Python

Nguyễn Quản Bá Hồng*

September 22, 2022

Abstract

A short lecture on Python programming language.

Contents

1 Wikipedia's

1.1 Wikipedia/Python (Programming Language)

"Python is a high-level, general-purpose programmming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed & garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented & functional programming. It is often described as a "batteries included" language due to its comprehensive standard library. Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language & 1st released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 & introduced new features such as list comprehensions, cycle-detecting garbage collection, reference counting, & Unicode support. Python 3.0, released in 2008, was a major revision that is not completely backward-compatible with earlier versions. Python 2 was discontinued with version 2.7.18 in 2020. Python consistently ranks as 1 of the most popular programming languages." – Wikipedia/Python (programming language)

*Independent Researcher, Ben Tre City, Vietnam
e-mail: nguyenquanbahong@gmail.com; website: https://nqbh.github.io.

1.1.1 History of Python

"Main article: Wikipedia/history of Python. Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language, which was inspired by SETL, capable of exception handling & interfacing with the Amoeba operating system. Its implementation began in Dec 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until Jul 12, 2018, when he announced his "permanent vacation" from his responsibilities as Python's "benevolent dictator for life", a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. In Jan 2019, active Python core developers elected a 5-member Steering Council to lead the project.

Python 2.0 was released on Oct 16, 2000, with many major new features. Python 3.0, released on Dec 3, 2008, with many of its major features backported to Python 2.6.x & 2.7.x. Releases of Python 3 include the `2to3` utility, which automates the translation of Python 2 code to Python 3.

Python 2.7's end-of-life was initially set for 2015, then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3. No further security patches or other improvements will be released for it. With Python 2's end-of-life, only Python 3.6.x & later were supported. Later, support for 3.6 was also discontinued. In 2021, Python 3.9.2 & 3.8.8 were expedited as all versions of Python (including 2.7) had security issues leading to possible remote code execution & web cache poisoning.

In 2022, Python 3.10.4 & 3.9.12 were expedited & so were older releases including 3.8.13, & 3.7.13 because of many security issues. When Python 3.9.13 was released in May 2022, it was announced that the 3.9 series (joining the older series 3.8 & 3.7) will only receive security fixes going forward. On Sep 7, 2022, 4 new releases were made due to a potential denial-of-service attack: 3.10.7, 3.9.14, 3.8.14, & 3.7.14." – Wikipedia/Python (programming language)/history

1.1.2 Design philosophy & features

"Python is a multi-paradigm programming language. Object-oriented programming & structured programming are fully supported, & many of its features support functional programming & aspect-oriented programming (including metaprogramming & metaobjects [magic methods]). Many other paradigms are supported via extensions, including design by contract & logic programming. Python uses dynamic typing & a combination of reference counting & a cycle-detecting garbage collector for memory management. It uses dynamic name resolution (late binding), which binds method & variable names during program execution. Its design offers some support for functional programming in the Lisp tradition. It has `filter`, `map`, & `reduce` functions; list comprehensions, dictionaries, sets, & generator expressions. The standard library has 2 modules (`itertools` & `functools`) that implement functional tools borrowed from Haskell & Standard ML.

Its core philosophy is summarized in the document *The Zen of Python* (*PEP 20*), which includes aphorisms such as: • Beautiful is better than ugly. • Explicit is better than implicit. • Simple is better than complex. • Complex is better than complicated. • Readability counts.

Rather than building all of its functionality into its core, Python was designed to be highly extensible via modules. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications. Van Rossum's vision of a small core language with a large standard library & easily extensible interpreter stemmed from his frustrations with ABC, which espoused the opposite approach.

Python strives for a simpler, less-cluttered syntax & grammar while giving developers a choice in their coding methodology. In contrast to Perl's "there is more than 1 way to do it" motto, Python embraces a "there should be one – & preferably only 1 – obvious way to do it" philosophy. Alex Martelli, a Fellow at the Python Software Foundation & Python book author, wrote: "To describe something as 'clever' is *not* considered a compliment in the Python culture."

Python's developers strive to avoid premature optimization & reject patches to non-critical parts of the CPython reference implementation that would offer marginal increases in speed at the cost of clarity. When speed is important, a Python programmer can move time-critical functions to extension modules written in languages such as C; or use Pypy, a just-in-time compiler. Cython is also available, which translates a Python script into C & makes direct C-level API calls into the Python interpreter.

Python's developers aim for it to be fun to use. This is reflected in its name – a tribute to the British comedy group Monty Python – & in occasionally playful approaches to tutorials & reference materials, such as examples that refer to spam & eggs (a reference to a Monty Python sketch) instead of the standard foo & bar.

