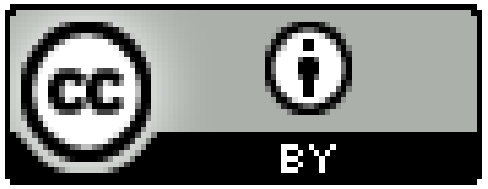
Python

Aliasing



An *alias* is a second name for a piece of data

An *alias* is a second name for a piece of data

Often easier (and more useful) than making a second copy

An *alias* is a second name for a piece of data

Often easier (and more useful) than making a second copy

If the data is immutable, aliases don't matter

An *alias* is a second name for a piece of data

Often easier (and more useful) than making a second copy

If the data is immutable, aliases don't matter

Because the data can't change

An *alias* is a second name for a piece of data

Often easier (and more useful) than making a second copy

If the data is immutable, aliases don't matter

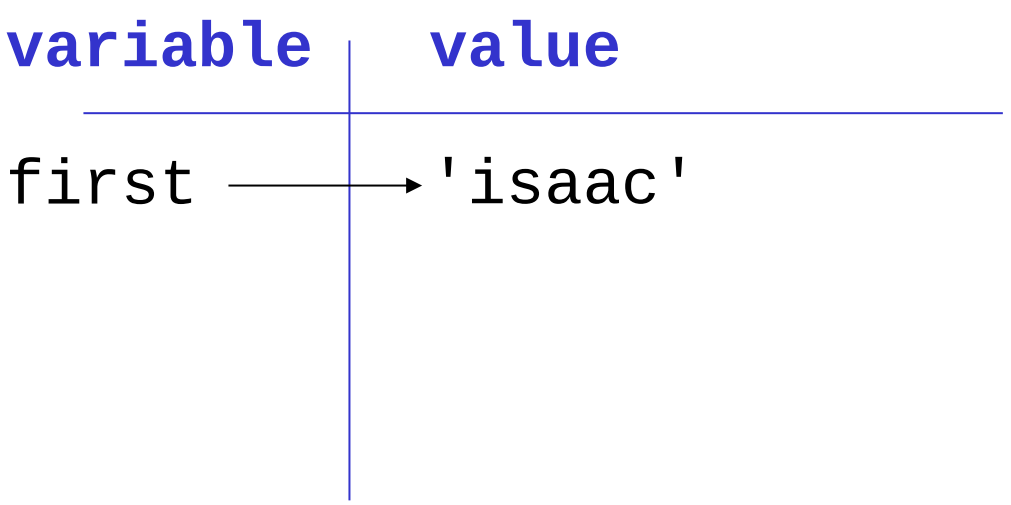
Because the data can't change

But if data *can* change, aliases can result in a lot of hard-to-find bugs

Aliasing happens whenever one variable's value is assigned to another variable

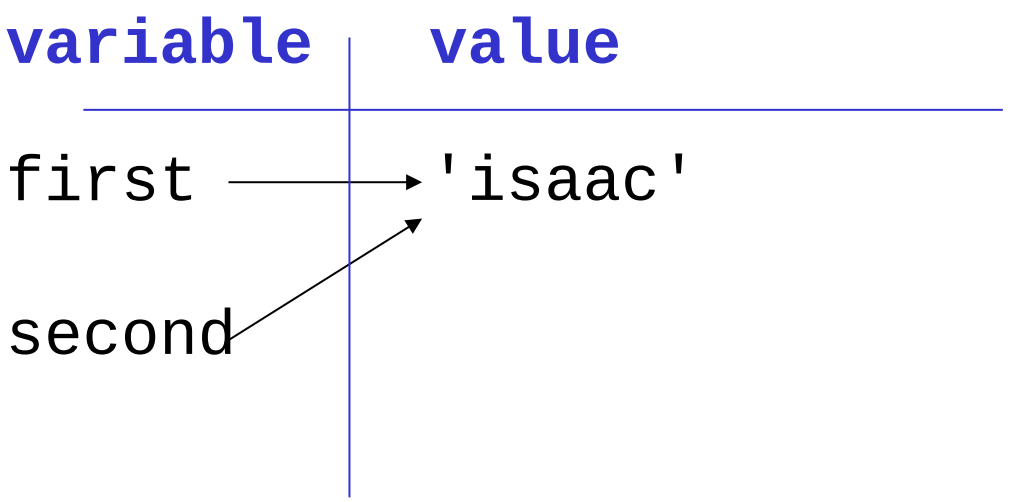
Aliasing happens whenever one variable's value is assigned to another variable

```
first = 'isaac'
```



Aliasing happens whenever one variable's value is assigned to another variable

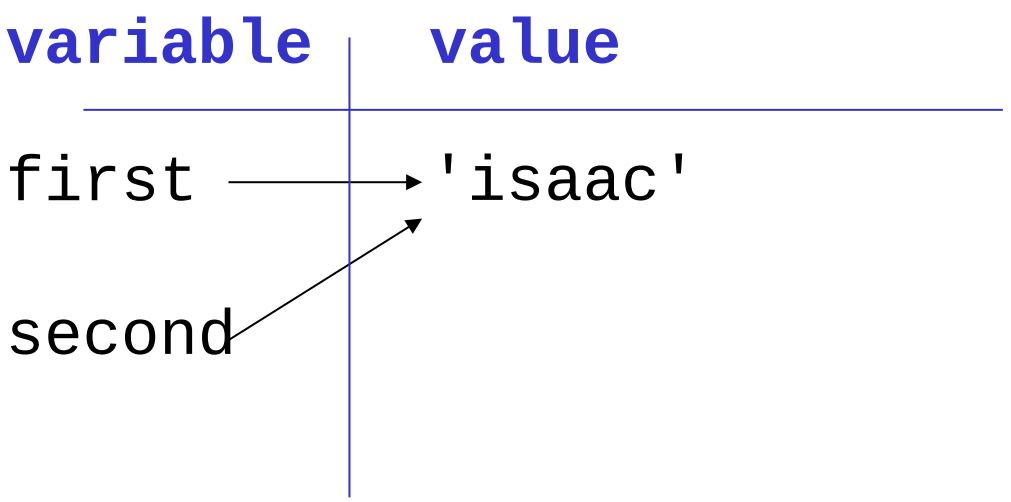
```
first = 'isaac'
second = first
```



Aliasing happens whenever one variable's value is assigned to another variable

```
first = 'isaac'
second = first
```

But as we've already seen…

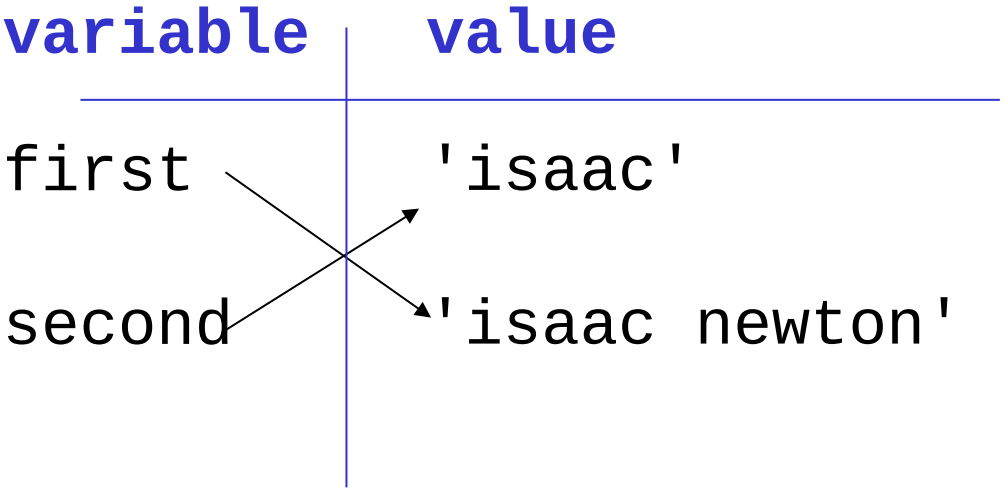


Aliasing happens whenever one variable's value is assigned to another variable

```
first = 'isaac'
second = first
```

