PEP 457: Syntax For Positional-Only Parameters

by Zachary Doyle, Niklas Fejes, Samuel Carnes

December 8th, 2014

Introduction

This PEP (Python Enhancement Proposal) proposes a change in how the syntax for parameter passing is defined, in a way such that documentation can be expressed more consistently, and such that the language allows positional-only parameters.

One of the important ideas of the proposal is that:

currently in the documentation there's no way to tell whether a function takes positional-only parameters.

Note that while it is possible to simulate positional-only parameters, one has to implement the argument parsing and the function signature does not tell what arguments are accepted.

Proposal info

- https://www.python.org/dev/peps/pep-0457/
- Last updated 2013-10-09
- Larry Hastings
- Status: draft

The original PEP was proposed at October 8, 2013 by Larry Hasting, but is grounded in earlier ideas by GvR (Guido van Rossum) himself. While 457 is the first actual PEP to be based on the idea of positional-only arguments, in early 2012, Guido van Rossum wrote:

I would actually like to see a syntactic feature to state that an argument cannot *be given as a keyword argument (just as we already added syntax to state that it* must *be a keyword).*

This "syntactic feature" would evolve into the now-proposed "/" operand, also originally suggested by van Rossum [3] ; it circulated in the Python-ideas

mailing list for a relatively brief time (most of 2012, as opposed to the almost decades-long germinating times of other PEPs) before officially being proposed by Mr. Hastings. As discussed below, a major point of contention was whether or not the slash-operator would be applied as a parameter operand, e.g.:

```
def somefunc(pos_only,/,p_or_k):
...
```

Or as a sigil:

```
def somefunc(/pos_only, p_or_k):
...
```

While the latter was originally favored by many aside from van Rossum himself, on the basis of reducing the "noise" in declarations, the former syntax prevailed - due to the fact that, by the nature of positional args, any argument placed before the last positional-only argument would *need* to itself be positional, meaning only a single delineating symbol was needed.

Motivation

In Python there are different semantics for function parameters declaration. This can be confusing for the programmer, especially when the documentation specifies two or more different versions of the same function that takes different numbers of arguments. This can be confusing because the actual implementation must be made as one single function, so if the programmer looks at the source code for the function, the parameters will most likely not match the ones in the documentation.

To cite the PEP,

- *Documentation can clearly, unambiguously, and consistently express exactly how the arguments for a function will be interpreted.*
- *The syntax is reserved for future use, in case the community decides someday to add positional-only parameters to the language.*
- *Argument Clinic can use a variant of the syntax as part of its input when defining the arguments for built-in functions.*

As an example, the `range()` function is documented with the signatures

```
range(stop)
range(start, stop[, step])
```

which with the new syntax would be

```
range([start,] stop, [step,] /)
```

One important aspect to note here is that this syntax is ambiguous, so the left-most optional parameter (`[start,]`) would be preferred over the right optional parameter (`[step,]`). However, if we specify multiple arguments in the optional group we can create more complex parameter passings. An example of this is the `window.addch()` function which is currently documented with the two following function signatures:

```
addch(ch[, attr])
addch(y, x, ch[, attr])
```

With PEP 457, the documentation for this function could be exactly the same as the function definiton, namely:

```
def addch([y, x,] ch, [attr,] /):
```

As opposed to the `range()` function, this parameterization is not ambiguous, but it shows one of the biggest advantages of this syntax. The right optional argument group is chosen if 2 arguments are given; the groups are determined by the number of arguments and in case of ambiguity the leftmost argument is used.

