# Python syntax and semantics

The **syntax of the Python programming language** is the set of rules that defines how a Python program will be written and interpreted (by both the runtime system and by human readers). Python was designed to be a highly readable language.[1] It has a relatively uncluttered visual layout and uses English keywords frequently where other languages use punctuation. Python aims towards simplicity and generality in the design of its syntax, encapsulated in the mantra "There should be one — and preferably only one — obvious way to do it", from "The Zen of Python".[2]

This mantra is deliberately opposed to the Perl and Ruby mantra, "there's more than one way to do it".

## Indentation

Python uses whitespace to delimit program blocks, following the off-side rule. Its uncommon block marking convention is a feature that many programmers, otherwise unfamiliar with Python, have heard of. Python borrows this feature from its predecessor ABC — instead of punctuation or keywords, it uses indentation to indicate the run of a block.

In so-called "free-format" languages, that use the block structure derived from ALGOL, blocks of code are set off with braces (`{ }`) or keywords. In most coding conventions for these languages, programmers conventionally indent the code within a block, to visually set it apart from the surrounding code (prettyprinting).

Consider a function, `foo`, which is passed a single parameter, `x`, and if the parameter is 0 will call `bar` and `baz`, otherwise it will call `qux`, passing `x`, and also call itself recursively, passing `x-1` as the parameter. Here are implementations of this function in both C and Python:

`foo` function in C with K&R indent style:

<syntaxhighlight lang="c"> void foo(int x) {

```
 if (x == 0) {
     bar();
     baz();
 } else {
     qux(x);
     foo(x - 1);
 }
```

} </syntaxhighlight>

`foo` function in Python: <syntaxhighlight lang="python"> def foo(x):

```
 if x == 0:
     bar()
     baz()
 else:
     qux(x)
     foo(x - 1)
```

</syntaxhighlight>

Python mandates a convention that programmers in ALGOL-style languages often follow. Moreover, in free-form syntax, since indentation is ignored, good indentation cannot be enforced by an interpreter or compiler. Incorrectly indented code can be understood by human reader differently than does a compiler or interpreter. For example:

Misleading indentation in C: <syntaxhighlight lang="c"> for (i = 0; i < 20; ++i)

```
    a();
    b();
```

c(); </syntaxhighlight>

This code was intended to call functions `a()` and `b()` 20 times. However, the iterated code block is just `{a();}`. The code calls `a()` 20 times, and then calls `b()` and `c()` only one time each. Later readers of the code may be fooled by the misalignment of the calls to functions b and c.

Both space characters and tab characters are currently accepted as forms of indentation in Python. Since many tools do not visually distinguish them, mixing spaces and tabs can create bugs that take specific efforts to find (a perennial suggestion among Python users has been removing tabs as block markers — except, of course, among those Python users who propound removing spaces instead). Moreover, formatting routines which remove whitespace — for instance, many Internet forums — can destroy the syntax of a Python program, whereas a program in a bracketed language would merely become more difficult to read.

Many popular code editors handle Python's indentation conventions seamlessly, sometimes after a configuration option is enabled.

## Data structures

Since Python is a dynamically typed language, Python *values,* not variables, carry type. This has implications for many aspects of the way the language functions.

All variables in Python hold references to objects, and these references are passed to functions; a function cannot change the value of variable references in its calling function. Some people (including Guido van Rossum himself) have called this parameter-passing scheme "Call by object reference."

Among dynamically typed languages, Python is moderately type-checked. Implicit conversion is defined for numeric types (as well as booleans), so one may validly multiply a complex number by a long integer (for instance) without explicit casting. However, there is no implicit conversion between (e.g.) numbers and strings; a string is an invalid argument to a mathematical function expecting a number.

### Base types

Python has a broad range of basic data types. Alongside conventional integer and floating point arithmetic, it transparently supports arbitrary-precision arithmetic, complex numbers, and decimal floating point numbers.

Python supports a wide variety of string operations. Strings in Python are immutable, so a string operation such as a substitution of characters, that in other programming languages might alter a string in place, returns a new string in Python. Performance considerations sometimes push for using special techniques in programs that modify strings intensively, such as joining character arrays into strings only as needed.

### Collection types

One of the very useful aspects of Python is the concept of *collection* (or *container*) types. In general a collection is an object that contains other objects in a way that is easily referenced or *indexed*. Collections come in two basic forms: *sequences* and *mappings*.