But as we've already seen…

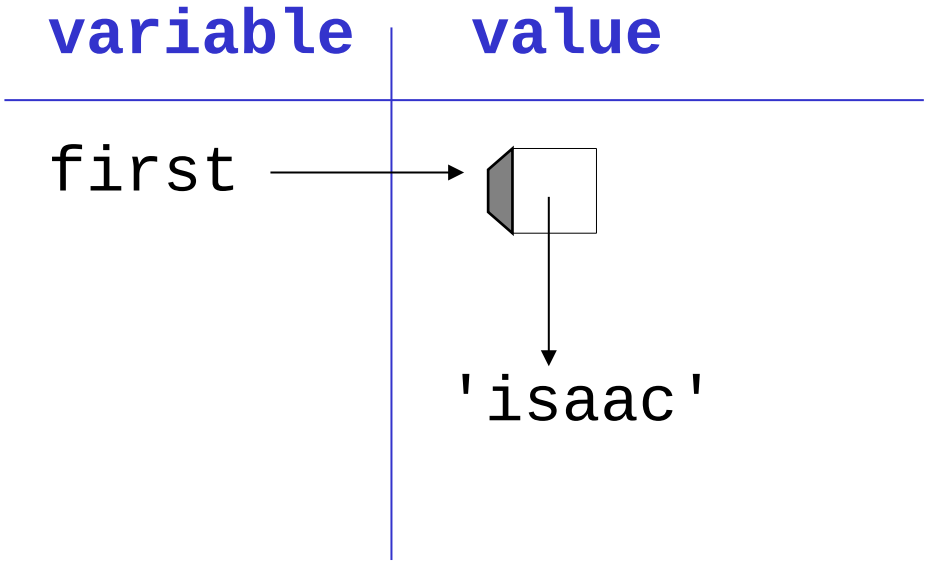
```
first = first + ' newton'
```



But lists are mutable

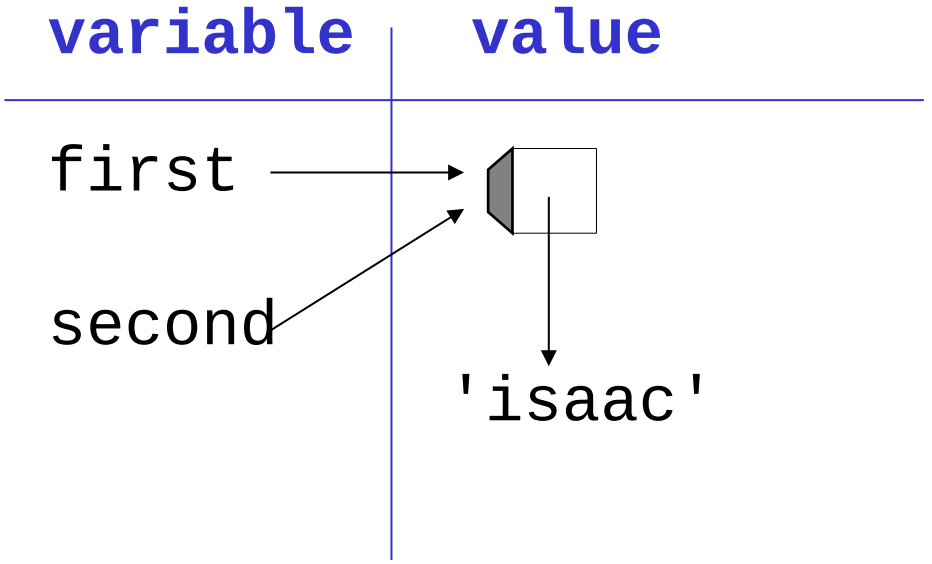
But lists are mutable

```
first = ['isaac']
```



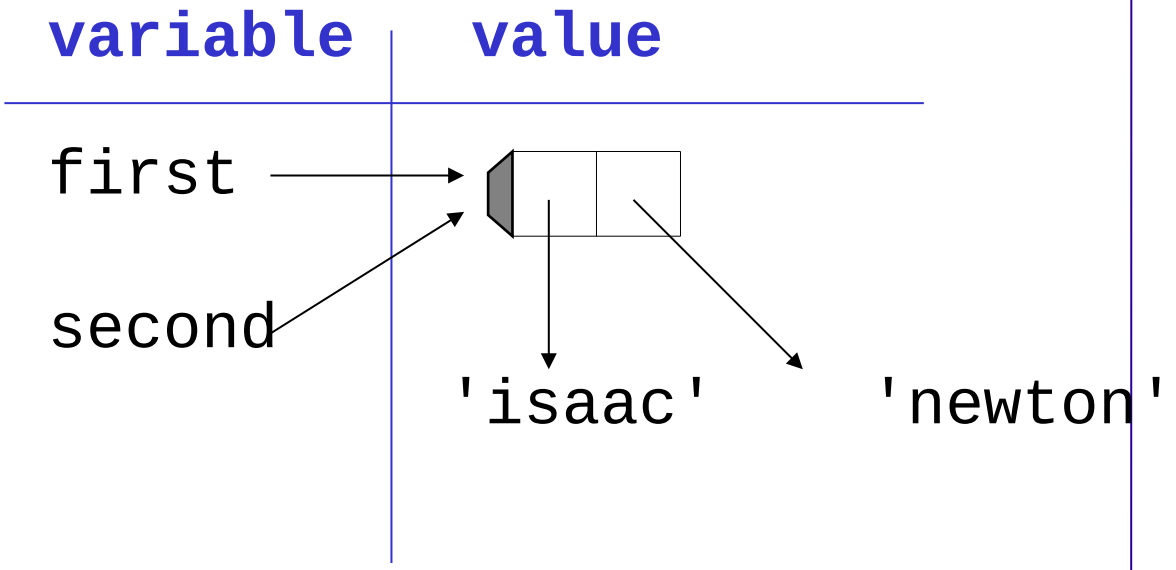
But lists are mutable

```
first = ['isaac']
second = first
```



