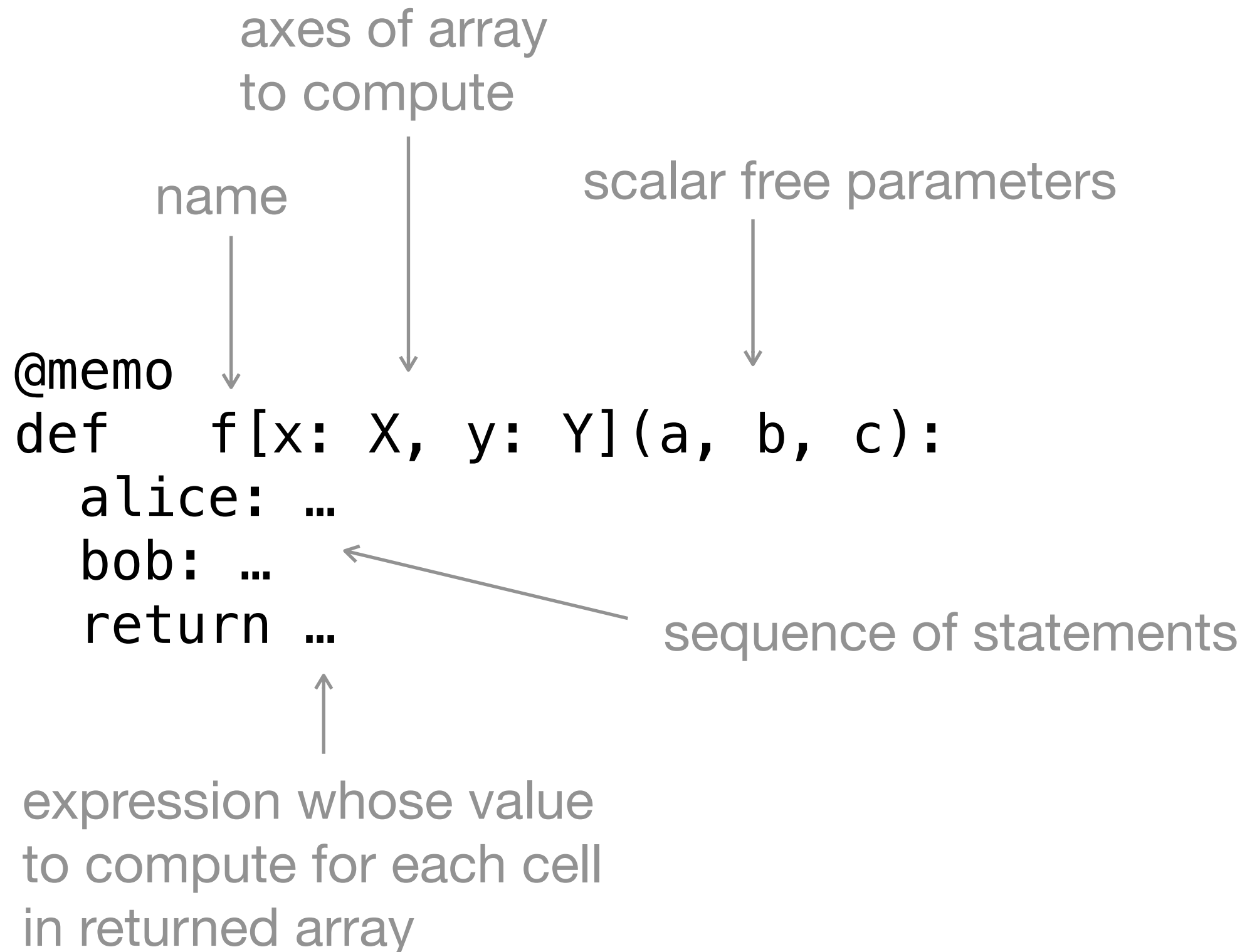


The memo handbook



Overall 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(β *utility(a)))

