer API functions.





```
someresult = abc(-20)
```



- Inline `gt_eq` and `do_multiplication` functions according to types.
- Unbox constant values.



this used to be boxed

```
def eval_frame_abs(frame):  
    cmp_result = BoolObject(  
        frame.fast_variables[0].low_level_value >= 0  
    )  
    if cmp_result.low_level_value:  
        return frame.fast_variables[0]  
    else:  
        return LongObject(  
            -1*frame.fast_variables[0].low_level_value  
        )
```

this used to be boxed



```
def eval_frame_abs(frame):
    cmp_result = BoolObject(
        frame.fast_variables[0].low_level_value >= 0
    )
    if cmp_result.low_level_value:
        return frame.fast_variables[0]
    else:
        return LongObject(
            -1*frame.fast_variables[0].low_level_value
        )
```



```
def eval_frame_abs(frame):  
    if frame.fast_variables[0].low_level_value >= 0:  
        return frame.fast_variables[0]  
    else:  
        return LongObject(  
            -1*frame.fast_variables[0].low_level_value  
        )
```



```
someresult = abc(-20)
```



```
def eval_frame_abs(frame):  
    return LongObject(  
        -1*frame.fast_variables[0].low_level_value  
    )
```



How about inline caching?





```
cached_value = LongObject(20)
```

```
def eval_frame_abs(frame):  
    return cached_value
```



How about throwing code away?



```
def eval_frame_abs(frame) :  
    pass
```



Sometime this happens, you never know...

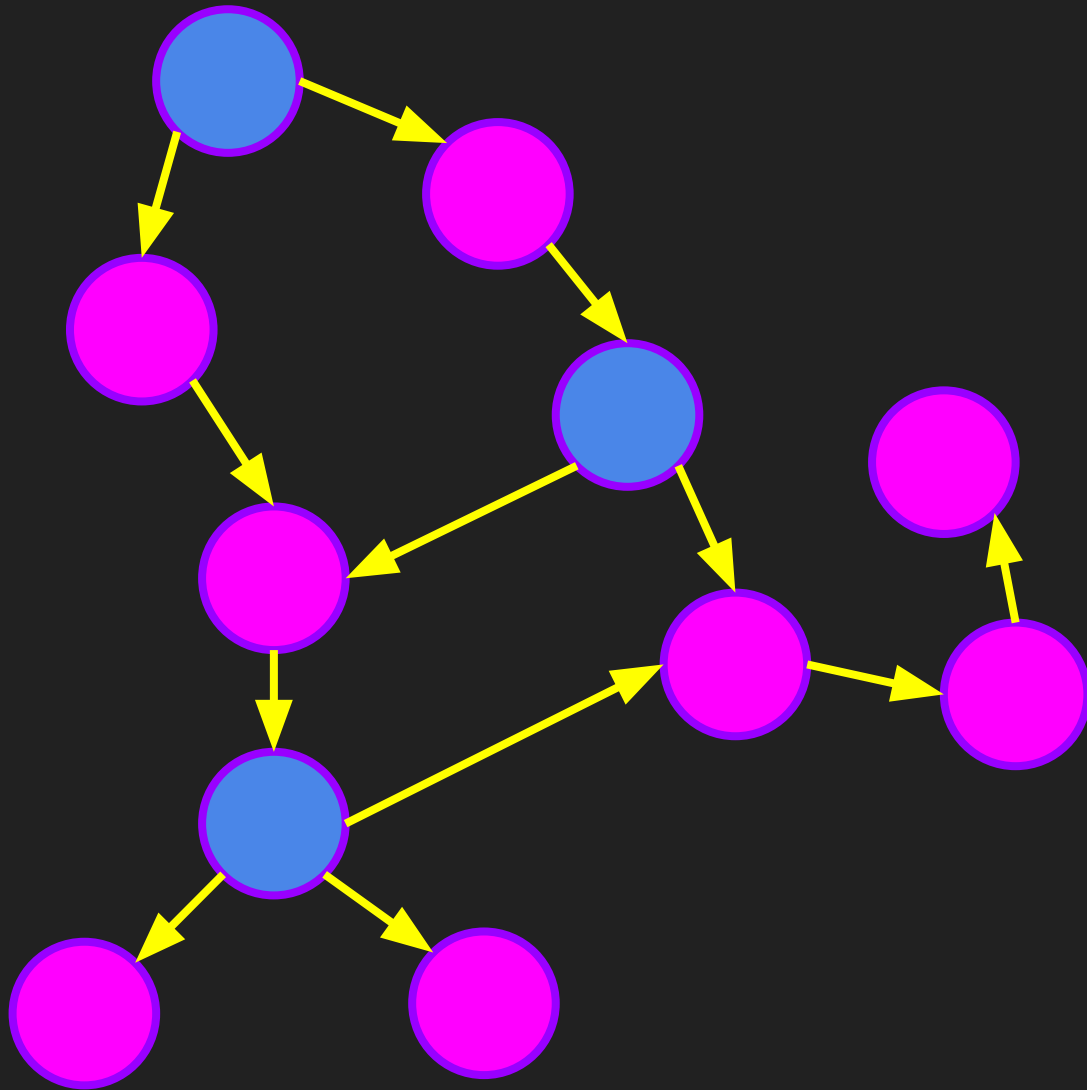
```
def abs(value):  
    if value >= 0:  
        return value  
    else:  
        return -1 * value
```

