CHAPTER 1

Formal Semantics

1.1 How does Python work?

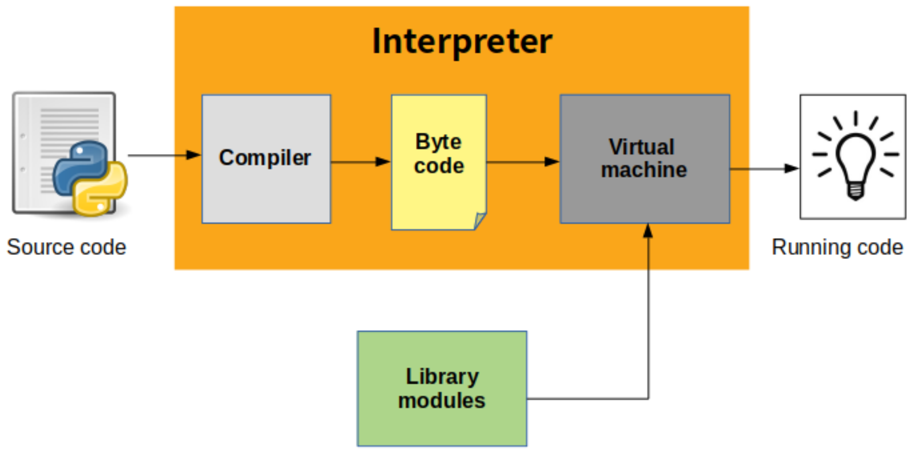

Fig. 1.1: Python Code Execution Mechanism

Coming up with formal semantics rules requires us to understand how Python code gets executed. In this section, we explain the process of executing Python code according to **cpython**, which is the default implementation of Python (see Figure 1.1).

Python code execution process runs in mainly two stages, translation and execution. The translation consists in transforming Python source code into an object that the Virtual Machine (interpreter) can execute. This object is called **code object**. Code objects have the necessary information that a Virtual Machine needs in order to execute the code, for example the **Bytecode**, a **list of constants used in the code**, a **list of the locally defined names**, and a **list of non-local names used in the code**.

Once the code object is generated, a Virtual Machine must be created for execution. The Virtual Machine contains the code object to be executed and a **call stack**. The call stack is used to manage **frames**. Frames are created and deleted while executing the Bytecode, and they are used to provide context for Bytecode blocks. For example, each function is translated into a separate Bytecode block and it has its own frame. Each frame contains its own **data stack** that is used to manipulate data while executing the

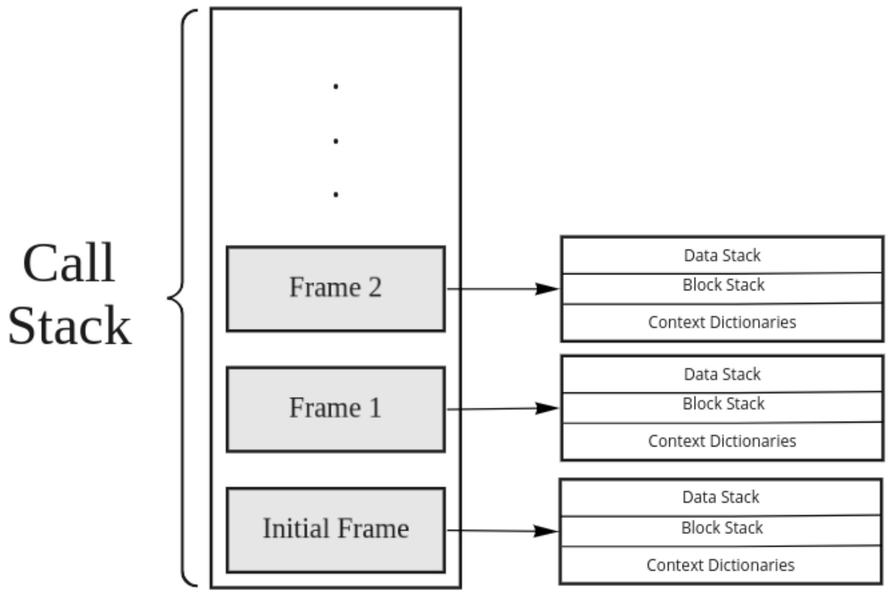

Fig. 1.2: Python Virtual Machine

Bytecode instructions, it also has a **block stack** that is used to handle some specific exceptions and loops cases, and **context dictionaries** that provide the frame with the global and local names it has access to (see Figure 1.2). In other words, a frame acts as the program state of a given Bytecode block.

Once the Virtual Machine is created it spawns the initial frame to run the Bytecode of the code object in it. The Virtual Machine starts the execution process which involves spawning new frames, executing them and then returning the results to caller frames. Once the initial frame returns a result, the execution process ends and the interpreter returns the result.

