Workshop Instructions

Go to **http:/tinyurl.com/ghc24vectorization**

Option 1: (if you're familiar with GitHub and already have Python on your laptop)

- Fork & clone the repo
- Follow set-up instructions to run locally with your own Python

Option 2: (Requires a Google account)

- Go to https://colab.research.google.com
- Start a new notebook
- Follow set-up instruction, copy-pasting the code from GitHub

ANITA B.ORG 2024

GRACE HOPPER CELEBRATION

me +we

ANITA B.ORG 2024 GRACE HOPPER CELEBRATION

How for loops are “highkey” killing your code performance

Justine Wezenaar, Engineering Team Lead
Tamara Rosenberg, Senior Software Engineer
Bloomberg

me +we

ANITA B.ORG 2024
GRACE HOPPER CELEBRATION

Intros

Justine Wezenaar
Team Lead
Environmental, Social & Governance (ESG) Quant Engineering


B.Sc., Math & Physics, McGill University
M.Sc., Mathematics & Statistics, McGill University
Keywords: ESG, data science, bond pricing, pharma
Hobbies: Running, languages, baking
LinkedIn: in/justine-wezenaar-62357857

Tamara Rosenberg
Senior Software Engineer
Environmental, Social & Governance (ESG) Data Integration

B.S. Computer Science, The University of Texas
Keywords: data engineering, microservices, performance optimization
Hobbies: Trail running, board games, improv
LinkedIn: /in/tamarawarton/

What's so wrong with for loops!?

What's wrong with for loops?

"Row-based" operations

Image by brgfx on Freepik

Code patterns to look for:

```
for i,row in df.iterrows():
    do_somthing(row['x'])

df["y"]=df["x"].apply(lambda x:my_func(x))

df.apply(lambda row:
    my_function(row["x1"],row["x2"], axis=1)
```

df: pd.DataFrame

ID	Year	Col1	Col2
ID1	2023	Y	0.342
ID2	2023	Y	0.34
ID1	2024	N	0.235
ID2	2024	Y	0.192

What to do instead: Vectorization

"Column-based" operations

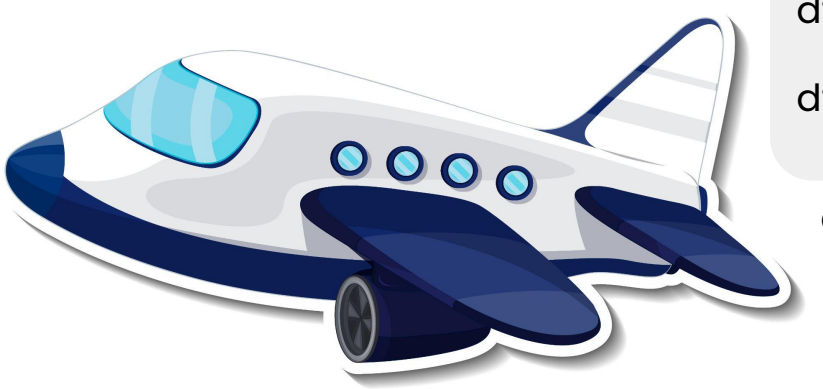


Image by brgfx on Freepik

New code patterns:

```
df["y"] = my_vec_func1(df["x1"], df["x2"])

df = my_vec_func2(df)
```

df: pd.DataFrame

ID	Year	Col1	Col2
ID1	2023	Y	0.342
ID2	2023	Y	0.34
ID1	2024	N	0.235
ID2	2024	Y	0.192

pandas

NumPy

Building Blocks: Foundations

Building Blocks: Foundational Strategies

Get data into np.array, pd.Series and/or pd.DataFrame

Use library functions with np.array & pd.Series args

Replace if/else using set theory np.isin() np.where() np.any()

np.array,
pd.Series,
pd.DataFrame

Get Everything in pd.DataFrame, pd.Series, or np.array

- **Vectorization** = element-wise operations on an entire array
- Already-optimized functions in **Numpy** and **Pandas** make this possible by leveraging LPACK, BLAS, and SIMD under the hood! 🏎️
- We can convert our data to:
  - np.array([1, 2, 3])
  - np.full(10, 1.0, dtype=float)
  - pd.Series(np.array([1, 2, 3]))
  - pd.DataFrame.from_dict(
    {“colName”: [“rowVal1”, “rowVal2”, “rowVal3”]},
    orient=”columns”
)

pandas

NumPy

Use library functions with np.array & pd.Series args

Building Blocks: Basic Operations (+ - * /) & math.log()

Replace this:

```
def slow_plus(array,inc=4):
  out=[]
  for x_i in array:
    out.append(x_i+inc)
  return np.array(out)
```