The ordered sequential types are lists (dynamic arrays), tuples, and strings. All sequences are indexed positionally (0 through *length* − 1) and all but strings can contain any type of object, including multiple types in the same sequence. Both strings and tuples are immutable, making them perfect candidates for dictionary keys (see below). Lists, on the other hand, are mutable; elements can be inserted, deleted, modified, appended, or sorted in-place.

On the other side of the collections coin are mappings, which are unordered types implemented in the form of *dictionaries* which "map" a set of immutable keys, to corresponding elements much like a mathematical function.

The keys in a dictionary must be of an immutable Python type such as an integer or a string. For example, one could define a dictionary having a string `"toast"` mapped to the integer 42 or vice versa. This is done under the covers via a hash function which makes for faster lookup times, but is also the culprit for a dictionary's lack of order and is the reason mutable objects (i.e. other dictionaries or lists) cannot be used as keys. Dictionaries are also central to the internals of the language as they reside at the core of all Python objects and classes: the mapping between variable names (strings) and the values which the names reference is stored as a dictionary (see Object system). Since these dictionaries are directly accessible (via an object's `__dict__` attribute), metaprogramming is a straightforward and natural process in Python.

A set collection type was added to the core language in version 2.4. A set is an unindexed, unordered collection that contains no duplicates, and implements set theoretic operations such as union, intersection, difference, symmetric difference, and subset testing. There are two types of sets: `set` and `frozenset`, the only difference being that `set` is mutable and `frozenset` is immutable. Elements in a set must be hashable and immutable. Thus, for example, a `frozenset` can be an element of a regular `set` whereas the opposite is not true.

Python also provides extensive collection manipulating abilities such as built in containment checking and a generic iteration protocol.

### Object system

In Python, everything is an object, even classes. Classes, as objects, have a class, which is known as their metaclass. Python also supports multiple inheritance and mixins (see also MixinsForPython).

The language supports extensive introspection of types and classes. Types can be read and compared—types are instances of `type`. The attributes of an object can be extracted as a dictionary.

Operators can be overloaded in Python by defining special member functions—for instance, defining `__add__` on a class permits one to use the `+` operator on members of that class.

## Literals

### Strings

Python has various kinds of string literals.

#### Normal string literals

Either single or double quotes can be used to quote strings. Unlike in Unix shell languages, Perl or Perl-influenced languages such as Ruby or Groovy, single quotes and double quotes function identically, i.e. there is no string interpolation of *$foo* expressions. However, interpolation can be done using the % string-format operator, e.g. the Perl statement <syntaxhighlight lang="Perl"> print "I just printed $num pages to the printer $printer\n" </syntaxhighlight>

is equivalent to the Python statement

<syntaxhighlight lang="Python"> print("I just printed %s pages to the printer %s" % (num, printer)) </syntaxhighlight>

However, the current standard for this sort of string formatting is to use the *format* method of strings:

<syntaxhighlight lang="Python"> print("I just printed {0} pages to the printer {1}".format(num, printer)) </syntaxhighlight>

### Multi-line string literals

There are also multi-line strings, which begin and end with a series of three single or double quotes and function like here documents in Perl and Ruby.

A simple example with variable interpolation (using the % string-format operator) is:

<syntaxhighlight lang="Python"> print("""Dear %(recipient)s,

I wish you to leave Sunnydale and never return.

Not Quite Love, %(sender)s """ % {'sender': 'Buffy the Vampire Slayer', 'recipient': 'Spike'}) </syntaxhighlight>

### Raw strings

Finally, all of the previously-mentioned string types come in "raw" varieties (denoted by placing a literal *r* before the opening quote), which do no backslash-interpolation and hence are very useful for regular expressions; compare "@-quoting" in C#. Raw strings were originally included specifically for regular expressions. Due to limitations of the tokenizer, raw strings may not have a trailing backslash.[3] Creating a raw string holding a Windows path ending with a backslash requires some variety of workaround (commonly, using forward slashes instead of backslashes, since Windows accepts both).