Positional only parameters

Currently the way parameter passing works (not including the `*args` and `**kwargs` syntax), the way keyword-mode is specified is:

```
def name(positional_or_keyword_parameters, *, keyword_only_parameters):
```

This allows the user to specify all parameters as keywords, which might not always be optimal. In PEP 457 this issue is addressed by adding the `/` parameter separator, where the preceding parameter are positional only and can not be keyworded. The new syntax would be as follows.

```
def name(positional_only_parameters, /, positional_or_keyword_parameters,
         *, keyword_only_parameters):
```

There are a few cases when positional only parameters are needed not only to ensure that the user puts the parameters in the order specified in the function definition. Most importantly, the optional parameter syntax requires the the parameters can not be keyworded. As an example, how would the code below be interpreted?

```
def foo([a, b,] c, [d,]):
    ...
foo(1,2,3,b=4)
```

Advanced semantics

To cite the PEP,

- *Although positional-only parameter technically have names, these names are internal-only; positional-only parameters are never externally addressable by name. (Similarly to* `*args` *and* `**kwargs` *.)*
- *It's possible to nest option groups.*
- *If there are no required parameters, all option groups behave as if they're to the right of the required parameter group.*
- *For clarity and consistency, the comma for a parameter always comes immediately after the parameter name. It's a syntax error to specify a square bracket between the name of a parameter and the following comma. (This is far more readable than putting the comma outside the square bracket, particularly for nested groups.)*
- *If there are arguments after the* `/` *, then you must specify a comma after the* `/` *, just as there is a comma after the* `*` *denoting the shift to keyword-only parameters.*
- *This syntax has no effect on* `*args` *or* `**kwargs` *.*

The process by which python determines which positional arguments are to be assigned in a given method call can be viewed fairly intuitively as "filling up" the parametersa available from left to right. Consider this example:

```
def ex_func([a, b,] c, [d,] [e,] /):
   print("["+str(a)+"], ["+str(b)+"], ["+str(c)+"], ["+str(d)+"], ["+str(e)+"]")
```

ex_func can be passed anywhere between 1 and 5 arguments. If passed 0, the non-optional positional-only argument, c, has not been passed, and an exception is thrown, and if 1 argument is passed, then that argument alone is filled. With 2 arguments, the first "paired" couple of optional arguments is skipped, as at least two optional arguments (in addition to the required one) are required to "enter" those square brackets. So, it skips to d, filling it, leaving us with a, b, and e undefined. With 3 arguments, we have enough to fill a and b, so d and e are left unfilled. With 4, we have enough to fill everything except e (we fill c, then a and b, then d), and, predictably, with 5 arguments, all 5 parameters are defined.

Alternative notations

As we discussed above, prior to this PEP, several other semantics for positional only parameters has been proposed. Two examples that are discussed by Guido van Rossum in [3] is

```
def spam(~x, ~y, ~z, ~a=2/3):
   ...
def spam(/x, /y, /z, /a=2/3):
   ...
```

where each positional only parameter is preceded by either `~` or `/`. However this syntax was rejected since it clutters the defintion more than necessary. Something that is noted in the discussion in [3] is that if parameter `c` in `def foo(a,b,c,d)` is positional only, then `a` and `b` must also be positional only to. Thus it is not necessary to specify that each parameter is positional only, but we can group them together with the PEP 457 `/` syntax.

Problems with the proposal

With the new syntax, it would be possible to create very complex function parameter declarations, especially if nested optional groups are used. This might be a disadvantage since it might actually be harder to read such declarations compared to using multiple function definitions in the documentation.

For example, say that you have a function `bar(a,b,c,d)`, where you want to be able to call * `bar(a,b,c,d)` * `bar(b,c,d)` * `bar(c,d)` * `bar(d)`

The syntax for this definition would be complicated with nested optional groups, and could cause more issues than it solves.

```
def bar([[[a,] b,] c,] e, /)
```

On the other hand, with the current documentation syntax this function would have four different definitions, which might be worse than the above.

All this being said, it should be noted that PEP457 does not *invalidate* any existing Python syntax; thus, while some (indeed, quite a lot) unintuitiveness could result from changing core function signatures, independent developers should not have any of their own code broken (directly) by this update. In many ways, taking advantage of this new syntax is a challenge similar to integrating the disparate worlds of Python2 and Python3.

References

1. PEP 457
2. Python-ideas: keyword arguments everywhere (stdlib) - issue8706 (Guido van Rossum)
3. Python-ideas: keyword arguments everywhere (stdlib) - issue8706 (Guido van Rossum)

4. Python-ideas: keyword arguments everywhere (stdlib) - issue8706 (Guido van Rossum)
5. Python 2 documentation
6. Python 3 documentation