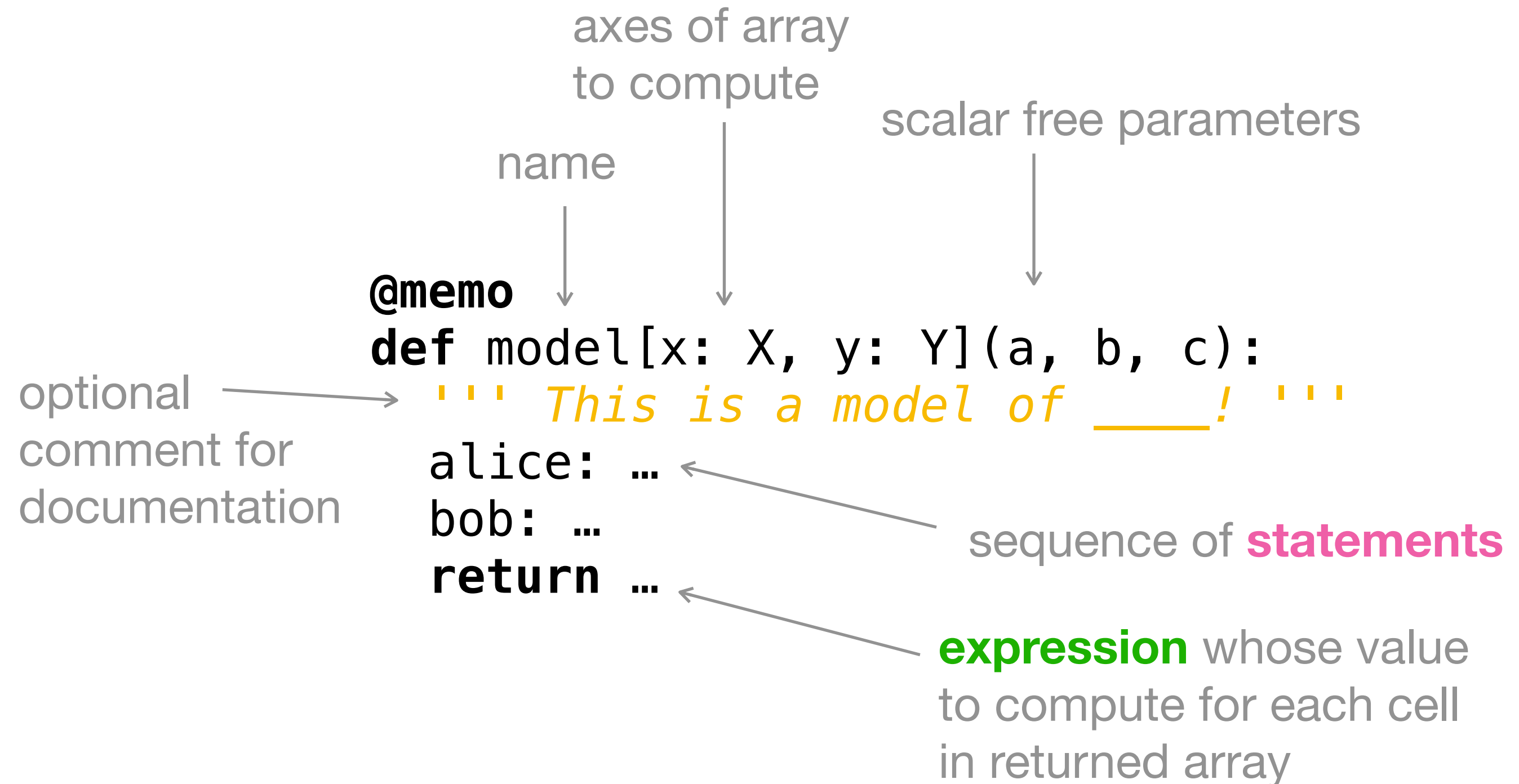


# The memo handbook

v1.2.0



# Anatomy of a memo



# Statements

# chooses

Domain of choice (name of Python list/enum or JAX array)

Agent making choice

**bob: chooses(a in Actions, wpp=exp( $\beta$ \*utility(a)))**

Name of choice

"With probability proportional to"  
For softmax, use wpp=exp(...)  
For uniform choice, use wpp=1

Can make multiple choices simultaneously

**bob: chooses(x in X, y in Y, wpp=joint(x, y))**

# more ways to choose

```
bob: chooses(a in Actions, to_maximize=utility(a))
```

↑  
For `argmax` use `to_maximize`  
For `argmin` use `to_minimize`

Aliases of "chooses" that don't imply agency/goal-orientation  
(These all mean the same, but can make your model easier to read.)

