Some comments about the Python standard documentation – list n° 4

Author:
Jacques Ducasse, PARIS, FRANCE
jacko21@aliceadsl.fr
March 08, 2016

All of these comments concern the standard documentation of Python **3.5.1**, on Mar 07, 2016, distributed on the official Python site.

The Python Library Reference

2. `Built-in Functions`

♦ `eval()`: "If the *globals* dictionary is present and lacks '`__builtins__`', the current globals are copied into *globals* before *expression* is parsed."
This is not true. Not all the current globals, only `__buitins__` is copied. In the following example, the global variable 'a' is not copied into the *globals* dictionary, while `__builtins__` is:

```
>>> a=3
>>> eval("print(list(globals()))",{'b':9})
['b', '__builtins__']
```

Note that this is the same for the `exec()` function, where it is correctly explained: "If the *globals* dictionary does not contain a value for the key `__builtins__`, a reference to the dictionary of the built-in module `builtins` is inserted under that key" I suggest to copy the same to the `eval()` description.

6.2. `re`

♦ **6.2.1. Regular Expression Syntax**: in the `\number` section: "to match one of the first 99 groups".
♦ **6.2.4. Match Objects**: in the `group()` section: "if it is in the inclusive range [1..99]".
But in the What's New In Python 3.5, we read: "The number of capturing groups in regular expression is no longer limited by 100".
It seems the new feature (99 is no longer a limit) was not reported to the `re` module documentation.
Moreover, if we return to the `\number` section in 6.2.1: "If the first digit of number is 0, or number is 3 octal digits long, it will not be interpreted as a group match, but as the character with octal value number". Well, now the number of groups can be > 99 (therefore more than 2 digits, like '123'), I don't understand how Python can distinguish between the group number 123 and the octal character specification 123?

8.8. `weakref`

♦ `finalize()`: "…a finalizer will always survive until the reference object is collected…"
I think it should be: the referenced object.

♦ `WeakKeyDictionary()`: "…additional methods. These expose…"
Since Python 3.0, there is only one additional method `keyref()`: therefore singular is required: "…additional method. This exposes…"

♦ `WeakValueDictionary()`: same thing.

10.2. `functools`

♦ `total_ordering()`: "this class decorator supplies the rest".
Maybe it could be useful to precise it supplies the rest except the `__ne__()` method. No need because the default method, inherited from `object` class, does the right job (i.e. `not ==`).

12.6. `sqlite3`

♦ `connect()`: the parameter `check_same_thread` is not described.

26.4. `unittest`

♦ `run()` method of `TestCase`: "If result is omitted or None, a temporary result object is created".
The word "temporary" must be removed: this was true in the past because the result object wasn't returned by the method: the object was then destroyed when the method ends. Since Python 3.3, `run()` returns the result object to the calling program: therefore, this object is no longer "temporary".

♦ `_removeTestAtIndex()` method of `TestSuite`: This method is mentioned twice; and one more time in the What's New in Python 3.4. However, it is not described. If I want to override this method in a subclass (as it is proposed), what is its signature and what does it have to return?

27.4. `The Python Profiler`

♦ `profile.Profile()`: the `subcalls` and `builtins` parameters are not described.

32.1. `parser`

♦ `st2list()` and `st2tuple()`: the `col_info` parameter is not described. Seems obvious, but how is it retuned in combination (of not) with `line_info`?

The Python Language Reference

2.3. Identifiers and keywords

♦ **2.3.2. Reserved classes of identifiers**: The section _* mixes up two cases: _* and _, which have no connection. I think it should be split:

_*

Not imported by `from module import *`. See section The import statement.

_

The special identifier _ is used in the interactive interpreter … no special meaning and is not defined.
(remove "See section The import statement")
Note:
The name _ is often used… on this convention.