Examples are

<syntaxhighlight lang="Python">

1. A Windows path, even raw strings cannot end in a backslash

r"C:\Foo\Bar\Baz\ ".rstrip()

1. A regular expression matching a quoted string with possible backslash quoting

r'"([^"\\]|\\.)*"'

1. Reverse the arguments in a two-arg function call, e.g. foo(2, bar) -> foo(bar, 2);
2. will fail if either argument has parens or commas in it

re.sub(r'\(([^,]*?),([^,]*?)\)', r'(\2, \1)', code) </syntaxhighlight>

## Numbers

Numeric literals in Python are of the normal sort, e.g. `0`, `-1`, `3.4`, `3.5e-8`.

Python has arbitrary-length integers and automatically increases the storage size as necessary. Prior to Python version 3, there were two kinds of integral numbers: traditional fixed size integers and "long" integers of arbitrary range. The conversion to "long" integers was performed automatically when required, and thus the programmer usually didn't have to be aware of the two integral types. In newer language versions the fixed-size integers are completely gone.

Python supports normal floating point numbers, which are created when a dot is used in a literal (e.g. `1.1`), when an integer and a floating point number are used in an expression, or as a result of some mathematical operations ("true division" via the `/ operator, or exponentiation with a negative exponent).`

Python also supports complex numbers natively. Complex numbers are indicated with the `J` or `j` suffix, e.g. `3 + 4j`.

### Lists, tuples, sets, dictionaries

Python has syntactic support for the creation of container types.

Lists (class `list`) are mutable sequences of items of arbitrary types, and can be created either with the special syntax <syntaxhighlight lang="Python"> a_list = [1, 2, 3, "a dog"] </syntaxhighlight> or using normal object creation <syntaxhighlight lang="Python"> a_second_list = list() a_second_list.append(4) a_second_list.append(5) </syntaxhighlight>

**Tuples** (class `tuple`) are immutable sequences of items of arbitrary types. There is also a special syntax to create tuples <syntaxhighlight lang="Python"> a_tuple = 1, 2, 3, "four" </syntaxhighlight> Although tuples are created by separating items with commas, the whole construct is usually wrapped in parentheses to increase readability. An empty tuple is denoted by `()`.

**Sets** (class `set`) are mutable containers of items of arbitrary types. The items are not ordered, but sets support iteration over the items. A syntax for set creation appeared in Python 2.7/3.0 <syntaxhighlight lang="Python"> some_set = {0, (), False} </syntaxhighlight> In earlier Python versions, sets would be created by calling initializing the set class with a list argument. Python sets are very much like mathematical sets, and support operations like set intersection and union.

Python also features a `frozenset` class for immutable sets.

**Dictionaries** (class `dict`) are mutable mappings tying keys and corresponding values. Python has special syntax to create dictionaries (`{key: value}`) <syntaxhighlight lang="Python"> a_dictionary = {"key 1":"value 1", 2:3, 4:[]} </syntaxhighlight> The dictionary syntax is similar to the set syntax, the difference is the presence of colons. The empty literal `{}` results in an empty dictionary rather than an empty set, which is instead created using the non-literal constructor: `set()`.

## Operators

### Arithmetic

Python includes the `+`, `-`, `*`, `/`, `%` (modulus), and `**` (exponentiation) operators, with their usual mathematical precedence.

Traditionally, `x / y` performed integer division if both `x` and `y` were integers (returning the floor of the quotient), and returned a float if either was a float. However, because Python is a dynamically-typed language, it was not always possible to tell which operation was being performed, which often led to subtle bugs. For example, with

<syntaxhighlight lang="Python">def mean(seq):

```
   return sum(seq) / len(seq)</syntaxhighlight>
```

A call to `mean([3.0, 4.0])` would return 3.5, but `mean([3, 4])` would return 3. If this was not the intended behavior, it was necessary to use a workaround such as

<syntaxhighlight lang="Python">def mean(seq):

```
   return float(sum(seq)) / len(seq)</syntaxhighlight>
```

To avoid this issue, a proposal [4] was made to change the behavior of the Python division operator. In Python 2.2, a new operator `//` was introduced for floor division, both for integer and floating-point arguments. The `/` operator was changed so that the quotient of two integers returned a float, but for backwards compatibility, this behavior had to be explicitly requested until Python 3.0.

