Rusty Python

Python interpreter, reimagined.

About the dude standing here

- Juhun “RangHo” Lee
- Professional procrastinator
- Commits bullshits for living

- Oh shit another hipster again

Twitter: @RangHo_777
GitHub: @RangHo

I like programming languages.

Like, a lot.

Rust is pretty cool..... Right?

- Compile-time memory safety
- Zero-cost abstraction
- Fearless concurrency
- Runtime engine not required
- FFI-able design
- Embed-able toolchain

- IT IS THE BEST COMPILED LANGUAGE EVER!!!11!!1!!!!

And Python is quite nice..... Isn't it?

- Feature-rich standard library
- Partially prototype-based OOP
- (Relatively) Easy metaprogramming
- Multi-paradigm
- Syntax is hip af
- It works on your machine as well™

- IT IS THE BEST SCRIPTING LANGUAGE EVER!!!!!11!!

Imagination time!

"자. 재밌는 상상 한 번 해 보자고." - Some random dude

- I have a Rust
  - which is hip in and of itself
- I have a Python
  - which used to be hip in and of itself

UH!

- Rust + Python! (or something)
  - which has to be hip af right?

SO INTERESTING OMFG

~~Heap~~ Hip overflow — how?

- The big question: HOW DO WE DOUBLE THE HIP?

- Two of many ways to achieve hip²
  1. Python-based: Building Rust for Python
  2. Rust-based: Building Python for Rust

Stage 1

Building Rust for Python

Python ❤️ Extension

- Python can be extended using C or C++
- Extensions - Python function with native code
- Python is slow af
  - Keras
  - TensorFlow
  - PyTorch
  - NumPy
  - SciPy
  - ...anything that requires heavy calculations

Where Python loses its strength...

- C/C++ extension requires...
  - Manual memory management and fun times free()-ing stuff
  - Manual reference counting with Py_INCREF and Py_DECREF macros
  - Saying bye bye to memory safety

“Why bother using Python, when you have C?”

Here comes a new challenger!

- Rust can fix these issues
  - Memory management? -> Leave that to the Borrow Checker™
  - Reference counting? -> Leave that to the Borrow Checker™
  - Memory safety? -> Leave that to the Borrow Checker™

- Make native Python more Python-y!

Sure, but we want the juice of it

- The most important stuff: PERFORMANCE!

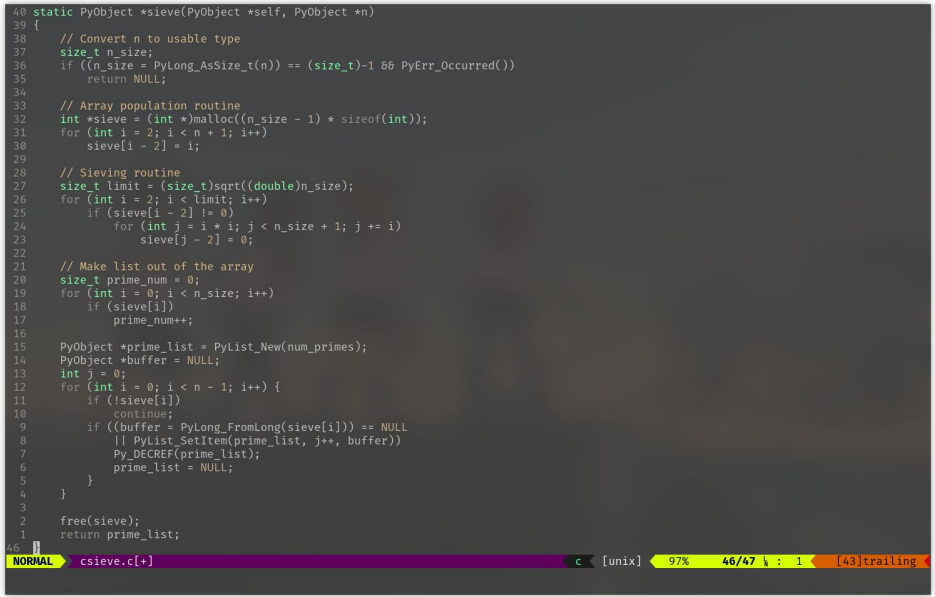
- Simple implementation of *Sieve of Eratosthenes*

Performance Battle (ver. Python)

- Only 11 lines!
- Basically looks like a pseudocode
  - (and it is basically a pseudocode)

