Beyond values

Notes on sameness in Python distributed systems

Ben Clifford

July 22, 2023

Contents

Chapter 1

Sameness

Lots of us probably have some gut feeling for what a *value* is in Python: for example, `3`, `"hello!"`, or `[1,2,3,4,5]`.

And probably have ideas about values are *the same* as each other. For example, `3` is the same as `3` but different to `4`.

Probably we'd also generally think that the integer `3` is mostly the same as the floating point `3.0` (for example, evaluating the Python expression `3 == 3.0` tells us that part of the Python runtime thinks so...)

We'd also probably think that the dictionary `{"a": 1, "b": 2}` is the same as the dictionary `{"b": 2, "a": 1}`.

Maybe if you're very Python, you'd also believe `1 == True` (as the Python runtime does) even though you can clearly tell them apart.

Often sameness of Python values is easy (like `3 == 3`) but the other examples above are intended to hint at a lot of complexity just below the surface. In these notes I want to dig into that complexity, as has affected my work on a couple of distributed systems written in Python (Parsl and Globus Compute).

1.1 Referential transparency

TODO: maybe this belongs in the rules-of-equality section now? with mention of referential transparency only there? But with simple examples in the "sameness" section, not focusing too much on strict rules...

I'm going to talk about this implication:

$$x = y \implies f(x) = f(y) \qquad \forall x, y, f$$

This looks like some weird logic formula... but it's capturing something that when you're think about mathematical values and functions, you assume is true without thinking about it much.

Here's an example, which I'll write in Python:

```
>>> def f(v):
...   return v+1

>>> x = 3
>>> f(x)
4
```

What is this going to evaluate to?

```
>>> y = 3
>>> f(y)
```

It's (obviously?) going to return `4` - but there's two ways we can reason about this: we can type that into the Python interpreter and look at the result of executing `f(y)`...

... or we can say well, f(x) must be the same as f(y) because x=y and I already know that f(x) is 4, so f(y) must also be 4.

1.2 functools.cache

In essence that's what's happening when you use the `@cache` decorator from the `functools` package like this:

```
>>> @functools.cache
... def g(v):
...     print(f"Calculating for {v}")
...     return v+1

>>> g(x)
Calculating for 3
4

>>> g(y)
4
```

The second time we invoke `g`, the decorator sees that it's already been invoked with an equal parameter, and doesn't run the underlying body, instead relying on the return value it cached.

In Parsl, at a high level, this is also what happens when you turn on memoization and checkpointing to avoid re-executing Parsl task invocations.

1.3 Functions with effects

When you are doing this in a language like Python, rather than in the abstract maths world, it relies on the function (in the Python sense) behaving (mostly) like a function (in the maths sense) - it has to be *pure*, which means it does not have *effects*.

But, it also relies on equality working correctly - something that we usually don't think about. For example, here are some obviously True (or False)" Python relations: `3 == 3`, `3 != 4`, `[1,2] == [1,2]`, `[1,2] != [5,6,7]`, and so on.

In the rest of this note I want to dig into some specific cases where this doesn't work.

First, functions can have effects (sometimes called side-effects). There's already one of those in the example I pasted above: I put a debugging `print` statement into the definition of `g` so that we could tell if it ran or not.

In this case, it was deliberate, and we can say it's ok to have debugging prints, because they don't affect the essential functionality: to return an incremented number. But this behaviour would be wrong if the effect was part of the essential behaviour.

For example, a function `h(count)` that operate hardware to dispense a given number of cat treats and return `True` if that worked ok: it would be incorrect to return `True` without dispensing anything. In this case, it's ok to describe `h` as a *Python* function, but it's not a function in the *mathematical* sense.

Effects don't have to change the world to make a Python function not be a mathematical function any more: for example, a Python function that observes the time (via `time.time()`) might return different values depending on when it is invoked - and so the output wouldn't be entirely defined by the input argument.

Other easy examples that I run into in this space are computing random numbers (`random.random()`) and inspecting the runtime environment (`platform.node()`).

Chapter 2

Objects

Values live in objects, and we can think about sameness of objects, distinct from sameness of values. Sometimes we don't think about this distinction very hard.