1.2 Establishing the semantics

Even though Python doesn't have formal semantics, it has a default implementation (**cpython**). We focus on cpython version 3.9, which has been released on October 5th, 2020. Through analyzing the source code of the Bytecode evaluation unit in cpython and its documentation, and by experimenting with cpython on small, hand-crafted programs, we understood the behavior of Python and formalized it using **small-step semantics** rules. Our formal semantics rules consist of two levels (i.e. two types of rules), **Frame Instructions**, and **Bytecode Instructions**.

As alluded to previously, execution of Bytecode happens in two different levels. The first level is the level of frames, where Frame Instructions rules describe how frames get managed and how they interact with each other (i.e. data flow between frames). The second level is Bytecode Instructions, where formal semantics rules describe how a frame can take a step after executing a specific Bytecode instruction in it (i.e. execution within the frames).

1.3 Notation

In this section we introduce the notation that will be used in the rules.

The call stack state. The call stack has two states, which are represented by the following notation.:

$\mathsf{K} \triangleright \mathsf{f}$	:	Evaluation State
$\mathsf{K} \triangleleft \mathsf{ret(v)}$	:	Return State

During the Evaluation State we evaluate the top frame `f`, and when it becomes `ret(v)` we switch to the Return State. Moreover, an empty stack is represented as ϵ.

Frame syntax. Next we show the notation used for representing a frame:

$\langle \varphi, \Gamma, i, \beta, \Delta \rangle$	:	Frame
$\varphi \triangleq \langle \Sigma_{\mathsf{g}}, \Sigma_{\mathsf{l}}, \Sigma_{\mathsf{l}_+} \rangle$	:	Contexts $\triangleq$ $\langle$ Global Names, local names, local+ $\rangle$
$\Gamma \triangleq \langle \Pi, \Sigma_{\mathsf{c}}, \Sigma_{\mathsf{v}}, \Sigma_{\mathsf{n}} \rangle$	:	Code Object $\triangleq$ $\langle$ Bytecode, constant, varnames, names $\rangle$
i	:	Program Counter
$\beta \triangleq \langle \mathsf{t}, \mathsf{l}, \mathsf{h} \rangle$	:	Block Object $\triangleq$ $\langle$ type, level, handler $\rangle$
Δ	:	Data Stack

Stepping. Frames level Semantics use $\longmapsto$ to indicate stepping, while Bytecode instructions level use $\xmapsto{\Gamma.\Pi[i]}$ to indicate stepping.

Special Terms. The rules also make use of some special terms that we explain below:

$\texttt{TypeOf}(obj)$	:	Object destructor $\triangleq$ Takes an object as an input and returns $\texttt{C}$(v)
$\texttt{C}(v)$	:	Type constructor, where C $\in T$ or $\texttt{C}(v) \triangleq$ `USERDEF` (i.e., user defined class)
T	:	Set of builtin types $\triangleq \langle \texttt{INT}, \texttt{BOOL}, \texttt{STRING}, \texttt{LIST}, \texttt{TUPLE}, \texttt{DICT}, \texttt{FUNCTION}, \texttt{NONE} \rangle$
$\texttt{CreateObj}(t)$	:	Object constructor $\triangleq$ Takes in $\texttt{C}$(v) and construct an object

Equality In our rules equality is commutative.

1.4 Frame Instructions

To demonstrate how these rules work, we explain how rule 1.4 works as an example. In rule 1.4, the call stack is in a Return State where the frame $\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle$ just finished being evaluated and it is returning to its caller stack. Rule 1.4 states that as long as the top element in the returned stack is not a new spawned frame, we do the following: first, we pop the top element in the data stack of the returned frame and we push it to the data stack of the caller frame and we delete the returned stack, lastly we switch the call stack state to evaluation stack and we start to evaluate the caller stack.

$\boxed{\text{Frame Rules}}$

$$\frac{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle}{K \ \triangleright \ \langle \varphi, \Gamma, i, \beta, \Delta \rangle \longmapsto K \ \triangleright \ \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle} \tag{1.1}$$

$$\frac{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle)}{K \ \triangleright \ \langle \varphi, \Gamma, i, \beta, \Delta \rangle \longmapsto K \ \triangleleft \ \mathsf{ret}(\langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle)} \tag{1.2}$$

$$\frac{}{K \ \triangleleft \ \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle :: \Delta \rangle) \longmapsto K; \langle \varphi, \Gamma, i+1, \beta, \Delta \rangle \ \triangleright \ \langle \varphi_n, \Gamma_n, i_n, \beta_n, \Delta_n \rangle} \tag{1.3}$$

$$\frac{v \neq \texttt{FRAMEOBJECT}(fo)}{K; \langle \varphi_p, \Gamma_p, i_p, \beta_p, \Delta_p \rangle \ \triangleleft \ \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle) \longmapsto K \ \triangleright \ \langle \varphi_p, \Gamma_p, i_p, \beta_p, v :: \Delta_p \rangle} \tag{1.4}$$

1.5 Bytecode Instructions

In this section we show the rules of how the currently supported Bytecode instructions get executed within a frame. As the number of rules for this section is big, we only include an example of rules in each class. The rest of the rules follow the same structure as the mentioned rules so they are not included.