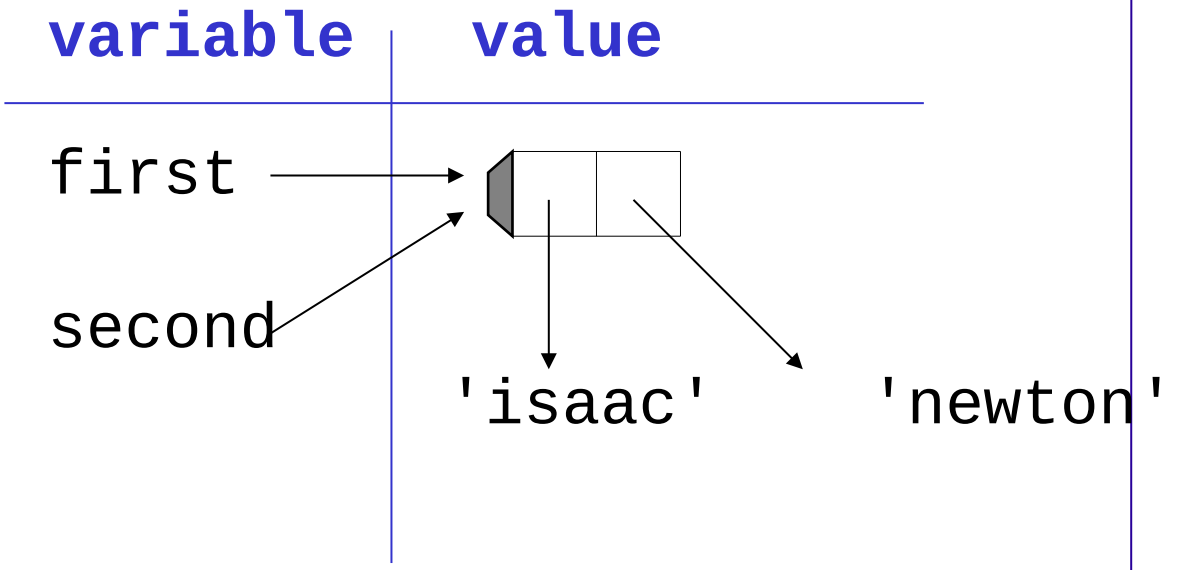
But lists are mutable

```
first = ['isaac']
second = first
first = first.append('newton')
print first
['isaac', 'newton']
```



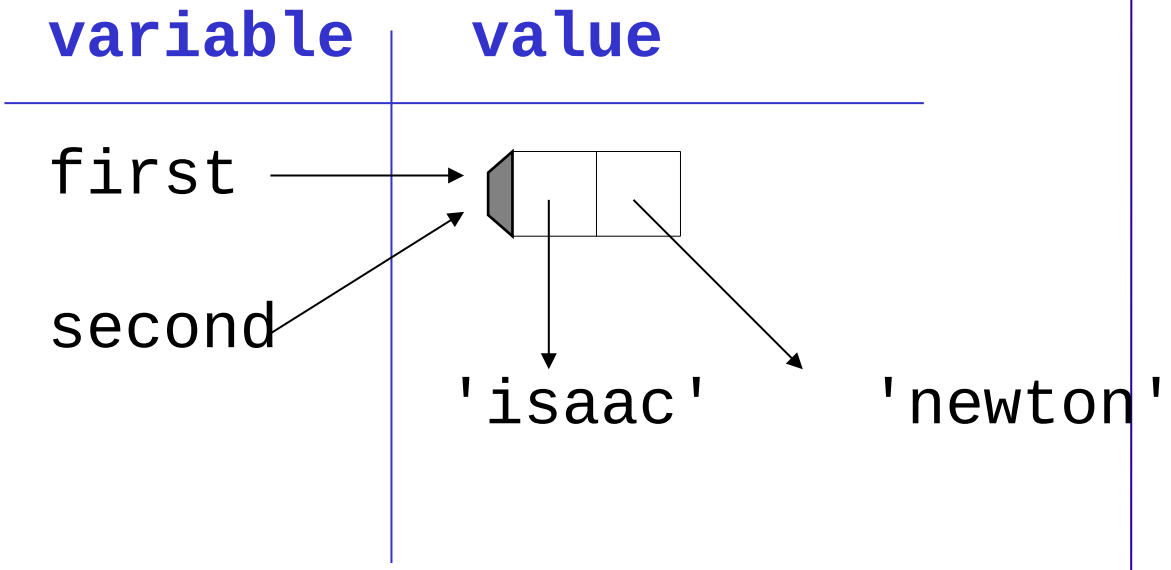
But lists are mutable

```
first = ['isaac']
second = first
first = first.append('newton')
print first
['isaac', 'newton']
print second
['isaac', 'newton']
```



But lists are mutable

```
first = ['isaac']
second = first
first = first.append('newton')
print first
['isaac', 'newton']
print second
['isaac', 'newton']
```

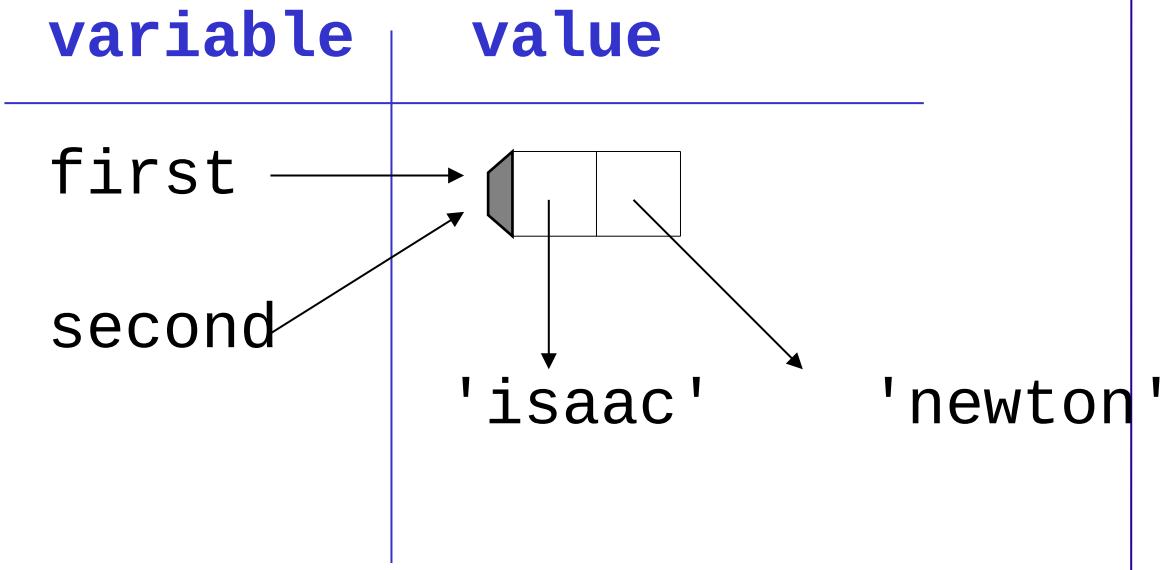


Didn't explicitly

modify second

But lists are mutable

```
first = ['isaac']
second = first
first = first.append('newton')
print first
['isaac', 'newton']
print second
['isaac', 'newton']
```



