Python Efficiency

Split up multiple rules

```
if n == 0:
        return None
if W == 0:
        return None
if n != len(values):
        return None
```

is ~2x as fast as making all of the stipulations on one line

```
if n == 0 or W == 0 or n != len(values):
        return None
```

Results

```
Function: SIK
----------------------------------------
Time elapsed:             0.0001001 s
Memory usage:             0.0 Mbs
Peak Memory usage:        0.0 Mbs
++++++++++++++++++++++++++++++++++++++++

Function: SIK2
----------------------------------------
Time elapsed:             3.934e-05 s
Memory usage:             0.0 Mbs
Peak Memory usage:        0.0 Mbs
++++++++++++++++++++++++++++++++++++++++
```

Making a pandas dataframe is expensive

```
%%timeit
table = pd.DataFrame(index = [x for x in range(n)],
            columns = [x for x in range(W+1)])
```

```
908 µs ± 12.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
%%timeit
table = [[0 for x in range(W+1)] for i in range(n)]
```

```
4.71 µs ± 168 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

range and enumerate are pretty even

```
%%timeit
[lst[x] for x in range(100)]
```

```
7.81 µs ± 496 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

```
%%timeit
[x for n,x in enumerate(lst,0)]
```

```
6.92 µs ± 215 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%%timeit
[(n,x,lst2[n]) for n,x in enumerate(lst,0)]
```

```
14.8 µs ± 388 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

```
%%timeit
[(x, lst[x], lst2[x]) for x in range(100)]
```

```
14.7 µs ± 278 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Use 'in' instead of regex when possible

```
%%timeit
for x in patts:
    if x not in s:
        patts.remove(x)
```

```
161 ns ± 19 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
%%timeit
for x in patts:
    if not re.search(f"", s):
        patts.remove(x)
```

```
2.16 µs ± 181 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

List comps vs for-loops

List comps are significantly faster for making lists

```
%%timeit
lst = []
for n1 in range(1000):
    for n2 in range(1000):
        lst.append(n2)
```

```
113 ms ± 3.21 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
lst = []
for n1 in range(1000):
    lst.extend([n2 for n2 in range(1000)])
```

```
76.2 ms ± 2.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
lst = [[n2 for n2 in range(1000)] for n1 in range(1000)]
```

```
71.8 ms ± 1.27 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

For-loops are faster for only iteration

- This is because the list comp will still store the data as a list

```
ts = time()
for n in range(1000000):
    pass
print(f"Iterative: {time() - ts:0.2e}")

##########################################

ts = time()
[n for n in range(1000000)]
print(f"ListComp: {time() - ts:0.2e}")

##########################################
"""
Iterative: 5.96e-02
ListComp: 8.48e-02
"""
```

List comp is faster for operations within the loop

```python
patts = ["cat", "dog", "xxx", "test"]
s = "xxxxxcatdangtest"

ts = time()
for patt in patts:
    if patt not in s:
        patts.remove(patt)
print(f"Iterative: {time() - ts:0.2e}")

########################################
patts = ["cat", "dog", "xxx", "test"]
s = "xxxxxcatdangtest"

ts = time()
[patts.remove(patt) for patt in patts \
 if patt not in s]
print(f"ListComp: {time() - ts:0.2e}")

########################################
"""
Iterative: 5.01e-06
ListComp: 3.34e-06
"""
```