Some comments about the Python standard documentation – list n° 3

Author:

Jacques Ducasse, PARIS, FRANCE
jacko21@aliceadsl.fr
October 10, 2014

All of these comments concern the standard .chm documentation of Python 3.4.0, last updated Mar 16, 2014, distributed on the official Python site.

The Python Library Reference

2. Built-in Functions

`compile()`

♦ "...that are in effect in the code that is calling compile."
"`compile`" should be in keyword typography.
Note also that *`flags`* could be the special flag described in the `ast` module (PyCF_ONLY_AST) in addition of the `compiler_flag` in the `__future__` module as described here.

`format()`

♦ "A call to `format(value, format_spec)` is translated to `type(value).__format__(format_spec)` ."
I think the translation is rather: `type(value).__format__(value,format_spec)`."

`slice()`

♦ The `.indices()` method is described only in the "Standard type hierarchy" section of the "Python Language Reference" manual. May be it would be more useful to inform the reader about this method here (as well as other attributes `start`, `stop` and `step`).

4.4. Numeric Types

♦ Just above the operators table: " (operations in the same box have the same priority; "
Not applicable: in this table, there is only one operation in each box. (the only table with several operators in the boxes is the table in The Python Language Reference - 6.15. Operator precedence).

4.4.1. Bitwise Operations on Integer Types

♦ Just above the operators table: " (operations in the same box have the same priority) "
Not applicable again: in this table, there is only one operation in each box.

4.4.2. Additional Methods on Integer Types

♦ "In addition, it provides one more method."
Two methods and one class method are described.

4.6.1. Common Sequence Operations

♦ Second paragraph above the operators table: " (operations in the same box have the same priority) "
Not applicable again: in this table, there is only one operation in each box.

4.11. Context Manager Types

♦ `__exit__()` : "…This allows context management code (such as `contextlib.nested`) …"
There is no more function `nested()` in module `contextlib`. This function was deprecated in version 3.1, and definitively removed since version 3.2 (in favor of a plain `with` statement which can accept multiple context managers).

5. Built-in Exceptions

♦ In the introduction : "When raising (or re-raising) an exception in an `except` clause `__context__` is automatically…" : this occurs when an exception is raised in an `except` clause or in a `finally` clause. (see "Changes To Exceptions" in "What's new in Python 3.0").

♦ **`BaseException`** : The comment of the `args` attribute refers the `IOError`, which is kept only for compatibility reason and is now an alias of `OSError`.

7.1. `struct`

♦ Several functions write to or read from a *buffer*. But the nature of this buffer is not explained. In the past, the `buffer()` built-in was used. Today, it would be useful to describe what is this *buffer*, especially referring to the `memoryview()` built-in.

8.3. `collections`

8.3.5. `namedtuple()`

♦ The last block about the recipe is not in the usual typography of the "See also" blocks.

8.4 `collections.abc`

♦ "it only necessary to supply…" → "it is only necessary to supply…"

8.12. `reprlib`

♦ **`Repr.repr_TYPE()`**

The functions `join()` and `split()` are gone from module `string` since a long time. Moreover, a ")" is misplaced. Consequently:

```
string.join(string.split(type(obj).__name__, '_'))
```

should be rewrote:

```
'_'.join(type(obj).__name__.split())
```

♦ **8.12.2. Subclassing Repr Objects**

The given example of `MyRepr` doesn't work in Python 3.x! This is because the type of `sys.stdin` is now `TextIOWrapper` and not `file` (as it was in Python 2.x). Thus, the method should be named `repr_TextIOWrapper()` instead of `repr_file()`. But, not so nice!

11.6. `tempfile`

♦ Just above `tempdir`, we can read: "The module uses two global variables that tell it how to construct a temporary name."
But there is only one : `tempfile.tempdir` (the other one was `tempfile.template`, but it disappeared since version 3.0).

11.10 `shutil`

♦ `make_archive()` : the parameters `verbose` and above all `dry_run` are not described.