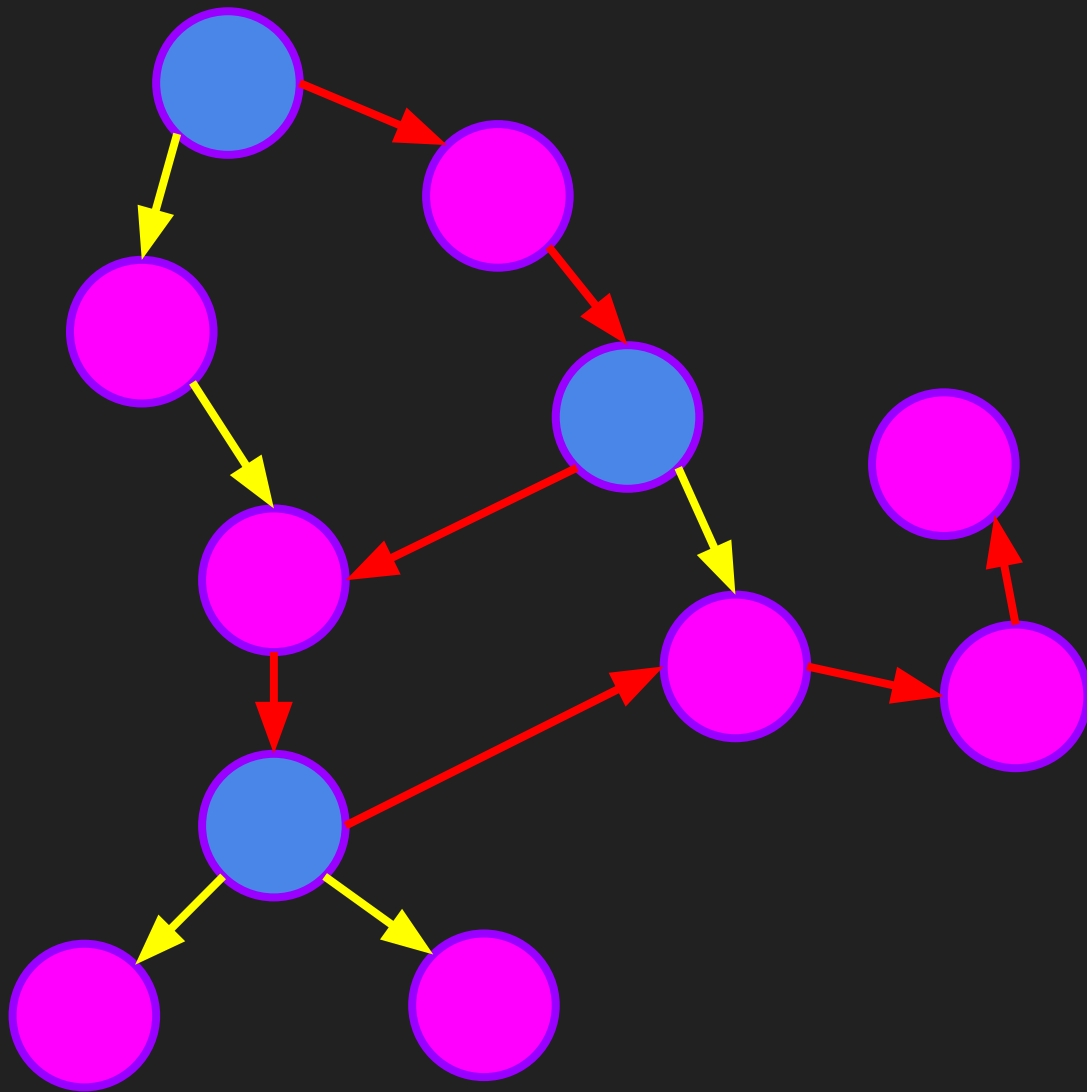
```
for value in range(10**9):  
    abs(value)
```

