# Semantics for CPython Virtual Machine

July 6, 2020

## Variables & Scopes

Python has 4 kinds of variables.

The scope of Python is divided by function boundaries. No other Python language constructs create/destroy a scope.

Any name in a scope may denote 4 kinds of variables.

1. **local-only variable**
   A bound variable, but used only in the scope
2. **cell variable**
   A bound variable, the its definition scope has nested functions referencing this variable. This variable is in form of *cell* data structure, in case of future mutations
3. **free variable**
   A cell variable comes to a nested function referencing it, and becomes a free variable. It's also a *cell*
4. **global variable**
   Defined in the top-level of a Python module

These variables other than global variables are stored in variable *slots* created per function. When a function was created, an array of variables slots holding *local-only*, *cell* and *free* variables will be created as well. The index of a variable in the slots is statically decided by the bytecode compiler.

Particularly, **global variables** are stored in a special hash table, which could be accessed by a Python expression **globals**(), and is separately maintained per Python module. Users can modify global variables outside its defined module, by modifying the special hash table.

## CoreCPY: A Minimal Language for CPython Bytecode Instructions

A core part of Python VM instructions are given as follow

$$
\begin{array}{rcl}
\langle\text{var}\rangle & ::= & \textit{localvar} \mid \textit{cellvar} \mid \textit{freevar} \mid \textit{globalvar} \\
\langle\text{instr}\rangle & ::= & \textit{LOAD}\ \langle\text{var}\rangle \\
 & & \mid\ \textit{STORE}\ \langle\text{var}\rangle \\
 & & \mid\ \textit{JUMP\_IF\_TRUE int} \\
 & & \mid\ \textit{JUMP int} \\
 & & \mid\ \textit{CALL int} \\
 & & \mid\ \textit{POP int} \\
 & & \mid\ \textit{ROT int}
\end{array}
$$

Except for exception handling, the whole Python VM instruction set can be described in a set of primitive Python objects and this language.

We call this language Core CPython(**CoreCPY**), which is also we'd discuss in the following contents.

Extending this to support exception handling is not difficult, but it turns out to be annoying to do this, as the notations will loss conciseness and intuitions.

*A translation from CPython 3.9 to CoreCPY is WIP.*

To introduce the semantics of **CoreCPY**, we have to introduce Python function calls, a few built-in Python objects, in another word, *special objects*.

## Python Function Calls

Python function calls are some **opaque** operations, it can be anything happened in the computer, but through an **opaque** mechanism. Only one of its property needs attention, that if we have another Python function object defined with an instruction suite, when we call it via the **CALL** instruction, it should at least execute that instruction suite with suitable arguments.

## Special Objects

Special objects are given as follow

$$\begin{aligned}\textbf{special\_object} = {} & \textbf{int} \\ \mid\ & \textbf{str} \\ \mid\ & \textbf{bool} \\ \mid\ & \textbf{True} \\ \mid\ & \textbf{False} \\ \mid\ & \textbf{None} \\ \mid\ & \textbf{type} \\ \mid\ & \textbf{getattr} \\ \mid\ & \textbf{setattr} \\ \mid\ & \textbf{globals} \\ \mid\ & \textbf{locals} \\ \mid\ & \textbf{make\_cell} \\ \mid\ & \textbf{make\_frozen}\end{aligned} \tag{1}$$

Beyond above special objects, all other Python objects can be implemented in Python itself or extensions written in C language. Only special objects concern Python's semantics.

All special objects other than **make_cell** and **make_frozen** are primitives in Python, and all special objects should support following functionalities, note that $\cdot(\cdot)$ means a Python function call

1. An **int** object should represent an integer with arbitrary precision
2. An object **s** typed **str** can support random accessing with any **int** index **i**, in form of **getattr**(**str**, "__getitem__")(s, i)
3. The **bool** object should satisfy **type**(True) **is bool** and **type**(False) **is bool**
4. The **type** object is a function to access an object's type object
5. **getattr**(s, attr) looks up object **s**'s attribute named **attr**; **getattr**(s, attr, default) looks up object **s**'s attribute named **attr**, if not found, returns **default**
6. **setattr**(s, attr, value) tries to set object **s**'s attribute named **attr** with **value**;
7. make_cell() makes a cell object, which is expected to be modified and read by **setattr**(cell, "cell_contents", val) and **getattr**(cell, "cell_contents")
8. make_cell(o) tells Python object **o** to reject any further modification to its attributes.
9. WIP