Didn't explicitly

modify `second`

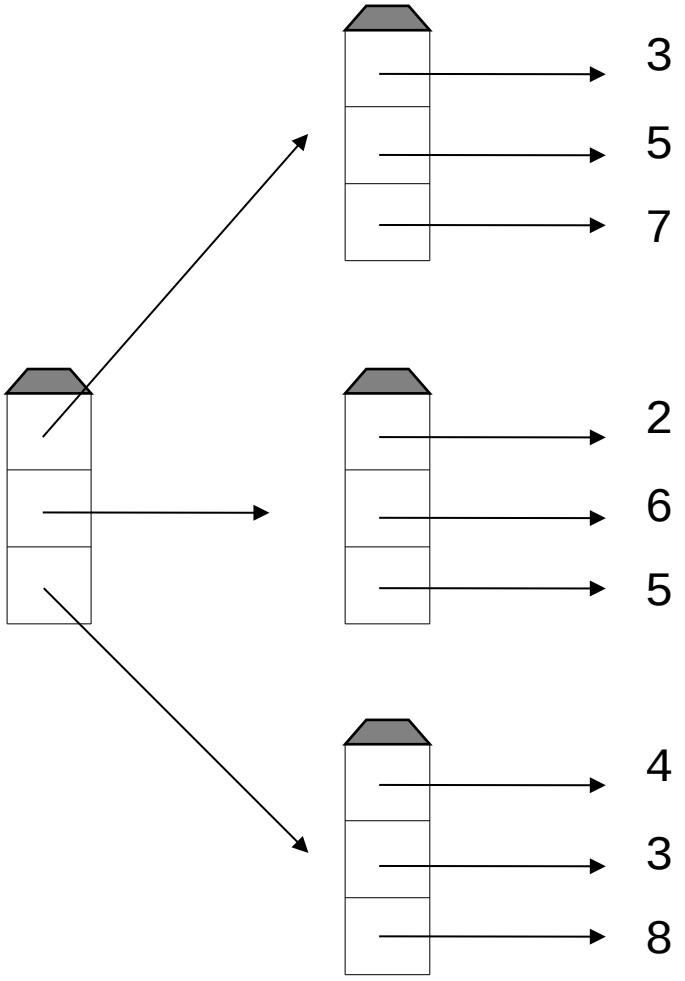
A *side effect*

Example: use lists of lists

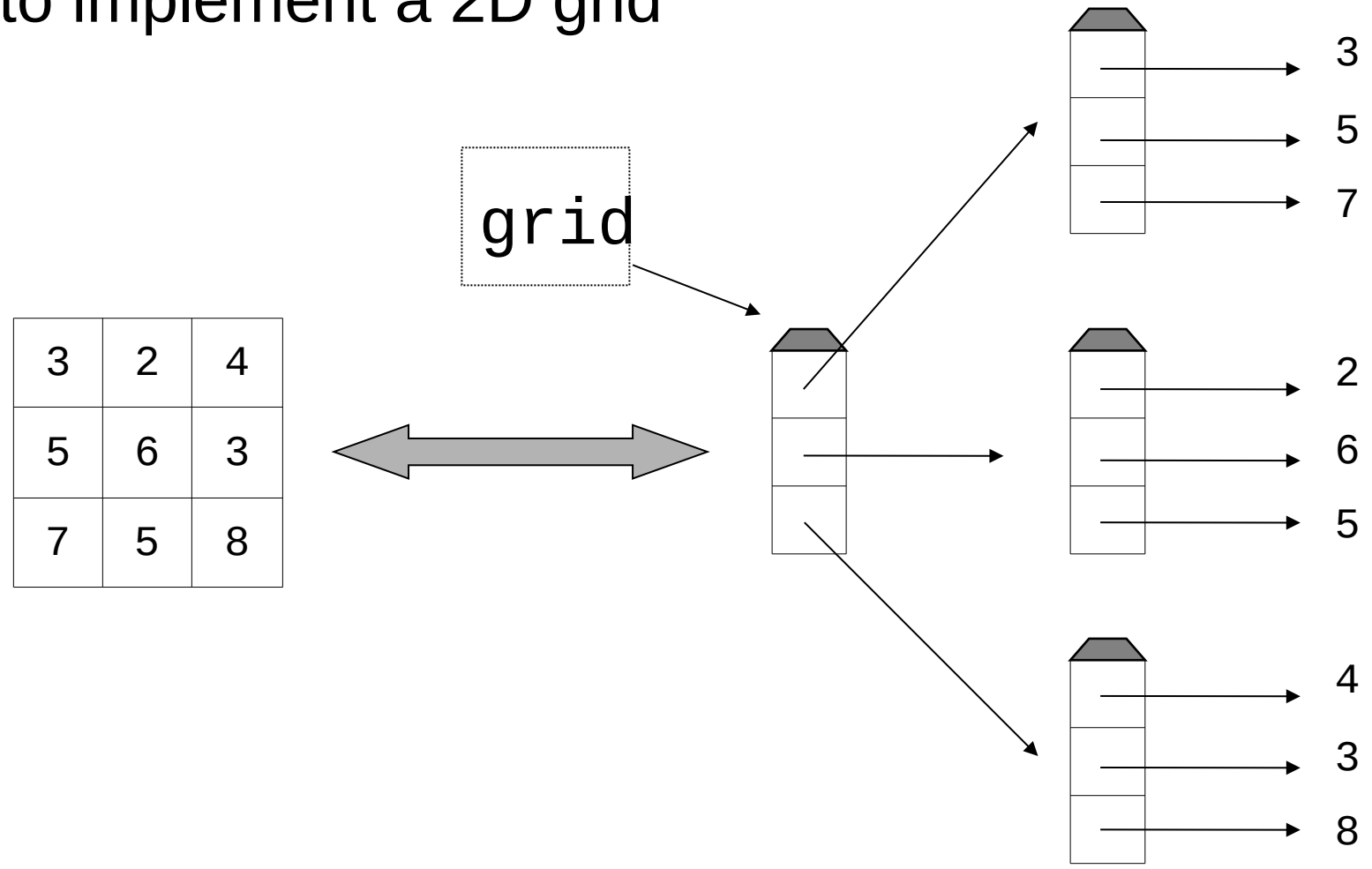
to implement a 2D grid

Example: use lists of lists to implement a 2D grid

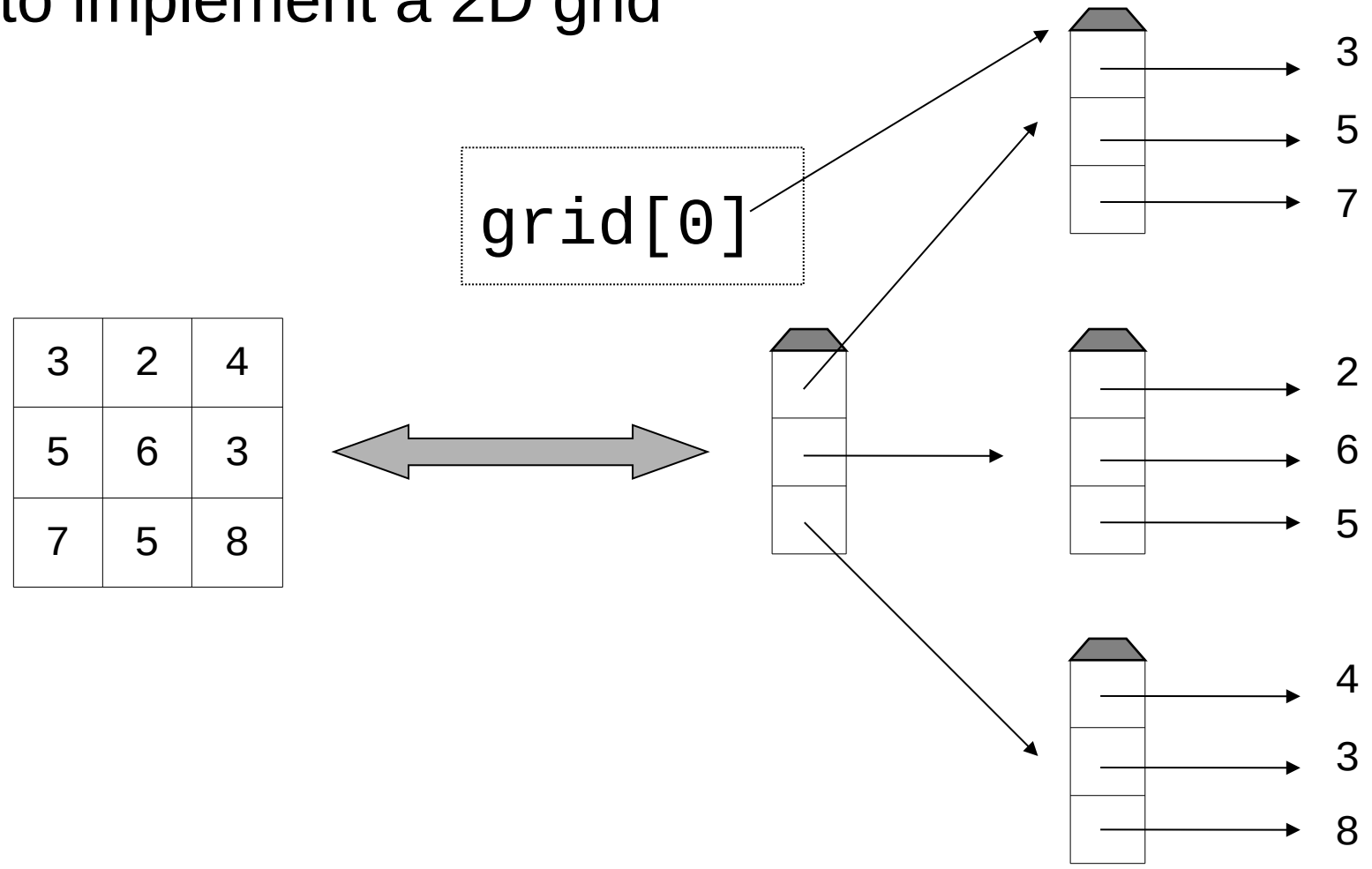
3	2	4
5	6	3
7	5	8



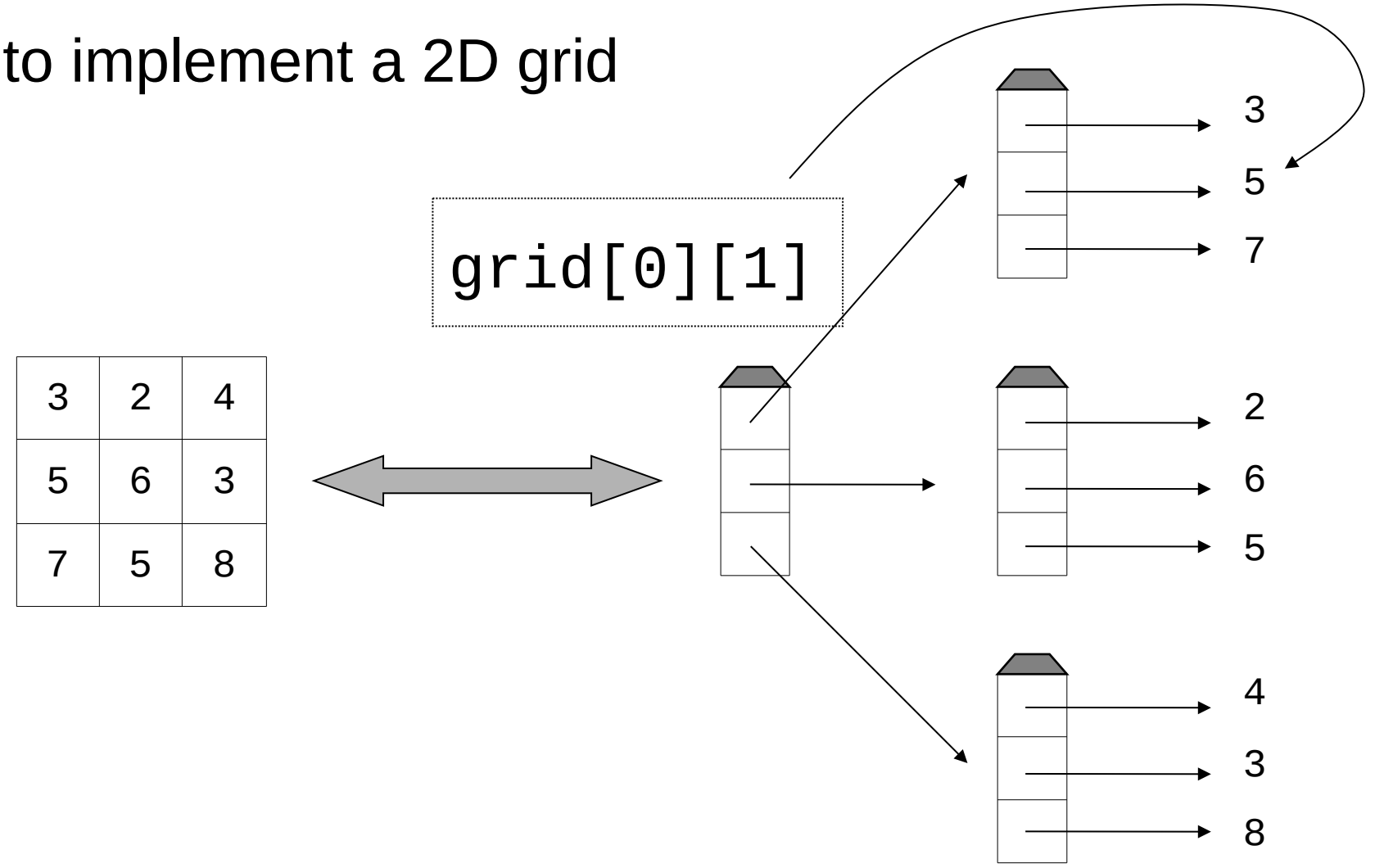
Example: use lists of lists to implement a 2D grid



Example: use lists of lists to implement a 2D grid



Example: use lists of lists to implement a 2D grid



```
# Correct code
grid = []
for x in range(N):
  temp = []
  for y in range(N):
    temp.append(1)
  grid.append(temp)
```

```
# Correct code
grid = []
for x in range(N):
  temp = []
  for y in range(N):
