If I were hiring, I'd be looking for people who ask the "why" questions, not just the "how." That is, I'd want to find people who have thought critically about their tools, instead of accepting them blindly just because they are in style. The "how" questions matter too, of course; but asking the "why" questions suggests dedication to improve a field, not just earn a paycheck in it, and tends to imply potential to innovate.

With that in mind, here are a few queries drawn from both departments that I'd put to candidates, along with common replies, and the answers I'd hope to hear instead. I hope they help you evaluate the companies interviewing you, too; after all, a company that seems to be only about the "how" might not be the sort of place that fosters new ideas, or career growth in general.

How do Python 2.X and 3.X differ, and why should you care?

What people might say: "The print statement becomes a built-in function in 3.X."

That's true, but it's one of the most trivial differences, and may reflect a superficial understanding of a major pragmatic dilemma Python programmers face today. The more significant 3.X changes are:

- Differing and more pervasive Unicode support
- Mandatory usage of new-style classes
- Deeper integration of iterables and functional programming tools.
- Change, replacement, and deletion of many built-in tools (not just print!).

Of these, the 3.X Unicode model may have the largest impact, as it touches on strings, files, and a host of application-level interfaces in the standard library and 3rd-party domain. The new-style class model elevates topics such as the MRO, descriptors, and metaclasses from optional topic to required reading; and the more widespread role of iterables demands more careful use of tools like zip() results and dictionary key lists, for display, multiple traversals, and list-like operations.

You need to care about 2.X/3.X differences because the Python world still uses *both* lines, and the vast body of existing 2.X code will probably be a permanent part of the Python ecosystem. Even today, some 6 years after 3.X's initial release, there is still likely more 2.X code in production. In fact, by some measures, 3.X has yet to even achieve parity with 2.X in terms of user base. Python 3.X is a complete and entirely usable tool—and may even be better when it comes to Unicode support. But unless your code will never use a 2.X-only library, and your users will all have 3.X installed, you must still use 2.X. Even if that's not your story, you may still need to understand how 2.X differs, so you can maintain or port old code, or write new code that works on either line agnostically. Python 2.X may someday become the Fortran 77 of Python, but it's far too soon to discount millions of lines of working code.

A dynamic typing pop-quiz:

Quick: Given these three statements run in series:

```
A = []
B = A
A += [1]
```

does the third statement change B?

What people might say: The worst answer is "No, only A changes" (that reflects a lack of fundamental Python knowledge). But most people who've looked at Python in any depth would probably say "Yes—B changes, because it prints as [1] after the last statement runs, not []."

And yet, that's not quite right either. Yes, B does print a different value if displayed, and thus in some sense differs. But really, the better answer here is that the *variable* B has not changed at all, and still references the same object it did after the second statement. Rather, the *object* that B (and A) references differs at the end because it has been changed in-place through variable A.

This distinction between *variables* (a.k.a. names) like B—that reference objects, and *objects* like lists—which may be changed in place if mutable, turns out to be a central concept in all Python work. In larger programs, shared objects are often deliberately changed in-place in potentially far-flung bits of code, to update long-lived state. If you don't understand this model, it can lead to fairly painful debugging sessions when it occurs unexpectedly. If you do, it shows deeper and qualifying Python knowledge.

What's the point of using classes and OOP in Python?

What people might say: "Because of polymorphism?" (No, seriously; that's been a common response when I've asked this in classes I teach.)

This question is a sort of litmus test to gauge depth of experience and understanding. Polymorphism is a mechanism of OOP, not its purpose. The common answer is like saying that the purpose of carpentry is sawing! Since OOP—and classes, its concrete realization in Python code—represents a fundamental design choice, it's crucial to understand when it is and isn't warranted.

For smaller scripts and programs (under a few hundred lines, perhaps), OOP may not be justified at all. It's okay to code functions, modules, and even top-level script code for such tasks—but only as long as you don't expect them to be flexible enough to be reused in other programs. Top-level script code is always a one-program effort, because it runs immediately and has no container object. Functions can be imported and reused to some extent, but they don't directly support growth by extension, and must rely on arguments and single-copy global data for recording state information.

In contrast, OOP and classes become indispensable as programs grow larger: the extra *structure* that OOP can bring to your code makes it much easier to organize your work, know where to look for parts of an implementation, and factor code to avoid redundancy—the nemesis of software evolution. By naturally supporting multiple *copies* of application components, OOP also avoids the pitfalls of global data that vex much function-based code.

