Specification of Python §4—2026 edition

Martin Henz, Wang Yuyang, Lucas Ng, Jonas Ong

National University of Singapore
School of Computing

July 15, 2026

The language "Python §x" is a sublanguage of Python 3.13 and defined in the documents titled "Python §x", where x refers to the respective textbook chapter.

1 Changes

Compared to Python §3, Python §4 has the following changes:

- Python §4 adds the functions `tokenize`, `parse`, and `apply_in_underlying_python`. For details, see Section "Interpreter Support" below.

2 Syntax

A Python program is a *program*, defined using Backus-Naur Form[1] as follows:

[1] We adopt Henry Ledgard's BNF variant that he described in *A human engineered variant of BNF*, ACM SIGPLAN Notices, Volume 15 Issue 10, October 1980, Pages 57-62. In our grammars, we use **`bold`** font for keywords, *italics* for syntactic variables, ϵ for nothing, $x \mid y$ for x or y, [x] for an optional x, x... for zero or more repetitions of x, and (x) for clarifying the structure of BNF expressions.

program	::=	*import-stmt*... *block*	program
import-stmt	::=	**from** *dotted-name* **import** *import-clause*	import statement
dotted-name	::=	*name* (**.** *name*)...	dotted identifier chain
import-clause	::=	*import-as-names*	
	|	**(** *import-as-names* **)**	wildcard or name list
import-as-names	::=	*import-as-name* (**,** *import-as-name*)...	clause name list
import-as-name	::=	*name* [**as** *name*]	clause name and alias
statement	::=	*name* **=** *expression*	variable declaration
	|	**def** *name* **(** *names* **) :** *block*	function declaration
	|	**return** *expression*	return statement
	|	*if-statement*	conditional statement
	|	**while** *expression* **:** *block*	while loop
	|	**for** *name* **in range(** *range-args* **) :** *block*	for loop
	|	**break**	break statement
	|	**continue**	continue statement
	|	**global** *name*	global statement
	|	**nonlocal** *name*	nonlocal statement
	|	*expression*	expression statement
range-args	::=	*expression* [**,** *expression*] [**,** *expression*]	for loop range arguments
names	::=	ϵ | *name* (**,** *name*)...	name list
rest-names	::=	ϵ | *****name | *name* (**,** *name*)... [**,** *****name]	name list (rest)
if-statement	::=	**if** *expression* **:** *block*	
		(**elif** *expression* **:** *block*)...	
		[**else :** *block*]	conditional statement
block	::=	*statement*...	block statement
expression	::=	*integer*	primitive integer expression
	|	*float*	primitive float expression
	|	*complex*	primitive complex expression
	|	**True** | **False**	primitive boolean expression
	|	**None**	primitive None expression
	|	*string*	primitive string expression
	|	*name*	name expression
	|	*expression binary-operator expression*	binary operator combination
	|	*unary-operator expression*	unary operator combination
	|	*expression binary-logical expression*	logical composition
	|	*expression* **(** *spread-expressions* **)**	function application
	|	**lambda** (*name* | *names*) **:** *expression*	lambda expression
	|	*expression* **if** *expression* **else** *expression*	conditional expression
	|	*expression***[***expression***] =** *expression*	list assignment
	|	*expression***[***expression***]**	list access
	|	**[** *expressions* **]**	literal list expression
	|	**(** *expression* **)**	parenthesised expression
binary-operator	::=	**+** | **-** | ***** | **/** | ****** | **%** | **is** | **is not**	
	|	**==** | **!=** | **>** | **<** | **>=** | **<=**	binary operator
unary-operator	::=	**not** | **-** | **+**	unary operator
binary-logical	::=	**and** | **or**	logical composition symbol
expressions	::=	ϵ | *expression* (**,** *expression*)...	element expressions
spread-expressions	::=	ϵ | *spread-expression* (**,** *spread-expression*)...	argument expressions
spread-expression	::=	*expression* | ***** *expression*	argument expressions (spread)

Indentation

In Python, indentation is syntactically significant and strictly enforces code block structure. Unlike languages using braces, Python employs whitespace (spaces or tabs) to delimit blocks for control flow (e.g., `if`, `for`, `while`), function definitions, and class declarations. Key rules:

- Consistency: Use 4 spaces per indentation level (PEP 8 recommendation). Mixing tabs and spaces is prohibited.
- Alignment: Statements within the same block must align vertically. Misaligned indentation raises an `IndentationError`.
- Nesting: Each nested block increases indentation by one level. Dedenting resumes the outer block.
- Line Continuation: For multi-line statements, align wrapped lines with the opening delimiter or use an extra level.

Restrictions

- In Python, a logical line is the smallest unit of code that the interpreter can execute, typically corresponding to a complete statement. A physical line, by contrast, is a single line in the source file, usually ending with a newline character. A logical line may span multiple physical lines depending on whether line joining is used.

  By default, each physical line corresponds to a logical line. However, when a statement is too long, Python allows it to be split across multiple physical lines using line continuation. Python provides two ways to join lines: **explicit line joining** and **implicit line joining**.

