

Specification of Python §2 Stepper—2026 edition

Martin Henz

National University of Singapore
School of Computing

October 11, 2026

Introduction

The *stepper* shows how a Python §1 or Python §2 program is evaluated by the *substitution model*, as used in Chapters 1 and 2 of SICPy: the program itself is rewritten, one small step at a time, until it cannot be rewritten any further. In contrast to the CSE machine, no environments exist: a name is replaced by its value, and a function application is replaced by the function’s body in which the arguments have been substituted for the parameters. The substitution model does not support assignment to names that have already been used; that is the reason for Python §3 to move on to the environment model, visualized by the CSE machine and the environment stepper. This document specifies the rewriting that the stepper performs. It follows the specification of the Source §2 stepper, with the differences that arise from Python’s syntax: programs are sequences of statements without semicolons, function bodies are suites that end in **return**, and a statement that consists of a value has no result.

Reduction

The *reducer* $\Rightarrow$ is a partial function from programs/statements/expressions to programs/statements/expressions (with slight abuse of notation via overloading) and $\Rightarrow^*$ is its reflexive transitive closure. A *reduction* is a sequence of programs $p_1 \Rightarrow \cdots \Rightarrow p_n$, where p_n is not reducible, i.e. there is no program q such that $p_n \Rightarrow q$. Here, the program p_n is called the *result of reducing* p_1 .

If p_n , the result of reducing a program, is the empty sequence of statements, the evaluation is *complete*. Reduction can get “stuck”: the result can be a program that is not empty, which corresponds to a runtime error. In both cases the stepper ends the reduction with a last step that says “Evaluation complete” or “Evaluation stuck”, respectively. Unlike in Source, a statement that consists of a value does not survive as the result of the program: it is removed when it is reached (see **Program-discard**); the stepper therefore always ends with the empty program, unless a runtime error occurs.

A *value* is an integer, float, complex or string literal, **True**, **False**, **None**, a **lambda** expression, a function definition (**def**), a built-in function, or a pair $[v_1, v_2]$ of values. (The stepper shows the result of **pair**(v_1, v_2) in this notation; **None** is the empty linked list.) A function that was declared with a name is shown by that name and reveals its definition when hovered.

The *substitution* function $p[n := v]$ on programs/statements/expressions replaces every free occurrence of the name n in p by value v . See the section on substitution below. Care must be taken to introduce and preserve co-references in this process; substitution can introduce cyclic references in the result of the substitution. For example, n may occur free in v , in which case every occurrence of n in p will be replaced by v such that n in v refers cyclically to the node at which the replacement happens.

Expressions are reduced from left to right, and inner expressions before the expressions that contain them: in $e_1 + e_2$, the expression e_2 is only reduced when e_1 is a value. Each reduction step is called a *contraction*. The stepper shows two states for each contraction: first the program with the expression that is about to be contracted highlighted (the *redex*), and then the program after the contraction with the result highlighted.

Programs

Program-intro: In a sequence of statements, we can always reduce the first statement.

$$\frac{\text{statement} \Rightarrow \text{statement}'}{\text{statement statement}_2 \dots \Rightarrow \text{statement}' \text{statement}_2 \dots}$$

Program-discard: When the first statement of a program is an expression statement whose expression is a value, the statement is removed. In Python, a statement does not have a result.

$$\frac{v \text{ is a value}}{v \text{ statement} \dots \Rightarrow \text{statement} \dots}$$

Eliminate-function-definition: A function definition as the first statement is substituted in the remaining statements, in which its name is replaced by the function.

$$\frac{f = \text{def name}(x_1, \dots, x_n) : \text{suite}}{f \text{ statement} \dots \Rightarrow \text{statement} \dots [\text{name} := f]}$$

Eliminate-assignment: An assignment of a value as the first statement is substituted in the remaining statements.

$$\frac{v \text{ is a value}}{\text{name} = v \text{ statement} \dots \Rightarrow \text{statement} \dots [\text{name} := v]}$$

Pass: A `pass` statement is removed.

$$\text{pass statement} \dots \Rightarrow \text{statement} \dots$$

Import: An import statement is removed, and the imported names are substituted in the remaining statements by the functions of the module.

$$\text{from module import name statement} \dots \Rightarrow \text{statement} \dots [\text{name} := f]$$