With this:

```
def vec_plus(array,inc=4):
  return array+inc
```

Performance on 10k records

```
n_reps=20; data_size= 10000 records
slow_plus_array: 0.0221s +/- 0.002s
slow_plus_df:    0.0023s +/- 0.0003s
vec_plus_df:     0.0004s +/- 0.0s
speedup:         61.2X  +/- 5.6X
```

Replace this:

```
def slow_log(array):
  out=[]
  for x_i in array:
    out.append(math.log(x_i,2))
  return np.array(out)
```

With this:

```
def vec_log(array):
  return np.log2(array)
```

Performance on 10k records

```
n_reps=20; data_size= 10000 records
slow_log_array:  0.0126s +/- 0.0027s
slow_log_df:  0.0041s +/- 0.0003s
vec_log_df:  0.0005s +/- 0.0001s
speedup:  27.8X  +/- 6.6X
```

Building Blocks: Library example (scipy.interpolate)

Use library functions with np.array & pd.Series args

```
def setup_library_fn(n: int):
    np.random.seed(12345)
    x = np.random.random(n)
    scale_fn = sp.interpolate.interp1d([0,1], [0,10], kind='linear')
```

Replace this:

```
def library_fn(x:np.array):
  y=[]
  for x_i in x:
    y.append(scale_fn(x_i))
  return y
```

With this:

```
def library_fn_vec(x:np.array):
  y = scale_fn(x)
  return y
```

Performance on 1M rows

```
tests/koans/vectorization.py::test_library_fn
for n=1000000:
time_before: 12.3905, time_vectorized: 0.0073.
improvement: 1697.3288x
```

Replace if/else using set theory

Building Blocks: If/Else

Replace this:

```
def slow_ifelse(x):
  if x['discount_code']=='SUBWAY':
    return x['subtotal']*0.85
  elif x['discount_code']=='FRIDAY':
    return x['subtotal']*0.9
  else:
    return x['subtotal']*0.95

def slow_ifelse_wrapper(X):
  return X.apply(
      lambda row: slow_ifelse(row), axis=1)
```

With this:

```
def vec_ifelse(X):
  idx_subway = X['discount_code']=='SUBWAY'
  idx_friday = X['discount_code']=='FRIDAY'
  idx_none = ~(np.any([idx_subway,idx_friday],axis=0))

  final_price = X['subtotal'].copy()
  final_price[idx_subway] = final_price[idx_subway]*0.85
  final_price[idx_friday] = final_price[idx_friday]*0.9
  final_price[idx_none] = final_price[idx_none]*0.95
  return final_price
```

Performance on 10k records

```
n_reps=20; data_size= 10000 records
slow_ifelse: 0.0973s +/- 0.0062s
vec_ifelse:  0.0068s +/- 0.001s
speedup:     14.6X  +/- 1.8X
```

Replace if/else using set theory

Building Blocks: Set membership (np.where, np.isin)

Replace this:

```
def set_membership(x, vals:set):
  y=[]
  for x_i in x:
    if x_i in vals:
      y.append(1)
    else:
      y.append(0)
  return y
```

With this:

```
def set_membership_vec(x,vals:np.array):
  # NOTE: list or np.array much faster than set
  y = np.where(np.isin(x,vals),1,0)
  return y
```

$$\text{Set Members} = \{2, 4\}$$

$$x = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}, idx = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Performance on 10M rows

```
tests/koans/vectorization.py::test_set_membership
for n=10000000:
time_before: 1.1364, time_vectorized: 0.0619.
improvement: 18.3586x
```

Using Multiple Blocks

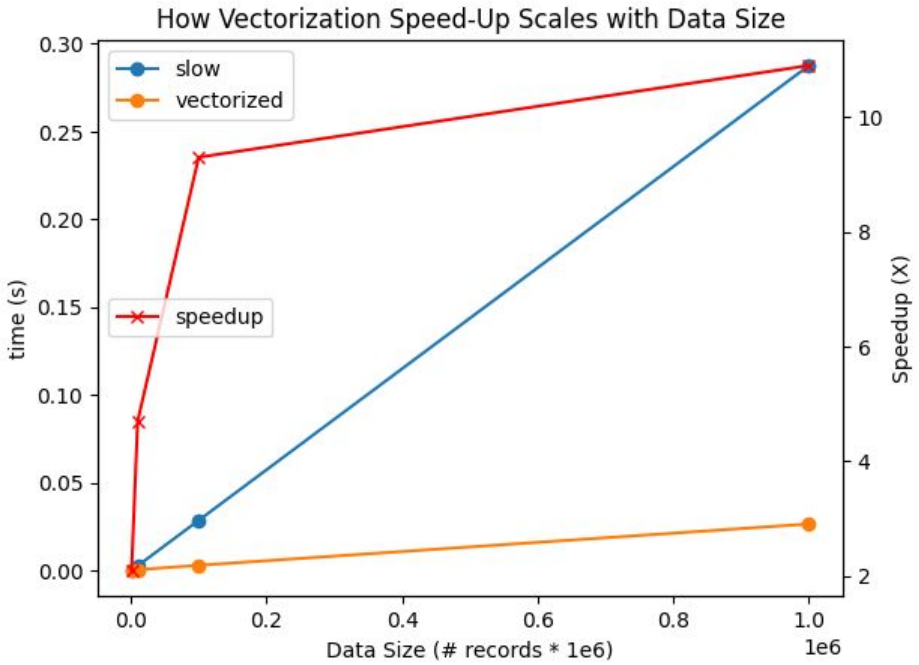
```python
def slow_esg_example(x:pd.Series , params: dict=country_params):
  if x['COUNTRY'] in params['countries_eligible_to_score']:
    if not pd.isna(x['FIELD1']):
      return sp.stats.norm.cdf(x['FIELD1'], loc=params['mu'], scale=params['scale'])
    else:
      return sp.stats.norm.cdf(x['FIELD2'], loc=params['mu'], scale=params['scale'])
  else:
    return np.nan

def slow_esg_example_on_DF(X:pd.DataFrame, params: dict=country_params):
  scores = []
  for i,row in X.iterrows():
    scores.append(slow_esg_example(row,params))
  return scores
```