To demonstrate how these rules work, we explain how rule 1.37 works as an example. Rule 1.37 states that if the current Bytecode instruction that the program counter `i` pointing to is `BUILD_LIST(n)`, and the data stack contains at least n elements, then we pop these n elements and we create a list that contains them $\texttt{LIST}([v_1, ..., v_n])$. Following that, we push the created list back to the data stack, and we increment the program counter by 1.

$$\frac{}{\langle \varphi, \Gamma, i, \beta, \texttt{ERR}(s) :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]} \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, \texttt{ERR}(s) :: \Delta \rangle)} \tag{1.5}$$

$\boxed{\text{General Instructions}}$

$$\frac{}{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{NOP}} \langle \varphi, \Gamma, i+1, \beta, \Delta \rangle} \tag{1.6}$$

$$\frac{}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{POP\_TOP}} \langle \varphi, \Gamma, i+1, \beta, \Delta \rangle} \tag{1.7}$$

$$\frac{}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{DUP\_TOP}} \langle \varphi, \Gamma, i+1, \beta, v :: v :: \Delta \rangle} \tag{1.8}$$

$$\boxed{\text{Unary Instructions}}$$

$$\frac{\texttt{not v} = \texttt{C}(v') \quad v'' = \texttt{CreateObj}(\texttt{C'}(v'))}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{UNARY\_NOT}} \langle \varphi, \Gamma, i+1, \beta, v'' :: \Delta \rangle} \tag{1.9}$$

$$\frac{v = \texttt{INT}(\bar{v}) \quad \bar{v} = 0}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.10}$$

$$\frac{v = \texttt{INT}(\bar{v}) \quad \bar{v} \neq 0}{\texttt{not } v = \texttt{BOOL}(false)} \tag{1.11}$$

$$\frac{v = \texttt{STRING}(\bar{s}) \quad \bar{s} = \texttt{""}}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.12}$$

$$\frac{v = \texttt{STRING}(\bar{s}) \quad \bar{s} \neq \texttt{""}}{\texttt{not } v = \texttt{BOOL}(false)} \tag{1.13}$$

$$\frac{v = \texttt{BOOL}(false)}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.14}$$

$$\frac{v = \texttt{BOOL}(true)}{\texttt{not } v = \texttt{BOOL}(false)} \tag{1.15}$$

$$\frac{v = \texttt{LIST}(\bar{l}) \quad \bar{l} = \texttt{[]}}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.16}$$

$$\frac{v = \texttt{LIST}(\bar{l}) \quad \bar{l} \neq \texttt{[]}}{\texttt{not } v = \texttt{BOOL}(false)} \tag{1.17}$$

$$\frac{v = \texttt{DICT}(\bar{d}) \quad \bar{d} = \{\}}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.18}$$

$$\frac{v = \texttt{DICT}(\bar{d}) \quad \bar{d} \neq \{\}}{\texttt{not } v = \texttt{BOOL}(false)} \tag{1.19}$$

$$\frac{v = \texttt{TUPLE}(\bar{l}) \quad \bar{l} = \texttt{[]}}{\texttt{not } v = \texttt{BOOL}(true)} \tag{1.20}$$

$$\frac{v = \texttt{TUPLE}(\bar{l}) \quad \bar{l} \neq \texttt{[]}}{\texttt{not}\ v = \texttt{BOOL}(false)} \tag{1.21}$$

$$\frac{v = \texttt{FUNCTION}(\bar{f})}{\texttt{not}\ v = \texttt{BOOL}(false)} \tag{1.22}$$

$$\frac{v = \texttt{NONE}}{\texttt{not}\ v = \texttt{BOOL}(true)} \tag{1.23}$$

$$\frac{v = \texttt{USERDEF}}{\texttt{not}\ v = \texttt{BOOL}(false)} \tag{1.24}$$