- It takes about **25ms** to sieve out 100,000 numbers
  - i7-9700K, btw

Performance Battle (ver. C)



```c
static PyObject *sieve(PyObject *self, PyObject *n)
{
    // Convert n to usable type
    size_t n_size;
    if ((n_size = PyLong_AsSize_t(n)) == (size_t)-1 && PyErr_Occurred())
        return NULL;

    // Array population routine
    int *sieve = (int *)malloc((n_size - 1) * sizeof(int));
    for (int i = 2; i < n + 1; i++)
        sieve[i - 2] = i;

    // Sieving routine
    size_t limit = (size_t)sqrt((double)n_size);
    for (int i = 2; i < limit; i++)
        if (sieve[i - 2] != 0)
            for (int j = i * i; j < n_size + 1; j += i)
                sieve[j - 2] = 0;

    // Make list out of the array
    size_t prime_num = 0;
    for (int i = 0; i < n_size; i++)
        if (sieve[i])
            prime_num++;

    PyObject *prime_list = PyList_New(num_primes);
    PyObject *buffer = NULL;
    int j = 0;
    for (int i = 0; i < n - 1; i++) {
        if (!sieve[i])
            continue;
        if ((buffer = PyLong_FromLong(sieve[i])) == NULL
            || PyList_SetItem(prime_list, j++, buffer))
            Py_DECREF(prime_list);
            prime_list = NULL;
        }
    }

    free(sieve);
    return prime_list;
}
```

```
NORMAL  csieve.c[+]   c  [unix]  97%  46/47 ㏑ :  1  [43]trailing
```

- Relatively massive
- Some `if` statements for memory safety
  - I forgot malloc safety check as well

- It takes about **700µs** to sieve out 100,000 numbers
  - Same, i7-9700K

Performance Battle (ver. Rust + PyO3)

- Simpler than the C code
- Resembles Python more
- Requires two functions
  - Usually they are separated

- It takes about **670µs** to sieve out 100,000 numbers
  - Again, i7-9700K

Performance battle result

- Jesus Christ, Python *is* slow
- In most cases, Rust is as fast as C
- In most cases, Rust requires less memory-related code

=> Rust is enough to replace C for Python extensions!

Stage 2

Building Python for Rust

There are loads of Pythons out there

- **CPython** - The “Reference”
- **PyPy** - The Ouroboros of Infinity
- MicroPython - The Featherweight Warrior

- Jython - Python with a cup of coffee
- IronPython - Python got Microsoft’d

Why another one?

“One of the reasons is that... I wanted to learn Rust.”

- Windel Bouwman

Learning Rust by making a Python interpreter

- Currently RustPython project has 5M+ lines of code
- In the beginning, it used to be really simple
  - https://github.com/windelbouwman/rspython

- Now it is fully (kinda) functional Python 3.5 interpreter

Yeah, but *why*?

Let's pull out the C-equivalent of **p[key]**, where p is dict:

```
PyObject *PyDict_GetItem(PyObject *p, PyObject *key);
```

...and return the borrowed reference

From the dict object

...find by key

Yeah, but *why*? (Cont'd)

```
PyObject *PyDict_GetItem(PyObject *p, PyObject *key);
```

...if no match,
return NULL
(no exception)

From the dict
object

...find by key

Yeah, but *why*? (Cont'd)

Because Python is GC'd language:

```
PyObject *PyDict_GetItem(PyObject *p, PyObject *key);
```

Keeps reference counter of its own

Keeps reference counter of its own

Keeps reference counter of its own

Yeah, but *why*? (Cont'd)

- Here are some Rust features:
  - Rust ships with Borrow Checker to enforce strict borrowing rules.
  - Rust has a type called `Result<T, E>` to indicate a recoverable error.
  - Rust has a type called `Rc` to simplify reference counting.

