Specification of Python §1—2024 edition

Han Yizhan, Tan Jia Jun, Ooi Ken Jin, Martin Henz

National University of Singapore
School of Computing

March 1, 2026

The language Python §1 is a language designed for use with the textbook *Structure and Interpretation of Computer Programs*, JavaScript Adaptation. Python §1 is a sublanguage of *Python 3.7*, defined in the documents titled "Python §x", where x refers to the respective textbook chapter. The language aims to be faithful to the original Structure and Interpretation of Computer Programs (SICP) book. The language also maintains compatibility with modules designed for *Source*, a subset of JavaScript intended for use with SICP on Source Academy.

1 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-directive*... *statement*...	program
import-directive	::=	`from` *string* `import` `(` *import-names* `)` `;`	import directive
import-names	::=	ϵ | *import-name* (`,` *import-name*)...	import name list
import-name	::=	*name* | *name* `as` *name*	import name
statement	::=	*name* `=` *expression*	constant declaration
	|	`def` *name* `(` *names* `)` `:` *suite*	function declaration
	|	`return` *expression*	return statement
	|	*if-statement*	conditional statement
	|	*expression*	expression statement
	|	`debugger`	breakpoint
names	::=	ϵ | *name* (`,` *name*)...	name list
if-statement	::=	`if` `(` *expression* `)` `:` *suite*	
		[`elif:` *suite*]	
		`else:` *suite*	conditional statement
block	::=	`NEWLINE` `INDENT` *statement*... `DEDENT`	block statement
expression	::=	*number*	primitive number expression
	|	`True` | `False`	primitive boolean 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* `(` *expressions* `)`	function application
	|	`lambda` *name* | `(` *names* `)` `:` *expression*	lambda expression
	|	*expression* `if` *expression* `else` *expression*	conditional expression
	|	`(` *expression* `)`	parenthesised expression
binary-operator	::=	`+` | `-` | `*` | `/` | `%` | `==` | `!=`	
	|	`>` | `<` | `>=` | `<=`	binary operator
unary-operator	::=	`not` | `-`	unary operator
binary-logical	::=	`and` | `or`	logical composition symbol
expressions	::=	ϵ | *expression* (`,` *expression*)...	argument expressions

Restrictions

- Single assignment language: Each variable can be assigned a value only once during its lifetime.
- Operators do not implicitly cast boolean values to integers. Booleans cannot be used interchangeably with integers in operations.
- Scope restriction: Each variable can only be accessed in the block it is declared.

Import directives

Import directives allow programs to import values from modules and bind them to names, whose scope is the entire program in which the import directive occurs. Import directives can only appear at the top-level. All names that appear in import directives must be distinct, and must also be distinct from all top-level variables. The specifications do not specify how modules are programmed.

Logical Composition

Conjunction

$$\textit{expression}_1 \; \texttt{and} \; \textit{expression}_2$$

stands for

$$\textit{expression}_2 \; \texttt{if} \; \textit{expression}_1 \; \texttt{else} \; \texttt{False}$$

Disjunction

$$\textit{expression}_1 \; \texttt{or} \; \textit{expression}_2$$

stands for

$$\textit{expression}_1 \; \texttt{if} \; \textit{expression}_1 \; \texttt{else} \; \textit{expression}_2$$

Names

Names start with _ or a letter[2] and contain only _ or alphanumeric characters[3]. Restricted words[4] are not allowed as names.
Valid names are `x`, `_x`, `X` and `X_`, 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.

Floating-point Numbers

We use decimal notation for numbers, with an optional decimal dot. “Scientific notation” (multiplying the number with 10^x) is indicated with the letter `e`, followed by the exponent x. Examples for numbers are `5432`, `-5432.109`, and `-43.21e-45`.

Integers

We represent integers as whole numbers without any decimal component. Integers can be positive, negative or zero. Integers have arbitrary-precision, meaning they can represent extremely large values without loss of precision. Integers are automatically converted to floats as needed.

[2]By *letter* we mean Unicode letters (L) or letter numbers (NI).
[3]By alphanumeric characters we mean A-z, 0-9
[4]By *restricted word* we mean any of:**`False`**, **`None`**, **`True`**, **`and`**, **`as`**, **`assert`**, **`async`**, **`await`**, **`break`**, **`class`**, **`continue`**, **`def`**, **`del`**, **`elif`**, **`else`**, **`except`**, **`finally`**, **`for`**, **`from`**, **`global`**, **`if`**, **`import`**, **`in`**, **`is`**, **`lambda`**, **`nonlocal`**, **`not`**, **`or`**, **`pass`**, **`raise`**, **`return`**, **`try`**, **`while`**, **`with`**, **`yield`**. These are all words that cannot be used without restrictions as names in the strict mode of ECMAScript 2020.

