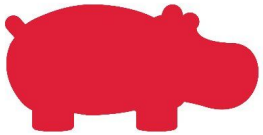
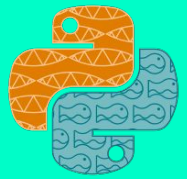


Hack The CPython

Batuhan Taskaya
@isidentical



europython
July 8-14 2019
BASEL

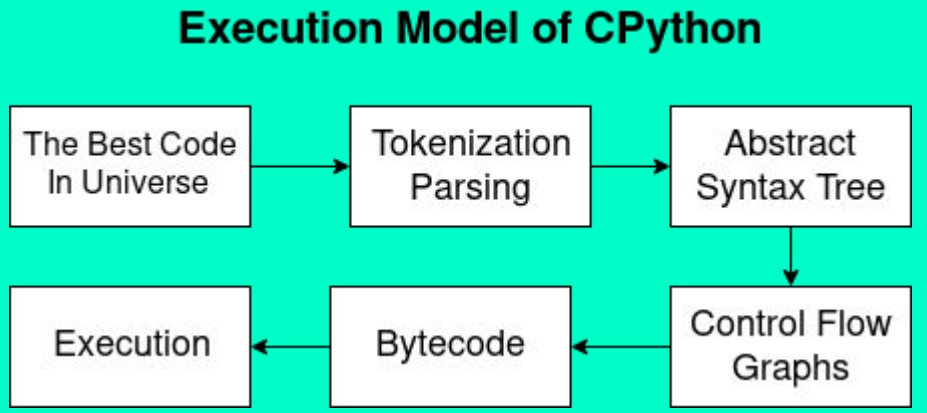
What is hacking?


Why do we hack?

Yes, we want FREEDOM!

We want to use PEP313!

