

Specification of Python §4 Environment Stepper—2026 edition

Martin Henz

National University of Singapore
School of Computing

October 11, 2026

Introduction

The *environment stepper* (“e-stepper”) shows how a Python §3 or Python §4 program is evaluated by the *environment model* of SICPy, Chapter 3. Like the substitution stepper (see the specification of the Python §2 stepper), it rewrites the program itself, one small step at a time, so that the shape of a process stays visible. In contrast to the substitution stepper, names are *not* replaced by their values at the time a binding is made. Instead, every part of the program that is under evaluation has an *environment* E in which it looks up names, and assignments change the frames of the environments. This is what makes assignment, local state and mutable data structures explainable: the substitution model cannot express them.

The environments and the data structures that the program refers to are drawn next to the program as in the CSE machine visualization. The rewriting rules below and the CSE machine describe the same computations: they agree on the result of every program, and on the kind of error, if any.

Configurations

A step shows a *configuration*: the *program term* P , and the *frames* and the *heap* that the program refers to.

Program term. The program term P is the remaining program: a sequence of statements. In it, expressions are rewritten to *values*:

- primitive values: integers, floats, complex numbers, strings, **True**, **False** and **None**, shown as in the substitution stepper;
- *references* $\#k$ to objects in the heap: function objects and lists. A reference to a named function object is shown by that name; other references are shown as a badge with the number of the object in the program and in the drawing;
- built-in functions, shown by their name.

A function body (or lambda body) under evaluation is wrapped as an *environment block* $\llbracket E', b \rrbracket$: the body b is evaluated in the environment E' . The stepper shows it as the body, surrounded by a bracket that carries the label of E' (for example $E3$) in the colour of the frame E' in the drawing.

Frames and heap. Besides primitive values, the program can refer to frames, and to function objects and lists, which together form the *heap*. We do not describe them as a mathematical structure; they are what the environment diagram next to the program shows. Frames and heap objects *refer to each other*:

- A *frame* binds names to values, or to $\perp$ (“unassigned”), and refers to its *parent* frame. The *global frame* G has no parent. We write $E(x)$ for the value that E binds to x , and $E = \{x_1 \mapsto v_1, \dots\}$ for a frame with these bindings.
- A *function object* $Fn(n, params, b, E)$ has an optional name n , parameters, a body b (with its **global** and **nonlocal** declarations), and refers to the frame E in which the function was created.

- A list $List[v_0, \dots, v_{n-1}]$ refers to its elements. A pair is a list of two elements, as in the CSE machine.

A value in a binding or a list that is an object is a *reference* $\#k$: it is the object, not a copy of it. Two references with the same number refer to the same object, and a change of the object is seen through every reference to it. This is the difference to the substitution model, in which every use of a name gets its own copy of the value.

Frames and heap can contain cycles. As in the substitution stepper, where a name that occurs in its own value refers cyclically to that value, frames and heap objects may refer to each other in a circle. The most important cycle is the one of every function defined in a frame: the frame binds the name of the function to the function object, and the function object refers to the frame in which it was created. A list can contain a reference to itself after `set_tail`. The rules never copy or unfold such a cycle; they only create objects and change what a binding or an element refers to. A step can change the frames and the heap in only three ways, which the rules state after the word *changing*: `new  $\#k = o$` creates a new object o (or a new frame); $E(x) := v$ makes the binding of x in the frame E refer to v ; and $\#k[i] := v$ makes the i -th element of the list $\#k$ refer to v .

The builtins environment. The parent of G is the *builtins environment* B , the root of every chain of frames. It binds the names of the built-in functions (for example `print`, `abs`, `pair`, `head`) to built-in functions, and the names of the library functions that are written in Python (for example `map` and `filter`) to function objects whose frame is B itself. It is not drawn. An assignment never reaches B : as in Python, assigning to a name that only B binds, for example `print = 0`, creates a binding in G that *shadows* the one in B from then on (see **Resolution**). Before the assignment, `print` is the built-in function. A built-in function is shown by its name, and the application of a library function is a single step.