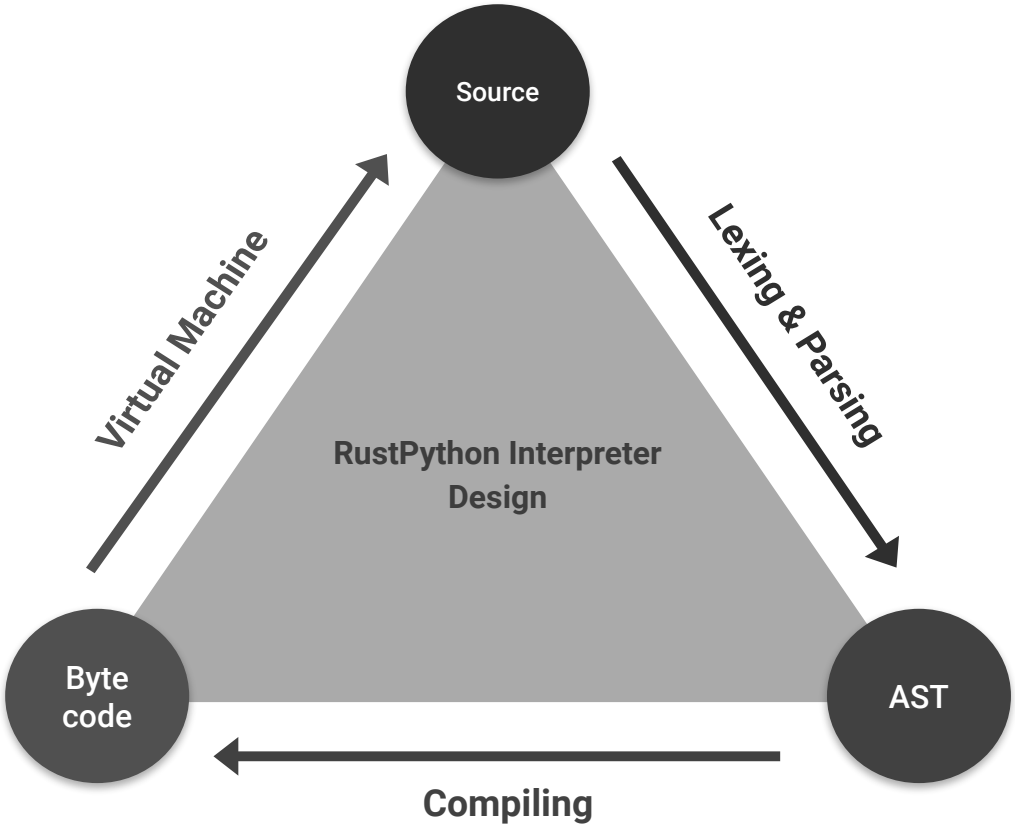
- Sounds familiar yet?

Boy-meets-girl: an all-time classic

- Python interpreter's benefit
  - Borrow Checker
  - Rust Standard Library
  - Cargo and Rust Packages
  - Memory Safety
  - WebAssembly

- The Holy Grail of Hipness

How RustPython sees Python



Lexer & Parser

- Python grammar is painful
- Not context-free language
  - i.e. Indentation
- Two ways to solve issue:
  1. Parser-lexer feedback loop
  2. “Terminalize” indentation