### Comparison operators

The basic comparison operators such as `==, <, >=, and so forth are used on all manner of values. Numbers, strings, sequences, and mappings can all be compared. Although disparate types (such as a str and an int) are defined to have a consistent relative ordering, this is considered a historical design quirk and will no longer be allowed in Python 3.0.`

Chained comparison expressions such as `a < b < c` have roughly the meaning that they have in mathematics, rather than the unusual meaning found in C and similar languages. The terms are evaluated and compared in order. The operation has short-circuit semantics, meaning that evaluation is guaranteed to stop as soon as a verdict is clear: if `a < b` is false, `c` is never evaluated as the expression cannot possibly be true anymore.

For expressions without side effects, `a < b < c` is equivalent to `a < b and b < c`. However, there is a substantial difference when the expressions have side effects. `a < f(x) < b` will evaluate `f(x)` exactly once, whereas `a < f(x) and f(x) < b` will evaluate it twice if the value of `a` is less than `f(x)` and once otherwise.

### Logical operators

Python 2.2 and earlier does not have an explicit boolean type. In all versions of Python, boolean operators treat zero values or empty values such as `""`, `0`, `None`, `0.0`, `[]`, and `{}` as false, while in general treating non-empty, non-zero values as true. In Python 2.2.1 the boolean constants `True` and `False` were added to the language (subclassed from 1 and 0). The binary comparison operators such as `==` and `>` return either `True` or `False`.

The boolean operators `and` and `or` use minimal evaluation. For example, `y == 0 or x/y > 100` will never raise a divide-by-zero exception. Note that these operators return the value of the last operand evaluated, rather than `True` or `False`. Thus the expression `(4 and 5)` evaluates to `5`, and `(4 or 5)` evaluates to `4`.

## Functional programming

As mentioned above, another strength of Python is the availability of a functional programming style. As may be expected, this makes working with lists and other collections much more straightforward.

### List comprehensions

One such construction is the list comprehension, as seen here in calculating the first five powers of two:

<syntaxhighlight lang="python"> powers_of_two = [2**n for n in range(1, 6)] </syntaxhighlight>

The Quicksort algorithm can be expressed elegantly, albeit inefficiently, using list comprehensions:

<syntaxhighlight lang="python"> def qsort(L):

```
if L == []:
    return []
pivot = L[0]
return (qsort([x for x in L[1:] if x < pivot]) +
        [pivot] +
        qsort([x for x in L[1:] if x >= pivot]))
```

</syntaxhighlight>

## First-class functions

In Python, functions are first-class objects that can be created and passed around dynamically.

Python's limited support for anonymous functions is the `lambda` construct. Since the availability of full anonymous functions is non-existent, the main use for lambda functions is named functions. Lambdas are limited to containing expressions rather than statements, although control flow can still be implemented less elegantly within lambda by using short-circuiting.[5]

## Closures

Python has had support for lexical closures since version 2.2. Here's an example: <syntaxhighlight lang="python"> def derivative(f, dx):

```
"""Return a function that approximates the derivative of f
using an interval of dx, which should be appropriately small.
"""
def function(x):
    return (f(x + dx) - f(x)) / dx
return function
```

</syntaxhighlight>

Python's syntax, though, sometimes leads programmers of other languages to think that closures are not supported. Variable scope in Python is implicitly determined by the scope in which one assigns a value to the variable, unless scope is explicitly declared with `global` or `nonlocal`.[6]

## Generators

Introduced in Python 2.2 as an optional feature and finalized in version 2.3, generators are Python's mechanism for lazy evaluation of a function that would otherwise return a space-prohibitive or computationally intensive list.

This is an example to lazily generate the prime numbers:

<syntaxhighlight lang="python"> from itertools import count

def generate_primes(stop_at=0):

```
primes = []
for n in count(2):
    if 0 < stop_at < n:
        return # raises the StopIteration exception
    composite = False
    for p in primes:
        if not n % p:
            composite = True
            break
        elif p**2 > n:
            break
    if not composite:
        primes.append(n)
        yield n
```

</syntaxhighlight>

To use this function simply call, e.g.:

<syntaxhighlight lang="python"> for i in generate_primes(): # iterate over ALL primes

```
if i > 100:
    break
print(i)
```

</syntaxhighlight>

The definition of a generator appears identical to that of a function, except the keyword `yield` is used in place of `return`. However, a generator is an object with persistent state, which can repeatedly enter and leave the same scope. A generator call can then be used in place of a list, or other structure whose elements will be iterated over. Whenever the `for` loop in the example requires the next item, the generator is called, and yields the next item.