$\boxed{\text{Binary Instructions}}$

$$\frac{\texttt{TypeOf}(v1) = \texttt{C}(\_) \quad C \in T \quad \texttt{TypeOf}(v2) = \texttt{C'}(\_) \quad C' \in T \quad v2/v1 = \texttt{C''}(v) \quad v' = \texttt{CreateObj}(\texttt{C''}(v))}{\langle \varphi, \Gamma, i, \beta, v1 :: v2 :: \Delta \rangle \xmapsto{\Gamma.\Pi[i] = \texttt{BINARY\_FLOOR\_DIVIDE}} \langle \varphi, \Gamma, i+1, \beta, v' :: \Delta \rangle} \tag{1.25}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{v1} \neq 0 \quad v' = \bar{v2}/\bar{v1}}{v2/v1 = \texttt{INT}(v')} \tag{1.26}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{v1} = 0}{v2/v1 = \texttt{ERR}(s)} \tag{1.27}$$

$$\frac{v1 = \texttt{BOOL}(\bar{b1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{b1} = true \quad \bar{b2} = true \quad v' = 1}{v2/v1 = \texttt{INT}(1)} \tag{1.28}$$

$$\frac{v1 = \texttt{BOOL}(\bar{b1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{b1} = true \quad \bar{b2} = false \quad v' = 0}{v2/v1 = \texttt{INT}(0)} \tag{1.29}$$

$$\frac{v1 = \texttt{BOOL}(\bar{b1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{b1} = false}{v2/v1 = \texttt{ERR}(s)} \tag{1.30}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{v1} = 0}{v2/v1 = \texttt{ERR}(s)} \tag{1.31}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{v1} \neq 0 \quad \bar{b2} = false \quad v' = 0}{v2/v1 = \texttt{INT}(0)} \tag{1.32}$$

$$\frac{v1 = \texttt{INT}(\bar{v1}) \quad v2 = \texttt{BOOL}(\bar{b2}) \quad \bar{v1} \neq 0 \quad \bar{b2} = true \quad v' = 1/\bar{v1}}{v2/v1 = \texttt{INT}(v')} \tag{1.33}$$

$$\frac{v1 = \texttt{BOOL}(\bar{b1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{b1} = false}{v2/v1 = \texttt{ERR}(s)} \tag{1.34}$$

$$\frac{v1 = \texttt{BOOL}(\bar{b1}) \quad v2 = \texttt{INT}(\bar{v2}) \quad \bar{b1} = true \quad v' = v2}{v2/v1 = \texttt{INT}(v')} \tag{1.35}$$

$$\frac{v1 \neq \texttt{INT}(\bar{v1}) \quad v1 \neq \texttt{BOOL}(\bar{b1}) \quad v2 \neq \texttt{INT}(\bar{v2}) \quad v2 \neq \texttt{BOOL}(\bar{b2})}{v2/v1 = \texttt{ERR}(s)} \tag{1.36}$$

$\boxed{\text{Miscellaneous opcodes}}$

$$\frac{v' = \Gamma.\Sigma_n\texttt{[idx]} \qquad \varphi' = \varphi \texttt{ with } \varphi.\Sigma_l\texttt{[}v'\texttt{]} = v}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{STORE\_NAME(idx)}} \langle \varphi', \Gamma, i+1, \beta, \Delta \rangle} \tag{1.37}$$

$$\frac{v = \Gamma.\Sigma_c\texttt{[idx]}}{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{LOAD\_CONST(idx)}} \langle \varphi, \Gamma, i+1, \beta, v :: \Delta \rangle} \tag{1.38}$$

$$\frac{l = \texttt{LIST}([v_1, ..., v_n])}{\langle \varphi, \Gamma, i, \beta, v_1 :: ... :: v_n :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{BUILD\_LIST(n)}} \langle \varphi, \Gamma, i+1, \beta, l :: \Delta \rangle} \tag{1.39}$$

$$\frac{i' = i + (c/2) + 1}{\langle \varphi, \Gamma, i, \beta, \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{JUMP\_FORWARD(c)}} \langle \varphi, \Gamma, i', \beta, \Delta \rangle} \tag{1.40}$$

$$\frac{}{\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{RETURN\_VALUE}} \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, v :: \Delta \rangle)} \tag{1.41}$$

$$\frac{v = \texttt{CODEOBJECT}(\bar{co}) \quad f = \langle\langle \varphi.\Sigma_g, \{\}, \texttt{[}v_n\texttt{, ..., }v_1\texttt{]} \rangle, \bar{co}, 0, \texttt{[]}, \texttt{[]} \rangle}{\langle \varphi, \Gamma, i, \beta, v_1 :: ... :: v_n :: v :: \Delta \rangle \xmapsto{\Gamma.\Pi[i]=\texttt{CALL\_FUNCTION(n)}} \mathsf{ret}(\langle \varphi, \Gamma, i, \beta, f :: \Delta \rangle)} \tag{1.42}$$