Current environment. The *current environment* E of a redex is the environment of the innermost environment block that contains it, or G if there is none. It is not written in the program, because the environment of a subexpression is the environment of its block. The rules below are written as

$$E \vdash e \Rightarrow e' \quad \text{changing: changes}$$

which reads: “in environment E , the expression e reduces to e' , making the *changes* to the frames and the heap”. If a rule does not change them, the changes are left out, and we write $E \vdash e \Rightarrow e'$. A rule for a sub-expression (such as **Block-reduce**) makes the changes of its premise.

The global frame. G is empty at the start, and the program is evaluated in G . G is the only frame that grows while the program runs: a global name is bound when its first assignment, function definition, `for` target or import is executed, and not earlier. This ensures that reading a global name before its first assignment is a `NameError`, and that a name that B binds, such as `print` or `map`, refers to the one in B until the program binds its own in G : `print(1); print = 0` first prints and then rebinds `print`. The frame of a function call, in contrast, never grows: it is created with all parameters and with all names that the function assigns to already present, the latter unassigned (see **Function-application** below).

Inside a function, a name that is declared `global` or `nonlocal` is *designated* by the function object, which remembers these declarations. The frames of its calls do not bind such names; instead, lookup and assignment of a designated name go directly to G (`global`) or to the nearest enclosing function frame that binds it (`nonlocal`). This is the only way in which a function accesses G other than by the ordinary search through its parent frames; see **Resolution** below.

Garbage. A frame or object is *garbage* if it cannot be reached from the program term P (through environment blocks and references) or from G (and B). Garbage is not removed, but drawn in grey, so that the effect of returning from a call is visible: the frame of a call stays alive if a returned function object refers to it, and becomes garbage otherwise. The stepper offers to remove the garbage from the drawing (“Clear dead frames”).

Reduction

The reducer $\Rightarrow$ and its transitive closure $\Rightarrow^*$ are used as in the specification of the Python §2 stepper. A *contraction* is a single step; the order of evaluation is the same: expressions are evaluated from left to right, inner expressions before the expressions that contain them. We only give the rules that are different from the substitution stepper, namely those that involve names, functions and data. The rules for the primitive operators, for **and**, **or**, conditional expressions, built-in functions on primitive values, and for **if**, **pass** and **breakpoint()** are the same, with the current environment E added to every premise.

Names

Lookup: A name is replaced by the value that it is bound to, in the frame that the name *resolves* to from E (see **Resolution** below). The search ends in the builtins environment B , which binds the built-in and library functions. Every name in the program is replaced in a step of its own, from left to right, and the stepper highlights the binding that was read. (Names that B binds, which no frame in the drawing binds, are looked up as part of the step that uses them.)

$$\frac{x \text{ resolves to } E' \text{ from } E \quad E'(x) = v \neq \perp}{E \vdash x \Rightarrow v}$$

If $E'(x) = \perp$, the reduction is stuck with `UnboundLocalError` if E' is E itself, and with `NameError` (“cannot access free variable”) if E' is an enclosing frame, as in the CSE machine. If no frame binds x and x is not a built-in function, it is a `NameError`; for a global name that has not been assigned yet, this is the case, because G has no binding for it yet.

Functions

Lambda: A lambda expression is replaced by a reference to a new function object that remembers the current environment.

$$\frac{k \text{ is fresh}}{E \vdash \text{lambda } x_1, \dots, x_n: e \Rightarrow \#k \quad \text{changing: new } \#k = Fn(\varepsilon, x_1 \dots x_n, e, E)}$$

Function-definition: A function definition is removed from the program. A new function object that remembers the current environment is created, and its name is bound to it in the frame that the name resolves to.

$$\frac{k \text{ is fresh} \quad f \text{ resolves to } E' \text{ from } E \text{ (growing } G \text{ if needed)}}{E \vdash \text{def } f(x_1, \dots, x_n): b \text{ statement} \dots \Rightarrow \text{statement} \dots \quad \text{changing: new } \#k = Fn(f, x_1 \dots x_n, b, E), \quad E'(f) := \#k}$$