  Explicit line joining uses a backslash (\) to join the current physical line with the next one, forming a single logical line. The backslash must be the last character on the line—no characters (including whitespace or comments) are allowed after it.

  ```
  total = 1 + 2 + 3 + \
          4 + 5
  ```

  Implicit line joining occurs when expressions are enclosed in parentheses `()`, square brackets `[]`, or braces `{}`. Within these delimiters, line breaks are allowed without using a backslash.

  ```
  total = (
              1 + 2 + 3 +
              4 + 5
          )
  ```

  Therefore, when breaking a statement across lines, you must either use a backslash for explicit continuation or wrap the expression in parentheses, brackets, or braces for implicit continuation. Unlike JavaScript, which uses automatic semicolon insertion, Python requires each line to be syntactically complete—it does not infer the end of a statement implicitly.

  For details on logical lines, see the official Python documentation: Python Lexical Analysis – Logical Lines.

- Return statements are only allowed in bodies of functions. Each return statement must appear on a single logical line.
- Lambda expressions are limited to a single logical line.

Loops

`for` loop through a range with one argument

The expression in the following syntax should evaluate to an integer:

`for` *name* **`in range(`** *expression* **`) :`** *block*

This roughly stands for:

```
_loop_i = 0
_end = expression

while _loop_i < _end:
    name = _loop_i
    block
    _loop_i = _loop_i + 1
```

`for` loop through a range with two arguments

The expressions in the following syntax should both evaluate to integers:

`for` *name* **`in range(`** $expression_1$**`,`** $expression_2$ **`)`** **`:`** *block*

This roughly stands for:

```
_loop_i = expression_1
_end = expression_2

while _loop_i < _end:
    name = _loop_i
    block
    _loop_i = _loop_i + 1
```

`for` loop through a range with three arguments

The expressions in the following syntax should all evaluate to integers:

`for` *name* **`in range(`** $expression_1$**`,`** $expression_2$**`,`** $expression_3$ **`)`** **`:`** *block*

This roughly stands for:

```
_loop_i = expression_1
_end = expression_2
_step = expression_3

if _step > 0:
    while _loop_i < _end:
        name = _loop_i
        block
        _loop_i = _loop_i + _step
else:
    while _loop_i > _end:
        name = _loop_i
        block
        _loop_i = _loop_i + _step
```

Names

Names[2] start with _, or a letter[3] and contain only _, letters or digits[4]. Reserved words[5] are not allowed as names.
Valid names are `x`, `_45`, and π, but always keep in mind that programming is communicating and that the familiarity of the audience with the characters used in names is an important aspect of program readability.

[2] In Python 3.13 Documentation, these names are called *identifiers*.
[3] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[4] By *digit* we mean characters in the Unicode categories Nd (including the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9), Mn, Mc and Pc.
[5] By *reserved word* we mean any of: **`False`**, **`await`**, **`else`**, **`import`**, **`pass`**, **`None`**, **`break`**, **`except`**, **`in`**, **`raise`**, **`True`**, **`class`**, **`finally`**, **`is`**, **`return`**, **`and`**, **`continue`**, **`for`**, **`lambda`**, **`try`**, **`as`**, **`def`**, **`from`**, **`nonlocal`**, **`while`**, **`assert`**, **`del`**, **`global`**, **`not`**, **`with`**, **`async`**, **`elif`**, **`if`**, **`or`**, **`yield`**. These are all reserved words, or keywords of the language that cannot be used as ordinary identifiers.

Numbers

Python supports three numeric types: integers (`int`), floats (`float`), and complex numbers (`complex`).

Integers

Integers can be represented in decimal notation, optionally prefixed with a sign (`+` or `-`). Additional base notations are supported, such as binary (`0b1010` or `0B1010`), octal (`0o777` or `0O777`), and hexadecimal (`0x1A3F` or `0X1A3F`). Underscores (`_`) may be used for readability (e.g., `1_000_000`). Examples include `42`, `-0b1101`, and `0x_FF_00`.

Floats

Floats use decimal notation with an optional decimal dot. Scientific notation (multiplying the number by 10^x) is indicated with the letter `e` or `E`, followed by the exponent `x`. Special values inf (infinity), -inf, and nan (not a number) are allowed. Examples include `3.14`, `-0.001e+05`, and `6.022E23`.

Complex Numbers

Complex numbers are written as `<real>±<imag>j`, where `j` (or `J`) denotes the imaginary unit. Both the real and imaginary parts are stored as floats. The imaginary part is mandatory (e.g., `5j`, `0j`, and `0+3j` is valid, `5` alone is real).
Examples include `2+3j`, `-4.5J`, `0j`, and `1e3-6.2E2J`.