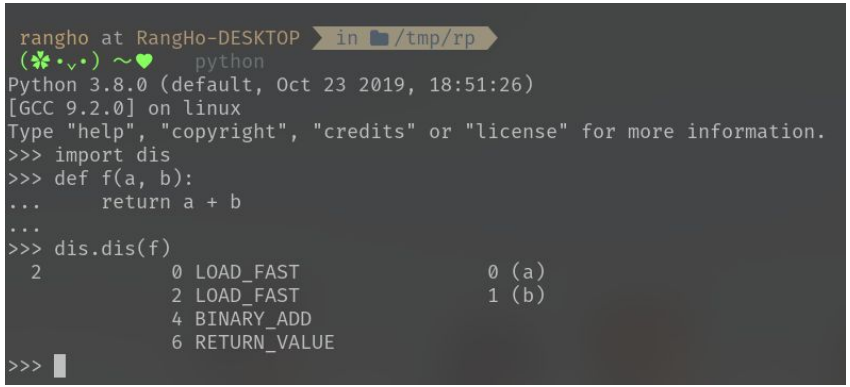
- Possible to form context-free grammar out of this spec
  - INDENT and DEDENT
  - Lexing rules are not CF
  - Parsing rules can be CF
  - LL(2) parser can be constructed

```
# It needs to be fully expanded to allow our LL(1) parser to work on it.

typedargslist: (
  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* ',' [TYPE_COMMENT] '/' [',' [ [TYPE_COMMEN
        ',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] [
        '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**'
      | '**' tfpdef [','] [TYPE_COMMENT]]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**' tfpd
  | '**' tfpdef [','] [TYPE_COMMENT]] )
|  (tfpdef ['=' test] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] [
   '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**' tfpde
  | '**' tfpdef [','] [TYPE_COMMENT]]])
  | '*' [tfpdef] (',' [TYPE_COMMENT] tfpdef ['=' test])* (TYPE_COMMENT | [',' [TYPE_COMMENT] ['**' tfpd
  | '**' tfpdef [','] [TYPE_COMMENT])
)
tfpdef: NAME [':' test]

# The following definition for varargslist is equivalent to this set of rules:
#
#     arguments = argument (',' argument )*
#     argument = vfpdef ['=' test]
#     kwargs = '**' vfpdef [',']
#     args = '*' [vfpdef]
#     kwonly_kwargs = (',' argument )* [',' [kwargs]]
#     args_kwonly_kwargs = args kwonly_kwargs | kwargs
#     poskeyword_args_kwonly_kwargs = arguments [',' [args_kwonly_kwargs]]
#     varargslist_no_posonly = poskeyword_args_kwonly_kwargs | args_kwonly_kwargs
#     varargslist = arguments ',' '/' [',' [(varargslist_no_posonly)]] | (varargslist_no_posonly)
#
# It needs to be fully expanded to allow our LL(1) parser to work on it.

varargslist: vfpdef ['=' test ](',' vfpdef ['=' test])* ',' '/' [',' [ (vfpdef ['=' test] (',' vfpdef [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']) ]] | (vfpdef ['=' test] (',' vfpdef ['=' test])* [',' [
        '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
      | '**' vfpdef [',']]]
  | '*' [vfpdef] (',' vfpdef ['=' test])* [',' ['**' vfpdef [',']]]
  | '**' vfpdef [',']
)
vfpdef: NAME

stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                     [('=' (yield_expr|testlist_star_expr))+ [TYPE_COMMENT]] )
annassign: ':' test ['=' (yield_expr|testlist_star_expr)]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
# For normal and annotated assignments, additional restrictions enforced by the interpreter
del_stmt: 'del' exprlist
pass_stmt: 'pass'
```

Compiler

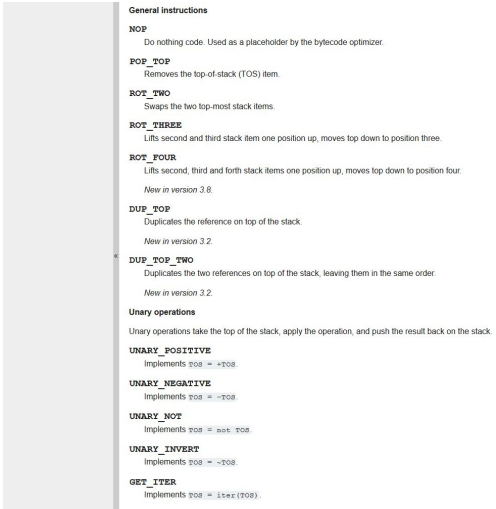
- Python virtual machine only understands bytecode
- AST to Bytecode

- Bytecode often reside in memory
- py_compile.compile(file)
  - Python2-> ./xxx.pyc
  - Python3-> ./__pycache__/xxx.pyc



Bytecode

- Python bytecode is not standardized
- Bytecode is subject to change
- Yet CPython has pretty comprehensive documents
  - https://docs.python.org/3/library/dis.html



General instructions

NOP
Do nothing code. Used as a placeholder by the bytecode optimizer.

POP_TOP
Removes the top-of-stack (TOS) item.

ROT_TWO
Swaps the two top-most stack items.

ROT_THREE
Lifts second and third stack item one position up, moves top down to position three.

ROT_FOUR
Lifts second, third and forth stack items one position up, moves top down to position four.

New in version 3.8.

DUP_TOP
Duplicates the reference on top of the stack.

New in version 3.2.

DUP_TOP_TWO
Duplicates the two references on top of the stack, leaving them in the same order.

New in version 3.2.

Unary operations

Unary operations take the top of the stack, apply the operation, and push the result back on the stack.

UNARY_POSITIVE
Implements `TOS = +TOS`.

UNARY_NEGATIVE
Implements `TOS = -TOS`.

UNARY_NOT
Implements `TOS = not TOS`.

UNARY_INVERT
Implements `TOS = ~TOS`.

GET_ITER
Implements `TOS = iter(TOS)`.