Before we hack,



Learn the internals

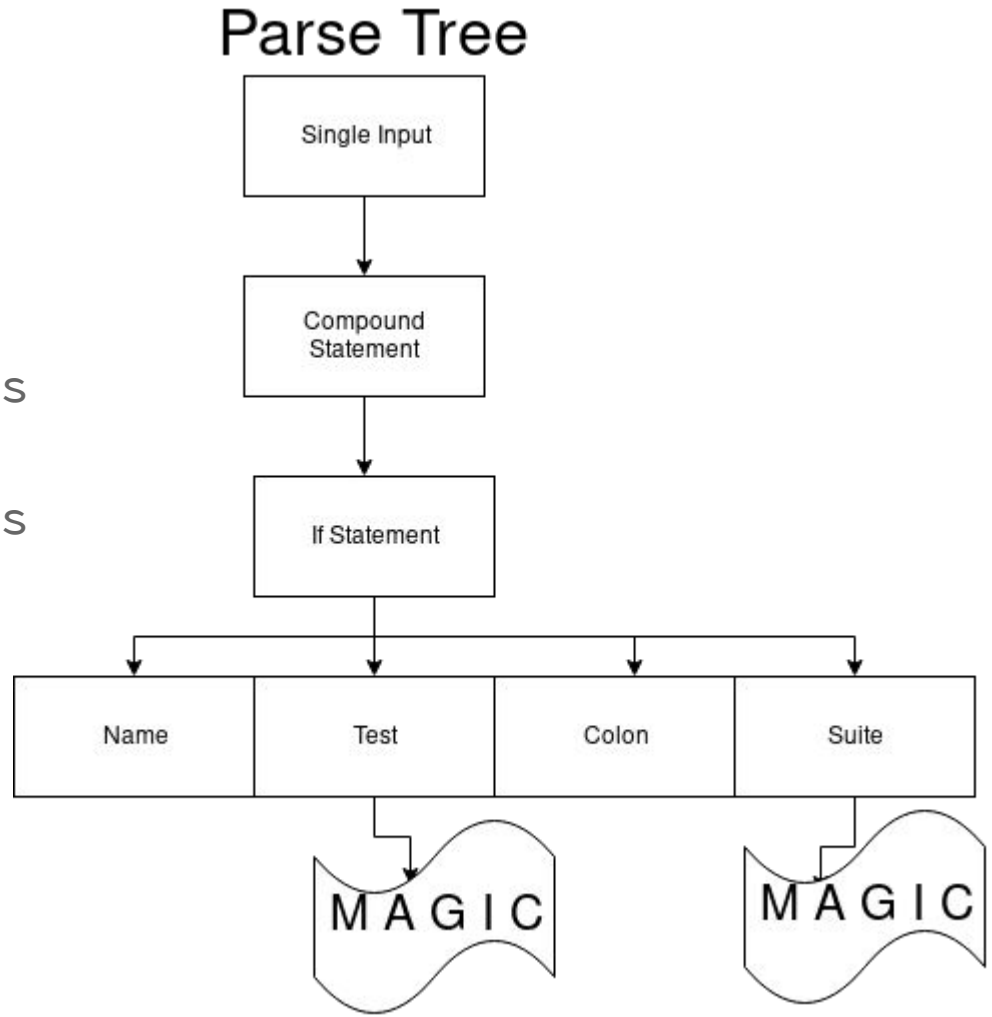
Lexing - Tokenization

- Read
- Split
- Set the first token

```
#define NEWLINE         4
#define INDENT          5
#define DEDENT          6
#define LPAR            7
#define RPAR            8
#define LSQB            9
#define RSQB            10
#define COLON           11
#define COMMA           12
```

Parsing - Parser

- Generated by PGen2
- Keeps record of structres in arcs, dfas etc.
- Keeps non-affect things (like whitespace)
- Constructs a CST



AST (where actual hack begins)

- Generated by ASDL
- A highly relational tree that constructed from CST
- Doesn’t keep any thing if it doesn’t need (like whitespace)
- Can be manipulated easily

```
class RewriteName(NodeTransformer):

    def visit_Name(self, node):

        return ast.Name(“a” +
    node.id, node.ctx)
```

Bytecode Generation

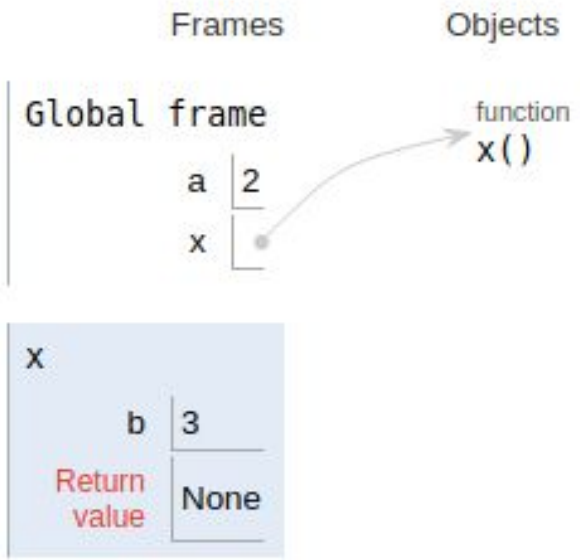
- CFG construction
- Compiling to a code object
- Peephole

```
>>> dis.dis("a.xyz(3)")

  1           0 LOAD_NAME                0 (a)
              2 LOAD_METHOD              1 (xyz)
              4 LOAD_CONST               0 (3)
              6 CALL_METHOD              1
              8 RETURN_VALUE
```

Evaluation

- A biiig for loop
- (with labeled goto's if gcc)
- Tons of structs tries to track everything
- Based on frame by frame execution atop on stacks
- Global & Local namespaces



Let's Hack

Walrus on Python 3.7

A project that allows you to use walrus operator on python 3.7 with using a new encoding

```
# coding: walrus37
name = "batuhan"
if (name := "new_name") == "batuhan":
    print("name changed as ", name)
else:
    print("no name doesnt equal to 'batuhan'")
```

The Strategy For Hacking

- Should run before the tokenization happen
- Needs a new tokenizer or modification to python’s tokenize module
- Should be tokenized with that tokenizer
- Needs an untokenizer that consumes sequence of tokens to construct source back
- Should stream that source to real tokenizer

Modifiying the Tokens

- Add a new token under `token` module (where python keep token names and ids)
- Add a new key to `tokenize.EXACT_TOKEN_TYPES` for getting token name when that token streamed
- Updating rule for tokenization (if not python will throw error tokens because it cant understand :=)

```
tokens.COLONEQUAL = 0xFF
tokens.tok_name[0xFF] = "COLONEQUAL"
tokenize.EXACT_TOKEN_TYPES[":="] =
tokens.COLONEQUAL


tokenize.PseudoToken =
tokenize.Whitespace + tokenize.group(
    r":=",
    tokenize.PseudoExtras,
    tokenize.Number,
    tokenize.Funny,
    tokenize.ContStr,
    tokenize.Name,
)
```