    temp.append(1)
  grid.append(temp)
```

Outer "spine" of structure

```
# Correct code
grid = []
for x in range(N):
  temp = []
  for y in range(N):
    temp.append(1)
  grid.append(temp)
```

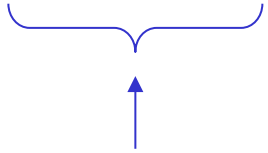
Add N sub-lists to outer list

```
# Correct code
grid = []
for x in range(N):
    temp = []
    for y in range(N):
        temp.append(1)
    grid.append(temp)
```

Create a sublist of N 1's

```
# Equivalent code
grid = []
for x in range(N):
  grid.append([])
  for y in range(N):
    grid[-1].append(1)
```

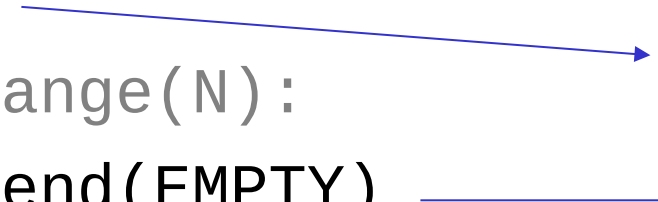
```
# Equivalent code
grid = []
for x in range(N):
  grid.append([])
  for y in range(N):
    grid[-1].append(1)
```