2.1 Observable object identity

Another effect is when Python code looks not at the "value" (whatever that is) of an object, but at the nature of the Python object itself. Most brazenly, there is the `id` function, which returns:

> Return the "identity" of an object. This is an integer which is guaranteed to be unique and constant for this object during its lifetime.
>
> – `https://docs.python.org/3/library/functions.html#id`

For example, with objects that represent large numbers, the values of two objects compare the same (with `==`) but the `id` function returns different values:

```
>>> x = 1000000

>>> y = 1000000
>>> x == y
True
>>> id(x) == id(y)
False
>>> id(x)
```

```
139863203106896
>>> id(y)
139863203107280
```

"Interestingly," this doesn't happen with small integers, at least in cpython: small integers (allegedly -5 to 256) are interned and are only ever represented by a single object.

2.2 Object identity as equality

TODO: combine with previous chapter? or rework the separation

This leads to a second notion of equality in Python: in addition to the `==` value comparison operator, there is `is`, an operator that checks if two objects are the same object.

This operator looks at object identities, not at values, but it is still in some sense an equality (or equivalence) operator. We can think about the original referential transparency implication using 'is' as a different equality on one side:

`x is y` $\implies$ `f(x) == f(y)`

This holds in some different cases, for example, `x is y` $\implies$ `id(x) == id(y)`, but note that this doesn't work: `x is y` $\implies$ `id(x) is id(y)` - although id will return the same number, that number might be represented by different objects in each case!

Different equalities (or equivalences - I'm being deliberately vague about the difference here) can be related to each other: for example, an equivalance can be *finer* than another equivalence.

For example, we might expect that: `a is b` $\implies$ `a == b`.

That's a way of saying that an object is always `==`-equal to itself. This is *reflexivity* and it's another "obvious" thing that we don't always think about but sometimes rely on.

There's at least one counterexample built into Python (and you can create your own in your own `__eq__` implementations) - it's an ongoing "controversy" with IEEE754, the standard for floating point arithmetic.

```
>>> x = math.nan
>>> x is x
True
>>> x == x
False
```

Where would you run into this in practice? If you were looking for a float in a data structure using `==`: you would never find `nan`.

Luckily people are culturally scared of using floats as keys, so actually not so much in practice.

2.3 Singletons

note (PEP-8?) coding styles with singletons - this requires the use of `is` with singleton classes such as None. There is only ever one None object so it's fine to use `is` here: there won't ever be another object "equal"(?) to None that is not that singleton object. (unless you do weird stuff with ==) – contrasts with next section, that not only is there only one representation, there is also only one object containing that representation.

A classic commonly-shown example of behaviour is here this potentially surprising behaviour: for x,y = 1, x is y, but for (suitably distinctly defined) x,y = 1000000, x is not y.

2.4 Different representations of equal values

Another place where the notion of value becomes a bit fuzzy is when there are different representations of the `==`-same value: such as with dictionaries.

Here are two `==`-equal dictionaries: the keys and values are all well behaved objects (`str`), they have the same keys and each key maps to the same value in both `x`, `y`.

```
>>> x = {"a":"one", "b":"two"}
>>> y = {"b":"two", "a":"one"}
>>> x == y
True
```

They're different according to `is`, because they are different objects, which shouldn't be too surprising:

```
>>> x is y
False
```

but we'd expect functions (and expressions) that look at the "value" of the dictionary to be the same for both `x` and `y`...

```
>>> x['a'] == y['a']
True
>>> len(x) == len(y)
```

```
True
```

But, here's a subtlety:

```
>>> list(x.keys()) == list(y.keys())
False
```

Somehow the keys of a `dict` are not value-like because we can use them to distinguish between the two `==`-equal objects, `x` and `y`.

Where would you run into this? Perhaps we're iterating over the entries in a dictionary - Parsl does this when computing the checkpoint hash of a `dict` object. You won't necessarily encounter the entries in the same order (so the parsl code *normalises* the list of keys by sorting - which restores the equality-behaviour [TODO code sample?]).