4.2. Naming and binding

♦ 4.2.4. Interaction with dynamic features: "If a variable is referenced in an enclosing scope, it is illegal to delete the name. An error will be reported at compile time."
This couple of sentences is not clear, and probably wrong. I think it should be removed. This is the reason:

1.
```
>>> def f():
...     a=3
...     def g():
...             del a
...     g()
```
In this example, there is no error at compile time. There is an `UnboundLocalError` error at run time, because 'a' is local in `g()` but unbound. This is true for all versions of Python (at least from 2.5).

2.
```
>>> def f():
...     a=3
...     def g():
...             nonlocal a
...             del a
...     g()
```

In this example, there is no error at compile time and no error at run time.

3. Maybe there is a mistake in the sentence, which perhaps would be: "If a variable is referenced in an enclosed scope, it is illegal to delete the name. An error will be reported at compile time" like this:

```
>>> def f():
...     def g():
...             print(a)
...     a=3
...     del a
```

But even in this case, there is no error at compile time and no error at run time, since Python 3.2. This changed in Python 3.2 (see What's New in Python 3.2); indeed, there was a compile time error in Python 3.1 and earlier: "SyntaxError: can not delete variable 'a' referenced in nested scope."

Thus, in all the cases, since Python 3.2, there is no error at compile time, and it is legal to delete the name.

7.2. Assignment statements

♦ "If the target list is a comma-separated list of targets: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets".
The underlined sentence is now restricted to the case "Else" below (i.e. without starred target). In fact, the sentence is exactly copied in the section "Else", and should be removed from the header.

7.5. The `del` statements

♦ "If the name is unbound, a `NameError` exception will be raised."
Rather a `UnboundLocaError` exception. Example:

```
>>> def f():
...     del a
...     a=3
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 'a' referenced before assignment
```

7.8. The `raise` statement

♦ The sentence "if given, the second expression must be another exception class or instance" is not fully correct since Python 3.3: it forgets the form "`from None`". This form is described nowhere in the documentation, except in the "What's new in Python 3.3", and therefore users may miss this interesting feature.

Proposition: "If the second *expression* in the `from` clause is `None`, the display of the context, if any, is suppressed." to be added just after the last example and before "Additional information on exceptions can be found…".

8.4. The `try` statement

♦ "`sys.exc_info()` values are restored to their previous values (before the call) when returning from a function that handled an exception."
In addition, it could be useful to precise that the same is true at the end of the `except` clause. Example:

```
>>> print(sys.exc_info())
(None, None, None)
>>> try: raise KeyError
... except:
...     print(sys.exc_info())
...     try: raise ValueError
...     except:
...             print(sys.exc_info())
...     print(sys.exc_info())
```

```
...
(<class 'KeyError'>, KeyError(), <traceback object at 0x00CDBBE8>)
(<class 'ValueError'>, ValueError(), <traceback object at 0x00CDBC10>)
(<class 'KeyError'>, KeyError(), <traceback object at 0x00CDBBE8>)      ← restored
>>> print(sys.exc_info())
(None, None, None)                                                      ← restored
```

Proposition: “`sys.exc_info()` values are also restored to their previous values (before the try statement) when ending the `try: except:` statement.”

♦ “The exception information <u>is not available</u> to the program during execution of the `finally` clause.”
This is not true, it is available through `sys.exc_info()`. Example:

```
>>> try: raise KeyError("Hello world")
... finally:
...     print("in finally:",sys.exc_info())
...
in finally: (<class 'KeyError'>, KeyError('Hello world',), <traceback object at 0x00CDBC10>)
```

♦ There is a footnote: “[1] The exception is propagated to the invocation stack unless there is a `finally` clause which happens to raise another exception. That new exception causes <u>the old one to be lost</u>.”
This is no longer totally true since Python 3.0: the old one is available on the `__context__` attribute of the new one.

8.6. Function definitions

♦ 4th paragraph from the end: “are keyword-only parameters and may only be passed <u>used</u> keyword arguments” → “using”.

That’s all folks!