Strings

Strings are of the form `"`*double-quote-characters*`"`, where *double-quote-characters* is a possibly empty sequence of characters without the character `"` and without the newline character; of the form `'`*single-quote-characters*`'`, where *single-quote-characters* is a possibly empty sequence of characters without the character `'` and without the newline character; and of the form `'''`*triple-single-quote-characters*`'''`, or `"""`*triple-double-quote-characters*`"""`, where *triple-double-quote-characters* is a possibly empty sequence of characters and can span multiple lines and may contain both single and double quotes without escaping.
The following characters can be represented in strings[6] as given:

- `\<newline>`: Backslash followed by a newline is ignored.
- `\\`: Represents a backslash (i.e., `\`).
- `\'`: Represents a single quote (').
- `\"`: Represents a double quote (").
- `\a`: Represents the ASCII Bell (BEL) character.
- `\b`: Represents the ASCII Backspace (BS) character.
- `\f`: Represents the ASCII Formfeed (FF) character.
- `\n`: Represents the ASCII Linefeed (LF) character.
- `\r`: Represents the ASCII Carriage Return (CR) character.
- `\t`: Represents the ASCII Horizontal Tab (TAB) character.
- `\v`: Represents the ASCII Vertical Tab (VT) character.

[6] In Python 3.13 Documentation, unless an 'r' or 'R' prefix is present, escape sequences in string and bytes literals are interpreted according to rules similar to those used by Standard C. *identifiers*.

3 Dynamic Type Checking

Expressions evaluate to integers, floats, complex numbers, boolean values, strings or function values. Implementations of Source generated error messages when unexpected values are used as follows.
Only function values can be applied using the syntax:

expression ::= *name* **(** *expressions* **)**

For compound functions, implementations need to check that the number of *expressions* matches the number of parameters.
The following table specifies what arguments Python's operators take and what results they return. Implementations need to check the types of arguments and generate an error message when the types do not match.

operator	argument 1	argument 2	result
`+,-,*,%,//`	int	int	int
`/`	int	int	float
`+,-,*,/,%,//`	int	float	float
`+,-,*,/,%,//`	float	int, float	float
`+,-,*,/`	int	complex	complex
`+,-,*,/`	float	complex	complex
`+,-,*,/`	complex	int, float, complex	complex
`+`	string	string	string
`**`	int	int >= 0	int
`**`	int	int < 0	float
`**`	int	float	float
`**`	int	complex	complex
`**`	float	int, float	float
`**`	float	complex	complex
`**`	complex	int, float, complex	complex
`==,!=,is,is not`	any	any	bool
`>,>=,<,<=`	int,float,bool	int,float,bool	bool
`>,>=,<,<=`	string	string	bool
`and`	bool	any	any
`or`	bool	any	any
`not`	bool		bool
`-`	int		int
`-`	float		float
`-`	complex		complex

Following **`if`** and **`elif`**, Python §x only allows boolean expressions. Lambda expressions can only be used as arguments to operators when they are parenthesized.
The operators **`==`**, **`!=`**, **`is`**, **`is not`**, **`>`**, **`>=`**, **`<`**, **`<=`** follow the Python 3.14 Documentation. As in Python, **`True`** and **`False`** are represented by the integers 1 and 0, respectively, for the purpose of any comparisons. The operators **`is`** and **`is not`** test for identity. As in Python, we can assume that there exists only a single instance of each of **`None`**, **`True`**, and **`False`** system-wide, and that all lists and functions have their own identity once created.

4 Standard Library

The standard library contains constants and functions that are always available in this language. In py-slang, the standard library functions are not implemented using a conventional split between primitive and predeclared functions. Unlike in Source, where predeclared functions are defined using built-in primitives, all standard library functions in py-slang are written directly in

TypeScript and embedded into the runtime of the CSE Machine. These functions are treated as part of the host environment rather than as user-level definitions, and they interact directly with the evaluator's internal control stack and environment. This design simplifies the interpreter architecture and allows for tighter integration with the execution model, enabling better support for visualization, error handling, and platform-specific extensions within the Source Academy learning environment.
The standard library for Python §x is documented in online documentation of Python §x.

MISC Library

The following names are provided by the MISC library:

- `str(x)`: *primitive*, converts the value `x` to a complex number, or a string representation of `x` respectively, and returns the respective value.
- `repr(s)`: *primitive*, returns a string representation of `s` that is unambiguous and suitable for debugging; for many values, this string can be used to reconstruct `s`.
- `is_boolean(x)`, `is_integer(x)`, `is_float(x)`, `is_string(x)`, `is_none(x)`, `is_function(x)`, `is_complex(x)`: *primitive*, returns `True` if the type of `x` matches the function name and `False` if it does not.
- `complex(r, i)`: *primitive*, takes in two numbers `r` and `i`, and returns the complex number with real part `r` and imaginary part `i`.
- `real(x)`, `imag(x)`: *primitive*, return the real and imaginary parts of a complex number `x` respectively.
- `input(s)`: *primitive*, pops up a window that displays the *string* `s`, provides an input line for the user to enter a text, a "Cancel" button and an "OK" button. The call of `input` suspends execution of the program until one of the two buttons is pressed. If the "OK" button is pressed, `input` returns the entered text as a string. If the "Cancel" button is pressed, `input` returns a non-string value.
- `min(x, y, *args)` and `max(x, y, *args)`: *primitive*, takes in two or more arguments and returns the smallest and largest of the arguments respectively. If multiple arguments are equally small or large, the first one encountered in the argument list is returned. The arguments must be mutually comparable (i.e., they must support the < and > operators).
- `abs(x)`: *primitive*, returns the absolute value of `x`. For an `int` input, it returns the non-negative integer equivalent. For a `float` input, it returns the positive floating-point number representing its magnitude. For a `complex` input, it returns the modulus of the complex number, which is a non-negative float.
- `print(*x)`: *primitive*, takes in any number of parameters, converts them to their string representations using `str` and writes them to the console, followed by a newline character.
- `error(*x)`: *primitive*, takes in any number of parameters, converts them to their string representations using `str` and writes them to the console, followed by a newline character. The evaluation of any call of `error` aborts the running program immediately.
- `breakpoint()`: *primitive*, evaluates to the `None` value with no effect on the program (the analogue of JavaScript's `debugger;` statement). In the stepper it additionally marks a breakpoint that the breakpoint navigation can jump to.
- `arity(f)`: *primitive*, takes in a function `f` and returns the minimum number of parameters that `f` can accept.
- `round(x)`[7]: *primitive*, takes in an int or float `x` and returns the nearest integer to `x`.
- `round(x, n)`: *primitive*, takes in an int or float `x` and an int `n`, and returns the value of `x` rounded to `n` digits of precision after the decimal point. If the second argument `n` is None, it is treated as `round(x)`.

[7] In Python 3.13, when the value to be rounded is exactly halfway between two possible rounded values, it rounds to the nearest even number. For example, `round(2.5)` would return 2, while `round(3.5)` would return 4.

- `len(s)`: *primitive*, takes in a container for a finite number of values (e.g., a string), and returns the number of elements in the container.
- `time_time()`: *primitive*, returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `random_random()`: *primitive*, returns a random float uniformly in the range $[0.0, 1.0)$.

All library functions can be assumed to run in $O(1)$ time, except `print`, `error`, `str` and `repr`, which run in $O(n)$ time, where n is the size (number of components such as pairs) of their arguments.

MATH Library

The following names are provided by the MATH library:

- `math_`*name*, where *name* is any name specified in the Python **`math`** library, see Python 3.14 Documentation. Examples:
  - `math_nan`: *primitive*, refers to the NaN ("Not a Number") value,
  - `math_inf`: *primitive*, refers to the Infinity value,
  - `math_pi`: *primitive*, refers to the mathematical constant π,
  - `math_sqrt(n)`: *primitive*, returns the square root of the *number* `n`.

All functions can be assumed to run in $O(1)$ time and are considered *primitive*.