2.5 UUIDs

Another related problem could be using UUIDs: A UUID represents a 128 bit number, but usually have multiple string representations (because the hex values a-f can each be written as a lower-case or upper-case letter, and TODO PERHAPS: leading zeroes can be omitted in each section) so the equality of UUIDs-rendered-as-strings is not the same equality as equality-of-UUIDs:

```
>>> import uuid
>>> s1 = "af240119-eaa7-4d3d-b5ee-0a1d9064e3bd"
>>> s2 = "aF240119-eAa7-4D3d-B5eE-0a1D9064e3Bd"
>>> s1 == s2
False
>>> uuid.UUID(s1) == uuid.UUID(s2)
True
```

The equalities are related: `str`-equality implies `UUID`-equality, but not the other way round. TODO: perhaps a forward reference to the section on relations between equalities here?

TODO: a note that one of the reasons for encapsulating values inside another class (eg making a UUID class rather than using str) is that equality is now the equality of your domain object, not of string – so then things that use equality (sets, membership testing, ...) now use the equality of your domain object.

2.6 bool as a subclass of int

Another equality awkwardness arises with subclasses: in Python, `bool` is a subclass of `int` (for, I think, historical reasons) and the two elements of that type, `False` and `True` are `==`-equal to `0` and `1` respectively.

TODO: reference for PEP that introduced the bool type, PEP-285, `https://peps.python.org/pep-0285/` and which contains lots of juicy text including about behaviour of equalities.

```
>>> False == 0
True
>>> True == 1
True
```

which can behave surprisingly with dictionaries:

```
>>> d = {}
>>> d[1] = str(1)
>>> d[True] = str(True)
>>> d
{1: 'True'}
```

This sort of behaviour can arise with user defined enums too – although in Python 3.11, for user defined `IntEnum` in this specific `str` case, that code now returns the underlying integer to make this less surprising - a change that has caused some different surprise in Parsl log file output.

2.7 Mutability

One final awkwardness comes with the passage of time, or mutability: objects in Python are often mutable, which means their value can be changed. Mutability means that the equality behaviour between two objects can change. For example, with the pretty strong `is` equality relation, we usually have x is x $\implies$ str(x) == str(x). But that doesn't work over time - for example if we make a function memoization cache keyed by a reference to the object x, then change the value of object x, the memoization will break: object identity over time doesn't imply value identity over time.

TODO: example from parsl here: open bug on returning a mutable object from @cache to a user, where the user can then perform arbitrary mutations - `https://github.com/Parsl/parsl/issues/`

2555. For two identical bytestrings x == y, deserialise(x) == deserialise(y) ... but that doesn't hold over time if the object returned by deserialise(x) is mutable.

Chapter 3

Rules of equality

3.1 Equivalence relations

I've talked about two different kinds of sameness: `==` and `is`, but there's a more general framework (with more examples to come) with some rules that make things look nice.

The mathematical structures are called *equivalence relations*.

I mentioned *reflexivity* before: We would like: $x = x$ for every value `x`.

There are a couple of others that we need too:

Transitivity: If $x = y$ and $y = z$, then $x = z$

A counter example to that might be if we define an equality for real numbers where x == y if x and y are within 0.1 of each other: this doesn't satisfy transitivity, because 0.0 = 0.07 and 0.07 = 0.14 but 0.0 != 0.14

symmetry: if $x = y$ then $y = x$

This means that it doesn't matter which way round we compare things.

3.2 referential transparency 2

the rules of equivalence relations don't tell us that two objects that are equivalent will behave the same: for example, (probably I'll have this earlier on too), 1 == True, but str(1) and str(True) don't give the same output.

That sort of behaviour, which is often desirable, is called referential transparency - in a purely functional world, it talks about how a function behaves when it's applied to two "equal" (or equivalent) values:

$$x = y \implies f(x) = f(y) \qquad \forall x, y, f$$

The referential transparency equation I started with, about how equality and functions should interact: If $x = y$ then $f(x) = f(y)$.

The example above shows that this doesn't work in general for Python but it's an interesting way (I think) to consider which functions will behave "right" when you're working with any particular notion of sameness.