Breakpoint: The statement `breakpoint()` is removed. As a side-effect, the step before this contraction is marked as a breakpoint, to which the *previous breakpoint* and *next breakpoint* buttons of the stepper jump. A statement on which the user has set a breakpoint in the editor is treated in the same way, in addition to its own contraction: the first step that reaches it is marked.

$$\text{breakpoint}() \text{ statement} \dots \Rightarrow \text{statement} \dots$$

Statements: Expression statements

Expression-statement-reduce: An expression statement is reducible if its expression is reducible.

$$\frac{e \Rightarrow e'}{e \Rightarrow e'}$$

Statements: Assignments

Evaluate-assignment: The right-hand expression in an assignment is evaluated.

$$\frac{\text{expression} \Rightarrow \text{expression}'}{\text{name} = \text{expression} \Rightarrow \text{name} = \text{expression}'}$$

Statements: Conditionals

Conditional-statement-predicate: A conditional statement is reducible if its predicate is reducible.

$$\frac{e \Rightarrow e'}{\text{if } e : \text{suite}_1 \text{ else: suite}_2 \Rightarrow \text{if } e' : \text{suite}_1 \text{ else: suite}_2}$$

An `elif` is the `else` suite of a conditional statement, which consists of a conditional statement; an `if` without `else` has an empty alternative suite.

Conditional-statement-consequent: A conditional statement whose predicate is **True** is replaced by the statements of the consequent suite.

$$\frac{}{\text{if True: suite}_1 \text{ else: suite}_2 \text{ statement} \dots \Rightarrow \text{suite}_1 \text{ statement} \dots}$$

Conditional-statement-alternative: A conditional statement whose predicate is **False** is replaced by the statements of the alternative suite.

$$\frac{}{\text{if False: suite}_1 \text{ else: suite}_2 \text{ statement} \dots \Rightarrow \text{suite}_2 \text{ statement} \dots}$$

If the predicate is a value other than **True** and **False**, the reduction is stuck with the error `TypeError: predicate type must be 'bool'`.

Function bodies

When a function with a suite as its body is applied, the application is replaced by a *block expression*: the statements of the body, with the arguments substituted. A block expression is reduced like a program, except for the following rules.

Block-expression-intro: A block expression is reducible if its first statement is reducible. The rules for statements above apply in the same way as for programs, including **Program-discard** for value statements that are not the last statement.

$$\frac{\text{statement} \Rightarrow \text{statement}'}{\{ \text{statement} \text{ statement}_2 \dots \} \Rightarrow \{ \text{statement}' \text{ statement}_2 \dots \}}$$

Block-expression-return: A block expression starting with a return statement reduces to the expression of the return statement. The statements after the return statement are dropped. A **return** without an expression returns **None**.

$$\frac{}{\{ \text{return } e \text{ statement} \dots \} \Rightarrow e}$$

Block-expression-fall-off: A block expression that has no statements left, or whose last statement is a value that is discarded, reduces to **None**.

$$\frac{}{\{ \} \Rightarrow \text{None}}$$

As a shortcut, a function whose body starts with a **return** statement, as well as every **lambda** expression, is applied by replacing the application directly with the returned expression (or the body of the lambda expression, respectively), without creating a block expression.

Expressions: Binary operators

Left-binary-reduce: An expression with a binary operator can be reduced if its left sub-expression can be reduced.

$$\frac{e_1 \Rightarrow e'_1}{e_1 \text{ binary-operator } e_2 \Rightarrow e'_1 \text{ binary-operator } e_2}$$

Right-binary-reduce: An expression with a binary operator can be reduced if its left sub-expression is a value and its right sub-expression can be reduced.

$$\frac{e_2 \Rightarrow e'_2}{v_1 \text{ binary-operator } e_2 \Rightarrow v_1 \text{ binary-operator } e'_2}$$

Binary-operator-reduce: A binary operator applied to two values is replaced by the result of the operation. Integers are arbitrary-precision; if one of the operands is a float, the other operand is converted to a float, and if one of them is complex, the other is converted to complex (see the numbers section of this specification). Operands of other types, for example a boolean operand of an arithmetic operator, cause a `TypeError`, and division by zero causes a `ZeroDivisionError`; the reduction is then stuck.

