Specification of Python §1—2025 edition

Martin Henz, Wang Yuyang

National University of Singapore
School of Computing

May 30, 2025

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 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
statement	::=	*name* **=** *expression*	variable declaration
	|	**def** *name* **(** *names* **) :** *block*	function declaration
	|	**return** *expression*	return statement
	|	*if-statement*	conditional statement
	|	*expression*	expression statement
names	::=	ϵ | *name* (**,** *name*)...	name list
if-statement	::=	**if** *expression***:** *block* (**elif** *expression***:** *block*)... **else :** *block*	conditional statement
block	::=	*statement*...	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

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.
- Re-declaration variables or functions is not allowed. Once a variable or function is defined, it cannot be redefined with the same name in the same scope [2].

[2] Scope refers to the region of a program in which a particular name (such as a variable, function, or class) is defined and can be accessed. In other words, it determines the part of the program where you can use that name without causing a name error. Scope is determined by the program's structure (usually its lexical or textual layout) and governs the visibility and lifetime of variables and other identifiers.

In Python, the *scope* of a declaration is determined lexically: a variable declared inside a function is local to that function; if it is declared outside any function, it is global (i.e., module-level). Moreover, if a variable is declared in an enclosing function, it is available to inner functions (the enclosing scope), and if not found in these scopes, Python looks into the built-in scope.

Names

Names[3] start with _, or a letter[4] and contain only _, letters or digits[5]. Reserved words[6] 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.

Numbers

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

Integers (`int`)

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 (`float`)

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`)

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 *backquote-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[7] 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.

[3] In Python 3.13 Documentation, these names are called *identifiers*.
[4] By *letter* we mean Unicode letters (L) or letter numbers (Nl).
[5] 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.
[6] 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.
[7] 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*.

- `\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.

2 Dynamic Type Checking

Expressions evaluate to integers, floats, complex numbers, boolean values, strings or function values. Implementations of Source generate 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 Source'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	float	float
`+,-,*`	int	complex	complex
`+,-,*`	float	int, float	float
`+,-,*`	float	complex	complex
`+,-,*`	complex	int, float, complex	complex
`+`	string	string	string
`/`	int	int, float	float
`/`	int	complex	complex
`/`	float	int, float	float
`/`	float	complex	complex
`/`	complex	int, float, complex	complex
`**`	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
`%`	int	int	int
`%`	int	float	float
`%`	float	float	float
`==`	int,float,complex	int,float,complex	bool
`==`	string	string	bool
`!=`	int,float,complex	int,float,complex	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
`<=`	int,float	int,float	bool
`<=`	string	string	bool
`and`	bool	bool	bool
`or`	bool	bool	bool
`not`	bool		bool
`-`	int		int
`-`	float		float
`-`	complex		complex

3 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.

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 adopts a more controlled and minimalistic REPL behavior: 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
```

This design aligns with the pedagogical goals of Source Academy, reinforcing the idea that output should be intentional and explicit, helping students better understand the role of side effects and output operations.