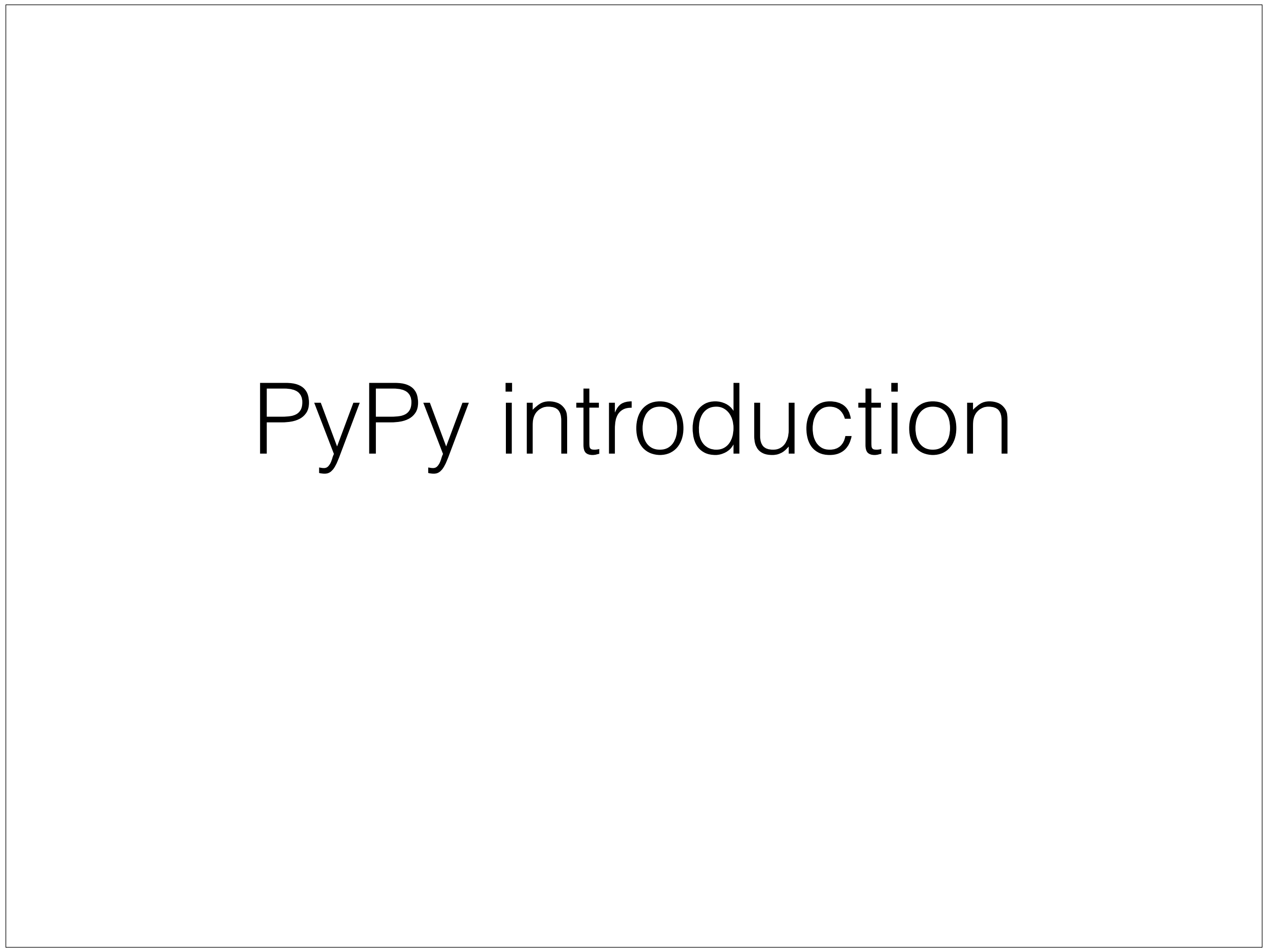


PyPy introduction

Hello everyone. My name is Maciej and I'll be doing hacking PyPy introduction here.

Format:

Short introduction - an hour or less

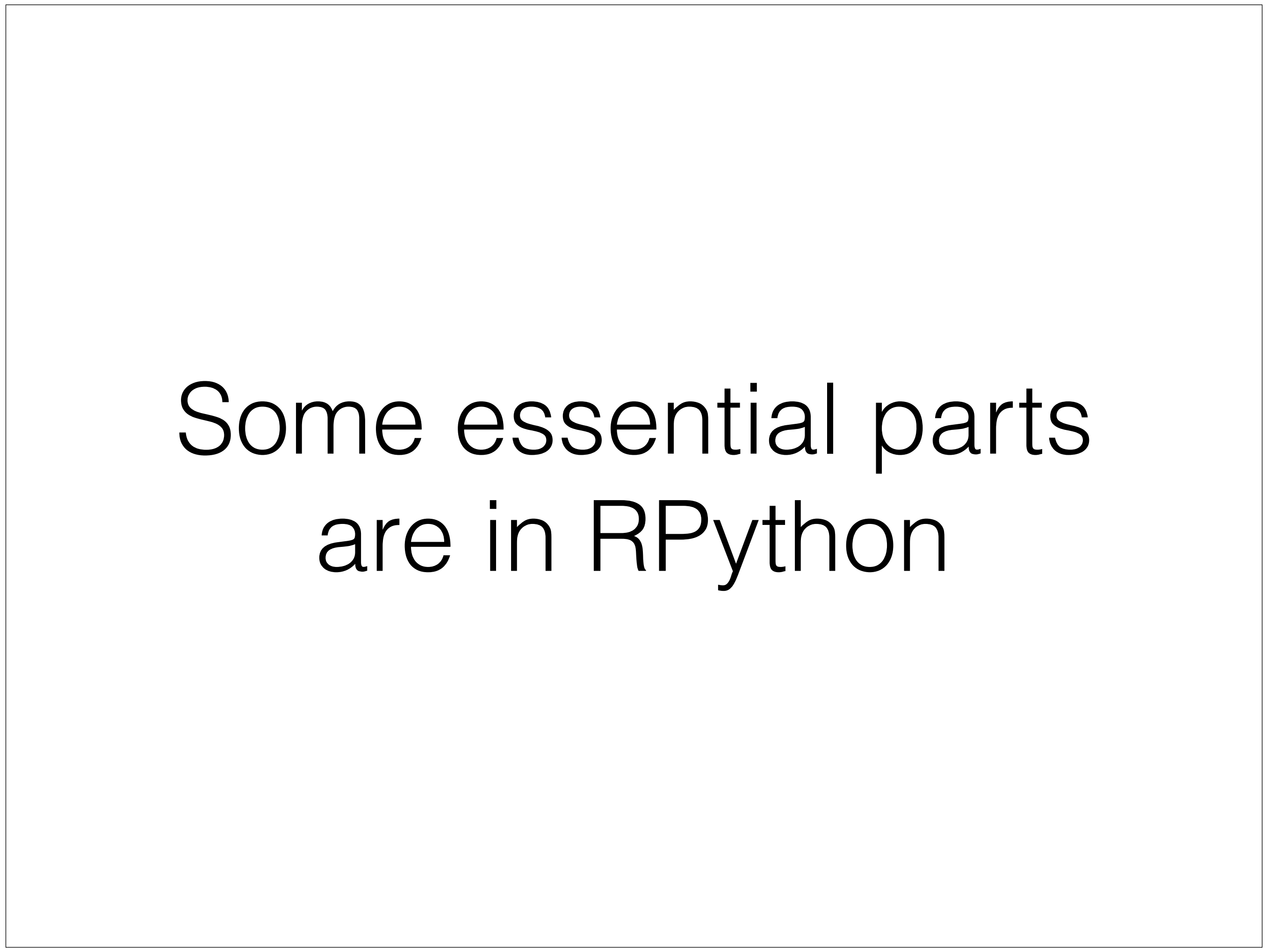
Live exercise

Four parts

I'll try to keep the formal introduction to things as short as possible, but I'll keep the projector here open so we can try various things. There will be four parts, with at most an hour introduction each, with live exercise being the main part of each part.

RPython introduction

The first part is introduction to RPython - the language in which PyPy is written. PyPy is two main parts - the Python interpreter and the RPython toolchain. They're a bit more intermingled than we would like, but RPython is a bit like C in CPython.