Generators don't have to be infinite like the prime-number example above. When a generator terminates, an internal exception is raised which indicates to any calling context that there are no more values. A `for` loop or other iteration will then terminate.

## Generator expressions

Further information: List comprehension

Introduced in Python 2.4, generator expressions are the lazy evaluation equivalent of list comprehensions. Using the prime number generator provided in the above section, we might define a lazy, but not quite infinite collection.

<syntaxhighlight lang="python"> from itertools import islice

primes_under_million = (i for i in generate_primes() if i < 1000000) two_thousandth_prime = islice(primes_under_million, 2000, 2001).next() </syntaxhighlight>

Most of the memory and time needed to generate this many primes will not be used until the needed element is actually accessed. Unfortunately, you cannot perform simple indexing and slicing of generators, but must use the *itertools* modules or "roll your own" loops. In contrast, a list comprehension is functionally equivalent, but is *greedy* in performing all the work:

<syntaxhighlight lang="python"> primes_under_million = [i for i in generate_primes(2000000) if i < 1000000] two_thousandth_prime = primes_under_million[1999] </syntaxhighlight>

The list comprehension will immediately create a large list (with 78498 items, in the example, but transiently creating a list of primes under two million), even if most elements are never accessed. The generator comprehension is more parsimonious.

## Dictionary and set comprehensions

While lists and generators had comprehensions/expressions, in Python versions older than 2.7 the other Python built-in collection types (dicts and sets) had to be kludged in using lists or generators:

<syntaxhighlight lang="python">>>> dict((n, n*n) for n in range(5)) {0: 0, 1: 1, 2: 4, 3: 9, 4: 16} </syntaxhighlight>

Python 2.7 and 3.0 unify all collection types by introducing dict and set comprehensions, similar to list comprehensions:

<syntaxhighlight lang="python">>>> [ n*n for n in range(5) ] # regular list comprehension [0, 1, 4, 9, 16] >>> >>> { n*n for n in range(5) } # set comprehension {0, 1, 4, 16, 9} >>> >>> { n: n*n for n in range(5) } # dict comprehension {0: 0, 1: 1, 2: 4, 3: 9, 4: 16} </syntaxhighlight>

## Objects

Python supports most object oriented programming techniques. It allows polymorphism, not only within a class hierarchy but also by duck typing. Any object can be used for any type, and it will work so long as it has the proper methods and attributes. And everything in Python is an object, including classes, functions, numbers and modules. Python also has support for metaclasses, an advanced tool for enhancing classes' functionality. Naturally, inheritance, including multiple inheritance, is supported. It has limited support for private variables using name mangling. See the "Classes" section of the tutorial [7] for details. Many Python users don't feel the need for private variables, though. The slogan "We're all consenting adults here" is used to describe this attitude.[8] Some consider information hiding to be unpythonic, in that it suggests that the class in question contains unaesthetic or ill-planned internals. However, the strongest argument for name mangling is prevention of unpredictable breakage of programs: introducing a new public variable in a superclass can break subclasses if they don't use "private" variables.

From the tutorial: *As is true for modules, classes in Python do not put an absolute barrier between definition and user, but rather rely on the politeness of the user not to "break into the definition."*

OOP doctrines such as the use of accessor methods to read data members are not enforced in Python. Just as Python offers functional-programming constructs but does not attempt to demand referential transparency, it offers an object system but does not demand OOP behavior. Moreover, it is always possible to redefine the class using *properties* so that when a certain variable is set or retrieved in calling code, it really invokes a function call, so that `spam.eggs = toast` might really invoke `spam.set_eggs(toast)`. This nullifies the practical advantage of accessor functions, and it remains OOP because the property `eggs` becomes a legitimate part of the object's interface: it need not reflect an implementation detail.

In version 2.2 of Python, "new-style" classes were introduced. With new-style classes, objects and types were unified, allowing the subclassing of types. Even entirely new types can be defined, complete with custom behavior for infix operators. This allows for many radical things to be done syntactically within Python. A new method resolution order [9] for multiple inheritance was also adopted with Python 2.3. It is also possible to run custom code while accessing or setting attributes, though the details of those techniques have evolved between Python versions.