♦ `register_archive_format()` :
- "description is used … archivers. Defaults to an empty list."
  Not an empty list, but an empty string.
- The signature of the *function* passed as argument is not described. (on the contrary, it is well described for `register_unpack_format()` : "The callable will receive…").

16.1. `os`

♦ Four variables named `supports_*` are said to give a `Set` object. In fact, it is rather a `set` object (with lower case); this is probably a typo: `Set` is an abstract class and cannot give an instance.

16.17 `ctypes`

16.17.2.2. Loading shared libraries

♦ Paragraph after the definition of `ctypes.DEFAULT_MODE`:
"Instances of these classes ...as attributes of by index."
should be : "…as attributes or by index".

16.17.2.5. Utility functions

♦ **`find_msvcrt()`:** "VC runtype library" --> "VC runtime library".

16.17.2.8. Structured data types

♦ **`ctypes.Structure`** : "Structure and union subclass constructors accept both positional and named arguments. Positional arguments are used to initialize the fields in the same order as they appear in the _fields_ definition, named arguments are used to initialize the fields with the corresponding name.
It is possible to defined sub-subclasses of structure types, they inherit the fields of the base class plus the _fields_ defined in the sub-subclass, if any."
These two sentences are given again later, at the end of § 16.17.2.8; the sentences are not exactly the same, but the information is the same. I think the later are the best, and the former could be removed.

24.2. `cmd`

♦ The section 24.2.2 "Cmd Example" shows how to use the `cmdqueue` attribute of the `Cmd` object. If it is legal to use this undocumented attribute, I think this so useful attribute must be described in the reference section 24.2.1, which is not the case.

26.1 `pydoc`

♦ The documentation explains how to use `pydoc`, but doesn't say what is displayed. In particular, for Python programmers who want to quickly include their modules in the `pydoc` system, it would be useful to know where the description of objects is taken. This is a proposal:
"For modules, classes, functions and methods, the displayed description is obtained from the *docstring* (the `__doc__` attribute) of the object. If there is no *docstring*, `pydoc` tries to obtain a description from the block of comment lines just above the definition of the class, function or method in the source file, or at the top of the module source file (see `inspect.getcomments()`)."

26.4. `unittest.mock`

♦ **26.4.2. The `Mock` Class**

- `side_effect`: description of the attribute:
  "This can either be a function to be called when the mock is called, or an exception (class or instance) to be raised."
  There is a third alternative: a sequence. Then, proposal: "This can either be a function to be called when the mock is called, an iterable returning the next value on each call, or an exception (class or instance) to be raised."

- Next line, we could read:
  "If you pass in a function it will be called with same arguments as the mock and unless the function returns the DEFAULT singleton the call to the mock will then return whatever the function returns. If the function returns DEFAULT then the mock will return its normal value (from the return_value)."
  and few lines below :
  "The side_effect function is called with the same arguments as the mock (so it is wise for it to take arbitrary args and keyword arguments) and whatever it returns is used as the return value for the call. The exception is if side_effect returns DEFAULT, in which case the normal return_value is used."
  The two paragraphs say the same thing! I think the first is clearer, and we can remove the second one.

♦ **26.4.3.1. patch**

"…the mock with be created with a spec…" → "…the mock will be created with a spec…".

27. Debugging and Profiling

♦ The chapters "27.8. Examples" and "27.9. API" are one level too high. They must be subchapters of "27.7. `tracemalloc`" instead.

29.12. `inspect`

♦ `formatargspec()` : the description of the `format*` parameters is erroneous : "The other five arguments are the corresponding optional formatting functions that are called to turn names and values into strings. The last argument is an optional function to format the sequence of arguments".
The right description should be (proposal): "`formatarg` is called to turn names in `args` and `kwonlyargs` into strings ; `formatvarargs` and `formatvarkw` are called to turn `varargs` and `varkw` into strings ; `formatvalue` is called to turn values in `defaults` and `kwonlydefaults` into strings ; `formatreturns` is called to turn the return annotation in `annotations` into string ; `formatannotations` is called to turn parameter annotations in `annotations` into strings."