Some remarks about Bytecode

- Python can be used without .py source code
  - .pyc file has all the info
- Recovery of original source code possible
  - e.g. DDLC
- Python bytecode is not optimized well
  - “Python is about having the simplest, dumbest compiler imaginable.”
    - Guido van Rossum, our Savior

Optimization Checklist

✓ Constant folding
✓ Immutable allocation optimization
✗ Unused local variable elimination
✗ Unnecessary intermediate object elimination
✗ Loop optimization
✗ Tail recursion optimization
✗ ...and pretty much anything else

RustPython Bytecode

- RustPython does not produce bytecode file…
  - Say bye to marshal
- Separated into a crate
  - rustpython-bytecode
- Rather simple architecture
  - no INPLACE_* or other advanced stuff
- Massive dispatch loop

```
match instruction {
    bytecode::Instruction::LoadConst { ref value } => {
        let obj = vm.ctx.unwrap_constant(value);
        self.push_value(obj);
        Ok(None)
    }
    bytecode::Instruction::Import {
        ref name,
        ref symbols,
        ref level,
    } => self.import(vm, name, symbols, *level),
    bytecode::Instruction::ImportStar => self.import_star(vm),
    bytecode::Instruction::ImportFrom { ref name } => self.import_from(vm, name),
    bytecode::Instruction::LoadName {
        ref name,
        ref scope,
    } => self.load_name(vm, name, scope),
    bytecode::Instruction::StoreName {
        ref name,
        ref scope,
    } => self.store_name(vm, name, scope),
    bytecode::Instruction::DeleteName { ref name } => self.delete_name(vm, name),
    bytecode::Instruction::Subscript => self.execute_subscript(vm),
    bytecode::Instruction::StoreSubscript => self.execute_store_subscript(vm),
    bytecode::Instruction::DeleteSubscript => self.execute_delete_subscript(vm),
    bytecode::Instruction::Pop => {
        // Pop value from stack and ignore.
        self.pop_value();
        Ok(None)
    }
    bytecode::Instruction::Duplicate => {
        // Duplicate top of stack
        let value = self.pop_value();
        self.push_value(value.clone());
        self.push_value(value);
        Ok(None)
    }
    bytecode::Instruction::Rotate { amount } => self.execute_rotate(*amount),
    bytecode::Instruction::BuildString { size } => {
        let s = self
            .pop_multiple(*size)
            .into_iter()
            .map(|pyobj| objstr::get_value(&pyobj))
            .collect::<String>();
        let str_obj = vm.ctx.new_str(s);
        self.push_value(str_obj);
        Ok(None)
    }
```

Virtual Machine

- Reads bytecode, executes the darn thing

- Keeps track of runtime info
  - Frames
  - Imported modules
  - Settings
  - Context
  - All builtins

```rust
/// Top level container of a python virtual machine. In theory you could
/// create more instances of this struct and have them operate fully isolated.
pub struct VirtualMachine {
    pub builtins: PyObjectRef,
    pub sys_module: PyObjectRef,
    pub stdlib_inits: RefCell<HashMap<String, stdlib::StdlibInitFunc>>,
    pub ctx: PyContext,
    pub frames: RefCell<Vec<FrameRef>>,
    pub wasm_id: Option<String>,
    pub exceptions: RefCell<Vec<PyBaseExceptionRef>>,
    pub frozen: RefCell<HashMap<String, bytecode::FrozenModule>>,
    pub import_func: RefCell<PyObjectRef>,
    pub profile_func: RefCell<PyObjectRef>,
    pub trace_func: RefCell<PyObjectRef>,
    pub use_tracing: RefCell<bool>,
    pub signal_handlers: RefCell<[PyObjectRef; NSIG]>,
    pub settings: PySettings,
    pub recursion_limit: Cell<usize>,
    pub codec_registry: RefCell<Vec<PyObjectRef>>,
    pub initialized: bool,
}
```

