`Is` vs `==`

- `is` operator compares the identity.
- `is` operator checks if both the variables refer to same memory location.
- `is` function compares the “identity” of an object. The identity can be generated using `id()` function.
- `is` operator, in most of the cases, should be used when comparing with `None`
- `id()` In short, this is the address of object in memory (CPython implementation). Function returns an integer which is guaranteed to be unique and constant for an object during its lifetime. Two objects with non-overlapping lifetimes may have the same `id()` value.
- `==` operator compares values instead of memory locations.

```
>>> x = 42
>>> id(x)

# check output?

>>> y = x
>>> id(x), id(y)

# check output?

>>> y = 78
>>> id(x), id(y)
```

`is` operator usage

```
>>> foo = ["a", "b", "c"]
>>> bar = foo
>>> bar is foo
```

```
True
>>> bar == foo
True

# Now, let's create a new list object using slicing
>>> bar = a[:]
>>> bar is foo
False
>>> bar == foo
True
```

Thumb rule of Object equality vs Identity

- `==` is for value equality.
- `is` is for reference equality.

BUT,

`is` operator behaves differently with integers

```
>>> a = 10
>>> b = 10
>>> a is b
True

>>> # Now, similar operation as we did above
>>> a = 1000
>>> b = 1000
>>> a is b
False

>>> # This is because ID changes.
>>> # Python maintains array of smaller integer objects in cache.
>>> # So those objects already exist
>>> # and they have same "identity" throughout the runtime.
```

Let’s try similar example with strings -

```
>>> a = "prashant"
>>> b = "prashant"
>>> a is b
True
```

```
>>>
>>> # Now, similar operation as we did above by just adding `space` in string
>>> a = "prashant jamkhande"
>>> b = "prashant jamkhande"
>>> a is b
False

>>> # Alphanumeric string literals always share memory
```