### Properties

Properties allow specially defined methods to be invoked on an object instance by using the same syntax as used for attribute access. An example of a class defining some properties is: <syntaxhighlight lang=Python> class MyClass(object):

```
def get_a(self):
   return self._a + 1
def set_a(self, value):
   self._a = value - 1
a = property(get_a, set_a, doc="Off by one a")
```

1. Python 2.6 style

class MyClass(object):

```
@property
def a(self):
   return self._a + 1
@a.setter # makes the property writable
def a(self, value):
   self._a = value - 1
```

</syntaxhighlight>

### Descriptors

A class that defines one or more of the special methods __get__(self,instance,owner), __set__(self,instance,value), __delete__(self,instance) can be used as a descriptor. Creating an instance of a descriptor as a class member of a second class makes the instance a property of the second class.

### Class and Static Methods

Python allows the creation of class methods and static method via the use of the @classmethod and @staticmethod decorators. The first argument to a class method is the class object instead of the self reference to the instance. A static method has no special first argument. Neither the instance, nor the class object is passed to a static method.

## Exceptions

Python supports (and extensively uses) exception handling as a means of testing for error conditions and other "exceptional" events in a program. Indeed, it is even possible to trap the exception caused by a syntax error.

Python style calls for the use of exceptions whenever an error condition might arise. Rather than testing for access to a file or resource before actually using it, it is conventional in Python to just go ahead and try to use it, catching the exception if access is rejected.

Exceptions can also be used as a more general means of non-local transfer of control, even when an error is not at issue. For instance, the Mailman mailing list software, written in Python, uses exceptions to jump out of deeply-nested message-handling logic when a decision has been made to reject a message or hold it for moderator approval.

Exceptions are often used as an alternative to the `if`-block, especially in threaded situations. A commonly-invoked motto is EAFP, or "It is Easier to Ask for Forgiveness than Permission,"[10] which is attributed to Grace Hopper[11] [12] In this first code sample, there is an explicit check for the attribute (i.e., "asks permission"):

<syntaxhighlight lang="python"> if hasattr(spam, 'eggs'):

```
ham = spam.eggs
```

else:

```
handle_error()
```

</syntaxhighlight>

This second sample follows the EAFP paradigm:

<syntaxhighlight lang="python"> try:

```
ham = spam.eggs
```

except AttributeError:

```
handle_error()
```

</syntaxhighlight>

These two code samples have the same effect, although there will be performance differences. When `spam` has the attribute `eggs`, the EAFP sample will run faster. When `spam` does not have the attribute `eggs` (the "exceptional" case), the EAFP sample will run slower. The Python profiler [13] can be used in specific cases to determine performance characteristics. If exceptional cases are rare, then the EAFP version will have superior average performance than the alternative. In addition, it avoids the whole class of time-of-check-to-time-of-use (TOCTTOU) vulnerabilities, other race conditions,[12] [14] and is compatible with duck typing.

## Comments and docstrings

Python has two ways to annotate Python code. One is by using comments to indicate what some part of the code does. Comments begin with the hash character ("#") and are terminated by the end of line. Python does not support comments that span more than one line.

Commenting a piece of code: <syntaxhighlight lang="python"> def getline():

```
   return sys.stdin.readline()       # Get one line and return it
```

</syntaxhighlight>

The other way is to use docstrings (documentation string), that is a string that is located alone without assignment as the first line within a module, class, method or function. Such strings can be delimited with `"` or `'` for single line strings, or may span multiple lines if delimited with either `"""` or `'''` which is Python's notation for specifying multi-line strings. However, the style guide for the language specifies that triple double quotes (`"""`) are preferred for both single and multi-line docstrings.

Single line docstring: <syntaxhighlight lang="python"> def getline():

```
   """Get one line from stdin and return it."""
   return sys.stdin.readline()
```

</syntaxhighlight>

Multi-line docstring: <syntaxhighlight lang="python"> def getline():

```
   """Get one line
      from stdin
      and return it."""
   return sys.stdin.readline()
```

</syntaxhighlight>

Docstrings can be as large as the programmer wants and contain line breaks. In contrast with comments, docstrings are themselves Python objects and are part of the interpreted code that Python runs. That means that a running program can retrieve its own docstrings and manipulate that information. But the normal usage is to give other programmers information about how to invoke the object being documented in the docstring.

There are tools available that can extract the docstrings to generate an API documentation from the code. Docstring documentation can also be accessed from the interpreter with the `help()` function, or from the shell with the pydoc command `pydoc`.