$$\frac{v_1 \text{ op } v_2 \text{ evaluates to } v}{v_1 \text{ binary-operator } v_2 \Rightarrow v}$$

And-shortcut: An expression with binary operator **and** whose left sub-expression is **False** is reduced to **False**, without evaluating the right sub-expression.

$$\frac{}{\mathbf{False} \text{ and } e \Rightarrow \mathbf{False}}$$

And-continue: An expression with binary operator **and** whose left sub-expression is **True** is reduced to the right sub-expression.

$$\frac{}{\mathbf{True} \text{ and } e \Rightarrow e}$$

Or-shortcut: An expression with binary operator **or** whose left sub-expression is **True** is reduced to **True**, without evaluating the right sub-expression.

$$\frac{}{\mathbf{True} \text{ or } e \Rightarrow \mathbf{True}}$$

Or-continue: An expression with binary operator **or** whose left sub-expression is **False** is reduced to the right sub-expression.

$$\frac{}{\mathbf{False} \text{ or } e \Rightarrow e}$$

The left operand of **and** and **or** is evaluated first, as in the rules for the other binary operators (**Left-binary-reduce**). A left operand that is a value other than **True** or **False** causes a `TypeError`.

Expressions: Unary operators

Unary-reduce: An expression with a unary operator can be reduced if its operand can be reduced.

$$\frac{e \Rightarrow e'}{\text{unary-operator } e \Rightarrow \text{unary-operator } e'}$$

Unary-operator-reduce: A unary operator (**-**, **+** or **not**) applied to a value is replaced by the result of the operation. **not** requires a boolean operand.

$$\frac{\text{op } v \text{ evaluates to } v'}{\text{unary-operator } v \Rightarrow v'}$$

Expressions: Conditional expressions

Conditional-expression-predicate: A conditional expression is reducible if its predicate is reducible.

$$\frac{e_1 \Rightarrow e'_1}{e_2 \text{ if } e_1 \text{ else } e_3 \Rightarrow e_2 \text{ if } e'_1 \text{ else } e_3}$$

Conditional-expression-consequent: A conditional expression whose predicate is **True** reduces to its consequent.

$$\frac{}{e_2 \text{ if } \mathbf{True} \text{ else } e_3 \Rightarrow e_2}$$

Conditional-expression-alternative: A conditional expression whose predicate is **False** reduces to its alternative.

$$\frac{}{e_2 \text{ if } \mathbf{False} \text{ else } e_3 \Rightarrow e_3}$$

Expressions: Function application

Application-functor-reduce: A function application can be reduced if its functor expression can be reduced.

$$\frac{e \Rightarrow e'}{e \text{ (expressions) } \Rightarrow e' \text{ (expressions)}}$$

Application-argument-reduce: A function application can be reduced if one of its argument expressions can be reduced and all preceding arguments are values.

$$\frac{e \Rightarrow e'}{v (v_1 \dots v_i, e, \dots) \Rightarrow v (v_1 \dots v_i, e', \dots)}$$

Function-definition-application-reduce: The application of a function definition, to as many arguments as it has parameters, all of which are values, is replaced by its body in which the parameters are replaced by the arguments, and the name of the function is replaced by the function itself (so that recursive calls can be applied). If the body is a suite that does not start with a **return**, the application is replaced by the block expression of the suite.

$$\frac{f = \text{def } n(x_1, \dots, x_k) : b}{f (v_1, \dots, v_k) \Rightarrow b[x_1 := v_1] \dots [x_k := v_k][n := f]}$$

Lambda-application-reduce: The application of a lambda expression, to as many arguments as it has parameters, all of which are values, is replaced by its body in which the parameters are replaced by the arguments. A lambda expression that was assigned to a name n (**n = lambda** ...) is applied in the same way as a function definition, including the substitution $[n := f]$.

$$\frac{f = \text{lambda } x_1, \dots, x_k : e}{f (v_1, \dots, v_k) \Rightarrow e[x_1 := v_1] \dots [x_k := v_k]}$$

If the number of arguments is different from the number of parameters, the reduction is stuck with a `TypeError`. The same holds for the application of a value that is not a function.