31.5. `runpy`

`run_path()`

♦ "__spec__ will be set to None if the supplied path is a direct path to a script (as source or as precompiled bytecode)."
This sentence is redundant with the just above sentence; it gives no more information and could be removed.

Extending and Embedding

1.10. Reference Counts

♦ Last paragraph : "The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (__del__() methods). When there are such finalizers, the detector exposes the cycles through the gc module (specifically, the garbage variable in that module)."
This was true until version 3.3. With the version 3.4 the method `__del__()` is no more a restriction for the deletion of garbage cycles by `gc`. See "What's new, PEP 442, Safe Object Finalization".

Python/C API Reference Manual

Utilities
Parsing arguments and building values
API Functions

♦ **`PyArg_Parse()`**
The text references the METH_OLDARGS parameter, but this parameter is removed from the API since version 3.0 (even out of the general index). Then, new reader of this documentation cannot understand what is the matter and needs a minimum of explanation. May be, the simplest way is to add a comment, like: "... which use METH_OLDARGS parameter parsing method, no more supported since version 3.0".

The Python Language Reference

2.4.1. String and Bytes literals

♦ There are two tables of escape sequences in this chapter. I think it is not obvious, for an inattentive reader, that the first one applies to `string` and `bytes` types, while the second applies to `string` type only. Especially because the word "string" is used in the both headers just above each of the tables, with different meanings.

3.3. Special method names
3.3.1. Basic customization
♦ `__del__()`
In the first box : "…the latter two situations can be resolved by storing `None` in `sys.last_traceback`.". This is a relic form version 2.7 and before: "the latter two situations can be resolved by storing `None` in `sys.exc_traceback` or `sys.last_traceback`.". When `sys.exc_traceback` was gone in 3.0, the sentence was carelessly rebuilt. This is a proposal:
"the second situation can be resolved by freeing the reference to the traceback obtained by `sys.exc_info()` as soon as it is no more useful ; the third situation can be resolved by storing `None` in `sys.last_traceback`.".

3.3.2.3. `__slots__`
♦ "If defined in a class, __slots__ reserves space…"
The beginning ("if defined in a class") could be removed: all the method described in chapter "Special method names" have sense only when they are defined in a class. This sentence is a relic from "If defined in a new style class,…" in release 2.x; today, all classes are new style classes.

4. Execution model
♦ **4.1. Naming and binding**
"The global statement must precede all uses of the name.".
The word "`global`" must be in keyword typography.

6.3. Primaries
6.3.3. Slicings
"The primary must evaluate to a mapping object". Why such a restriction ? This is a relic from the rebuilt documentation of version 2.x. In fact, the primary could be any object with a `__getitem__()` method, not only a mapping.

7. Simple statements

♦ **7.8. The raise statement**

- The form "`from None`" is not described. Details are available in the "What's new" section of version 3.3 "PEP 409: Suppressing exception context".

- “A similar mechanism works implicitly if an exception is raised inside an exception handler”: this occurs as well if the exception is raised inside a `finally` clause. (see my remark in **5. Built-in Exceptions** above).

♦ **7.11. The `import` statement**

"The `from` form with * may only occur in a module scope. The wild card form of import — `import *` — is only allowed at the module level."
The two sentences tell the same rule and are redundant (carelessly rebuilt relic from 2.x). They could be simplified as: "The wild card form of the `from` form — `import` * — is only allowed at the module level.".

9. Top-level components

♦ **9.4. Expression input**

"There are two forms of expression input."
There was two forms in version 2.x: one for `eval()`, the other for `input()`. This became false in 3.x because `input()` was gone (the new `input()` is the old `raw_input()`), and now there is only the solely form for `eval()`.

That’s all folks!