1.

Awesome study :D

I'm a functional programming aficionado, so some of the things I do in Python is not pythonic at all, however I think it reduces many of the downsides of python (immutability, higher order functions, composability etcetc) (google: functional python).

One of the things is always thinking about when doing Python is the type of variables and have a compact and efficient syntax to convey the even more complex types in the documentation of a method.
At my last company we defined sum types with the `|` operator, and list/array with `[T]` in the documentation. We didn't have a hard requirement on documentation but we had a hard requirement on defining the types of arguments and return values of a function.

```
a :: [(int, str)] | {int: str}
returns joined :: str
```

which means that the function takes either a list of int, str tuples or a dict with int keys and str values.
Hence I think, when practical, (2) Inconsistent argument types can be really useful and comparable to when you do overloading in typed languages.

However Python lacks proper support for sum types and matching the "enum" so it can be a bit error prone having the "if statements" similar to (3) to handle different types in the arguments. However if you have the type contract explicitely in the documentation it's should be farily straight forward to check that all cases are covered in code review (and if it's not the code is to complex and you should go back to the drawing board)

We also had an even harder restriction on (1) inconsisten assigment types: A thing cannot be two things at once so never use the same name for something else (even if it's the same type), if you have done something with it it's not the same thing and therefore should have the same name.

(6) Dynamic attribute access is really error prone, a class (type) should never have to visit an attribute via reflection, the only valid reason is when you write meta code.

Also one of the most important special cases of inconsistent types thing is that of None, ex. things that return None when not finding an element or something. This actually would be like sum type `str | None` it returns a string or None if missing. (Java has the same issue with Null). Rust has solved this really nicely by having Option and Result type which makes the "missing element" or an error occured (instead of throwing an error) makes the code really easy to reason about since everything is explicit.

2.

I'm glad to collaborate with you and your team :)

I would like to make some comments about the examples in the survey:

(1) Inconsistent Assignment Types: inconsistent types in assignment.
The problem here seems to come from re-using variables, the way it is coded makes it hard to understand and indeed can be detected by the inconsistency of the types.

(2) Inconsistent Argument Types: argument value has inconsistent types.
The example is a bad practice in API development, scikit-learn uses it a lot also, where they choose to support different use-cases within the same API call. Probably inside of the definition, among the first lines it would be a discussion if the first argument is of type string or tuple, which makes very hard to do automatic type inference. This can spot the bad practice.

(3) Inconsistent Variable Types: variable reference has inconsistent types.
I don't see how 'func' can be bool if 'run' is always available. This example is a way of writing in a very concise way code that would take several lines.

(4) Dynamic Element Deletion: one element is deleted from a container.
and
(5) Dynamic Attribute Deletion: one attribute is deleted from an object.
I found this also a bad practice, you shouldn't have to delete elements or attributes for extending functionality.
I think it tends to happen when "glueing" different codebases, I found myself also doing it:
https://github.com/math-a3k/django-ai/blob/a000165c8ecfd741b4a595d5271cb3e3d94db5b4/django_ai/bayesian_networks/m

odels.py#L668

and

https://github.com/math-a3k/django-ai/blob/1ffd6ebafeb35361ddc9fee2535e881b54a9cfbe/django_ai/systems/spam_filtering/models.py#L469

In the first case, is to contemplate bayespy API given how django-ai API was constructed, and in the second case, it is a way for dealing with defaults for arguments and how they should "arrive" to scikit-learn API call. Those weren't the best solutions :)

(6) Dynamic Attribute Access: one attribute is visited based on a dynamically determined name.

I found myself doing this a lot:

https://github.com/math-a3k/django-ai/search?utf8=%E2%9C%93&q=getattr&type=

and most of the times I haven't provided a default value as I'm sure that the attribute exists. Dynamic attribute access is a key element in metaprogramming and allows to construct very cool things, I don't think is bad, the problem might be not providing a default value for the case that it is non-existant, but I guess this technique make type inference even harder...

Your research seems very interesting, can you send me the paper? :)

3.

The research looks very interesting and relevant. Please feel free to discuss more with me or to include me in this interesting research work. I would like to chip in if it is of interest to me.

4.

a try/except on attribute access or delattr is just a way to ensure something is removed [eg to clear a cache]

True it’s not Java or another strongly typed language, but the upside is greater flexibility and ease of development.

5.

About that, I think that most of the problems concerning bad patterns are related to a increased complexity on code development, and many programmers tend to add those complexities in the wrong places.

Se a class, for example. Many progs usually create classes with few functions that do too much logic, when they could disagregate those functions into something more "functional".

This is my opinion based on the Zen of Python, because it says a lot about how you police yourself as a python programmer.

By the way, let me know when you publish your research, I'll be very glad to read your thesis!

6.

I completed your questionnaire, I hope it helps you with your research. If you'd like more information don't hesitate to ask!

Also, I would be delighted if you'd send me an email when you finish your research and start developing the tools you're speaking about.

7.

I just completed your questionnaire. I'm interested in tools for improving code quality too. I use flake8 and isort on all my Python projects. I've also personally created two plugins for flake8:

https://github.com/adamchainz/flake8-comprehensions
https://github.com/adamchainz/flake8-tidy-imports

I like flake8, it's just a shame it currently has no way of automating changes. isort is neat but obviously restricted to just imports.

Keep me informed with your progress :)

8.

just filled out the questionaire.

I'm really interested in your research.

Would like to know more and if you can point me to references, this would be very nice.

9.

Most of them can be justified depending on a specific context: for code readibility, not multiplicating intermediate variables (against pep8), mimicking function overloading, in a data driven programming paradigm...

10.

I used pycharm and its type checking from docstrings to avoid inconsistent type errors

11.

These are all downsides of dynamic languages that prove that strong testing is required to validate the program. Test, test, test. Keep test coverage high.

12.

Of course dynamic types can lead to bugs just the same way static types can lead to overly cluttered, verbose, and unmaintainable walls of code. They each have their place.

The sklearn example is totally fine btw. It is exactly as unsafe as a hash table in C++. The example I really agree with is 1 but only because I dislike overriding existing variable names.

13.

Bad Practice: @property style abstractions (accessing what looks like a variable actually calls a function)

14.

Duck typing is a widely-used Python idiom, and I think most Python developers would consider it a feature. Related to this, Python tends to encourage `try`-ing operations and catching the exceptions, over writing code that seeks to avoid exceptions in the first place.

I'd argue the actual bugs here are in the fixes to some of your examples. (Which makes me think maybe I'm not understanding some of your questions.)

For example, the problem in example 3 seems to be the developer is checking for an exact type, rather than simply trying to use `vars` as a mapping or iterable object (and catching the error if that doesn't work!).

Similarly, a better fix for example 6 might have been `try...except AttributeError` rather than `getattr(..., None)` -- though it's hard to say without more context. I use both approaches in my own code, as well as sometimes deliberately *not* handling the exception, as a way of signaling invalid input to the caller when a default isn't appropriate.

Put another way, not all uncaught exceptions are bugs. (And trying to avoid exceptions at all costs might actually introduce bugs.)"

15.

I look forward to hearing about the results of your study! Hopefully with the more wide-spread acceptance of type hinting some of these issues will go away. Fabric is definitely at fault here when it comes to less than ideal type. This is partially due fabric's very long history, which reaches back to Python 2.5 (if not further in some cases). Anyways, I hope my questionnaire results help out with your research!