Last element of outer list is the sublist currently being filled in

```
# Incorrect code
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```

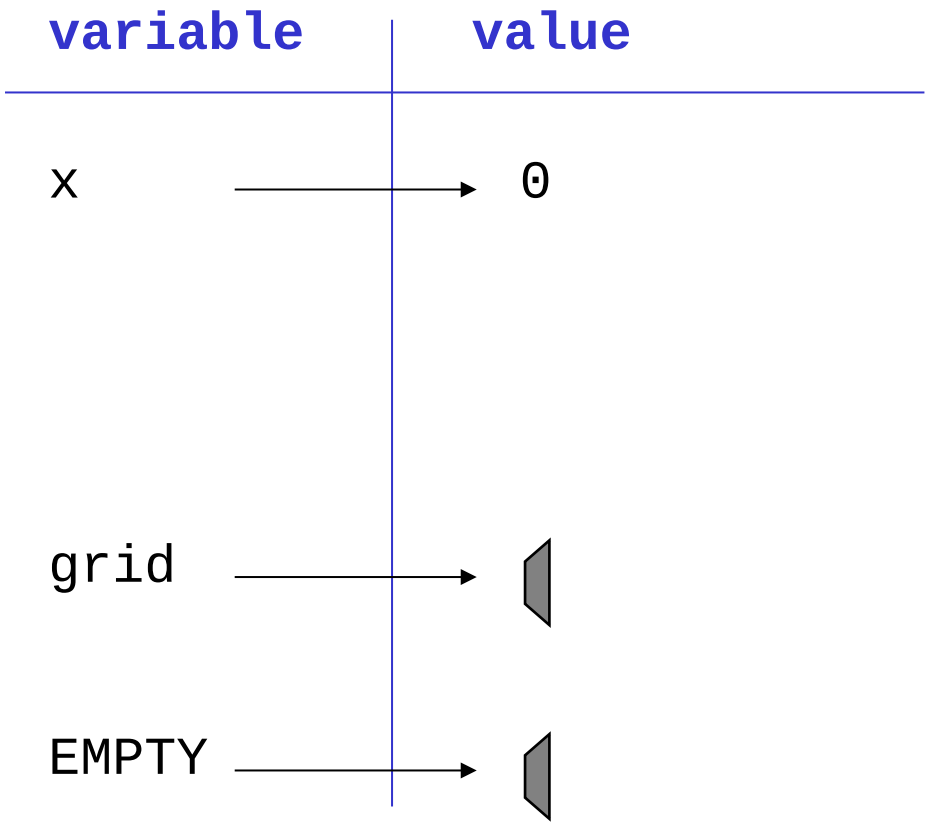
```
# Incorrect code
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```



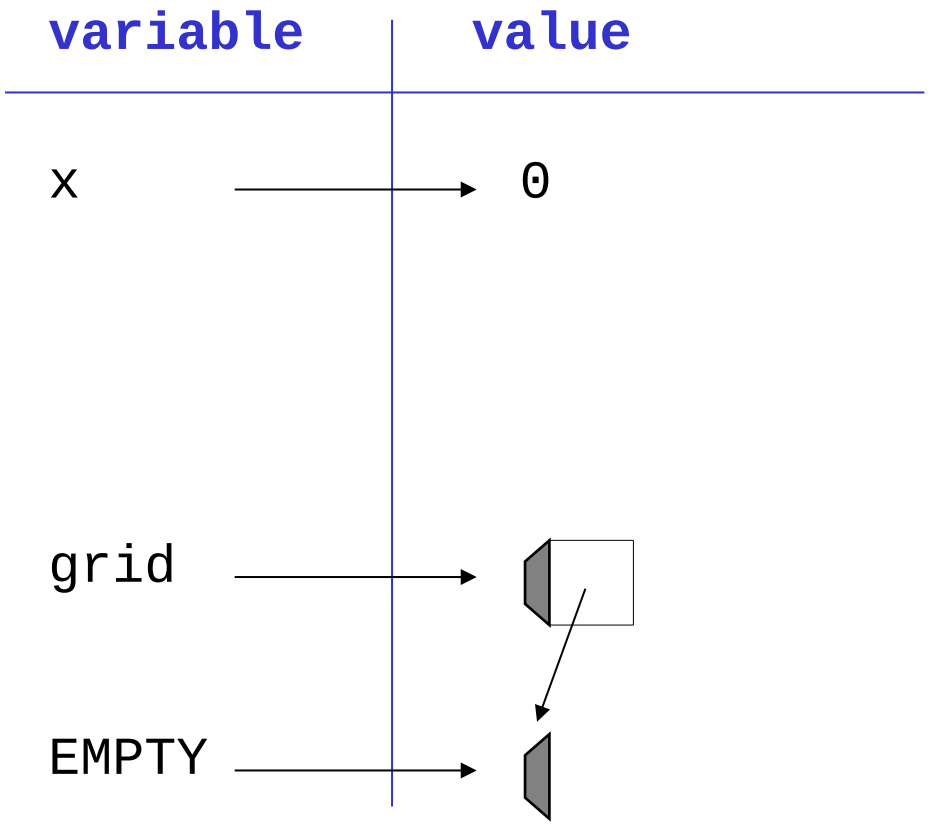
```
# Equivalent code
grid = []
for x in range(N):
  grid.append([])
  for y in range(N):
    grid[-1].append(1)
```

```
# Incorrect code
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```

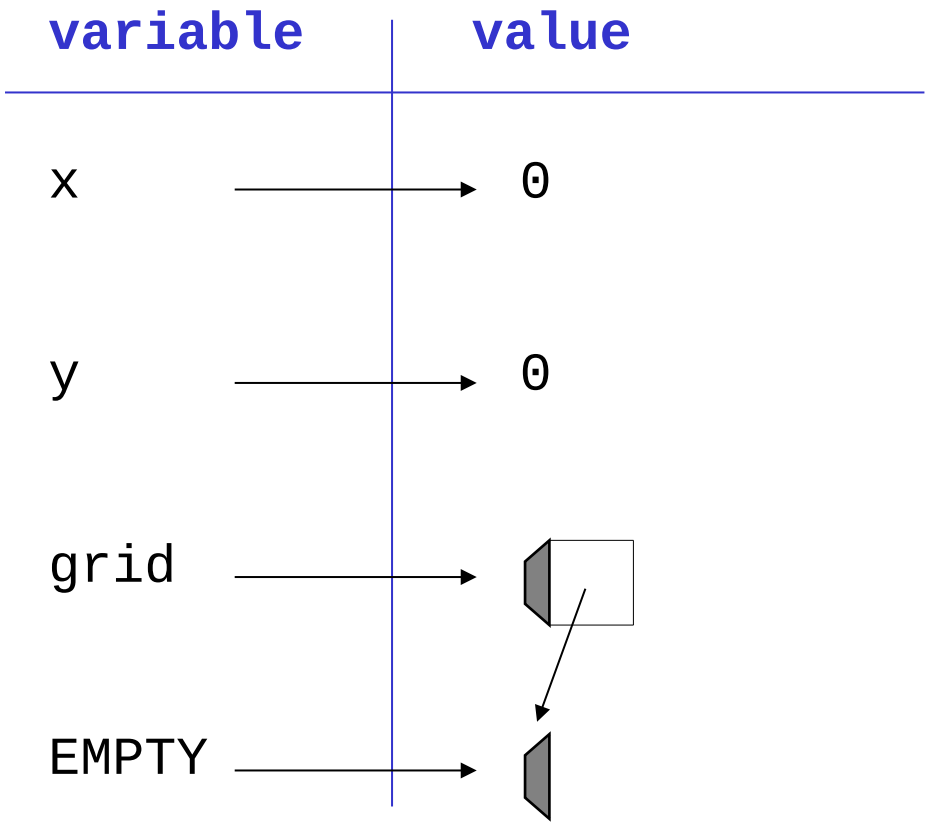
Aren't meaningful variable names supposed to be a good thing?



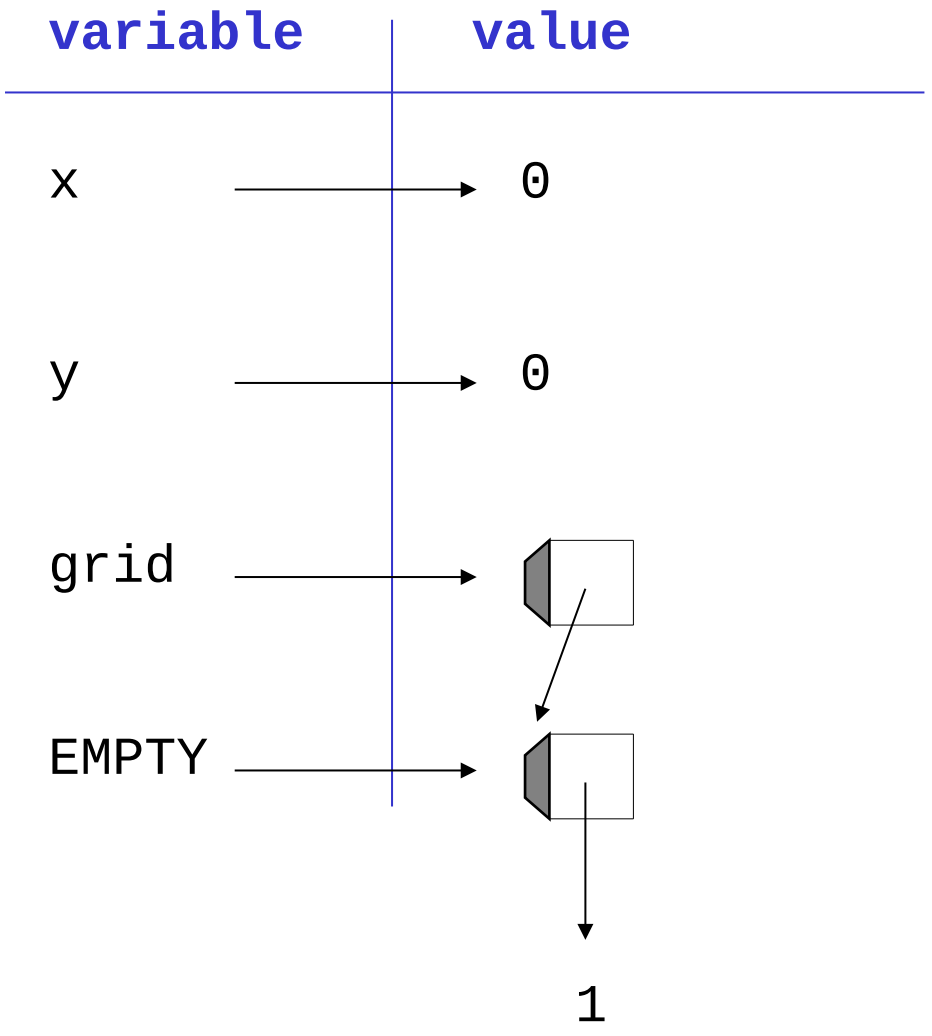