1. Set theory to identify indices
2. Leverage vectorized library function (scipy.stats.norm.cdf)
3. Apply logic to select indices

```python
def vec_esg_example(X: pd.DataFrame, params: dict=country_params):
  idx_eligible = X['COUNTRY'].isin(params['countries_eligible_to_score'])
  idx_field1 = X['FIELD1'].notna()
  score = np.full(X.shape[0], np.nan, dtype=float)
  score[idx_eligible & idx_field1] = scipy.stats.norm.cdf(X.loc[idx_eligible & idx_field1]['FIELD1'], loc=params['mu'], scale=params['scale'])
  score[idx_eligible & ~idx_field1] = scipy.stats.norm.cdf(X.loc[idx_eligible & ~idx_field1]['FIELD2'], loc=params['mu'], scale=params['scale'])
  return score
```

1
2
3
3

Performance Scaling with Data Size



The bigger the data**, the more speed-up

***subject to memory constraints*

Workshop Instructions

Go to **http:/tinyurl.com/ghc24vectorization**

Option 1: (if you're familiar with GitHub and already have Python on your laptop)

- Fork & clone the repo
- Follow set-up instructions to run locally with your own Python

Option 2: (Requires a Google account)

- Go to https://colab.research.google.com
- Start a new notebook
- Follow set-up instruction, copy-pasting the code from GitHub

Option 1: From your local Python3 shell:

0. Clone this repo (optionally: fork it to your github first)

```
git clone <>
```

1. Enter the directory and make your virtual environment (any Python3 version should be fine. Here I use 3.9)

```
cd ghc2024-vectorization-workshop
python3.9 -m venv venv
```

2. Activate your new virtual environment

```
source venv/bin/activate
```

3. Install required packages from the requirements.txt

```
python -m pip install -r requirements.txt
```

Option 2: In a Google Colab Notebook

1. Open a new Google Colab Notebook (https://colab.research.google.com). You will need to be signed into your own Google account

2. copy-paste the code from `utils.py` into the first cell and run

Writing and testing the functions

Option 1: From your local Python3 shell:

1. Open the `q#.py` in your text editor, according to the question (q1 through q7). For the given question, put your optimized code in the `vec_*` function where it says `pass # insert your code here`.
2. To test that your function works and compare the speed (for relevent questions) by running the `test_run.py` script:

```
python test_run.py
```

Note: if you're having trouble with one function and want to skip to the next one, just use `#` to comment out that line in the `test_run.py`

If your function matches the desired output, then you should see either a "Success" message, or a printout of the timing differences.

Option 2: In a Google Colab Notebook

1. Copy the code from the `q#.py` for your corresponding question (q1 through q7). Be sure to include the imports too, EXCEPT **do not include the line `from util import print_time_results, time_funcs`. You have already pasted these functions into your notebook during the Setup.
2. Update the `vec_*` function where it says `pass # insert your code here`.
3. Run the `test_*` function to determine if your output is satisfactory, and what the speedup is (for relevent questions). You won't use the `test_run.py` script if you're working in a Notebook; just copy-paste the `test_` functions directly and run them.

Q1: Convert list to np.array

```
def convert_list_to_array(input_list: list):
    pass  # insert your code here
```

Q1: ANSWER

```
def convert_list_to_array(input_list: list):
  return np.array(input_list)
```

Q2: Convert dict to DataFrame

```
def convert_dict_to_df(input_dict: dict):
    pass  # insert your code here
```

Q2: ANSWER

```
def convert_dict_to_df(input_dict: dict):
    return pd.DataFrame.from_dict(input_dict)
```

Q3: Write a vectorized function vec_power which has the same arguments are returns as slow_power

```
def slow_power(x, m=4):
    out = []
    for x_i in x:
        out.append(x_i**m)
    return np.array(out)

def vec_power(x, m=4):
    pass  # insert your new code here
```

Brag about your gains!