Appendix: Linked List library

Those linked list library functions that are not primitive functions are pre-declared as follows:

```
def pair(x, y):
    """
    PRIMITIVE
    Makes a pair whose head (first component) is x and whose
    tail (second component) is y.

    Parameters:
        x (value): given head
        y (value): given tail

    Returns:
        pair: pair with x as head and y as tail
    """
    pass


def is_pair(x):
    """
    PRIMITIVE
    Returns True if x is a pair and False otherwise.

    Parameters:
        x (value): given value

    Returns:
        boolean: whether x is a pair
    """
    pass


def head(p):
```

```
    """
    PRIMITIVE
    Returns head (first component) of given pair p.

    Parameters:
        p (pair): given pair

    Returns:
        value: head of p
    """
    pass


def tail(p):
    """
    PRIMITIVE
    Returns tail (second component) of given pair p.

    Parameters:
        p (pair): given pair

    Returns:
        value: tail of p
    """
    pass


def is_none(x):
    """
    PRIMITIVE
    Returns True if x is the empty linked list None, and False otherwise.

    Parameters:
        x (value): given value

    Returns:
        boolean: whether x is None
    """
    pass

def llist(*values):
    """
    PRIMITIVE
    Given n values, returns a linked list of length n. The elements of
    the linked list are the given values in the given order.

    Parameters:
        value (value1, value2, ...values): given values

    Returns:
        linked list: linked list containing all values
    """
    pass

def draw_data(value1, value2, *values):
    """
    PRIMITIVE
    Visualizes the arguments in a separate drawing area in the Source
    Academy using box-and-pointer diagrams.
```

```
    Parameters:
        value1 (value1, value2, ...values): given values
    """
    pass

def is_llist(xs):
    """
    Returns True if xs is a linked list as defined in the textbook, and
    False otherwise.

    Parameters:
        xs (value): given value

    Returns:
        boolean: whether xs is a linked list
    """
    if is_none(xs):
        return True
    elif is_pair(xs):
        return is_llist(tail(xs))
    else:
        return False

def length(xs):
    """
    Returns the length of the linked list xs.
    """
    return _length(xs, 0)

def _length(xs, acc):
    return acc if is_none(xs) else _length(tail(xs), acc + 1)

def map(f, xs):
    """
    Returns a linked list that results from linked list xs by element-wise
    application of unary function f.
    """
    return _map(f, xs, None)

def _map(f, xs, acc):
    return (
        reverse(acc)
        if is_none(xs)
        else _map(f, tail(xs), pair(f(head(xs)), acc))
    )

def build_llist(fun, n):
    """
    Makes a linked list with n elements by applying the unary function fun
    to the numbers 0 to n - 1.
    """
    return _build_llist(n - 1, fun, None)

def _build_llist(i, fun, already_built):
    return (
        already_built
        if i < 0
        else _build_llist(i - 1, fun, pair(fun(i), already_built))
    )
```

```
def for_each(fun, xs):
    """
    Applies the unary function fun to every element of the linked list xs.
    """
    if is_none(xs):
        return True
    else:
        fun(head(xs))  # Side effect happens here if fun is not pure
        return for_each(fun, tail(xs))

def llist_to_string(xs):
    """
    Returns a string that represents linked list xs using the text-based
    box-and-pointer notation.
    """
    return _llist_to_string(xs, lambda x: x)

def _llist_to_string(xs, cont):
    if is_none(xs):
        return cont("None")
    elif is_pair(xs):
        return _llist_to_string(
            head(xs),
            lambda x_str: _llist_to_string(
                tail(xs), lambda y_str: cont("[" + x_str + ", " + y_str + "]")
            ),
        )
    else:
        return cont(repr(xs))

def reverse(xs):
    """
    Returns linked list xs in reverse order.
    """
    return _reverse(xs, None)

def _reverse(original, reversed_acc):
    return (
        reversed_acc
        if is_none(original)
        else _reverse(
            tail(original), pair(head(original), reversed_acc)
        )
    )

def append(xs, ys):
    """
    Returns a linked list that results from appending the linked list ys
    to the linked list xs.
    """
    return _append(xs, ys, lambda x: x)

def _append(xs, ys, cont):
    return (
        cont(ys)
        if is_none(xs)
        else _append(
            tail(xs), ys, lambda zs: cont(pair(head(xs), zs))
        )
    )
```

```
def member(v, xs):
    """
    Returns first postfix sub-linked list whose head is identical to v
    (using ==). Returns None if the element does not occur in the linked
    list.
    """
    if is_none(xs):
        return None
    elif v == head(xs):
        return xs
    else:
        return member(v, tail(xs))

def remove(v, xs):
    """
    Returns a linked list that results from xs by removing the first item
    from xs that is identical (==) to v.
    """
    return _remove(v, xs, None)

def _remove(v, xs, acc):
    if is_none(xs):
        return append(reverse(acc), xs)
    elif v == head(xs):
        return append(reverse(acc), tail(xs))
    else:
        return _remove(v, tail(xs), pair(head(xs), acc))

def remove_all(v, xs):
    """
    Returns a linked list that results from xs by removing all items from
    xs that are identical (==) to v.
    """
    return _remove_all(v, xs, None)

def _remove_all(v, xs, acc):
    if is_none(xs):
        return append(reverse(acc), xs)
    elif v == head(xs):
        return _remove_all(v, tail(xs), acc)
    else:
        return _remove_all(v, tail(xs), pair(head(xs), acc))

def enum_llist(start, end):
    """
    Makes a linked list with elements from start to end (inclusive).
    """
    return _enum_llist(start, end, None)

def _enum_llist(start, end, acc):
    return (
        reverse(acc)
        if start > end
        else _enum_llist(start + 1, end, pair(start, acc))
    )

def llist_ref(xs, n):
    """
    Returns the element of linked list xs at position n (0-indexed).
```

```
    """
    if n == 0:
        if is_none(xs):
            error("llist_ref: index out of bounds on None linked list")
        return head(xs)
    else:
        if is_none(xs):
            error("llist_ref: index out of bounds")
        return llist_ref(tail(xs), n - 1)


def reduce(f, initial, xs):
    """
    Applies binary function f to the elements of xs from right-to-left
    order.
    """
    return _reduce(f, initial, xs, lambda x: x)


def _reduce(f, initial, xs, cont):
    if is_none(xs):
        return cont(initial)
    else:
        # Recursive CPS call: Process tail, then apply f with head,
        # then pass to continuation
        return _reduce(
            f,
            initial,
            tail(xs),
            lambda x_reduced_from_tail: cont(
                f(head(xs), x_reduced_from_tail)
            ),
        )


def filter(pred, xs):
    """
    Returns a linked list that contains only those elements for which the
    one-argument function pred returns True.
    """
    return _filter(pred, xs, None)


def _filter(pred, xs, acc):
    if is_none(xs):
        return reverse(acc)
    else:
        if pred(head(xs)):
            return _filter(pred, tail(xs), pair(head(xs), acc))
        else:
            return _filter(pred, tail(xs), acc)
```

Pair Mutators

The following pair mutator functions are supported:

- `set_head(p, x)`: *primitive*, changes the pair `p` such that its head is `x`. Returns `None`.
- `set_tail(p, x)`: *primitive*, changes the pair `p` such that its tail is `x`. Returns `None`.