Function-application: The application of a reference to a function object, to as many arguments as it has parameters, all of which are values, is replaced by the body of the function in an environment block. The block’s environment is a new frame whose parent is the environment *in which the function was created* (and not the environment of the caller). The frame binds the parameters to the arguments (a rest parameter ***r** to a new list of the remaining arguments), and every other name that the body assigns to as unassigned. Names that the body declares **global** or **nonlocal** are not bound in the new frame. A **global** declaration holds for the whole body, wherever it stands (in an **if** or a loop, say), so these declarations are taken out of the body and placed, each once, at its front. **nonlocal** declarations are left out of the body: the frames that bind the names are in the environment already. If the number of arguments does not match, the reduction is stuck with a `TypeError`, as in the CSE machine.

$$\frac{\#k \text{ is a function object } Fn(n, x_1 \dots x_m, b, E') \quad E'' \text{ is fresh with parent } E'}{E \vdash \#k(v_1, \dots, v_m) \Rightarrow \llbracket E'', b \rrbracket \quad \text{changing: new } E'' = \{x_i \mapsto v_i\} \cup \{y \mapsto \perp \mid y \in \text{assigned}(b)\}}$$

Block-return: An environment block whose first statement is a **return** reduces to the returned expression, evaluated in E : the block disappears, the statements after the **return** are dropped, and the frame of the block stays (it may be garbage). A **return** without an expression returns `None`.

$$\frac{}{\llbracket E'', \text{return } e \text{ statement} \dots \rrbracket \Rightarrow e}$$

Block-value: An environment block whose body is a lambda body that has been reduced to a value v reduces to v . An environment block whose statements are all finished reduces to **None**.

$$\frac{\llbracket E'', v \rrbracket \Rightarrow v}{\llbracket E'', \varepsilon \rrbracket \Rightarrow \mathbf{None}}$$

Tail-call: When the first statement of an environment block is a **return** of the application of a function object to values (a tail call), the application is replaced by the callee's block as in **Function-application**, but the *caller's block disappears*: the callee's block takes its place, instead of being nested in it. The caller's frame is no longer part of the program; it stays in the frames, as garbage unless something refers to it (a function object created in it, for example). The same holds for a lambda body that is such an application, and for a **return** inside a loop.

$$\frac{\#k \text{ is a function object } Fn(n, x_1 \dots x_m, b, E') \quad E_3 \text{ is fresh with parent } E'}{E \vdash \llbracket E'', \mathbf{return} \#k(v_1, \dots, v_m) \text{ statement} \dots \rrbracket \Rightarrow \llbracket E_3, b \rrbracket} \\ \text{changing: new } E_3 = \{x_i \mapsto v_i\} \cup \{y \mapsto \perp \mid y \in \text{assigned}(b)\}$$

Block-reduce: The first statement of an environment block is reduced in the *frame of the block*, not in E .

$$\frac{E'' \vdash s \Rightarrow s' \quad \text{changing: changes}}{E \vdash \llbracket E'', s \text{ statement} \dots \rrbracket \Rightarrow \llbracket E'', s' \text{ statement} \dots \rrbracket \quad \text{changing: changes}}$$

A sequence of statements in an environment block is reduced statement by statement as a program, with environment E'' . A consumed statement disappears. Calls of library functions (such as **map**) are applied in a single step.

Assignment

Assignment: The right-hand expression of an assignment is evaluated. When it is a value, the statement is removed from the program, and the name is rebound to the value in the frame that the name resolves to (see **Resolution**); if that is G and x is not bound yet, G grows. If the frame was created before, this is a change of the frame that the substitution model cannot express.

$$\frac{x \text{ resolves to } E' \text{ from } E \text{ (growing } G \text{ if needed)}}{E \vdash x = v \text{ statement} \dots \Rightarrow \text{statement} \dots \quad \text{changing: } E'(x) := v}$$

Global-and-nonlocal: The declarations **global** and **nonlocal** are not statements that run: they have no step of their own, and they have been taken into account when the frame of the function was created. A **global** declaration *stays in the program*, at the front of the body of each call (see **Function-application**). The frame of the call cannot show what it decides: the name may not be bound in G yet, in which case an assignment to it makes G grow, and an enclosing function may bind the same name, which that assignment leaves alone. A **nonlocal** declaration is *not* in the body of a call: the frame of the call has no binding for the name, and the enclosing frame that binds it is in the environment already.