The programming language's name 'Python' came from the BBC Comedy series Monty Python's Flying Circus. Guido van Rossum thought he needed a name that was short, unique & slightly mysterious, & so, he decided to name the programming language 'Python'.

A common neologism in the Python community is *pythonic*, which has a wide range of meanings related to program style. "Pythonic" code may use Python idioms well, be natural or show fluency in the language, or conform with Python's minimalist philosophy & emphasis on readability. Code that is difficult to understand or reads like a rough transcription from another programming language is called *unpythonic*. Python users & admirers, especially those considered knowledgeable or experienced, are often referred to as *Pythonistas*." – Wikipedia/Python (programming language)/design philosophy & features

1.1.3 Syntax & semantics

"Main article: Wikipedia/Python syntax & semantics. Python is meant to be an easily readable language. Its formatting is visually uncluttered & often uses English keywords where other languages use punctuation. Unlike many other languages, it does not use curly brackets to delimit blocks, & semicolons after statements are allowed but rarely used. It has fewer syntactic exceptions & special cases than C or Pascal." – Wikipedia/Python (programming language)/syntax & semantics

1.1.3.1 Indentation. "Main article: Wikipedia/Python syntax & semantics/indentation. Python uses whitespace indentation, rather than curly brackets or keywords, to delimit blocks. An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block. Thus, the program's visual structure accurately represents its semantic structure. This feature is sometimes termed the off-side rule. Some other languages use indentation this way; but in most, indentation has no semantic meaning. The recommended indent size is 4 spaces." – Wikipedia/Python (programming language)/syntax & semantics/indentation

1.1.3.2 Statements & control flow. "Python's statements include: • The assignment statement, using a single equals sign `=`. • The `if` statement, which conditionally executes a block of code, along with `else` & `elif` (a contraction of else-if). • The `for` statement, which iterates over an iterable object, capturing each element to a local variable for use by the attached block. • The `while` statement, which executes a block of code as long as its condition is true. • The `try` statement, which allows exceptions raised in its attached code block to be caught & handled by `except` clauses (or new syntax `except*` in Python 3.11 for exception groups); it also ensures that clean-up code in a `finally` block is always run regardless of how the block exits. • The `raise` statement, used to raise a specified exception or re-raise a caught exception. • The `class` statement, which executes a block of code & attaches its local namespace to a class, for use in object-oriented programming. • The `def` statement, which defines a function or method. • The `with` statement, which encloses a code block within a context manager (e.g., acquiring a lock before it is run, then releasing the lock; or opening & closing a file), allowing resource-acquisition-is-initialization (RAII)-like behavior & replacing a common try/finally idiom. • The `break` statement, which exits a loop. • The `continue` statement, which skips the current iteration & continues with the next. • The `del` statement, which removes a variable – deleting the reference from the name to the value, & producing an error if the variable is referred to before it is redefined. • The `pass` statement, serving as a NOP, syntactically needed to create an empty code block. • The `assert` statement, used in debugging to check for conditions that should apply. • The `yield` statement, which returns a value from a generator function (& also an operator); used to implement coroutines. • The `return` statement, used to return a value from a function. • The `import` statement, used to import modules whose functions or variables can be used in the current program.

The assignment statement (`=`) binds a name as a reference to a separate, dynamically-allocated object. Variables may subsequently be rebound at any time to any object. In Python, a variable name is a generic reference holder without a fixed data type; however, it always refers to *some* object with a type. This is called dynamic typing – in contrast to statistically-typed languages, where each variable may contain only a value of a certain type.

Python does not support tail call optimization or 1st-class continuations, &, according to van Rossum, it never will. However, better support for coroutine-like functionality is provided by extending Python's generators. Before 2.5, generators were lazy iterators; data was passed unidirectionally out of the generator. From Python 2.5 on, it is possible to pass data back into a generator function; & from version 3.3, it can be passed through multiple stack levels." – Wikipedia/Python (programming language)/syntax & semantics/statements & control flow