Some essential parts are in RPython

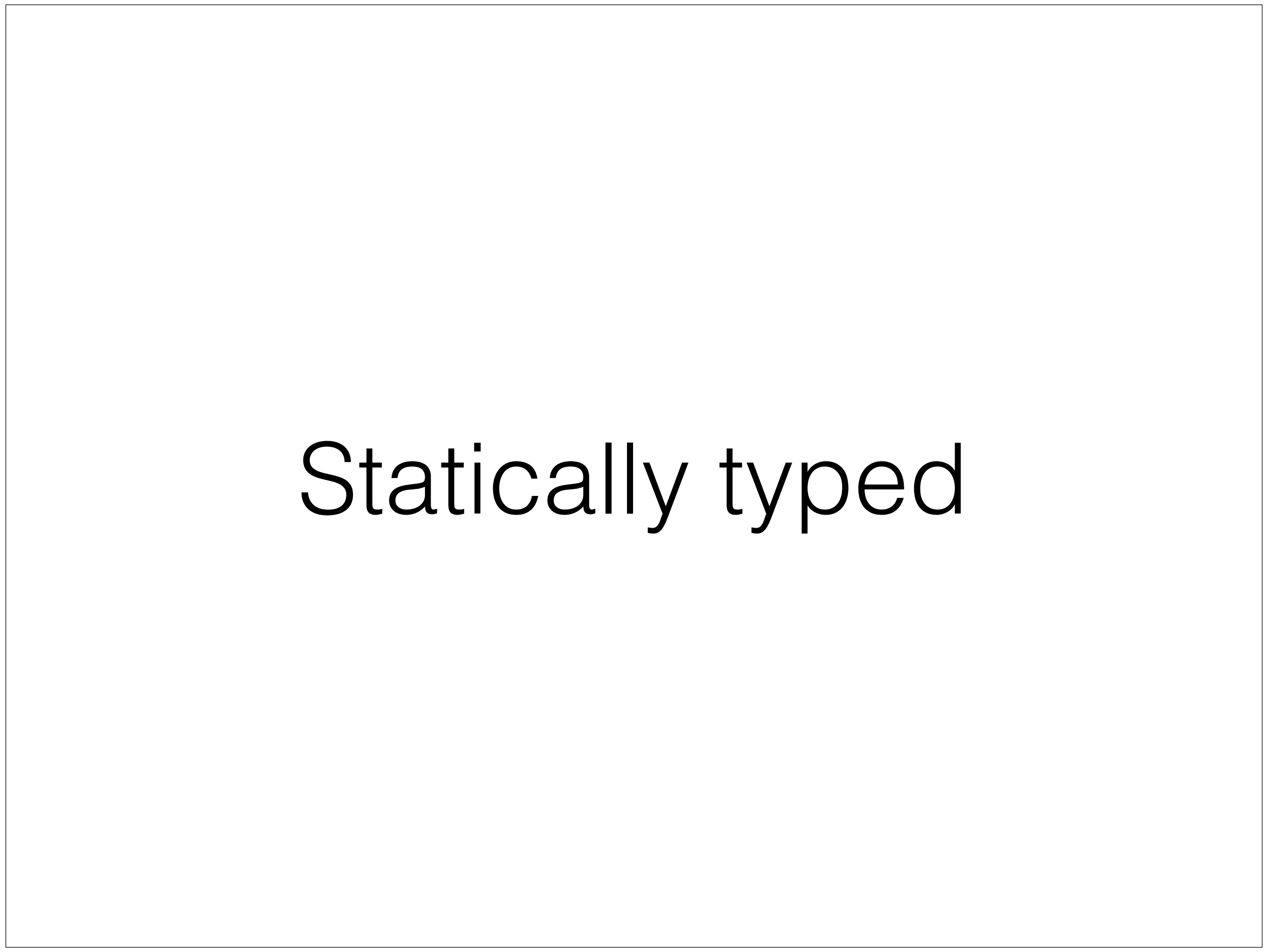
Maybe the most important part though is that some parts that are definitely very important, like the JIT and the garbage collector are written in RPython and not in PyPy part. That means that sometimes you need to cross-cut the layers and hack into the toolchain itself, but it also means certain things are possible that are not possible in other systems.

Motivation

The easiest way to find motivation is to read something like HolyJIT rationale - for implementing a language as complicated as Python we need something where we are not implementing semantics of the language 2 or 3 times, since it's too cumbersome. We also want a language where we can control macro-like features on sufficient level to be able to change details that are spread all over the place, like reference counting. In 2017 we might have picked Rust

Subset of Python

We chose a subset of Python. It might have been a terrible idea, but we're so far stuck with it and it works well. That let's us run (quickly!) tests of RPython-written interpreter without an extensive compilation process. This has proven essential due to long compile times



Statically typed

We want a language that can be compiled to C in an efficient way. We also want it analyzable, so whole program needs to be known at compile time

GCd, Analyzable, metaprogramming

We wanted high-ish level enough language that can be garbage collected. We wanted a language that can be analyzed and comes with powerful macros so we don't need to use eclipse

Code 1

Basic python works

Very basic python works - ints, floats, strings, classes, lists etc work as expected. There is global type inference, which means that variables, function calls etc need to follow the types. Lists need to be homogenous, tuples have to be of fixed size. Ints, floats, bools can't be mixed with None, neither can tuples. Classes have to follow hierarchies and share a base class. More insane rules follow.

Error messages

Global type inference create insane error messages. Let's see some simple ones.

Code 2, 3

Metaprogramming

One of the bigger differences is how you structure stuff like polymorphism. Instead of getting a function to accept anything, we generate a few functions.

Code 4, 5

rpython.readthedocs.org

There is quite a bit of documentation - be aware, RPython is pretty hard to master, but incredibly powerful to use. I'll cover more of the examples in the second part of the powers.

Writing interpreters

So the exercise before lunch is to write a simple interpreter. I believe that way we can work from the basics without the burdens of the whole pypy codebase. Jumping immediately into PyPy would cause a bit more frustration, so let's spend a day hacking a small interpreter.

github.com/fijal/intro-pypy

the interpreter spec and the test harness is there. Fork it, clone it and try to implement interpret(). I did not write the tests nor how the tests should work on purpose here