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.

```python
first = 'isaac'
```
Aliasing happens whenever one variable's value is assigned to another variable

```python
first = 'isaac'
second = first
```
Aliasing happens whenever one variable's value is assigned to another variable.

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

But as we've already seen...
Aliasing happens whenever one variable's value is assigned to another variable

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

But as we've already seen...

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

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>'isaac'</td>
</tr>
<tr>
<td>second</td>
<td>'isaac newton'</td>
</tr>
</tbody>
</table>
But lists are mutable
But lists are mutable

```python
first = ['isaac']
```
But lists are mutable

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

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td></td>
</tr>
<tr>
<td>second</td>
<td>'isaac'</td>
</tr>
</tbody>
</table>
But lists are mutable

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

Python Aliasing

<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>first</td>
<td>'isaac'</td>
</tr>
<tr>
<td>second</td>
<td>'isaac', 'newton'</td>
</tr>
</tbody>
</table>
But lists are mutable

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

![Diagram showing variable and value relationships]

- `first` and `second` are aliased, so changes to `first` are reflected in `second`.
- `first.append('newton')` modifies the list stored in `first`, which is also accessible through `second`.

**Python Aliasing**
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

```python
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

7 5 8
5 6 3
3 2 4

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

```
<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
```
Example: use lists of lists to implement a 2D grid

```
grid[0] = [7, 5, 8]
grid[1] = [5, 6, 3]
grid[2] = [3, 2, 4]
```
Example: use lists of lists to implement a 2D grid

```
grid[0][1] = 3
grid[1][1] = 5
grid[2][1] = 7
```

Python

Aliasing
# Correct code

```python
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

```python
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

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

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

# Equivalent code

```python
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)
<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
</tbody>
</table>

```python
grid = []
EMPTY = []

for x in range(N):
    grid.append(EMPTY)
    for y in range(N):
        grid[-1].append(1)
```
<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
</tr>
<tr>
<td>grid</td>
<td></td>
</tr>
<tr>
<td>EMPTY</td>
<td>1</td>
</tr>
</tbody>
</table>

```python
grid = []
EMPTY = []
for x in range(N):
    grid.append(EMPTY)
for y in range(N):
    grid[-1].append(1)
```
<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>2</td>
</tr>
</tbody>
</table>

```python
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

\[
\text{first} = [] \\
\text{second} = []
\]
<table>
<thead>
<tr>
<th>No Aliasing</th>
<th>Aliasing</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first = []</code></td>
<td><code>first = []</code></td>
</tr>
<tr>
<td><code>second = []</code></td>
<td><code>second = first</code></td>
</tr>
</tbody>
</table>
grid = []
for x in range(N):
    grid.append([])
    for y in range(N):
        grid[-1].append(1)
grid = []
for x in range(N):
    grid.append([])
    for y in range(N):
        grid[-1].append(1)
Python

grid = []
for x in range(N):
    grid.append([])
    for y in range(N):
        grid[-1].append(1)
grid = []
for x in range(N):
    grid.append([])
    for y in range(N):
        grid[-1].append(1)
grid = []
for x in range(N):
    grid.append([])
    for y in range(N):
        grid[-1].append(1)
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 really do want to update a structure in place