Appendix: Stream library

Those stream library functions that are not primitive functions are pre-declared as follows:

```
def stream_tail(xs):
    """
    stream_tail returns the second component of the given pair
    throws an error if the argument is not a pair
    """
    if is_pair(xs):
        the_tail = tail(xs)
        if is_function(the_tail):
            return the_tail()
        else:
            error(
                the_tail,
                "stream_tail(xs) expects a function as "
                + "the tail of the argument pair xs, "
                + "but encountered ",
            )

    else:
        error(
            xs,
            "stream_tail(xs) expects a pair as "
            + "argument xs, but encountered ",
        )


def is_stream(xs):
    """
    is_stream recurses down the stream and checks that it ends with the
    empty linked list None
    """
    return is_none(xs) or (
        is_pair(xs)
        and is_function(tail(xs))
        and arity(tail(xs)) == 0
        and is_stream(stream_tail(xs))
    )


def llist_to_stream(xs):
    """
    A stream is either None or a pair whose tail is
    a nullary function that returns a stream.
    """
    return (
        None
        if is_none(xs)
        else pair(head(xs), lambda: llist_to_stream(tail(xs)))
    )


def stream_to_llist(xs):
    """
    stream_to_llist transforms a given stream to a linked list
    Lazy? No: stream_to_llist needs to force the whole stream
    """
    return (
        None if is_none(xs) else pair(head(xs), stream_to_llist(stream_tail(xs)))
```

```
    )


def stream_length(xs):
    """
    stream_length returns the length of a given argument stream
    throws an exception if the argument is not a stream
    Lazy? No: The function needs to explore the whole stream
    """
    return 0 if is_none(xs) else 1 + stream_length(stream_tail(xs))


def stream_map(f, s):
    """
    stream_map applies first arg f to the elements of the second
    argument, assumed to be a stream.
    f is applied element-by-element:
    stream_map(f, llist_to_stream(llist(1,2)) results in
    the same as llist_to_stream(llist(f(1),f(2)))
    stream_map throws an exception if the second argument is not a
    stream, and if the second argument is a nonempty stream and the
    first argument is not a function.
    Lazy? Yes: The argument stream is only explored as forced by
                         the result stream.
    """
    return (
        None
        if is_none(s)
        else pair(f(head(s)), lambda: stream_map(f, stream_tail(s)))
    )


def build_stream(fun, n):
    """
    build_stream takes a function fun as first argument,
    and a nonnegative integer n as second argument,
    build_stream returns a stream of n elements, that results from
    applying fun to the numbers from 0 to n-1.
    Lazy? Yes: The result stream forces the applications of fun
                         for the next element
    """

    def build(i):
        return None if i >= n else pair(fun(i), lambda: build(i + 1))

    return build(0)


def stream_for_each(fun, xs):
    """
    stream_for_each applies first arg fun to the elements of the stream
    passed as second argument. fun is applied element-by-element:
    for_each(fun, llist_to_stream(llist(1, 2, None))) results in the calls fun(1)
    and fun(2).
    stream_for_each returns True.
    stream_for_each throws an exception if the second argument is not a
    stream, and if the second argument is a nonempty stream and the
    first argument is not a function.
    Lazy? No: stream_for_each forces the exploration of the entire stream
    """
```

```
    if is_none(xs):
        return True
    else:
        fun(head(xs))
        return stream_for_each(fun, stream_tail(xs))


def stream_reverse(xs):
    """
    stream_reverse reverses the argument stream
    stream_reverse throws an exception if the argument is not a stream.
    Lazy? No: stream_reverse forces the exploration of the entire stream
    """

    def rev(original, reversed):
        return (
            reversed
            if is_none(original)
            else rev(
                stream_tail(original), pair(head(original), lambda: reversed)
            )
        )

    return rev(xs, None)


def stream_append(xs, ys):
    """
    stream_append appends first argument stream and second argument stream.
    In the result, None at the end of the first argument stream
    is replaced by the second argument stream
    stream_append throws an exception if the first argument is not a
    stream.
    Lazy? Yes: the result stream forces the actual append operation
    """
    return (
        ys
        if is_none(xs)
        else pair(head(xs), lambda: stream_append(stream_tail(xs), ys))
    )


def stream_member(x, s):
    """
    stream_member looks for a given first-argument element in a given
    second argument stream. It returns the first postfix substream
    that starts with the given element. It returns None if the
    element does not occur in the stream
    Lazy? Sort-of: stream_member forces the stream only until the element is found.
    """
    return (
        None
        if is_none(s)
        else s
        if head(s) == x
        else stream_member(x, stream_tail(s))
    )


def stream_remove(v, xs):
```

```
    """
    stream_remove removes the first occurrence of a given first-argument element
    in a given second-argument linked list. Returns the original linked list
    if there is no occurrence.
    Lazy? Yes: the result stream forces the construction of each next element
    """
    return (
        None
        if is_none(xs)
        else stream_tail(xs)
        if v == head(xs)
        else pair(head(xs), lambda: stream_remove(v, stream_tail(xs)))
    )


def stream_remove_all(v, xs):
    """
    stream_remove_all removes all instances of v instead of just the first.
    Lazy? Yes: the result stream forces the construction of each next element
    """
    return (
        None
        if is_none(xs)
        else stream_remove_all(v, stream_tail(xs))
        if v == head(xs)
        else pair(head(xs), lambda: stream_remove_all(v, stream_tail(xs)))
    )


def stream_filter(p, s):
    """
    filter returns the substream of elements of given stream s
    for which the given predicate function p returns True.
    Lazy? Yes: The result stream forces the construction of
    each next element. Of course, the construction
    of the next element needs to go down the stream
    until an element is found for which p holds.
    """
    return (
        None
        if is_none(s)
        else pair(head(s), lambda: stream_filter(p, stream_tail(s)))
        if p(head(s))
        else stream_filter(p, stream_tail(s))
    )


def enum_stream(start, end):
    """
    enumerates numbers starting from start,
    using a step size of 1, until the number
    exceeds end.
    Lazy? Yes: The result stream forces the construction of
    each next element
    """
    return (
        None
        if start > end
        else pair(start, lambda: enum_stream(start + 1, end))
    )
```

```
def integers_from(n):
    """
    integers_from constructs an infinite stream of integers
    starting at a given number n
    Lazy? Yes: The result stream forces the construction of
    each next element
    """
    return pair(n, lambda: integers_from(n + 1))


def eval_stream(s, n):
    """
    eval_stream constructs the linked list of the first n elements
    of a given stream s
    Lazy? Sort-of: eval_stream only forces the computation of
    the first n elements, and leaves the rest of
    the stream untouched.
    """

    def es(s, n):
        return (
            llist(head(s))
            if n == 1
            else pair(head(s), es(stream_tail(s), n - 1))
        )

    return None if n == 0 else es(s, n)


def stream_ref(s, n):
    """
    Returns the item in stream s at index n (the first item is at position 0)
    Lazy? Sort-of: stream_ref only forces the computation of
    the first n elements, and leaves the rest of
    the stream untouched.
    """
    return head(s) if n == 0 else stream_ref(stream_tail(s), n - 1)
```