Strings

In Python, strings can be enclosed within single quotes (`'...'`), double quotes (`"..."`), or triple quotes (`'''...'''` or `"""..."""`). The choice between single and double quotes is generally based on convenience, such as opting for a quote type not present within the string to avoid escaping. Triple quotes allow strings to span multiple lines and include both single and double quotes without needing escape sequences. Triple quotes are used for multi-line strings or docstrings.

- Single and Double Quotes:
  - `'single-quote-characters'`: Can contain any character except the single quote (`'`) itself unless escaped.
  - `"double-quote-characters"`: Can contain any character except the double quote (`"`) itself unless escaped.
- Triple Quotes:
  - `'''backquote-characters'''` or `"""backquote-characters"""`: Can include any character, including newlines, without escaping.
- Special Characters:
  - Horizontal tab: `\t`
  - Vertical tab: `\v`
  - Null character: `\0`
  - Backspace: `\b`
  - Form feed: `\f`
  - Newline: `\n`
  - Carriage return: `\r`
  - Single quote: `\'` (necessary in single-quoted strings)
  - Double quote: `\"` (necessary in double-quoted strings)
  - Backslash: `\\`
- Unicode Characters:
  - Represented using `\u` followed by four hexadecimal digits (e.g., `\u03A9` for the Greek capital letter omega).
  - For characters outside the Basic Multilingual Plane (BMP), use `\U` followed by eight hexadecimal digits (e.g., `\U0001F604` for a smiling face with open mouth and smiling eyes).

Comments

In Python §1, any characters after "#" until the next newline character is ignored.

2 Dynamic Type Checking

Expressions evaluate to numbers, boolean values, strings or function values. Implementations of Python §1 generate error messages when unexpected values are used as follows.
Only function values can be applied using the syntax:

$$\textit{expression} \quad ::= \quad \textit{name}\,\texttt{(}\ \textit{expressions}\ \texttt{)}$$

For compound functions, implementations need to check that the number of *expressions* matches the number of parameters.
The following table specifies what arguments Python §1'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
+	float	int | float	float
+	string	string	string
−	int	int	int
−	float	int | float	float
*	int	int	int
*	float	int | float	float
*	string	int	string
/	int | float	int | float	float
%	int	int	int
%	float	int | float	float
**	int	int	int
**	float	int | float	float
//	int | float	int | float	int
==	any	any	bool
!=	any	any	bool
>	int | float	int | float	bool
>	string	string	bool
<	int | float	int | float	bool
<	string	string	bool
>=	int | float	int | float	bool
>=	string	string	bool
<=	int | float	int | float	bool
<=	string	string	bool
`and`	bool	bool	bool
`or`	bool	bool	bool
`not`	bool	N/A	bool
− (unary)	int	N/A	int
− (unary)	float	N/A	float

Note: Python supports more complex behaviors through operator overloading and special methods (e.g., __add__, __lt__, etc.), allowing custom objects to interact with these operators in user-defined ways. This table covers only the most basic and common use cases.

3 Standard Libraries

The following libraries are always available in this language.

MISC Library

The following names are provided by the MISC library:

- `get_time()`: *primitive*, returns number of milliseconds elapsed since January 1, 1970 00:00:00 UTC
- `parse_int(s, i)`: *primitive*, interprets the *string* `s` as an integer, using the positive integer `i` as radix, and returns the respective value, see ECMAScript Specification, Section 18.2.5.
- `is_boolean(x)`, `is_float(x)`, `is_int(x)` `is_string(x)`, `is_function(x)`: *primitive*, returns `true` if the type of `x` matches the function name and `false` if it does not. We specify that `is_float` returns `true` for `NaN` and `Infinity`.
- `prompt(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 `prompt` suspends execution of the program until one of the two buttons is pressed. If the “OK” button is pressed, `prompt` returns the entered text as a string. If the “Cancel” button is pressed, `prompt` returns a non-string value.
- `print(x)`: *primitive*, displays the value `x` in the console[5]; returns the argument `a`.

[5]The notation used for the display of values is consistent with JSON, but also displays `undefined` and function objects.

- `print(x, s)`: *primitive*, displays the string `s`, followed by a space character, followed by the value `x` in the console[5]; returns the argument `x`.
- `error(x)`: *primitive*, displays the value `x` in the console[5] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `error(x, s)`: *primitive*, displays the string `s`, followed by a space character, followed by the value `x` in the console[5] with error flag. The evaluation of any call of `error` aborts the running program immediately.
- `str(x)`: *primitive*, returns a string that represents[5] the value `x`.

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

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 3.11.8 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*.