3.3 equality dunder methods

TODO: notes about how `==` turns into invocations of `__eq__` on the left and right objects, with reference to the python documentation

in this model of equality, a class gets to play a part in deciding if its objects are equal to other objects. Python does not force a definition of equality. How does this work and what do we need to be careful of? (especially thinking about the rules in the previous section...)

`https://docs.python.org/3/reference/datamodel.html#object.__eq__`

Key points to summarise here:

notimplemented to passthrough to someone elses impl: (I should write what happens when you fall through with NotImplemented? because it isn't included in the quote)

> By default, `object` implements `__eq__()` by using `is`, returning `NotImplemented` in the case of a false comparison: `True if x is y else NotImplemented`
>
> – from datamodel.html

order of fall through of NotImplementeds -

> If the operands are of different types, and right operand's type is a direct or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority.

TODO: some practical code/examples of how these need to align/how they work in the real codebase? Perhaps IntEnum vs int comparison implementation? Ideally I'd like something that can behave

”surprisingly” – perhaps if you naively compare your own a,b values against the opposite, but the opposite is a subclass? that’s actually something this fallthrough rule deliberately tries to avoids... but i think if a subclass adds a field c and does not implement a new eq, then we’re screwed

(related to this is that what data classes can help with? I think thats what the `eq` paramter for DataClass is for... it makes an eq for you...)

another example: so what’s a better one? Something that breaks symmetry - would show that one way round we evaluate the code for x first and the other way round we evaluate the code for y first, and we need to hope/ensure that the two implementations agree... falling through is one way to do that...?

TODO: rephrase this note moved from the section on equivalence relations, about symmetry: Where that can have practical effect in Python is in choosing which `__eq__` gets to run first, `x.__eq__` or `y.__eq__`. Nothing forces those two implementations to return consistent results.

3.4 relations between equivalences

TODO - bring talk about relating equivalence relations to each other, eg. being finer, here into this more theoretical section - leaving the section that introduces `is` slightly cleaner.

from wikipedia `https://en.wikipedia.org/wiki/Equivalence_relation#Comparing_equivalence_relations`

if x =A b implies x =B y for all x,y then =A is *finer* than =B (or phrased the other way round, =B is *coarser* than =A).

Some of the equivalence relations/equalities in these notes are related to each other in this way, or are almost related, and where they aren’t is a source of bugs:

for example, `is` is finer than `==` if we ignore the problem object NaN (which we have to do anyway if we want to treat `==` as an equivalence relation at all, because of symmetry) – that means if x is y, then we can assume that x == y.

(and indeed, the next section deals with that NaN weirdness...)

3.5 x in y

`is` and `==` aren’t used only when explicitly invoked by a user, but are also used as part of other pieces of code that need to know if values are the same.

We've seen this already with `functools.cache` which needs to know about sameness in order to decide if a particular parameter already has a cached result or not.

A common use is inside the `in` operator: `v in c` is `True` when the value v is contained in some collection. To do this, `in` needs some notion of equality so it can, for example, visit every element of `c` and ask if it is equal to `v`.

(TODO: what's actual python abstract type for the RHS of c?)

We might expect `in` to use one of the two equalities already mentioned: `==`-equality or `is`-equality. But, a bit of testing will show it isn't either of those:

If `in` was using `==`-equality, then it would not be able to determine that math.nan was in a list, like this:

```
>>> import math
>>> math.nan == math.nan
False
>>> math.nan in [math.nan]
True
```

If `in` was using `is`-equality, then it would not see that the integer 3 and the floating point 3.0 are the same value:

```
>>> 3 is 3.0
False
>>> 3 in [3.0]
True
```

This actual equality test is implemented in C, in `PyObject_RichCompareBool` in cpython's `object.c`. This code has a shortcut for object identity that regards two objects as equal if they are either `is`-equal or `==`-equal.

That forms a useful equality which, along with the fact that math.nan is a singleton, fixes up the missing reflexivity of equality of math.nan.