The doctest standard module uses interactions copied from Python shell sessions into docstrings, to create tests.

## Decorators

A decorator is any callable Python object that is used to modify a function, method or class definition. A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition. Python decorators were inspired in part by Java annotations, and have a similar syntax; the decorator syntax is pure syntactic sugar, using `@` as the keyword:

<syntaxhighlight lang="python"> @viking_chorus def menu_item():

```
   print("spam")
```

</syntaxhighlight>

is equivalent to

<syntaxhighlight lang="python"> def menu_item():

```
    print("spam")
```

menu_item = viking_chorus(menu_item) </syntaxhighlight>

Decorators are a form of metaprogramming; they enhance the action of the function or method they decorate. For example, in the above sample, `viking_chorus` might cause `menu_item` to be run 8 times (see Spam sketch) for each time it is called:

<syntaxhighlight lang="python"> def viking_chorus(myfunc):

```
    def inner_func(*args, **kwargs):
        for i in range(8):
            myfunc(*args, **kwargs)
    return inner_func
```

</syntaxhighlight>

Canonical uses of function decorators are for creating class methods or static methods, adding function attributes, tracing, setting pre- and postconditions, and synchronisation,[15] but can be used for far more besides, including tail recursion elimination,[16] memoization and even improving the writing of decorators.[17]

Decorators can be chained by placing several on adjacent lines:

<syntaxhighlight lang="python"> @invincible @favorite_color("Blue") def black_knight():

```
    pass
```

</syntaxhighlight>

is equivalent to

<syntaxhighlight lang="python"> def black_knight():

```
    pass
```

black_knight = invincible(favorite_color("Blue")(black_knight)) </syntaxhighlight>

or, using intermediate variables

<syntaxhighlight lang="python"> def black_knight():

```
    pass
```

blue_decorator = favorite_color("Blue") decorated_by_blue = blue_decorator(black_knight) black_knight = invincible(decorated_by_blue) </syntaxhighlight>

In the above example, the `favorite_color` decorator factory takes an argument. Decorator factories must return a decorator, which is then called with the object to be decorated as its argument: <syntaxhighlight lang="python"> def favorite_color(color):

```
    def decorator(func):
        def wrapper():
            print(color)
            func()
        return wrapper
    return decorator
```

</syntaxhighlight> This would then decorate the `black_knight` function such that the color, "Blue", would be printed prior to the `black_knight` function running.

In Python prior to version 2.6, decorators apply to functions and methods, but not to classes. Decorating a (dummy) `__new__` method can modify a class, however.[18] Class decorators are supported[19] starting with Python 2.6.

Despite the name, Python decorators are not an implementation of the decorator pattern. The decorator pattern is a design pattern used in statically typed object-oriented programming languages to allow functionality to be added to objects at run time; Python decorators add functionality to functions and methods at definition time, and thus are a higher-level construct than decorator-pattern classes. The decorator pattern itself is trivially implementable in Python, because the language is duck typed, and so is not usually considered as such.

See also

Advice in Lisp.

## Easter eggs

Users of curly bracket programming languages, such as C or Java, sometimes expect or wish Python to follow a block-delimiter convention. Brace-delimited block syntax has been repeatedly requested, and consistently rejected by core developers. The Python interpreter contains an easter egg that summarizes its developers' feelings on this issue. The code `from __future__ import braces` raises the exception `SyntaxError: not a chance`. The `__future__` module is normally used to provide features from future versions of Python.

Another hidden message, The Zen of Python (a summary of Python philosophy), is displayed when trying to `import this`.

The message `Hello world...` is printed when the import statement `import __hello__` is used.

An `antigravity` module was added to Python 2.7 and 3.0. Importing it opens a web browser to an xkcd comic that portrays a humorous fictional use for such a module, intended to demonstrate the ease with which Python modules enable additional functionality.[20]

## References

[1] "Readability counts." - PEP 20 - The Zen of Python (http://www.python.org/dev/peps/pep-0020/)

[2] "PEP 20 - The Zen of Python" (http://www.python.org/dev/peps/pep-0020/). Python Software Foundation. 2004-08-23. . Retrieved 2008-11-24.

[3] http://docs.python.org/reference/lexical_analysis.html#string-literals

[4] http://www.python.org/dev/peps/pep-0238/

[5] David Mertz. "Functional Programming in Python" (http://gnosis.cx/publish/programming/charming_python_13.html). IBM developerWorks. .