```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```



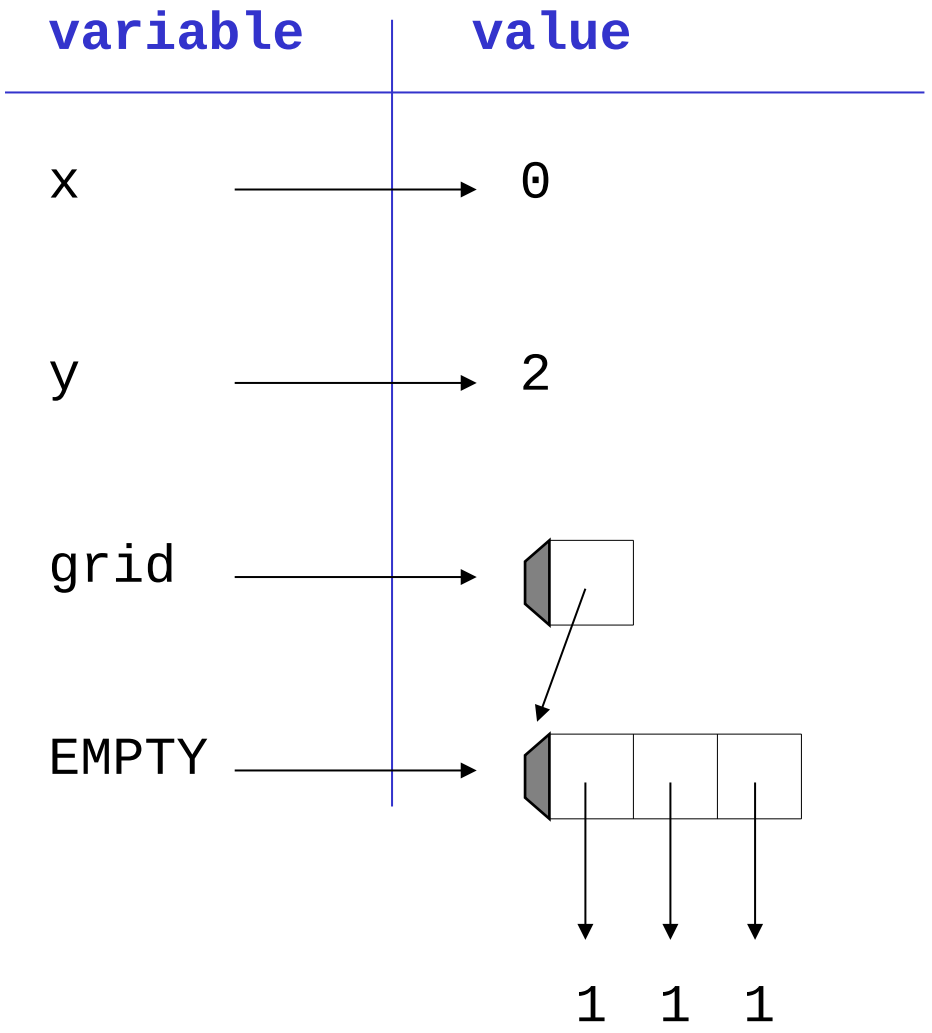

```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```



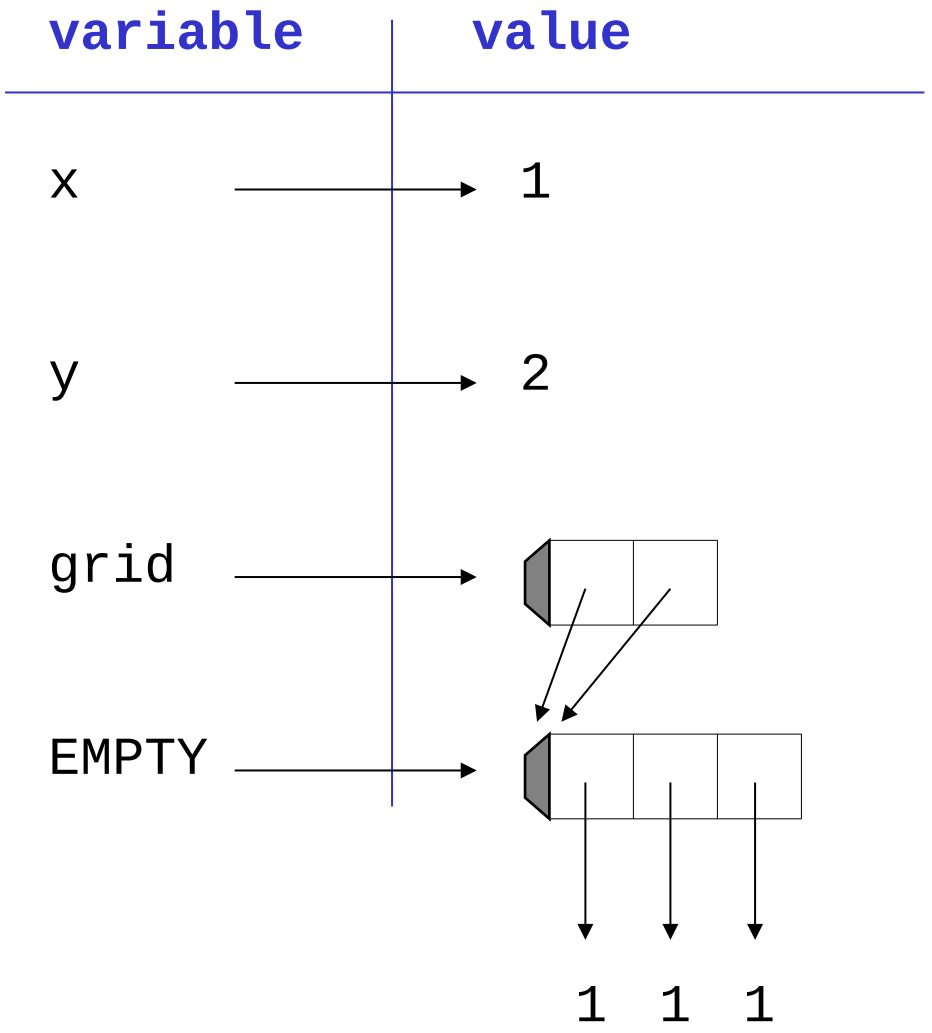