Interpreter Support

- `apply_in_underlying_python(f, xs)`: *primitive*, calls the function `f` with arguments `xs`. For example:

  ```
  def times(x, y):
      return x * y
  apply_in_underlying_python(times, llist(2, 3))  # returns 6
  ```

- `tokenize(x)`: *primitive*, returns the literal linked list of tokens that results from tokenizing the string `x` as a Python program. Each token is a string that contains the characters of the token as they appear in the program. Comments are ignored.

- `parse(x)`: *primitive*, returns the parse tree that results from parsing the string `x` as a Python program. The following two pages describe the shape of the parse tree. The tree is represented by the tagged linked lists on the right; the angle brackets denote recursive application of the transformation rules. Implementations are allowed to support more of Python than listed.

In addition, the Source Academy frontend predeclares the name `__program__` in all Python languages to refer to the string representation of the entrypoint file of the program in the editor that is being run using "Run". The entrypoint file is the file containing the code that acts as the entrypoint of the program being run. If `__program__` is used in the REPL, it refers to the string representation of the editor content at the time when "Run" was last pressed.

program::=	*statement* ...	`linked_list("sequence", linked_list of` ⟨*statement*⟩`)`
statement::=	*name* **=** *expression*	`linked_list("variable_declaration",` ⟨*name*⟩, ⟨*expression*⟩`)`
|	*declared_name* **=** *expression*	`linked_list("assignment",` ⟨*declared_name*⟩, ⟨*expression*⟩`)`
|	**def** *name* **(** *names* **)** **:** *block*	`linked_list("function_declaration",` ⟨*name*⟩, ⟨*names*⟩, ⟨*block*⟩`)`
|	**return** *expression*	`linked_list("return_statement",` ⟨*expression*⟩`)`
|	*if-statement*	see below
|	**while (** *expression* **)** **:** *block*	`linked_list("while_loop",` ⟨*expression*⟩, ⟨*block*⟩`)`
|	**for** *name* **in range** **(** *range-args* **)** **:** *block*	`linked_list("for_loop",` ⟨*range_args*⟩, ⟨*block*⟩`)`
|	**break**	`linked_list("break_statement")`
|	**continue**	`linked_list("continue_statement")`
|	**nonlocal**	`linked_list("nonlocal_statement")`
|	*block*	see below
|	*expression*	see below
range_args::=	$expression_1$ **[**, $expression_2$ **]** **[** , $expression_3$ **]**	`linked_list("range_args",` ⟨$expression_1$⟩, ⟨$expression_2$⟩, ⟨$expression_3$⟩`)`
names::=	ϵ | *name* (**,** *name*) ...	`linked_list of` ⟨*name*⟩
if-statement::=	**if** $expression_1$ **:** $block_1$	
	(**elif** $expression_2$ **:** $block_2$)...	
	else **:** $block_3$	`linked_list("conditional_statement",` ⟨$expression_1$⟩, ⟨$block_1$⟩, ⟨$block_3$⟩ or ⟨*if-statement*⟩`)`
block::=	*statement*...	`linked_list("block",` ⟨*statement*⟩*)*
declaration::=	*name* **=** *expression*	`linked_list("variable_declaration",` ⟨*name*⟩, ⟨*expression*⟩`)`
assignment::=	*declared_name* **=** *expression*	`linked_list("assignment",` ⟨*declared_name*⟩, ⟨*expression*⟩`)`
|	$expression_1$ **[**$expression_2$**]** **=** $expression_3$	`linked_list("object_assignment",` ⟨$expression_1$ **[**$expression_2$**]**⟩, ⟨$expression_3$⟩`)`