3.6 functools.cache, again

UGH all of the below is not right: functools cache has weird equality behaviour with shortcuts and relies on hash() quite a lot (not sure if that forces equality...)

weirdnesses: short cuts for int and str, which makes a simple example seem like it can distinguish always between 3 and 3.0 (which is not true) – there's a typed option to skip that.

so *mostly* it will use dict equality, but with an extra shortcut on the front that makes things behave differently in some simple special cases.

So it's a good thing we can turn on typing, but it's a bad thing (probably) that this 3 vs 3.0 shortcut codepath exists...

hashes aren't really relevant in this explanation (even though they're used as part of the hash key calculation... that's invisible to the equality relation)

The interesting takeaway, I think, is that functools.cache usually but not always respects ==-equality.

—

Let's revisit functools.cache.

The equalities we've seen so far might make you quite uncomfortable:

if `1 == True` we might expect functools.cache to behave quite badly with a function like this:

```
@functools.cache
def s(v):
    return str(v)
```

Uncached `s(1)` and `s(True)` would return different results (the strings `'1'` and `'True'`) - but with the various Python equalities we've seen so far, we might expect `functools.cache` to be unable to distinguish between them and share a cache entry, returning whichever was computed first.

But that's not what happens:

```
>>> s(1)
'1'
>>> s(True)
'True'
```

`functools.cache` uses its own equality, which tries harder to distinguish between values that other equalities would treat as the same.

Chapter 4

Hashes

TODO: this intro is about cross-project objects, not about hashes... so move it/integrate it elsewhere?

So far, I've talked about different equalities between objects in the same Python process. In that situation, we can invoke some Python code, such as the `__eq__` dunder method, to compute if two objects are equal or not.

We're not always so lucky: sometimes Python objects exist in different processes, separated by time or space (or both). In this situation, we can't pass two objects as parameters to some equality testing code, so what does equality mean in that situation?

TODO: maybe some more discussion here about what equality actually does mean here? (move some of the discussion from the serialisation section here...?)

4.1 Hashes as a coarser equivalence

Some objects can be hashed, and we can hope that that hash is a (mathematical) function of their value. Depending on the characteristics of the hash, there are a couple of ways we can use this:

For any hash, we can assume that: $\text{x} = \text{y} \implies \text{hash(x)} = \text{hash(y)}$ and so we can do things like put stuff in hash buckets: if we're looking for y, we know that if some x exists already where x = y, it will be in hash bucket hash(y) (aka. hash(x)) - we can forget about all the other hash buckets entirely.

TODO: we can assume it with python's hash(), `__hash__()` because it's a requirement in the data model:

> The only required property is that objects which compare equal have the same hash value
> – `datamodel.html#object.__hash__`

This works (slowly) even if the hash function is poor: for example, the constant hash(x) = 7 - where it's quite easy to see that this relationship holds: $x = y \implies 7 = 7$.

(There was a performance bug in the Globus Toolkit string hash function sometime around 2004, where it only hashed a prefix of strings, and that prefix was often the same - `globus_` - so the hash function effectively became a constant) TODO: github URL to the relevant commit? - that was essentially exactly this "constant hash" performance problem.

The meaning you can get from hashes is "if two hashes are different, then the vaues are different" – there's no implication that if two hashes are the same, the values are the same, so you still have to make some better equality test, but you can avoid that (usually more expensive) test in many cases because of this implication. Compare that to the `PyObject_RichCompareBool` test: that technique and this hashing technique both make a quick test that is inaccurate in a specific way (`is` might incorrectly say two values are different; a hash might incorrectly say that two values are the same) and then if needed perform a more expensive test to get the right answer.

TODO: note that dict (I think) requires that you can make a hash to use as a dictionary key (not value) ? - citation to source code, it's for performance reasons?

4.2 Strong hashes as equality

TODO: what's the technical term for this kind of hash? strong? (cryptographically secure is another related term)

With stronger hash functions, we want the relationship to go the other way too: $x = y \iff \text{hash}(x) = \text{hash}(y)$. With that we can do parsl style checkpointing, which makes a hash of the parameters to a function, and uses that as a cache key in an external database. This is only safe because of the stronger relationship between = and our checkpoint hash: we need to be sure that $x \mathrel{!=} y \implies \text{hash}(x) \mathrel{!=} \text{hash}(y)$. In some sense, we need the `==`-equality and the hash equality to be "the same" equality: that is they make the same decision about whether two objects are equal or not, in all cases.