Virtual Machine: the good parts

- Honestly bytecode routine is not fun at all
  - Read byte -> match instruction -> dispatch -> go back
- The real fun stuff
  - How PyObject is implemented
  - __builtin__ hellfire mess
  - ByRef and ByVal in Python
  - Rust HashMap v. Python __hash__
  - Metaprogramming

```rust
pub fn init(context: &PyContext) {
    extend_class!(context, &context.types.dict_type, {
        "__bool__" => context.new_rustfunc(PyDictRef::bool),
        "__len__" => context.new_rustfunc(PyDictRef::len),
        "__sizeof__" => context.new_rustfunc(PyDictRef::sizeof),
        "__contains__" => context.new_rustfunc(PyDictRef::contains),
        "__delitem__" => context.new_rustfunc(PyDictRef::inner_delitem),
        "__eq__" => context.new_rustfunc(PyDictRef::eq),
        "__ne__" => context.new_rustfunc(PyDictRef::ne),
        "__getitem__" => context.new_rustfunc(PyDictRef::inner_getitem),
        "__iter__" => context.new_rustfunc(PyDictRef::iter),
        (slot new) => PyDictRef::new,
        "__repr__" => context.new_rustfunc(PyDictRef::repr),
        "__setitem__" => context.new_rustfunc(PyDictRef::inner_setitem),
        "__hash__" => context.new_rustfunc(PyDictRef::hash),
        "clear" => context.new_rustfunc(PyDictRef::clear),
        "values" => context.new_rustfunc(PyDictRef::values),
        "items" => context.new_rustfunc(PyDictRef::items),
        "keys" => context.new_rustfunc(PyDictRef::keys),
        "fromkeys" => context.new_classmethod(PyDictRef::fromkeys),
        "get" => context.new_rustfunc(PyDictRef::get),
        "setdefault" => context.new_rustfunc(PyDictRef::setdefault),
        "copy" => context.new_rustfunc(PyDictRef::copy),
        "update" => context.new_rustfunc(PyDictRef::update),
        "pop" => context.new_rustfunc(PyDictRef::pop),
        "popitem" => context.new_rustfunc(PyDictRef::popitem),
    });

    PyDictKeys::extend_class(context, &context.types.dictkeys_type);
    PyDictKeyIterator::extend_class(context, &context.types.dictkeyiterator_type);
    PyDictValues::extend_class(context, &context.types.dictvalues_type);
    PyDictValueIterator::extend_class(context, &context.types.dictvalueiterator_type);
    PyDictItems::extend_class(context, &context.types.dictitems_type);
    PyDictItemIterator::extend_class(context, &context.types.dictitemiterator_type);
```

Alright, now what?

- Why RustPython instead of CPython?
  - Native WebAssembly support
  - Guaranteed memory safety
  - Clean, readable codebase
  - Lots of things to learn

- Some projects started picking up
  - pyckitup 2D game engine
  - codingworkshops.org educational website

Can it really stand against CPython?

- Native integration is not working yet
- Still lots of improvements needed
  - Some features work on 🅱️in🅱️ows only
  - It targets WASM but...
- Not very optimized
  - around x16 slower than CPython
  - Sub-optimal data structure design

```rust
/// This is an actual python object. It consists of a `typ` which is the
/// python class, and carries some rust payload optionally. This rust
/// payload can be a rust float or rust int in case of float and int objects.
pub struct PyObject<T>
where
    T: ?Sized + PyObjectPayload,
{
    pub typ: PyClassRef,
    pub dict: Option<RefCell<PyDictRef>>, // __dict__ member
    pub payload: T,
}
```

Final Thoughts

Alright, I know, let's face it

- Rust is not *the best*
  - Any *good* programmer can make manageable code
  - Compilation takes eternity
  - Honestly, wtf is borrow checking
  - "OK, here's langserver, deal with this crap"
  - Static linkage creates bloated binaries
  - Except libc, because reasons

Alright, I know, let's face it (cont'd)

- Python is not *the best*
  - Honestly the "scope-by-whitespace" is an unintelligible mess
  - lol no proper threadings
  - Why no switch-case? *WHY?*
  - async is a function but no, it's a generator, but still a function
  - Stop using Python2 already goddammit
  - Basically no optimization is made during compilation stage

Bottom line

- Rust can replace C, but cannot replace C++
  - Modern C++ has many features that help managing memories
  - Partial functional programming support is neat
  - Template-based metaprogramming is scalable enough
- Rust has potential
  - Data science
  - Compilation time is getting faster (for real)
  - Still better than golang, no?

Bottom line

- Python has its uses, but nothing more
  - Pseudocode must remain pseudocode
  - Great scripting engine to make simple CLI tools
    - **Ruby**, no one cares about that language anymore :(
    - “**Perl** is worse than Python because people wanted it worse.”
    - **JavaScript**, we don’t want another 120+MB of dependencies
    - **Shell Script** is useful, but complex logic is painful
- Stop giving Python a life support
  - Python 2 has fallen cold-dead, long live Python 3
  - Stop shoving more features into the poor thing

Still I am a
hipster-wannabe
madman.
(Objectively speaking)