Builtin-application-reduce: The application of a built-in function (for example **abs**, **math_sqrt**, **pair**, **head**, **tail** and the other functions of the linked list library), to arguments that are all values, is replaced by the result of the function in one step. This holds for the predefined functions that are written in Python, such as **length**, **map** and **reduce**, as well: the stepper does not show their implementation, nor the applications of a function argument (for example the function given to **map**) that they make. The functions **print** and **print_llist** additionally write to the output, which the stepper shows from the step after this contraction onwards.

$$\frac{f \text{ is a built-in function } \quad f(v_1, \dots, v_k) = v}{f (v_1, \dots, v_k) \Rightarrow v}$$

Substitution

Substitution $p[x := v]$ replaces the free occurrences of x in p by v . It is defined by the following cases; for the expressions that are not listed (operators, applications, pairs, conditional expressions), the substitution is applied to every sub-expression.

Identifier: An identifier with the same name as x is substituted with v ; other identifiers are left alone.

$$\frac{}{x[x := v] = v} \quad \frac{}{name[x := v] = name \quad (name \neq x)}$$

Compound expressions: All occurrences of x in the sub-expressions are substituted with v .

$$\frac{}{(e_1 \text{ binary-operator } e_2)[x := v] = e_1[x := v] \text{ binary-operator } e_2[x := v]}$$

Lambda expression: If x is one of the parameters, the lambda expression is left alone. Otherwise the body is substituted. If a parameter of the lambda expression occurs free in v , the parameter is renamed first, to avoid that the parameter captures that occurrence.

$$\frac{x \notin \{x_1, \dots, x_k\} \quad \{x_1, \dots, x_k\} \cap free(v) = \emptyset}{(\text{lambda } x_1, \dots, x_k : e)[x := v] = \text{lambda } x_1, \dots, x_k : e[x := v]}$$

Function definition: If x is the name of the function, or one of the parameters, the definition is left alone. Otherwise, the body is substituted, after the renaming of parameters and local names that occur free in v , as for lambda expressions. If x is assigned to in the body of the definition, x is local to the function (as in Python): v is not substituted into the body, and reading x before the assignment is an `UnboundLocalError`.

$$\frac{x \notin \{n, x_1, \dots, x_k\} \quad x \text{ is not assigned in } b}{(\text{def } n(x_1, \dots, x_k) : b)[x := v] = \text{def } n(x_1, \dots, x_k) : b[x := v]}$$

Statement sequences: In a sequence of statements, the statements are substituted from left to right. After a statement that declares x (an assignment to x , or a definition of x), the remaining statements are left alone, because they refer to that declaration.

$$\frac{\text{the first statement of } s \text{ declares } x}{(statement_1 \text{ statement}_2 \dots)[x := v] = statement_1[x := v] \text{ statement}_2 \dots}$$

Assignment: In an assignment, only the right-hand side is substituted.

$$\frac{}{(name = e)[x := v] = name = e[x := v]}$$

Free names

The set $free(p)$ of free names of a program, statement or expression p is defined as usual: an identifier is free; a lambda expression binds its parameters; a function definition binds its name, its parameters and the names that are assigned to in its body; and, in a sequence of statements, an assignment or function definition binds its name in the statements that follow it. Names of built-in functions are not free: they are never substituted.

Steps and their limit

The stepper presents an evaluation as a sequence of *steps*, numbered from 0 on the step slider:

- Step 0 shows the program before evaluation (“Start of evaluation”).

- Each contraction contributes two steps: the program with the redex highlighted (the explanation says what is about to happen, for example “Evaluating binary expression”), followed by the program after the contraction with the result highlighted (“Evaluated binary expression . . .”). The program after a contraction is identical to the program before the next contraction; only the highlighting differs. If a contraction removes a statement without leaving a result in its place (for example **Eliminate-assignment**), nothing is highlighted after it.
- The last step is “Evaluation complete”, “Evaluation stuck” (in which case the previous step explains the error), or “Maximum number of steps exceeded”.

The number of the last step does not exceed the *step limit* that the user can configure (default: 1000). A limit of i allows $\max(0, \lfloor (i-1)/2 \rfloor)$ contractions; the last step is numbered $2 \cdot \lfloor (i-1)/2 \rfloor + 1$ if the evaluation completes within the limit. The same limit applies to the environment stepper and to the CSE machine, whose step slider also counts from 0.