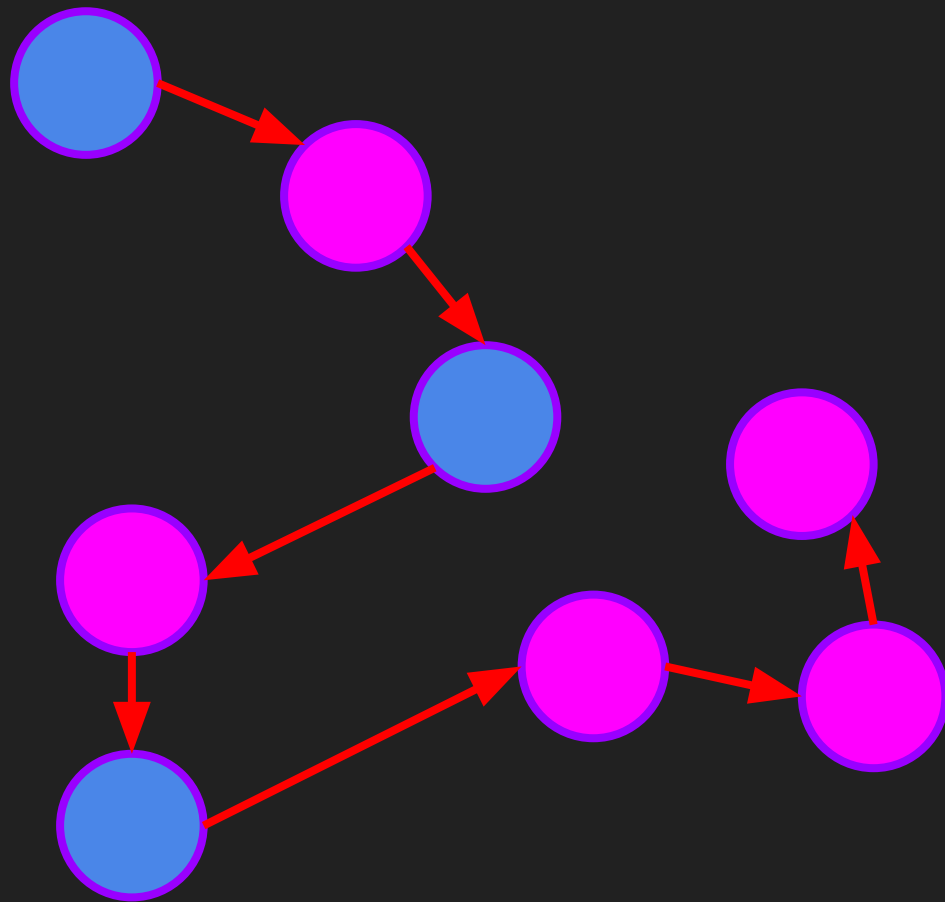


So, in general

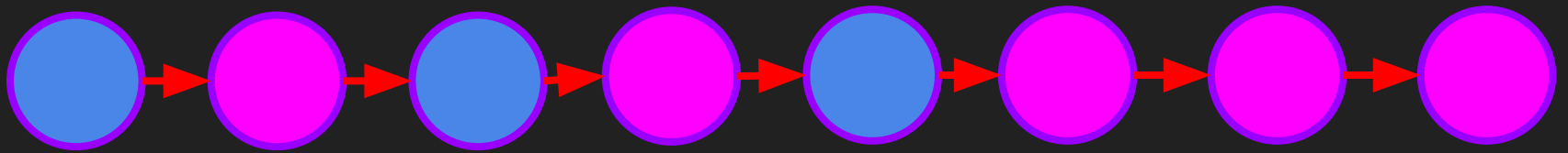












## Conclusions

- Dynamic languages are slow intrinsically.
- Performance comes with specialization.
- Availability of runtime feedback is crucial.



:3



POSITIVE TECHNOLOGIES