Resolution

Python looks up a name in four scopes, in the order **LEGB**: *Local, Enclosing, Global, Built-in*. In the e-stepper they are frames of the chain that starts at the current environment E :

- **Local:** E , the frame of the current call;
- **Enclosing:** the parent frames of E that are frames of calls, from the innermost outwards (the frames in which the enclosing functions were called);
- **Global:** G ;
- **Built-in:** B .

The frame that a name x *resolves to* from E , for both lookup and assignment, is the first frame in this order that binds x (to a value or to $\perp$). Two declarations change the order for the function that makes the call:

- if the function designates x as **global**, the search skips L and E and starts with G;
- if the function designates x as **nonlocal**, the search skips L and considers only E.

No search is needed to know that a parameter, or a name that the function assigns to, resolves to the local frame: that frame binds it from the moment it is created, so the search stops at L and never reaches an outer binding of the same name. In contrast, a name that no function frame binds is searched through E and G, and then B; it is found in G only if it has been assigned by now. If not even B binds x , lookup is a `NameError`.

Assignment resolves x in the same way, with one difference: it never goes as far as B. An assigned name is local, so an assignment in a function body is to L, unless the function designates the name as **global** (then it is to G) or **nonlocal** (then it is to E). At the top level, L is G. If G does not bind x yet, it *grows* by the new binding, even if B binds x : this is why `print = 0` creates a binding in G that shadows the one in B . No frame other than G ever gains a binding, and B never changes. In particular, a function that declares x **global** when G does not bind x yet makes G grow, even if an enclosing function binds an x of its own: that binding is not changed, because the search for a **global** name skips L and E.

```
p = print
print = 0
p(print)
print(3)
```

In the first line, `print` is looked up in B and `p` is bound in G to the built-in function. The second line creates the binding of `print` in G (G grows) and B is unchanged. In the third line, `p` is the built-in function and `print` is 0 (found in G , before B): the program prints 0. In the fourth line, the function is 0, and the reduction is stuck with `TypeError: 'int' object is not callable`.

Lists and pairs

List-literal: A list expression whose elements are all values is replaced by a reference to a new list.

$$\frac{k \text{ is fresh}}{E \vdash [v_0, \dots, v_{n-1}] \Rightarrow \#k \quad \text{changing: new } \#k = \text{List}[v_0, \dots, v_{n-1}]}$$

List-access: The access of an element of a list is replaced by the element. A negative index counts from the end. An index that is out of range causes an `IndexError`, and an index that is not an integer, a `TypeError`; the reduction is then stuck. Strings can also be accessed; the result is a string of one character.

$$\frac{\#k \text{ is a list } \text{List}[v_0, \dots, v_{n-1}] \quad -n \leq i < n}{\#k[i] \Rightarrow v_{i \bmod n}}$$

List-assignment: The assignment to an element of a list is removed from the program, and the element is changed in the heap.

$$\frac{-n \leq i < n}{E \vdash \#k[i] = v \text{ statement} \dots \Rightarrow \text{statement} \dots \quad \text{changing: } \#k[i \bmod n] := v}$$

Pair-construction: The application of `pair`, `llist` (and, in Python §3, `stream`) to values is replaced by a reference to the outermost new list; all the cells are created in the same step.

$$\frac{k_1, \dots, k_n \text{ are fresh}}{E \vdash \text{pair}(v_1, v_2) \Rightarrow \#k_1 \quad \text{changing: new } \#k_1 = \text{List}[v_1, v_2]}$$

Pair-mutation: The application of `set_head` or `set_tail` to a reference to a pair and a value is replaced by `None`, and the first or second element of the pair is changed in the heap.

$$\frac{\#k \text{ is a pair } List[v_0, v_1]}{E \vdash \text{set_head}(\#k, v) \Rightarrow \text{None} \quad \text{changing: } \#k[0] := v}$$

The other built-in functions of the linked list and stream libraries, and the functions that read lists such as `head`, `tail` and `is_pair`, work as in the substitution stepper, on the lists in the heap instead of on list literals.