Name of choice

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

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

For argmax use to_maximize

thinks

Agent doing the thinking



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

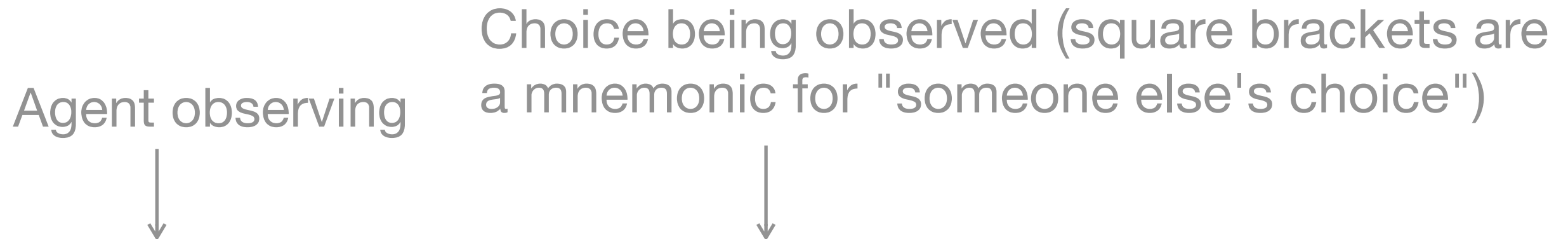


What that agent thinks
(notice the commas!)

observes

Agent observing

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



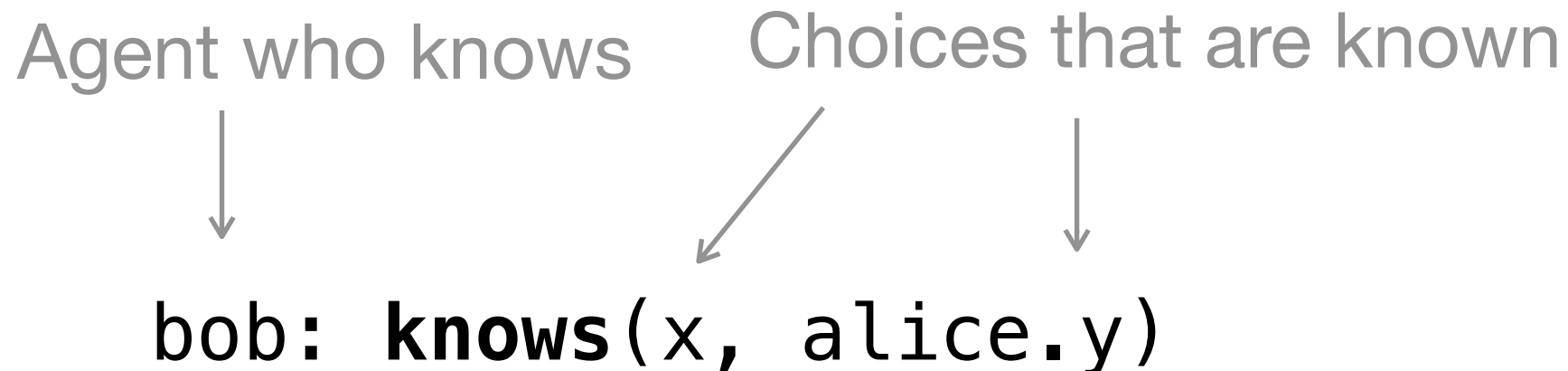
bob: observes [alice.x] is y

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.

