Design and History FAQ

Why does Python use indentation for grouping of statements?

Guido van Rossum believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after a while.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```
if (x <= y)
        x++;
        y--;
z++;
```

Only the `x++` statement is executed if the condition is true, but the indentation leads you to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. If you're used to reading and writing code that uses one style, you will feel at least slightly uneasy when reading (or being required to write) another style.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

Why am I getting strange results with simple arithmetic operations?

See the next question.

Why are floating-point calculations so inaccurate?

Users are often surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It's not. This has little to do with Python, and much more to do with how the underlying platform handles floating-point numbers.

The `float` type in CPython uses a C `double` for storage. A `float` object's value is stored in binary floating-point with a fixed precision (typically 53 bits) and Python uses C operations, which in turn rely on the hardware implementation in the processor, to perform floating-point operations. This means that as far as floating-point operations are concerned, Python behaves like many popular languages including C and Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating-point. For example, after:

```
>>> x = 1.2
```

the value stored for `x` is a (very good) approximation to the decimal value `1.2`, but is not exactly equal to it. On a typical machine, the actual stored value is:

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

which is exactly:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

The typical precision of 53 bits provides Python floats with 15–16 decimal digits of accuracy.

For a fuller explanation, please see the floating point arithmetic chapter in the Python tutorial.

Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as “elemental” as numbers. No amount of activity will change the value 8 to anything else, and in Python, no amount of activity will change the string “eight” to anything else.

Why must ‘self’ be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it’s more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don’t know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you’d have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren’t explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn’t have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don’t have to search the instance’s directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.

Why can’t I use an assignment in an expression?

Many people used to C or Perl complain that they want to use this C idiom:

```
while (line = readline(f)) {
    // do something with line
}
```

where in Python you’re forced to write this:

```
while True:
    line = f.readline()
    if not line:
        break
    ...  # do something with line
```

The reason for not allowing assignment in Python expressions is a common, hard-to-find bug in those other languages, caused by this construct:

```
if (x = 0) {
    // error handling
}
else {
    // code that only works for nonzero x
}
```

The error is a simple typo: `x = 0`, which assigns 0 to the variable `x`, was written while the comparison `x == 0` is certainly what was intended.

Many alternatives have been proposed. Most are hacks that save some typing but use arbitrary or cryptic syntax or keywords, and fail the simple criterion for language change proposals: it should intuitively suggest the proper meaning to a human reader who has not yet been introduced to the construct.

An interesting phenomenon is that most experienced Python programmers recognize the `while True` idiom and don’t seem to be missing the assignment in expression construct much; it’s only newcomers who express a strong desire to add this to the language.

There’s an alternative way of spelling this that seems attractive but is generally less robust than the “while True” solution:

```
line = f.readline()
while line:
    ...  # do something with line...
    line = f.readline()
```

The problem with this is that if you change your mind about exactly how you get the next line (e.g. you want to change it into `sys.stdin.readline()`) you have to remember to change two places in your program – the second occurrence is hidden at the bottom of the loop.

The best approach is to use iterators, making it possible to loop through objects using the `for` statement. For example, file objects support the iterator protocol, so you can write simply:

```
for line in f:
    ...  # do something with line...
```

Why does Python use methods for some functionality (e.g. list.index()) but functions for other (e.g. len(list))?

The major reason is history. Functions were used for those operations that were generic for a group of types and which were intended to work even for objects that didn't have methods at all (e.g. tuples). It is also convenient to have a function that can readily be applied to an amorphous collection of objects when you use the functional features of Python (`map()`, `zip()` et al).

In fact, implementing `len()`, `max()`, `min()` as a built-in function is actually less code than implementing them as methods for each type. One can quibble about individual cases but it's a part of Python, and it's too late to make such fundamental changes now. The functions have to remain to avoid massive code breakage.

> **Note:** For string operations, Python has moved from external functions (the `string` module) to methods. However, `len()` is still a function.

Why is join() a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
", ".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: “It looks really ugly using a method of a string literal (string constant)”, to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: “I am really telling a sequence to join its members together with a string constant”. Sadly, you aren’t. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space).

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself. Similar methods exist for bytes and bytearray objects.

How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn’t the case, you coded it like this:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

For this specific case, you could also use `value = dict.setdefault(key, getvalue(key))`, but only if the `getvalue()` call is cheap enough because it is evaluated in all cases.

Why isn't there a switch or case statement in Python?

You can do this easily enough with a sequence of `if... elif... elif... else`. There have been some proposals for switch statement syntax, but there is no consensus (yet) on whether and how to do range tests. See **PEP 275** for complete details and the current status.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```
def function_1(...):
    ...

functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1, ...}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```
def visit_a(self, ...):
    ...
...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

It's suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.

Can’t you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is Stackless Python, which has a completely redesigned interpreter loop that avoids the C stack.

Why can’t lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python’s syntactic framework can’t handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you’re too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally-defined function is that you don’t need to invent a name for the function – but that’s just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!

Can Python be compiled to machine code, C or some other language?

Cython compiles a modified version of Python with optional annotations into C extensions. Nuitka is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language. For compiling to Java you can consider VOC.

How does Python manage memory?

The details of Python memory management depend on the implementation. The standard implementation of Python, CPython, uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and de-

letes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Other implementations (such as Jython or PyPy), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

In some Python implementations, the following code (which is fine in CPython) will probably run out of file descriptors:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython's reference counting and destructor scheme, each new assignment to *f* closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of memory management scheme:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

Why doesn't CPython use a more traditional garbage collection scheme?

For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard malloc() and free() with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for malloc() and free(), and may not want Python's. Right now, CPython works with anything that implements malloc() and free() properly.

Why isn't all memory freed when CPython exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.

Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.

How are lists implemented?

Python's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly;

when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

How are dictionaries implemented?

Python's dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, "Python" could hash to -539294296 while "python", a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time – O(1), in computer science notation – to retrieve a key. It also means that no sorted order of the keys is maintained, and traversing the array as the `.keys()` and `.items()` do will output the dictionary's content in some arbitrary jumbled order that can change with every invocation of a program.

Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

would raise a KeyError exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn't work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.

- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.

- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if `o1 == o2` (ie `o1.__eq__(o2) is True`) then `hash(o1) == hash(o2)` (ie, `o1.__hash__() == o2.__hash__()`), re-

gardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of ListWrapper, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

Why doesn't list.sort() return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, `list.sort()` sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won't be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

If you want to return a new list, use the built-in `sorted()` function instead. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections.abc` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components. There is also a tool, PyChecker, which can be used to find problems due to subclassing.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple "self test." Even modules which use complex ex-

ternal interfaces can often be tested in isolation using trivial “stub” emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `append()` implementation will actually do this correctly, but it’s trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code with an eye to making it easily tested. One increasingly popular technique, test-directed development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.

Why is there no goto?

You can use exceptions to provide a “structured goto” that even works across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the “go” or “goto” constructs of C, Fortran, and other languages. For example:

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

This doesn’t allow you to jump into the middle of a loop, but that’s usually considered an abuse of goto anyway. Use sparingly.

Why can’t raw strings (r-strings) end with a backslash?

More precisely, they can’t end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you're trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt")  # works fine!
```

If you're trying to build a pathname for a DOS command, try e.g. one of

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

Why doesn't Python have a "with" statement for attribute assignments?

Python has a 'with' statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some language have a construct that looks like this:

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it's possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler *always* knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```
def foo(a):
    with a:
        print(x)
```

The snippet assumes that “a” must have a member attribute called “x”. However, there is nothing in Python that tells the interpreter this. What should happen if “a” is, let us say, an integer? If there is a global variable named “x”, will it be used inside the with block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of “with” and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

write this:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.

Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```
if a == b
    print(a)
```

versus

```
if a == b:
    print(a)
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it’s a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.

Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it's easier to add more elements because you don't have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

This list looks like it has four elements, but it actually contains three: "fee", "fiefoo" and "fum". Always adding the comma avoids this source of error.

Allowing the trailing comma may also make programmatic code generation easier.