↓

```
bob: given(r in Roles, wpp=1)  
bob: draws(r in Roles, wpp=1)  
bob: assigned(r in Roles, wpp=1)  
bob: guesses(r in Roles, wpp=1)
```

# thinks

Agent doing the thinking



```
bob: thinks[  
    alice: chooses(...),  
    charlie: chooses(...),  
    ...  
]
```

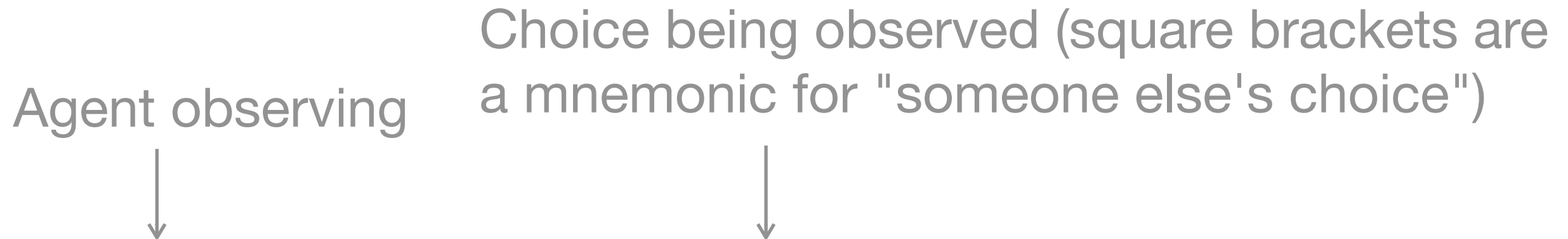


What that agent thinks  
(notice the commas!)

# observes

Agent observing

Choice being observed (square brackets are a mnemonic for "someone else's choice")



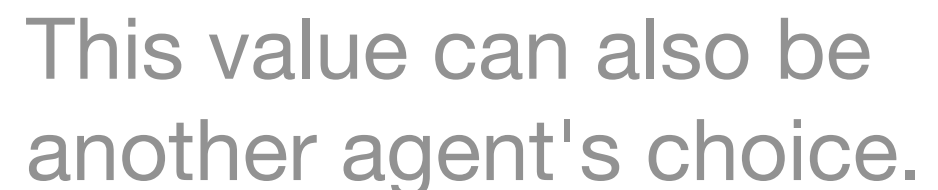
**bob: observes [alice.x] is y**

What the choice is observed to actually be.  
Can create false beliefs this way!

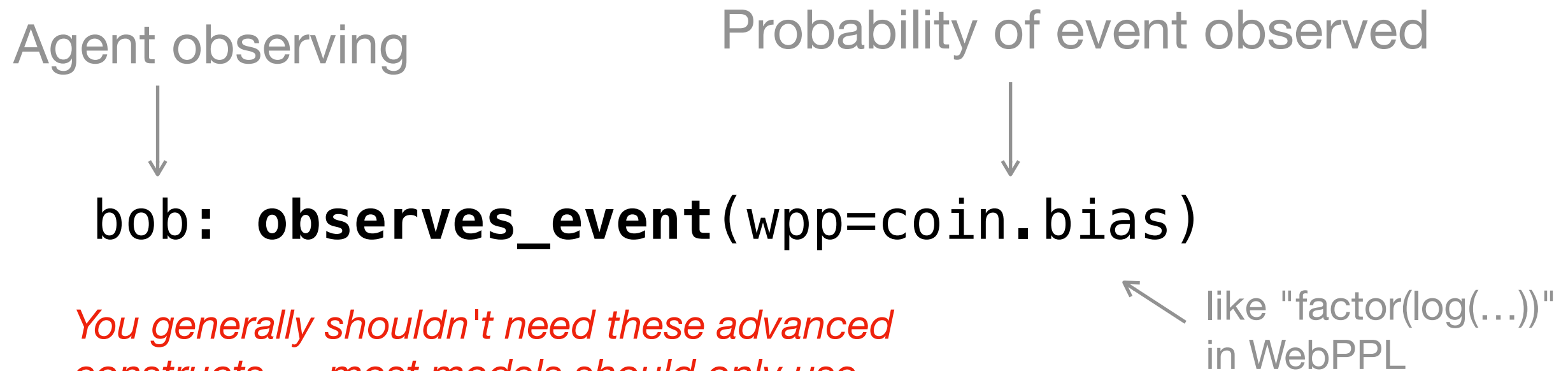
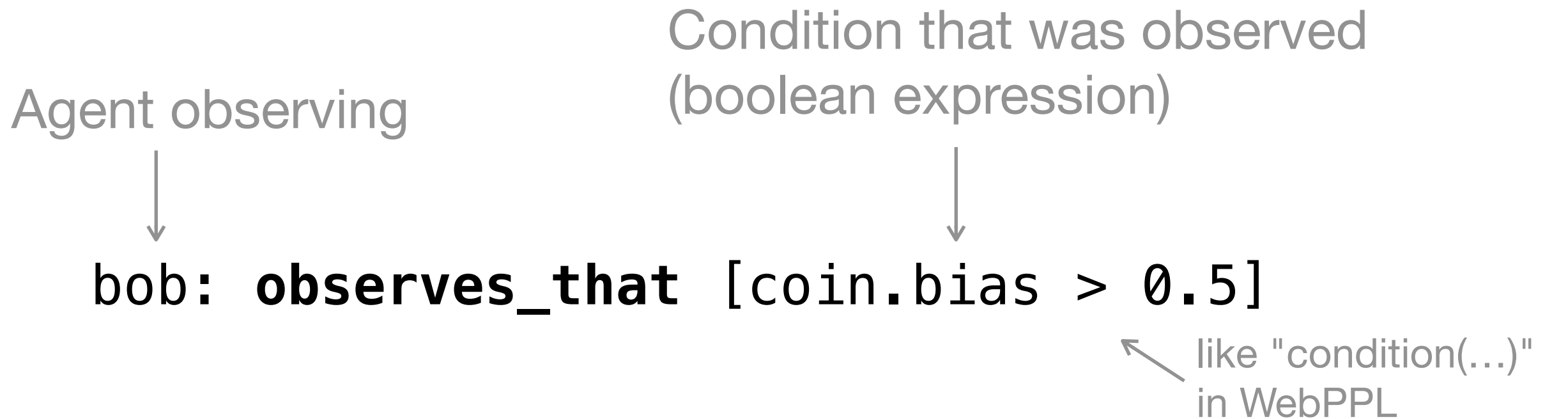


