Specification of Python §1—2025 edition

Authors

National University of Singapore
School of Computing

May 4, 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.

<table>
<tr><td>*program*</td><td>::=</td><td>*import-stmt... block*</td><td>program</td></tr>
<tr><td>*import-stmt*</td><td>::=</td><td>**from** *dotted-name* **import** *import-clause*</td><td>import statement</td></tr>
<tr><td>*dotted-name*</td><td>::=</td><td>*name* (**.** *name*)...</td><td>dotted identifier chain</td></tr>
<tr><td>*import-clause*</td><td>::=</td><td>*import-as-names*</td><td></td></tr>
<tr><td></td><td>|</td><td>**(** *import-as-names* **)**</td><td>wildcard or name list</td></tr>
<tr><td>*import-as-names*</td><td>::=</td><td>*import-as-name* (**,** *import-as-name*)...</td><td>clause name list</td></tr>
<tr><td>*import-as-name*</td><td>::=</td><td>*name* [**as** *name*]</td><td>clause name and alias</td></tr>
<tr><td>*statement*</td><td>::=</td><td>*name* **=** *expression*</td><td>variable declaration</td></tr>
<tr><td></td><td>|</td><td>**def** *name* **(** *names* **)** **:** *block*</td><td>function declaration</td></tr>
<tr><td></td><td>|</td><td>**return** *expression*</td><td>return statement</td></tr>
<tr><td></td><td>|</td><td>*if-statement*</td><td>conditional statement</td></tr>
<tr><td></td><td>|</td><td>*expression*</td><td>expression statement</td></tr>
<tr><td></td><td>|</td><td>**debugger()**</td><td>breakpoint</td></tr>
<tr><td>*names*</td><td>::=</td><td>ϵ | *name* (**,** *name*)...</td><td>name list</td></tr>
<tr><td>*if-statement*</td><td>::=</td><td>**if** *expression*: *block*
(**elif** *expression*: *block*)...
else **:** *block*</td><td>conditional statement</td></tr>
<tr><td>*block*</td><td>::=</td><td>*statement...*</td><td>block statement</td></tr>
<tr><td>*expression*</td><td>::=</td><td>*number*</td><td>primitive number expression</td></tr>
<tr><td></td><td>|</td><td>**true** | **false**</td><td>primitive boolean expression</td></tr>
<tr><td></td><td>|</td><td>*string*</td><td>primitive string expression</td></tr>
<tr><td></td><td>|</td><td>*name*</td><td>name expression</td></tr>
<tr><td></td><td>|</td><td>*expression binary-operator expression*</td><td>binary operator combination</td></tr>
<tr><td></td><td>|</td><td>*unary-operator expression*</td><td>unary operator combination</td></tr>
<tr><td></td><td>|</td><td>*expression binary-logical expression*</td><td>logical composition</td></tr>
<tr><td></td><td>|</td><td>*expression* **(** *expressions* **)**</td><td>function application</td></tr>
<tr><td></td><td>|</td><td>**lambda** (*name* | **(** *names* **)**) **:** *expression*</td><td>lambda expression</td></tr>
<tr><td></td><td>|</td><td>*expression* **if** *expression* **else** *expression*</td><td>conditional expression</td></tr>
<tr><td></td><td>|</td><td>**(** *expression* **)**</td><td>parenthesised expression</td></tr>
<tr><td>*binary-operator*</td><td>::=</td><td>**+** | **-** | ***** | **/** | ****** | **%** |</td><td></td></tr>
<tr><td></td><td>|</td><td>**==** | **!=** | **>** | **<** | **>=** | **<=**</td><td>binary operator</td></tr>
<tr><td>*unary-operator*</td><td>::=</td><td>**not** | **-**</td><td>unary operator</td></tr>
<tr><td>*binary-logical*</td><td>::=</td><td>**and** | **or**</td><td>*logicalcompositionsymbol*</td></tr>
<tr><td>*expressions*</td><td>::=</td><td>ϵ | *expression* (**,** *expression*)...</td><td>argument expressions</td></tr>
</table>

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

- Return statements are only allowed in bodies of functions.
- Line breaks within a statement must be explicitly continued using a backslash or be enclosed in parentheses, brackets, or braces. Python does not perform implicit statement termination like automatic semicolon insertion; each logical line—that is, each complete statement as interpreted by the interpreter, which may span multiple physical lines when properly continued—must be syntactically complete.
- 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].

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. All names that appear in import directives must be distinct, and must also be distinct from all top-level variables. The Source specifications do not specify how modules are programmed.

[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`.

[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.

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.
- `\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.
- `\ooo`: Represents the character with the octal value `ooo`.
- `\xhh`: Represents the character with the hexadecimal value `hh`.

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:

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

For compound functions, implementations need to check that the number of *expressions* matches the number of parameters.
The following table specifies what arguments 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.

[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*.

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 §x, 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: 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.