## Operational Semantics for CoreCPY

By introducing

- a value stack **S**,
- a store $\sigma$, and
- an instruction suite **I**,

we get capable of clarifying the semantics of **CoreCPY**.

$$\frac{n \in \textbf{localvar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n)} \quad \frac{n \in \textbf{cellvar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n).cell\_contents}$$

$$\frac{n \in \textbf{freevar}}{\sigma \vdash_{r-val} \Rightarrow \sigma(n).cell\_contents} \quad \frac{n \in \textbf{globalvar}}{\sigma \vdash_{r-val} \Rightarrow globals()[n]}$$

$$\frac{n \in \textbf{localvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.v} \quad \frac{n \in \textbf{cellvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.\sigma(n).cell\_contents \leftarrow v}$$

$$\frac{n \in \textbf{freevar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.\sigma(n).cell\_contents \leftarrow v} \quad \frac{n \in \textbf{globalvar}}{\sigma \vdash_{l-var} \Rightarrow \lambda v.globals()[n] \leftarrow v}$$

$$\frac{S = a_1 :: a_2 :: \cdots :: a_n :: S^{'}}{S, \sigma \vdash_{instr} \textbf{POP}\ n \Rightarrow (S', \sigma)}$$

$$\frac{\sigma \vdash_{r-val} n \Rightarrow obj}{S, \sigma \vdash_{instr} \textbf{LOAD}\ n \Rightarrow (obj :: S, \sigma)}$$

$$\frac{\sigma \vdash_{l-val} n \Rightarrow L \quad S = obj :: S^{'}}{S, \sigma \vdash_{instr} \textbf{STORE}\ n \Rightarrow (S', \sigma[n := L(obj)])}$$

$$\frac{S = a_1 :: a_2 :: \cdots :: a_n :: S^{'}}{S, \sigma \vdash_{instr} \textbf{ROT}\ n \Rightarrow (a_n :: a_1 :: a_2 :: \cdots :: a_{n-1} :: S', \sigma)}$$

$$\frac{I(i) = \textbf{JUMP}\ \mathit{off} \quad I, S, \sigma \vdash_{jump} \mathit{off} => (return, \sigma^{'})}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{S = f :: a_1 :: a_2 :: \cdots :: a_n :: S^{'} \quad f(a_1, a_2, \cdots, a_n) = e}{S, \sigma \vdash_{instr} \textbf{CALL}\ n \Rightarrow (e :: S', \sigma)}$$

$$\frac{\begin{array}{l} I(i) = \textbf{JUMP\_IF\_TRUE}\ \mathit{off} \quad S = obj :: S^{'} \\ type(obj).\_\_bool\_\_(obj) = True \quad I, S^{'}, \sigma \vdash_{jump} \mathit{off} => (return, \sigma^{'}) \end{array}}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{\begin{array}{l} I(i) = \textbf{JUMP\_IF\_TRUE}\ \mathit{off} \quad S = obj :: S^{'} \\ type(obj).\_\_bool\_\_(obj) = False \quad I, S^{'}, \sigma \vdash_{jump} i + 1 => (return, \sigma^{'}) \end{array}}{I, S, \sigma \vdash_{jump} i \Rightarrow (return, \sigma')}$$

$$\frac{\begin{array}{l} I(i) = instr \quad instr \neq \textbf{JUMP}\ \mathit{off}_1 \wedge instr \neq \textbf{JUMP\_IF\_TRUE}\ \mathit{off}_2 \\ S, \sigma \vdash_{instr} \Rightarrow (S^{'}, \sigma^{'}) \quad I, S^{'}, \sigma^{'} \vdash_{jump} i + 1 \Rightarrow (return, \sigma'') \end{array}}{I, S, \sigma \vdash_{jump} \Rightarrow (return, \sigma'')}$$