Hi Stephen,

**Section 6.5:**

New enum section looks good. Not sure if the
`enum.unique` class decorator might be worth mentioning when
discussing the perils of aliasing (it informs the runtime that the
class isn't supposed to contain any value aliases, so duplicate values
become an error)

**Section 6.41:**

I don't think you can realistically cover the degree to
which metaclass programming can make the runtime behaviour of a class
hierarchy incomprehensible, so the current stopping point and the
general recommendation to use a static type checker makes a lot of
sense.

For ease of understanding, it may worth mentioning the formal name of
the MRO algorithm, which is C3 linearisation:
<https://en.wikipedia.org/wiki/C3_linearization>

Name mangling (the `\_\_name` anti-collision feature) has a unique
vulnerability all its own: if a subclass (or peer class in a multiple
inheritance scenario) uses the same class name (e.g. in a different
module), even their mangled names will collide at runtime. So even
though the feature isn't going anywhere, actually using it trades a
relatively obvious namespace collision risk for a far more obscure
one, so a lot of folks (including me) think it isn't worth using.

**Section 6.43:**

The guidance to use specific class names for calls to
parent class methods is only correct in the case of fixed single
inheritance. If using multiple inheritance, only super() will
correctly traverse the MRO to the end.

**Section 6.44:**

The guidance doesn't really seem to follow from the new
text. I was expecting something more along the lines of "only call
unbound methods with instances of the relevant class". While there are
valid reason for breaking that rule (e.g. mock interfaces for testing,
runtime API proxy injection for app behaviour monitoring), in the vast
majority of cases, unbound methods should be used to adapt OO APIs to
functional implementations, not to play games with the type of the
first argument.

**Section 6.56:**

I believe mutating the result of locals() at function
scope (and equivalently, the result of vars() when it returns
locals()) would currently qualify as undefined behaviour (see
<https://www.python.org/dev/peps/pep-0558/> for more background on
that). This isn't the case for vars(obj), though. That always returns
obj.\_\_dict\_\_, and the object itself controls whether or not mutation
is permitted. That makes the behaviour \*variable\*, since it is type
dependent, but it's not undefined.

**Section 6.58:**

This section on deprecated features caught my eye while
reviewing the preceding section on implementation dependent features:
one Python specific trap is that DeprecationWarning is silent by
default outside \_\_main\_\_ modules. See
<https://www.python.org/dev/peps/pep-0565/#limitations-on-pep-scope> for
a discussion of several known problems with these warnings (written as
of Python 3.7, but still current as of 3.10).

Cheers,
Nick.