In the Parsl checkpointing code, one of the biggest complications is computing such a hash for complex objects, using the parsl `id_for_memo` function: we need a hash value that is consistent in the face of different representations of the same object (see the discussion about `dict` above) and we need a hash that is consistent *between Python processes*, so that when a Parsl workflow is re-run, function invocations with the "same" parameters will be found in the checkpoint database.

TODO: git commit IDs (and other similar IDs used inside git) are a place outside of Python where you might have encountered this. (comedy ensued when someone found a collision: TODO hyperlink to that)

Chapter 5

Equality by construction

5.1 Serialization

I put "same" in quotes there, because here's a new, more abstract kind of equality. The kinds I've talked about before, `==`, `is` and hashing are all equalities that you can compute inside a single Python process (by evaluating the Python expression `x == y` in that Python process, for example).

But what does it mean for values to be equal when they are kept inside different Python processes? At the code level, there's no longer a way to compute whether two objects represent the same value by passing into some comparison operator or function (like `==`) and they are definitely different objects so the `is` operator is meaningless.

We've got serialisation, and what we care about in Parsl is something involving serialization and equality: if in my first process I have a function and a value, and I send the function and the value to another process, and in that second process apply the function to the value, and then send the result back, then I end up with a value that is equal to if I applied the function to the value locally. (like with checkpointing, we're describing a computation and then instead of performing that computation, doing something that we believe to be equivalent)

TODO: looping diagram of (f,v) -¿ r, -¿ (f',v') -¿ r' -¿ r

So there's a notion of equality-by-serialisation: if I take an value in one process and deserialise it so that it behaves the same, in the above sense, then that value in one process is equal to the value on the other process. This is *by construction*, not *by comparison* - you can construct an equal object on the remote system, but not test an existing distant object for equality directly.

Getting serialisation right (in this value equality sense) can be quite complicated. In the Parsl and Globus Compute world, the most complexity comes from getting functions and classes from one place to the other, with three different ways in use: if a function or class seems likely to be installed remotely (based on some not-always-right heuristics in `dill`) then it will be referenced by name; otherwise `dill` can attempt to send Python's internal representation (which does not work well when each end is a different version of Python) and finally a Globus Compute specific method will try to send the source code for a function (which doesn't work when a function doesn't have source code).

This latter method has echoes of Python's `repr` function:

> For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval();
> – TODO citation of quote

see also the "serpent" serialiser that pretty much targets being a repr() style serialiser intended to use pythons ast string evaluator as the de-serialiser.

5.2 Proxying objects

This pretty loose model of serialization leads to a potential parsl serialization API that is similarly loose - for plugging in novel systems for moving objects around (such as ProxyStore TODO citation), this loose view of "what we want is an object that behaves 'the same' but it doesn't really matter how that object appears" model means a store-like system like proxystore is also a serializer. - there's an interesting related issue in ProxyStore (and in making "proxies" in general: `https://github.com/proxystore/proxystore/issues/311` - where a proxy object tries hard to pretend to behave like the object it's proxying... but it is a distinct object, so it's still possible to detect it's different, and in the case of this bug, especially, for singletons like None, where coding style in PEP-8 (reference?) is to use `x is None` rather than `x == None` - where we are caring about object identity not value identity... a proxy object cannot "be" a singleton in this way. In general when you "deserialise" (in the abstract sense) a singleton, you need to return the singleton, not create a new singleton.

5.3 Duplicating a configuration

TODO: maybe a section on parsl's config mechanism: "active" objects (that are not like data-classes, aka. structs, that do stuff... so you can't "copy" them easily... and structures that are constructed actively - for example, with a helper like "get local host's IP addresses"...

what does it mean to copy this config, to get "the same" config? should we store the IP address that we were configured with, or should we store that the IP address was acquired by a particular helper function?

leads to `fresh_config()` notion in Parsl test suite... "every time you run this function, you'll get 'the same' configuration, where sameness is configuration-sameness, not individual-field-sameness"

Index