Modifying The Source

- A function that reads walrused source and returns the 3.7 adapted source
- Tokenizes the walrused source with new modifications
- Creates a copy of that tokens
- Uses real one for detection and the copy for modification

```
def generate_walrused_source(readline):
    source_tokens = list(tokenize(readline))
    modified_source_tokens =
source_tokens.copy()

    for index, token in
enumerate(source_tokens):
        if token.exact_type ==
tokens.COLONEQUAL:
            <code for replacing that token>

    return untokenize(modified_source_tokens)
```

Creating decode function for Encoding

- Reads source
- Decodes with the actual decoding
- Streams into `generate_walrused_source`
- Returns the clean source back

```
def decode(input, errors="strict",
encoding=None):
    if not isinstance(input, bytes):
        input, _ = encoding.encode(input,
errors)


    buffer = io.BytesIO(input)
    result =
generate_walrused_source(buffer.readline)
    return encoding.decode(result)
```

Adding a search function

- `codecs.register` takes a search function that returns the `codecs.CodecInfo` if the given name is the codec's name else returns `None`
- For using walrus37 with other encodings then utf8 allow user to specify encoding and bind that encoding into `decode` function

```
def search(name):
   if "walrus37" in name:
       encoding =
name.strip("walrus37").strip("-") or
"utf8"
       encoding = lookup(encoding)
       decoder = <partial decoder with
given encoding>

       walrus_codec = CodecInfo(...)
       return walrus_codec
```

Implementing Rejected PEPs

A project that allows you to use features of rejected peps

```
from pepallow.allow import Allow

with Allow(313):
    assert IV == 4
```

The Strategy For Hacking

- Should run when imported
- Should be effective only with-in the Allow(<pep num>) space
- If the syntax is used outside the scope should raise the proper error (for an example if I used without the pep313 scope it should raise NameError)

Implementing Peps (Example PEP313)

- Should go through all names (a, x, obtainer, I, IV, test)
- If the name is a valid roman literal
- Get the value of that literal and then replace it with proper number

```
class PEP313(HandledTransformer):
    def visit_Name(self, node):
        number = roman(node.id)
        if number:
            return ast.Num(number)
        return node
```

Scoping

- Should go through all with statements
- Find with's name and check if name is `Allow`
- Get args of `Allow` (PEP Number)
- Dispatch the elements of that with to proper PEP handler

```
class PEPTransformer(Transformer):
    def visit_With(self, node):
        if <name check>:
            pep = <get first arg>
            new_node = <get node>

            copyloc(new_node, node)
            fix_missing(new_node)
        return node
```

Runtime

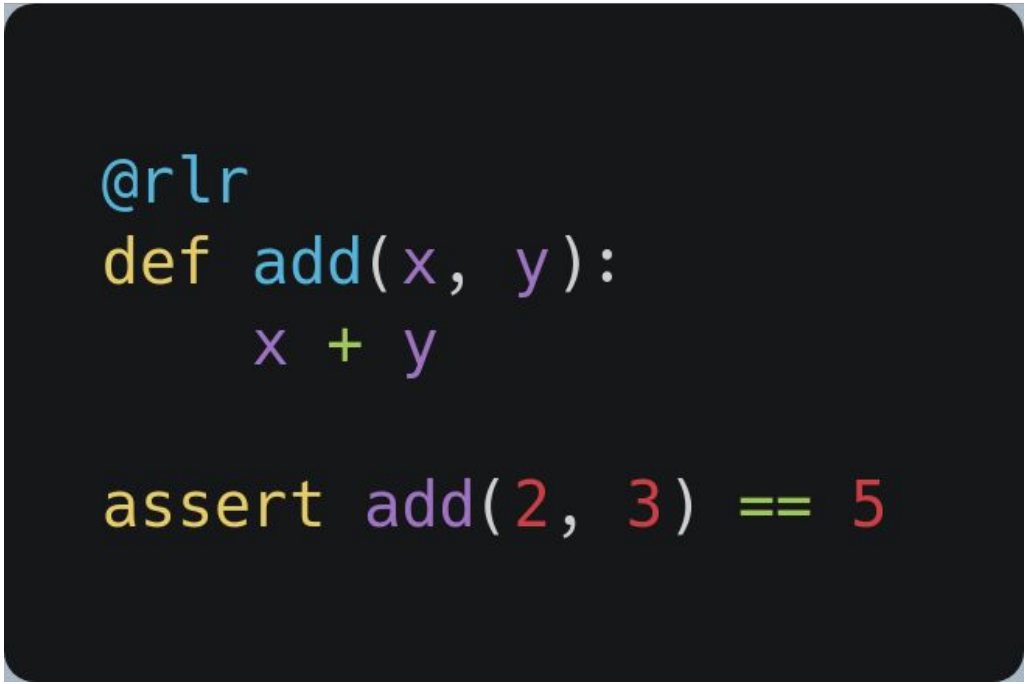
- Run when imported
- Get the source code of the file it is imported
- Transform that source into AST
- Dispatch AST to Scoping Handler
- Get back the AST
- Compile AST to bytecode
- Run the bytecode

```
def allow():
    main = __import__("__main__")
    tf = PEPTransformer()
    f = main.__file__
    main_ast = ast.parse(<open>)
    main_ast = tf.visit(main_ast)
    fix_missing_locations(main_ast)
    bc= compile(main_ast, f, "exec")
    exec(bc, main.__dict__)


allow()
```