**bob: observes [alice.x] is charlie.y**

This value can also be  
another agent's choice.



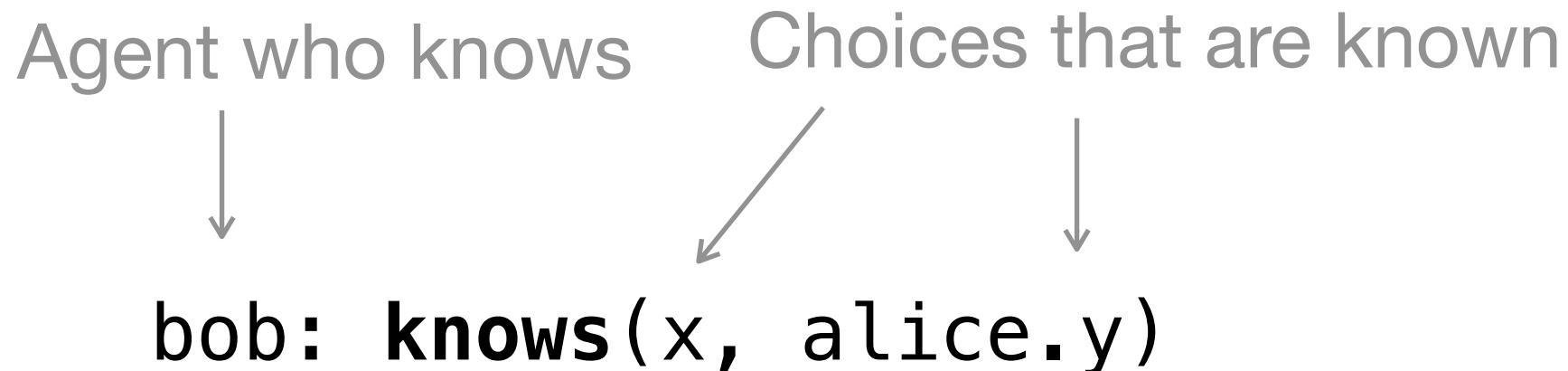
# advanced use of observes



*You generally shouldn't need these advanced constructs — most models should only use `observes` as documented on the previous page.*



# knows



This utility is useful for the common case of "pushing" a variable into an agent's frame of mind. Roughly shorthand for this:

↓

*bob: thinks[ alice: chooses(y in Y, wpp=...) ]*  
*bob: observes [alice.y] is alice.y*

# snapshots\_self\_as

Agents can remember "snapshots" of their past selves.  
Useful for counterfactuals and hypotheticals, especially  
when used with "imagine" expressions (see below...).