1.1.3.3 Expressions. "Some Python expressions are similar to those in languages such as C & Java, while some are not: • Addition, subtraction, & multiplication are the same, but the behavior of division differs. There are 2 types of divisions in Python: floor division (or integer division) `//` & floating-point `/` division. Python also uses the `**` operator for exponentiation. • The `@` infix operator. It is intended to be used by libraries such as NumPy for matrix multiplication. • The syntax `:=`, called the "walrus operator", was introduced in Python 3.8. It assigns values to variables as part of a larger expression. • In Python, `==` compares by value, vs. Java, which compares numerics by value & objects by reference. Python's `is` operator may be used to compare object identities (comparison by reference), & comparisons may be chained – e.g., `a <= b <= c`. • Python uses `and`, `or`, & `not` as boolean operators rather than the symbolic `&&`, `||`, `!` in Java & C. • Python has a type of extension called a *list comprehension*, as well as a more general expression called a *generator expression*. • Anonymous functions are implemented using lambda expressions; however, there may be only 1 expression in each body. • Conditional expressions are written as `x if c else y` (different in order of operands from the `c ? x : y` operator common to many other languages). • Python makes a distinction between lists & tuples. Lists are written as `[1, 2, 3]`, are mutable, & cannot be used as the keys of dictionaries (dictionary keys must be immutable in Python). Tuples, written as `(1, 2, 3)`, are immutable & thus can be used as keys of dictionaries, provided all of the tuple's elements are immutable. The `+` operator can be used to concatenate 2 tuples, which does not directly modify their contents, but produces a new tuple containing the elements of both. Thus, given the variable t initially equal to `(1, 2, 3)`, executing `t = t + (4, 5)` 1st evaluates `t + (4, 5)`, which yields `(1, 2, 3, 4, 5)`, which is then assigned back to `t` – thereby effectively "modifying the contents" of `t` while conforming to the

immutable nature of tuple objects. Parentheses are optional for tuples in unambiguous contexts. • Python features *sequence unpacking* where multiple expressions, each evaluating to anything that can be assigned (to a variable, writable property, etc.) are associated in an identical manner to that forming tuple literals – &, as a whole, are put on the LHS of the equal sign in an assignment statement. The statement expects an *iterable* object on the RHS of the equal sign that produces the same number of values as the provided writable expressions; when iterated through them, it assigns each of the produced values to the corresponding expression on the left. • Python has a "string format" operator `%` that functions analogously to `printf` format strings in C – e.g., `"spam=%s eggs=%d" % ("blah", 2)` evaluates to `"spam=blah eggs=2"`. In Python 2.6+ & 3+, this was supplemented by the `format()` method of the `str` class, e.g., `"spam={0} eggs={1}".format("blah", 2)`. Python 3.6 added "f-strings": `spam = "blah"; eggs = 2; f'spam={spam} eggs={eggs}'`. • Strings in Python can be concatenated by "adding" them (with the same operator as for adding integers & floats), e.g. `"spam" + "eggs"` returns `"spameggs"`. If strings contain numbers, they are added as strings rather than integers, e.g., `"2" + "2"` returns `"22"`. • Python has various string literals: ◦ Delimited by *single/double quote marks*. Unlike in Unix shells, Perls, & Perl-influenced languages, single & double quote marks function identically. Both use the backslash (\) as an escape character. String interpolation became available in Python 3.6 as "formatted string literals". ◦ *Triple-quoted* (beginning & ending with 3 single or double quote marks), which may span multiple lines & function like here documents in shells, Perl, & Ruby. ◦ Raw string varieties, denoted by prefixing the string literal with `r`. Escape sequences are not interpreted; hence raw strings are useful where literal backslashes are common, such as regular expressions & Windows-style paths. (Compare "`@`-quoting" in C#.) • Python has array index & array slicing expressions in lists, denoted as `a[key]`, `a[start:stop]` or `a[start:stop:step]`. Indexes are zero-based, & negative indexes are relative to the end. Slices take elements from the *start* index up to, but not including, the *stop* index. The 3rd slice parameter, called *step* or *stride*, allows elements to be skipped & reversed. Slice indexes may be omitted – e.g., `a[:]` returns a copy of the entire list. Each element of a slice is a shallow copy.

In Python, a distinction between expressions & statements is rigidly enforced, in contrast to languages such as Common Lisp, Scheme, or Ruby. This leads to duplicating some functionality. E.g.: • List comprehensions vs. `for`-loops. • Conditional expressions vs. `if` blocks. • The `eval()` vs. `exec()` built-in functions (in Python 2, `exec` is a statement); the former is for expressions, the latter is for statements.

Statements cannot be a part of an expression – so list & other comprehensions or lambda expressions, all being expressions, cannot contain statements. A particular case is that an assignment statement such as `a = 1` cannot form part of the conditional expression of a conditional statement. This has the advantage of avoiding a classic C error of mistaking an assignment operator `=` for an equality operator `==` in conditions: `if (c = 1) { ... }` is syntactically valid (but probably unintended) C code, but `if c = 1: ...` causes a syntax error in Python." – Wikipedia/Python (programming language)/syntax & semantics/expressions

1.1.3.4 Methods. "Methods on objects are functions attached to the object's class; the syntax `instance.method(argument)` is, for normal methods & functions, syntactic sugar for `Class.method(instance, argument)`. Python methods have an explicit `self` parameter to access instance data, in contrast to the implicit self (or `this`) in some other object-oriented programming language (e.g., C++, Java, Objective-C, Ruby). Python also provides methods, often called *dunder methods* (due to their names beginning & ending with double-underscores), to allow user-defined classes to modify how they are handled by native operations including length, comparison, in arithmetic operations & type conversion." – Wikipedia/Python (programming language)/syntax & semantics/methods

1.1.3.5 Typing. "Python uses duck typing & has typed objects buts untyped variable names. Type constraints are not checked at compile time; rather, operations on an object may fail, signifying that it is not of a suitable type. Despite being dynamically typed, Python is strongly typed, forbidding operations that are not well-defined (e.g., adding a number to string) rather than silently attempting to make sense of them.

