Crimes Against `jax.jit`

CS 5757

JAX is an excellent library for scientific computing, as it supports JIT compilation, automatic differentiation, vectorization and more. But it also produces some of the most inscrutable error messages known to man.

This document covers common "crimes" against JAX's tracing system that will land you in Python jail (i.e., debugging a `ConcretizationTypeError` at 3am). For a comprehensive treatment, see JAX's doc on The Sharp Bits.

Before we get into the crimes, a quick note on why we need these "laws." JAX's `jit`, `vmap`, and `grad` run your function once with abstract placeholder values ("tracers") to record the sequence of operations, then compile that sequence. Your function must describe the *same computation* regardless of concrete input values. Many standard Python patterns violate this.

Remember: Friends don't let friends write non-pure functions.

Crime 1: Side effects

The crime: Writing non-pure functions (any function that mutates variables outside its scope).

```python
results = []

def f(x):
    y = x ** 2
    results.append(y) # CRIME: side effect captures tracer
    return y

jax.jit(f)(3.)
print(results[0] + 1) # UnexpectedTracerError: tracer leaked out
```

Side effects happen at *trace time*, not run time. The tracer gets captured in `results`, then causes errors when you try to use it later as a real number.

JAX functions must be *pure*: all inputs come in as arguments, all outputs go out as return values. No exceptions.

Fix: Return everything explicitly.

```python
def f(x):
    y = x ** 2
    return y # return all outputs; no external state
```

Crime 2: Branching on traced values

The crime: Using Python control flow that depends on traced values.

```
def f(x):
    if x > 0: # CRIME: x is traced, can't branch on it
        return x
    else:
        return -x

jax.vmap(f)(jnp.array([-1., 2., 3.])) # ConcretizationTypeError
```

Python's `if` requires evaluating `x > 0` to a concrete boolean at trace time. But `x` is a tracer—a symbolic placeholder—not a real number.

Same goes for `assert` statements: `assert x[0] > 0` is secretly a branch.

Fix: Use `jnp.where` for simple cases, `jax.lax.cond` for general branching.

```
def f(x):
    return jnp.where(x > 0, x, -x) # OK: both branches always evaluated
```

Note: branching on *static* properties (shapes, dtypes) is fine: `if x.shape[0] > 0` works.

Crime 3: Python loops over traced values

The crime: Using traced values as loop bounds or conditions.

```
def f(x, n):
    y = 0.
    for i in range(n): # CRIME if n is traced
        y = y + x[i]
    return y
```

If `n` is a tracer, Python can't determine the iteration count at trace time.

Fix: Use `jax.lax.fori_loop`, `jax.lax.while_loop`, or `jax.lax.scan`.

```
def f(x, n):
    return jax.lax.fori_loop(0, n, lambda i, y: y + x[i], 0.)
```

Loops over static quantities (`range(10)`) are fine, since they just "unroll" at trace time.

Crime 4: Dynamic shapes

The crime: Creating arrays whose shape depends on traced values.

```
def nansum(x):
    mask = ~jnp.isnan(x)
    return x[mask].sum() # CRIME: output shape depends on values

jax.jit(nansum)(jnp.array([1., jnp.nan, 3.])) # NonConcreteBooleanIndexError
```

The number of non-NaN elements isn't known at trace time. JAX requires all shapes to be determinable from input *shapes*, not input *values*.

Fix: Use `jnp.where` to keep shapes static.

```
def nansum(x):
    return jnp.where(jnp.isnan(x), 0., x).sum() # OK: shape unchanged
```

Crime 5: In-place mutation

The crime: Assigning to array elements.

```
def f(x):
    x[0] = 0. # CRIME: JAX arrays are immutable
    return x
```

JAX arrays cannot be mutated. This errors even outside JIT.

Fix: Use `.at[].set()` for functional updates.

```
def f(x):
    return x.at[0].set(0.) # OK: returns a new array
```

Crime 6: Extracting scalars

The crime: Casting traced values to Python types.

```
def f(x):
    idx = int(jnp.argmax(x)) # CRIME: int() requires concrete value
    return x[idx]
```

Calling `int()`, `float()`, `bool()`, or `.item()` forces concretization.

Fix: Keep values as JAX arrays.

```
def f(x):
    idx = jnp.argmax(x)
    return x[idx] # OK: indexing with traced array works
```

Crime 7: Using `numpy` instead of `jax.numpy`

The crime: Calling `numpy` functions on traced values.

```
import numpy as np

def f(x):
    return np.sin(x) # CRIME: np.sin doesn't understand tracers

jax.jit(f)(jnp.array([1., 2., 3.])) # TracerArrayConversionError
```

`numpy` functions try to convert inputs to `numpy` arrays, which fails on tracers.

Fix: Use `jax.numpy` inside transformed functions.

```
import jax.numpy as jnp

def f(x):
    return jnp.sin(x) # OK
```

The Golden Rule

Your function must describe the *same computation graph* for all possible input values of a given shape and dtype.

Mental model: Imagine JAX runs your function with "?" values—they have shape and dtype but no actual numbers. Every line must work without knowing concrete values.

Debugging Tips

When things go wrong:

Print traced values with `jax.debug.print`, not `print`:

```
def f(x):
    jax.debug.print("x = {}", x) # prints at run time, not trace time
    return x ** 2
```

Disable JIT temporarily to get normal Python errors:

```
with jax.disable_jit():
    result = my_function(x) # runs eagerly, normal debugging works
```

Check for escaped tracers by calling your function outside JIT first. If it works eagerly but fails under `jit`, you have a tracing crime.

Read the error message. JAX errors are verbose but usually tell you exactly which line and which value caused the problem. Look for "was created on line..." in `UnexpectedTracerError`.

Quick Reference

Crime	Fix
mutating external state	return all outputs explicitly
`if x > 0`	`jnp.where(x > 0, ...)` or `lax.cond`
`for i in range(n)` (traced `n`)	`lax.fori_loop` or `lax.scan`
`x[mask]` (traced boolean mask)	`jnp.where(mask, x, 0)`
`x[0] = 5`	`x.at[0].set(5)`
`int(x)`, `x.item()`	keep as JAX array
`np.sin(x)`	`jnp.sin(x)`

Further Reading

- Control flow and logical operators with JIT
- JAX—The Sharp Bits
- JAX Errors (explains each error message)
- How JAX Tracing Works