Identity: `v is w` is replaced by `True` if v and w are the same primitive value or the same reference, and by `False` otherwise; `is not` is the negation.

$$\overline{\#k \text{ is } \#k \Rightarrow \text{True}}$$

Loops

While-unfold: A `while` loop is unfolded by the textbook rule before its test is evaluated. The loop stays in the program, in full, while the test is evaluated; so the program, the frames and the heap always contain what is needed to continue by hand. The `if` is then reduced by the rules for conditional statements.

$$\overline{\text{while } e: b \Rightarrow \text{if } e: b \text{ while } e: b}$$

For-range: A `for` loop over `range(a, b, s)` first evaluates the arguments of `range` to values. The loop is then tested step by step: while the current value c is in the range, the target is assigned c and the body is run, followed by the loop with the next value $c + s$; otherwise the loop ends.

$$\frac{c \text{ is in the range } a, b, s}{\text{for } x \text{ in range}(c, b, s): \text{body} \Rightarrow x = c \text{ body for } x \text{ in range}(c + s, b, s): \text{body}}$$

Break-and-continue: Within an iteration, `break` removes the rest of the iteration and the loop, and `continue` removes the rest of the iteration and continues with the loop.

Worked example

```
def make_withdraw(balance):
    def withdraw(amount):
        nonlocal balance
        if balance >= amount:
            balance = balance - amount
            return balance
        else:
            return 'Insufficient funds'
    return withdraw
```

```
W1 = make_withdraw(100)
W1(50)
```

Initially, G is empty.

1. **Function-definition:** the definition of `make_withdraw` is consumed. A function object `#1` is created for `make_withdraw`, with G as its frame, and G binds `make_withdraw` to `#1`.
2. **Lookup:** in `W1 = make_withdraw(100)`, the name `make_withdraw` is replaced by `#1`, in a step of its own.
3. **Function-application:** a new frame $E_1 = \{\text{balance} \mapsto 100, \text{withdraw} \mapsto \perp\}$ with parent G . The program becomes $W1 = \llbracket E_1, \text{def withdraw...; return withdraw} \rrbracket$.

4. **Function-definition** in E_1 : a function object #2 is created for **withdraw**, with E_1 as its frame, and E_1 binds **withdraw** to #2. This is a cycle: E_1 refers to #2, and #2 refers to E_1 .
5. **Lookup**, then **Block-return**: the block is replaced by the value of **withdraw**, which is #2 (shown as **withdraw**).
6. **Assignment**: $G(w1) = \#2$. E_1 stays alive, since #2 refers to it.
7. **Function-application**: **w1(50)** is looked up and applied. A new frame $E_2 = \{\text{amount} \mapsto 50\}$ with parent E_1 (the frame in which #2 was created, and not the caller's). E_2 has no binding for **balance**: **nonlocal balance** decided that.
8. **Lookup**: in **balance >= amount**, **balance** is found in E_1 (100), then **amount** in E_2 (50), each in its own step. **100 >= 50** is **True**, and the **if** runs its first branch.
9. **Assignment**: **balance = balance - amount** reduces to **balance = 50**. With **nonlocal**, the binding is changed in E_1 : this is the step that the substitution model cannot express.
10. **Block-return**: **return balance** gives 50 and the block disappears. E_2 is now garbage; E_1 is not.

A second call **w1(50)** creates E_3 with parent E_1 and finds **balance** = 50 there.

Steps and their limit

Steps are numbered, explained and limited as in the Python §2 stepper: step 0 shows the program before evaluation, each contraction contributes two steps (the redex highlighted, then the result highlighted), and the last step is “Evaluation complete”, “Evaluation stuck” or “Maximum number of steps exceeded”. The number of the last step does not exceed the step limit. Each step also shows the frames and the heap as they are at that step, in the environment diagram, with the frame of the current redex highlighted.

A gutter breakpoint in the editor, and the statement **breakpoint()**, mark the step in which the statement is reached (for a gutter breakpoint, the first step of the statement); the *previous breakpoint* and *next breakpoint* buttons jump between these steps.

Python §4

The e-stepper does not yet support the metacircular evaluator support of Python §4 (**parse**, **tokenize** and **apply_in_underlying_python**), nor constructs outside the Python §3 syntax. When the reduction reaches such a construct, it stops with the message that the construct is not supported.