expression::=	*integer*	`linked_list("literal", `*integer*`)`
|	*float*	`linked_list("literal", `*float*`)`
|	*complex*	`linked_list("literal", `*complex*`)`
|	**`True`** | **`False`**	`linked_list("literal", `*True*`)`
|	**`None`**	`linked_list("literal", `*None*`)`
|	*string*	`linked_list("literal", `*string*`)`
|	*name*	`linked_list("name", string)`
|	*expression*$_1$ *log-op expression*$_2$	`linked_list("logical_composition", ⟨`*log-op*`⟩, ⟨`*expression*$_1$`⟩, ⟨`*expression*$_2$`⟩)`
|	*expression*$_1$ *bin-op expression*$_2$	`linked_list("binary_operator_combination", ⟨`*bin-op*`⟩, ⟨`*expression*$_1$`⟩, ⟨`*expression*$_2$`⟩)`
|	*un-op expression*	`linked_list("unary_operator_combination", ⟨`*un-op*`⟩, ⟨`*expression*`⟩)`
|	*expression* **`(`** *expressions* **`)`**	`linked_list("application", ⟨`*expression*`⟩, `*linked_list of* `⟨`*expression*`⟩)`
|	**`lambda`** (*name* | *names*) **`:`** *expression*	`linked_list("lambda_expression", ⟨`*name*`⟩, ⟨`*names*`⟩)`
|	*expression*$_1$ **`if`** *expression*$_2$ **`else`** *expression*$_3$	`linked_list("conditional_expression", ⟨`*expression*$_1$`⟩, ⟨`*expression*$_2$`⟩, ⟨`*expression*$_3$`⟩)`
|	*expression*$_1$ **`[`***expression*$_2$**`]=`** *expression*$_3$	`linked_list("list_assignment", ⟨`*expression*$_1$`⟩, ⟨`*expression*$_2$`⟩, ⟨`*expression*$_3$`⟩)`
|	*expression*$_1$ **`[`***expression*$_2$**`]`**	`linked_list("list_access", ⟨`*expression*$_1$`⟩, ⟨`*expression*$_2$`⟩)`
|	**`[`** *expressions* **`]`**	`linked_list("list_expression", linked_list of ⟨`*expression*`⟩)`
|	**`(`** *expression* **`)`**	parenthesised *expression*
log-op::=	**`and`** | **`or`**	*string representing operator*
bin-op::=	**`+`** | **`-`** | **`*`** | **`/`** | **`**`** | **`%`** | **`is`**	*string representing operator*
|	**`==`** | **`!=`** | **`>`** | **`<`** | **`>=`** | **`<=`**	*string representing operator*
un-op::=	**`not`** | **`-`**	`"-unary"`
|	**`+`**	`"+unary"`
expressions::=	ϵ | *expression* (**`,`** *expression*) ...	`linked_list of ⟨`*expression*`⟩`

Deviations from native Python

We intend the Python §x to be a conservative extension of native Python: Every correct Python §x program should behave *exactly* the same using a Python §x implementation, as it does using a native Python implementation. We assume, of course, that suitable libraries are used by the TypeScript implementation, to account for the predefined names of Python §x. This section lists some exceptions where we think a Python §x implementation should be allowed to deviate from the native Python specification, for the sake of internal consistency and esthetics.

Output Behavior Differences Between py-slang and Standard Python REPL:

Python provides an interactive mode, where users can type and immediately evaluate Python expressions line by line. This mode is entered when you simply type python in a terminal without specifying a file. This is commonly referred to as the Python REPL. In the standard Python REPL, any evaluated expression automatically has its result printed to the console, even if the user does not explicitly call `print()`. For example:

```
>>> 1 + 2
3
>>> "hello"
'hello'
```

This is because Python's REPL implicitly displays the return value of each expression, unless it is None.

In contrast, py-slang implements the behavior of non-interactive Python. Only expressions explicitly passed to `print()` produce output. If the expression is evaluated without a print call, no output will appear, even though a value is computed.

For example, in py-slang:

```
print(1 + 2)     # Outputs: 3
```

But

```
1 + 2            # Outputs nothing
```