knows



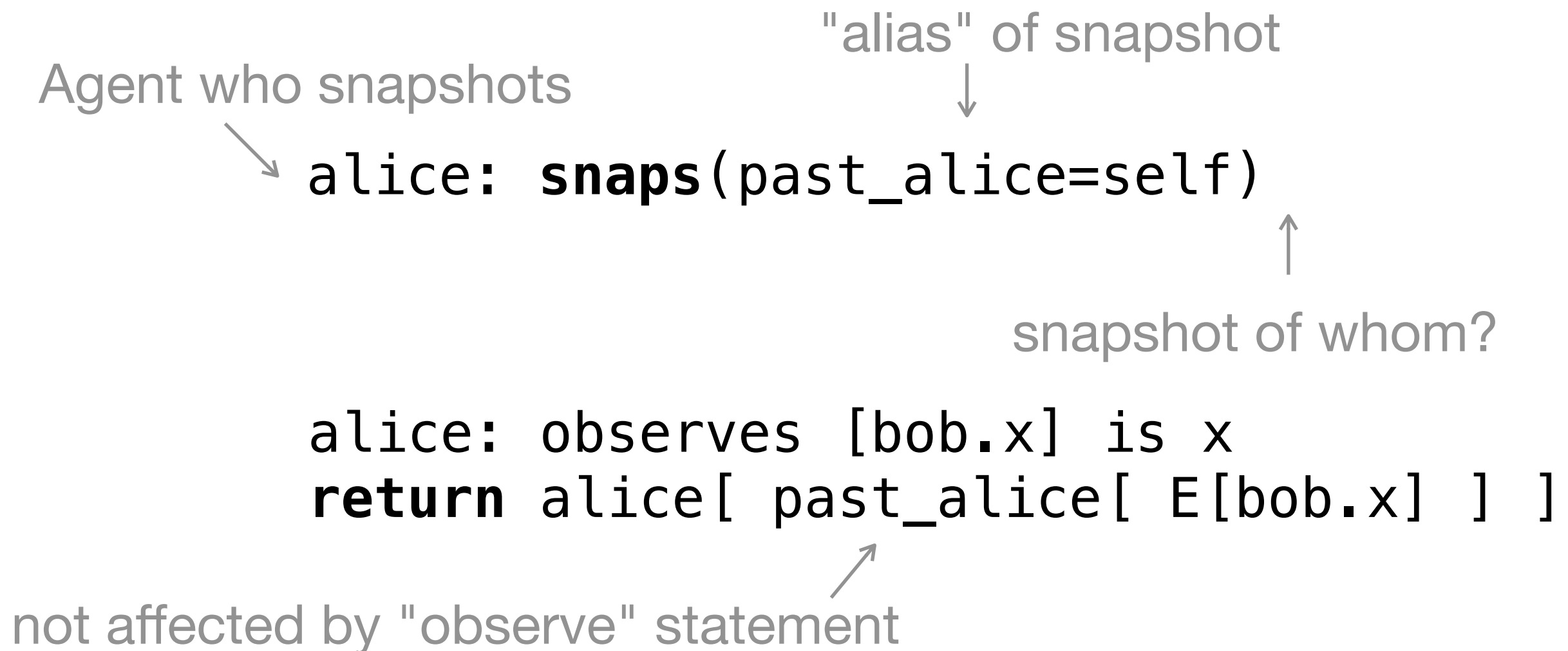
This utility is useful for the common case of "pushing" a variable into an agent's frame of mind. Roughly shorthand for this:

↓

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

snaps

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



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
@jax.jit (scalar-in-scalar-out)

↓
@jax.jit
def f(x):
 return np.cos(x)

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

choices

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



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

probabilistic operators

expectation



E[alice.x + bob.z]

probability



Pr[alice.y >= 0]

(mutual) entropy between choices



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

variance



Var[alice.y * 2]

queries

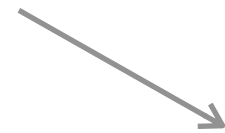
```
Var[alice[abs(x) * 2]]  
alice[bob.y == 7]
```



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): ...
```

```
@memo  
def g():  
  alice: chooses(x in X, wpp=f[x](3.14))
```



can reference one memo from another,
syntax evokes array indexing.
need to pass parameters, too!

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

```
N = 5
```

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



use braces for inline reference
to a global Python variable

Things to do with a memo

Running a memo

call it like a function with params
(returns an array w/ 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")`

Autodiff (useful for fitting)

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

returns tuple of value + gradient wrt params a & b



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