The largest answer here, though, is that, when used well, OOP lets you program by reusing and *customizing* what's already been written, thereby cutting development time radically. Classes provide a hierarchy that fosters extension in ways that functions and other tools cannot. An interviewee who doesn't express this probably hasn't moved beyond the trivial programs phase in the learning cycle; one who does is likely demonstrating experience with substantial and realistic software development.

What do you think about Python's "Batteries Included" paradigm?

What people might say: "I totally agree with it. You should use as much existing open-source code as you possibly can; why spend time reinventing the wheel when wheels are available for free?"

This one is as much about common sense as it is about engineering. The *pros* are easy—if you use already-developed code available in the open source world, you write less code yourself, and thus might cut overall development time. When this model works well, it's a clear net win. The *cons* are fairly major, though—by using third-party code, you create a major dependency for your company, on code of unknown quality, written by a person of unknown skill and discipline, and provided by someone who probably never worked for your company and has no interest in its success.

In truth, some such code could become your codebase's weakest link. Unless you're very careful (if not lucky) you may find yourself with an extra and ongoing task maintaining that wheel you supposedly got for free—fixing bugs in it that may break your product, upgrading to its new releases that may change in arbitrarily incompatible ways, and so on. Python 3.X's own slow uptake should be enough to prove the point: the cost of porting code may preclude migration for many, even after 2.X support is dropped. Given the constant flux in open source software, "free" isn't always as free as you may think.

This is a subjective topic, of course, and I'd very much welcome a differing opinion from an interviewee; interviews should be intellectually rewarding experiences on both sides of the table. But cut-and-paste development comes with major consequences. Blindly parroting the mantra that code reuse always beats writing new code could be a sign of a shallow perspective that might just produce code maintenance nightmares in the long run.

Why would you use the super() call, and why would you not?

What people might say: "super() is awesome, because it works just like it does in Java; you should use it whenever you can, instead of calling methods by class name."

That opinion is common; it's also uniformed or worse. This is a fairly heavy technical question, but it's fair game, given the growing prevalence of super() in code you're likely to stumble across these days. In short, super() has two primary roles (there are others, but they're more obscure, and can usually be achieved with simpler techniques):

- In *single-inheritance* class trees, super() can indeed be used to invoke a method in a superclass generically. This role is essentially as it is in Java, at least for trees that will never grow to include multiple inheritance.
- In *multiple-inheritance* class trees, super() can also be used for cooperative method-call dispatch, which routes a method call to each class just once in conforming trees. This role is more unique to Python, and works by always selecting a next class on the MRO following the caller and having the requested attribute.

So why and why not deploy it? super()'s *upsides* are the two roles just mentioned—you don't need to list any involved class by name (in Python 3.X, at least), and can route superclass method calls coherently through larger trees. Unfortunately, super()'s second role is also a massive *downside*—its automatic method routing makes for a wildly implicit code invocation model, one that can obscure a program's meaning, create deep class coupling, hinder customization, and complicate debugging.

In fact, super() might not invoke a superclass at all, and the class it calls will normally vary from tree to tree in ways you may not expect. This downside manifests itself most sharply in multiple inheritance trees, but that's a common pattern in realistic Python code, and even single inheritance trees often grow to have multiple parents. By adding a special case for attribute lookup which skips normal inheritance altogether, it also adds to your knowledge requirements. The traditional Python pattern of calling methods by explicit superclass name can often yield better control and clearer programs.

If you really want to impress, super() has three requirements that are behind most of its usability issues—*call-chain anchors, uniform argument signatures*, and *universal deployment*—but I wouldn't expect most junior-level programmers to have memorized such things. An interviewee that can describe *any* of super()'s downsides in the abstract would get my vote. On the other hand, a person who only lauds the Java-like role in single-inheritance trees would strike me as someone who will probably code Java in Python, and worse, may be prone to pepper a code base with complex and obscure tools in some misguided effort to prove personal prowess—usually at the expense of intellectual property, coworkers, and common sense. That's not good software engineering.

Have you ever written a perfect program?

What people might say: "Yes! And you can expect more if I'm hired!"

Of course not! Nobody has ever written "perfect" software, and responding otherwise might just suggest a towering ego that could someday sink an entire project. Perfection doesn't happen in code. Trust me on this; after getting two degrees in the field, working in it for 30 years, and writing 14 books about it, I still make programming mistakes on a regular basis. On a theoretical level, this is probably inherent in the models we use to program machines today. Given that this won't be changing any time soon, though, the best one can do is be forgiving of the mistakes of others; expect the same in return; and test like crazy.