Rusty Return

Implicitly return the last expression (like rust)



The Strategy For Hacking

- Should run when function decorated
- Should be return the last expression
- Should support infinite branching

Transforming AST (1)

- Visit the function definition
- Remove the @rlr from the decorators list (for preventing infinite recursion)

```
class RLR(ast.NodeTransformer):
    def visit_FunctionDef(self, fn):
        self._adjust(fn)
        ds = filter(lambda d: d.id
!= "rlr", fn.decorator_list)
        fn.decorator_list = list(ds)
        return fn
```

Transforming AST (2)

- If the last node is an expression should replace last node with `ast.Return`
- Call itself back while the last statement is `ast.If`

```
   def _adjust(self, container: ast.AST, items: str =
"body") -> None:
       items = getattr(container, items) if items is
not None else container
       last_stmt = items[-1]


       if isinstance(last_stmt, ast.Expr):

items.append(ast.Return(value=items.pop().value))
       elif isinstance(last_stmt, ast.If):
           self._adjust(last_stmt)
           if len(last_stmt.orelse) > 0:
               self._adjust(last_stmt.orelse, None)
       else:
           return None
```

Poophole Optimizer

An extra bytecode optimizer for python

```
@Poophole.optimize(elem_local_vars = True)
def some_func():
  a = 5
  b = 3
  return b + 6
```

The Strategy For Hacking

- Should run when function decorated
- Should go through bytecode and only apply the optimizations the user specified
- Should re-set the optimized bytecode

Optimize Function

- A decorator that takes a set of options
- Creates a `dis.Bytecode` from function
- Call optimizers by checking the given options
- Re-set the bytecode
- Return the function

```
@classmethod
def optimize(cls, el):
    def wrapper(func):
        buffer = Bytecode(func)
        if el:
            buffer = elem(buffer)
        reset_bytecode(func, buffer)
        return func
    return wrapper
```

Optimizers 1 (Example Elem Local Vars)

- Go over bytecode buffer
- Keep a dict of variables their value is a constant (like a int or string)
- Find unused variables

```
    def _elem_locals(self, buffer,
function):
        constant_loaded = False
        stack, symbols = [], {}
        for instr in buffer:
          <create a list of symbols>

        unuseds = [(unused[0],
unused[1]) for unused in
symbols.values() if unused[2] == 0]
```

Optimizers 2 (Example Elem Local Vars)

- Remove unused parts from bytecode
- Remove unnecessary constants
- Remove unnecessary symbols

```
unused_consts, unused_varnames =
[], []
offset = 0
for value, unused in unuseds:
    <replace code>

<remove consts>
<remove names>
```

Catlizor v1-extended

Assign hooks to python functions without mutating functions

```
@Hook.pre
class PreLoggingHook(Hook):
    methods = ['add_task']
    callbacks = [lambda result: print(result.args, result.kwargs)]
```

The Strategy For Hacking

- Should not mutate the function itself
- Should notify before a function call
- Should notify during a function call

  (result = notify(call(x)))

- Should notify after a function call

Hooking

- Write onto the memory address of default function call function
- Written by @dutc

```
#pragma pack(push, 1)
    jumper = {
      .push = 0x50,
      .mov  = {0x48, 0xb8},
      .jmp = {0xff, 0xe0}
    };
#pragma pack(pop)
```

```
lpyhook(_PyFunction_FastCallKeywords, &hookify_PyFunction_FastCallKeywords);
```

Modifiying

- Adding hooks for pre, on call and post actions
- Calling catlizor interface when these hooks activated

```
PyObject *
hookify_PyFunction_FastCallKeywords
(PyObject *func, PyObject * const
*stack, Py_ssize_t nargs, PyObject
*kwnames)
{
    <code>
    <code>
}
```

Thanks

@isidentical