```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```



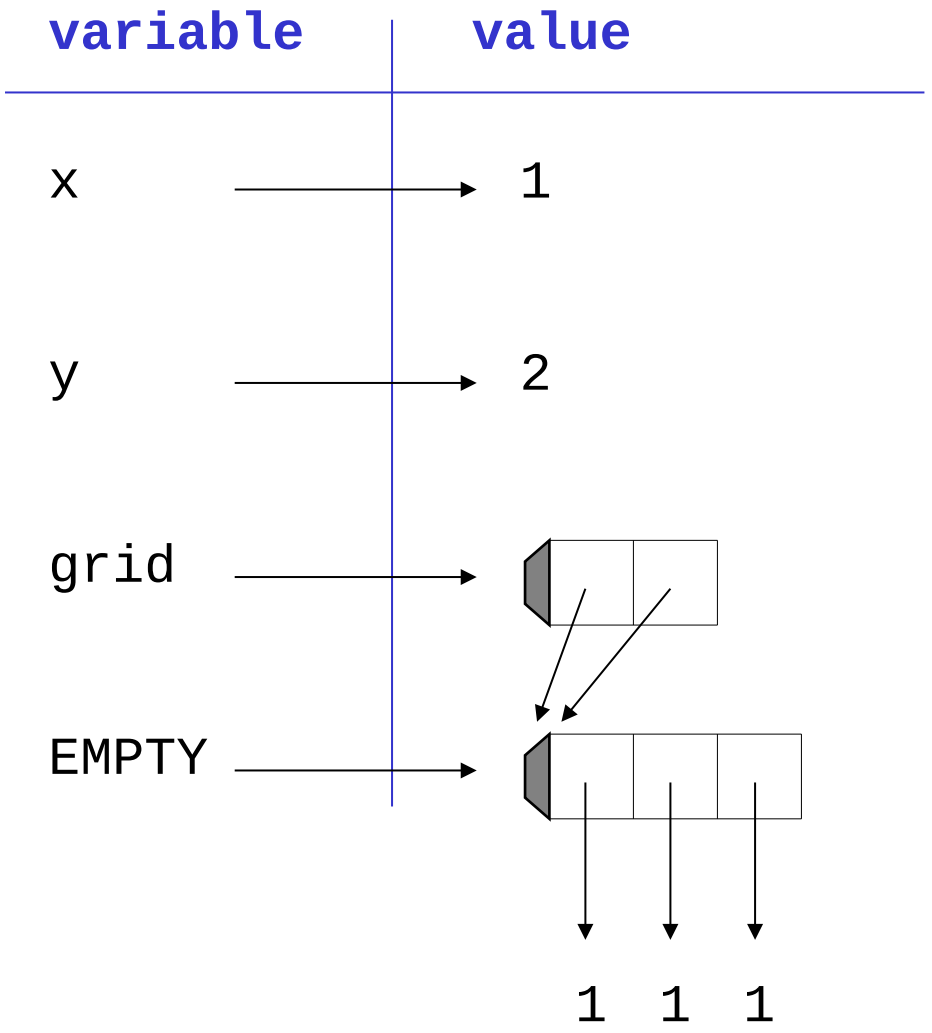

```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```




```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```




```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```




```
grid = []
EMPTY = []
for x in range(N):
  grid.append(EMPTY)
  for y in range(N):
    grid[-1].append(1)
```

You see the problem...

No Aliasing

```
first = []
second = []
```

No Aliasing	Aliasing
`first = []`	`first = []`
`second = []`	`second = first`

If aliasing can cause bugs, why allow it?

If aliasing can cause bugs, why allow it?

1. Some languages don't

If aliasing can cause bugs, why allow it?

1. Some languages don't

   Or at least appear not to

If aliasing can cause bugs, why allow it?

1. Some languages don't

   Or at least appear not to

2. Aliasing a million-element list is more efficient

   than copying it

If aliasing can cause bugs, why allow it?

1. Some languages don't

   Or at least appear not to

2. Aliasing a million-element list is more efficient

   than copying it

3. Sometimes you really do want to update a structure

   in place

If I know I don't want to create an alias

what can I do?

If I know I don't want to create an alias

what can I do?

Python's **copy.deepcopy** function will make a full copy of an object to want to replicate.

If I know I don't want to create an alias

what can I do?

Python's **copy.deepcopy** function will make a full copy of an object to want to replicate.

```
from copy import deepcopy
new_obj = deepcopy(my_obj)
```

created by

Greg Wilson

October 2010