Python allows programmers to define their own types using classes, most often used for object-oriented programming. New instances of classes are constructed by calling the class (e.g., `SpamClass()` or `EggsClass()`), & the classes are instances of the metaclass `type` (itself an instance of itself), allowing metaprogramming & reflection.

Before version 3.0, Python had 2 kinds of classes (both using the same syntax): *old-style* & *new-style*, current Python versions only support the semantics new style.

The long-term plan is to support gradual typing. Python's syntax allows specifying static types, but they are checked in the default implementation, CPython. An experimental optional static typechecker, *mypy*, supports compile-time type checking. Table: Summary of Python 3's built-in types." – Wikipedia/Python (programming language)/syntax & semantics/typing

1.1.3.6 Arithmetic operations. "Python has the usual symbols for arithmetic operators (`+`, `-`, `*`, `/`), the floor division operator `//` & the modulo operation `%` (where the remainder can be negative, e.g., `4 % -3 == -2`). It also has `**` for exponentiation, e.g., `5**3 == 125` & `9**0.5 == 3.0`, & a matrix-multiplication operator `@`. These operators work like in traditional math; with the same precedence rules, the operators infix (`+` & `-` can also be unary to represent positive & negative numbers respectively).

The division between integers produces floating-point results. The behavior of division has changed significantly over time: • Current Python (i.e. since 3.0) changed `/` to always be floating-point division, e.g., `5/2 == 2.5`. • The floor division `//` operator was introduced. So `7//3 == 2`, `-7//3 == -3`, `7.5//3 == 2.0`, & `-7.5//3 == -3.0`. Adding `from __future__ import division` causes a module used in Python 2.7 to use Python 3.0 rules for division.

In Python terms, `/` is *true division* (or simply *division*), & `//` is *floor division*. `/` before version 3.0 is *classic division*.

Rounding towards negative infinity, though different from most languages, adds consistency. E.g., it means that the equation `(a + b)//b == a//b + 1` is always true. It also means that the equation `b*(a//b) + a%b == a` is valid for both positive & negative values of `a`. However, maintaining the validity of this equation means that while the result of `a%b` is, as expected, in the half-open interval $[0, b)$, where `b` is a positive integer, it has to lie in the interval $(b, 0]$ when `b` is negative.

Python provides a `round` function for rounding a float to the nearest integer. For tie-breaking, Python 3 uses round to even: `round(1.5)` & `round(2.5)` both produce `2`. Versions before 3 used round-away-from-zero: `round(0.5)` is `1.0`, `round(-0.5)` is `-1.0`.

Python allows boolean expressions with multiple equality relations in a manner that is consistent with general use in mathematics. E.g., the expression `a < b < c` tests whether `a` is less than `b` & `b` is less than `c`. C-derived languages interpret this expression differently: in C, the expression would 1st evaluate `a < b`, resulting in 0 or 1, & that result would then be compared with `c`.

Python uses arbitrary-precision arithmetic for all integer operations. The `Decimal` type/class in the `decimal` module provides decimal floating-point numbers to a pre-defined arbitrary precision & several rounding modes. The `Fraction` class in the `fractions` module provides arbitrary precision for rational numbers.

Due to Python's extensive mathematics library, & the 3rd-party library NumPy that further extends the native capabilities, it is frequently used as a scientific scripting language to aid in problems such as numerical data processing & manipulation." – Wikipedia/Python (programming language)/syntax & semantics/arithmetic operations

1.1.4 Libraries

1.1.5 Development environments

1.1.6 Implementations

1.1.6.1 Reference implementation.

1.1.6.2 Other implementations.

1.1.6.3 Unsupported implementations.

1.1.6.4 Cross-compilers to other languages.

1.1.6.5 Performance.

1.1.7 Development

1.1.8 API documentation generators

1.1.9 Naming

1.1.10 Popularity

References. Knuth, 1997; Matthes, 2019.

References

Knuth, Donald Ervin (1997). *The Art of Computer Programming. Volume 1: Fundamental Algorithms*. 3rd edition. Addison-Wesley Professional, pp. xx+652.

Matthes, Eric (2019). *Python Crash Course: A Hands-on, Project-based Introduction to Programming*. 2nd edition. No Starch Press, pp. xxxvi+506.