Agent who snapshots      "aliases" of snapshots  
↓      ↓  
alice: **snapshots\_self\_as**(past\_alice, ...)

```
alice: observes [bob.x] is x  
return alice[ past_alice[ E[bob.x] ] ]
```

not affected by "observe" statement

# wants / EU

wants sets up a predicate or utility function ("goal") that is defined over *future* states of the world. You can reference choices that have not yet been made yet. The expected future value of the goal expression can be evaluated using an EU expression.

Agent who wants      name of goal (here, "win")      predicate/utility to evaluate in the future

↓      ↙

```
alice: wants(win=score > bob.score)
alice: chooses(a in A, to_maximize=EU[win])
alice: given(score in N, wpp=...)
```

Even though "score" is only defined here, the wants and EU expressions above are okay. Conceptually, the EU expression is evaluated here. ↗

↑  
name of goal to evaluate when making goal-directed choice

# INSPECT

At compile-time, print out the current state of an agent.  
This can be helpful for debugging tricky models.

Agent to inspect



`alice: INSPECT()`

# Expressions

# literals

floating-point numbers only



**3.14**

also references to declared free parameters



**a, b, c, ...**

# operators

memo supports most Python unary/binary ops

↓  
`1 + 1`

also some free bonus functions

↓  
`exp(...), log(...), abs(...)`

can also call any function tagged with `@jax.jit`

↓  
`@jax.jit`  
`def f(x):`

`return np.cos(x)`

useful for calling deep learning, etc.  
JAX is a big ecosystem

← note: can only take scalar inputs  
and can only return one scalar output

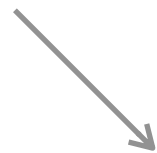
# choices

```
alice: chooses(x in X, wpp=1)
alice: chooses(y in Y, wpp=f(x, y))
```



you can refer to an agent's own  
choice as if it were simply a variable

or refer to other agents' choices with "dot" notation



```
alice.x + alice.y
```

equivalent to either of these



```
alice[x] + alice[y]
alice[x + y]
```



# probabilistic operators

expectation



**E**[alice.x + bob.z]

variance



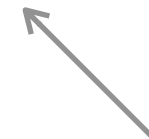
**Var**[alice.y \* 2]

probability



**Pr**[alice.y >= 0]

**Pr**[a.x > 0, b.y < 2]



can use both  
commas and  
"and" for joint

# information-theoretic operators

(mutual) entropy between choices



**H**[alice.x, bob.y, ...]

KL divergence



**KL**[alice.x | bob.y]

# queries

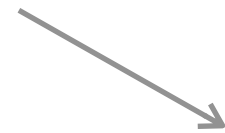
```
Var[alice[abs(x) * 2]]  
alice[bob.y == 7]
```



can "query" another agent for the value  
of an expression using square brackets

# hypotheticals

set up hypothetical world by  
running statements



```
imagine[  
    bob: chooses(y in Y, wpp=1),  
    alice: observes [bob.y] is bob.y,  
    alice[Pr[bob.x == 7]]  
]
```




last line = expression to  
evaluate in that world

# memo calls

```
@memo  
def f[x: X](a, b, c): ...
```



can reference one memo from another. (don't forget to pass parameters!)

```
@memo  
def g():  
  alice: chooses(x in X, wpp=f[x](1.0, 0.0, 3.1))
```



shorthand: can use "..." if f should be called with all of g's parameters; in this case, (a, b, c).

```
@memo  
def g(a, b, c):  
  alice: chooses(x in X, wpp=f[x](...))
```



# cost reflection

```
@memo def f[...](a, b, c): ...
```

```
cost @ f(3, 4, 5)
```



get number of FLOPs needed  
to evaluate f  
(note: no axes, params only!)

# reference to Python variable

```
class Action(IntEnum): WAIT = 0; ...
```

```
@memo def f[...](...):  
    return {Action.WAIT}
```



use braces for inline reference  
to a global Python variable

```
@memo def f[...](x):  
    return {x[0] << 4}
```



You can also refer to model parameters, and write raw Python expressions. You cannot refer to memo constructs, e.g. agents or their choices, in curly braces.

# Things to do with a memo



# Running a memo

call it like a function with params  
(returns an array with prescribed axes)



`f(a, b)`

pretty-print table of results



`f(a, b, print_table=True)`

`f(a, b, return_pandas=True)`

`f(a, b, return_xarray=True)`

get outputs in other formats

save "comic book" visualization of model via graphviz



`f(a, b, save_comic="file")`

# @memo def options

cache results (keyed by scalar parameters)



```
@memo ( cache=True )
```

trace execution, showing time taken



```
@memo ( debug_trace=True )
```

# Automatic differentiation (useful for fitting by gradient descent)

```
@memo  
def f[...] (a, b): ...
```

returns tuple of value + gradient wrt params a & b



```
jax.value_and_grad(f)(a, b)
```