[6] The `nonlocal` keyword was adopted by PEP 3104 (http://www.python.org/dev/peps/pep-3104/)

[7] http://www.python.org/doc/2.4.2/tut/node11.html#SECTION0011600000000000000000

[8] http://mail.python.org/pipermail/tutor/2003-October/025932.html

[9] http://www.python.org/download/releases/2.3/mro/

[10] EAFP (http://docs.python.org/glossary.html#term-eafp), Python Glossary

[11] Hamblen, Diane. "Only the Limits of Our Imagination: An exclusive interview with RADM Grace M. Hopper" (http://web.archive.org/web/20090114165606/http://www.chips.navy.mil/archives/86_jul/interview.html). Department of the Navy Information Technology Magazine. Archived from the original (http://www.chips.navy.mil/archives/86_jul/interview.html) on January 14, 2009. . Retrieved 2007-01-31.

[12] *Python in a nutshell,* Alex Martelli, p. 134 (http://books.google.com/books?id=JnR9hQA3SncC&pg=PA134)

[13] http://docs.python.org/lib/profile.html

[14] Alex Martelli (19 May 2003). "EAFP v. LBYL" (http://code.activestate.com/lists/python-list/337643/). python-list mailing list. .

[15] "Python 2.4 Decorators: Reducing code duplication and consolidating knowledge" (http://www.ddj.com/184406073#l11). *Dr. Dobb's*. 2005-05-01. . Retrieved 2007-02-08.

[16] "New Tail Recursion Decorator" (http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/496691). *ASPN: Python Cookbook*. 2006-11-14. . Retrieved 2007-02-08.

[17] "The decorator module" (http://www.phyast.pitt.edu/~micheles/python/documentation.html). . Retrieved 2007-02-08.

[18] David Mertz (2006-12-29). "Charming Python: Decorators make magic easy; A look at the newest Python facility for metaprogramming" (http://www-128.ibm.com/developerworks/linux/library/l-cpdecor.html). *IBM developerWorks*. . Retrieved 2007-02-08.

[19] "PEP 3129 - Class Decorators" (http://docs.python.org/whatsnew/2.6.html#pep-3129-class-decorators). *What's New in Python 2.6*. 2010-08-11. . Retrieved 2011-01-23.

[20] The referenced comic strip is xkcd #353 (http://xkcd.com/353/); the module was added to the Python trunk (http://svn.python.org/view/python/trunk/Lib/antigravity.py?rev=66902&view=markup) for the 3.0 release.

## External links

- Python tutorial (http://docs.python.org/tut/) - Tutorial written by the author of Python, Guido van Rossum.

# Article Sources and Contributors

**Python syntax and semantics** *Source*: http://en.wikipedia.org/w/index.php?oldid=455685574 *Contributors*: 17c, Abdull, Adriatikus, Agrdeuce, AntiRush, Arabic Pilot, Benhoyt, Benwing, Bogsat, Boshomi, Bromador, Cburnett, Ccfontes, Charles Merriam, ChorizoLasagna, Cmbankester, ColdShine, Comatose51, Conrad.Irwin, Crenner, CyberCerberus, Cybercobra, DanBishop, Darylbrowne, Decrypt3, Dodehoekspiegel, Dubbaluga, Długosz, EdC, Gerbrant, Grandmartin11, Hakeem.gadi, Hdante, Hydrargyrum, Illarionov, Imz, Intuicja7, InverseHypercube, JBsupreme, JLaTondre, JeDi, Jerryobject, JimD, Jogloran, Johnuniq, Keziah, Leafnode, LeonardoGregianin, Liesel Hess, Lulu of the Lotus-Eaters, Mike92591, MisterSheik, Mistercupcake, Moxfyre, Mykhal, Nbarth, Neo of ZW, Noah Salzman, Paddy3118, Paul Pogonyshev, Pingveno, ProveIt, Ragzouken, Raise exception, Rangi42, Rjwilmsi, Robin Stocker, Sam Pointon, Serprex, Simeon, Skagedal, Specious, Spoon!, Steveha, Strcat, Svick, Thumperward, Tomaxer, Upwestdon, Vadmium, Whiner01, Wiredsoul, ZacBowling, ZbySz, Zearin, 71 anonymous edits

# License