1. Go to the Google Form https://forms.gle/9xrjTUSEDozPm4kJ6 and report the speed-up factor you acheived for each function! We'll shout-out the biggest speed-ups live in the workshop.

Q3: ANSWER

```
def vec_power(x, m=4):
    return x ** m
```

Q4: Write a vectorized function, vec_addition, which adds two vectors of the same size

```python
def slow_addition(arr1, arr2):
    assert len(arr1) == len(arr2)
    out = [a1 + a2 for a1, a2 in zip(arr1, arr2)]
    return np.array(out)
```

```python
def vec_addition(arr1, arr2):
    pass  # insert your code here
```

Brag about your gains!

1. Go to the Google Form https://forms.gle/9xrjTUSEDozPm4kJ6 and report the speed-up factor you acheived for each function! We'll shout-out the biggest speed-ups live in the workshop.

Q4: ANSWER

```
def vec_addition(arr1, arr2):
    return arr1 + arr2
```

Q5: Write a vectorized function, vec_grade, which has the same arguments and return as slow_grade.

```
def slow_grade(grades):
    letter_grades = []
    for grade in grades:
        if grade >= 90:
            letter_grades.append("A")
        elif 80 <= grade <= 90:
            letter_grades.append("B")
        elif 70 <= grade <= 80:
            letter_grades.append("C")
        elif grade < 70:
            letter_grades.append("F")

    return np.array(letter_grades)
```

```
def vec_grade(grades):
    pass  # insert your code here
```

Brag about your gains!

1. Go to the Google Form https://forms.gle/9xrjTUSEDozPm4kJ6 and report the speed-up factor you acheived for each function! We'll shout-out the biggest speed-ups live in the workshop.

Q5: ANSWER

```
def vec_grade(grades):
    letter_grades = np.full(len(grades), "F")
    a_idx = grades >= 90
    b_idx = (80 <= grades) & (grades< 90)
    c_idx = (70 <= grades) & (grades < 80)

    letter_grades[a_idx] = "A"
    letter_grades[b_idx] = "B"
    letter_grades[c_idx] = "C"

    return np.array(letter_grades)
```

Q6: Write a vectorized function, vec_pass_fail, which has the same arguments and returns as slow_pass_fail.

```python
def slow_pass_fail(grades):
    pass_fail_grades = []
    for grade in grades:
        if grade == "A":
            pass_fail_grades.append("P")
        elif grade == "B":
            pass_fail_grades.append("P")
        elif grade == "C":
            pass_fail_grades.append("P")
        else:
            pass_fail_grades.append("F")

    return np.array(pass_fail_grades)
```

Hint: try using np.where and np.isin

```python
def vec_pass_fail(grades):
    pass  # insert your code here
```

Brag about your gains!

1. Go to the Google Form https://forms.gle/9xrjTUSEDozPm4kJ6 and report the speed-up factor you acheived for each function! We'll shout-out the biggest speed-ups live in the workshop.

Q6: ANSWER

```
def vec_pass_fail(grades):
    return np.where(np.isin(grades, ["A", "B", "C"]), "P", "F")
```

Q7: Write a vectorized function vec_insurance_factor which has the same arguments are returns as slow_insurance_factor_wrapper

```python
def slow_insurance_factor(car_df: pd.DataFrame):

    def slow_insurance_factor_row(year: int, color: str, model: str, mileage: float):
        """Return insurance factor based on car attributes."""
        factor = 1
        if year < 2000:
            factor *= 4
        elif year < 2015:
            factor *= 3
        elif year < 2020:
            factor *= 2
        if color in ["Red"]:
            factor *= 2
        if color == "Blue":
            factor *= 1.5
        factor += max((mileage - 100000) // 1000, 1)
        return factor

    return car_df.apply(
        lambda row: slow_insurance_factor_row(
            row["year"], row["color"], row["model"], row["mileage"]
        ),
        axis=1,
    )
```

```python
def vec_insurance_factor(car_df: pd.DataFrame):
    pass  # insert your code here
```

Brag about your gains!

1. Go to the Google Form https://forms.gle/9xrjTUSEDozPm4kJ6 and report the speed-up factor you acheived for each function! We'll shout-out the biggest speed-ups live in the workshop.

Q7: ANSWER

```python
def vec_insurance_factor(car_df: pd.DataFrame):
    result = np.full(len(car_df), 1.0, dtype=float)

    idx_car_lt_2000 = car_df["year"] < 2000
    idx_car_lt_2015 = (car_df["year"] < 2015) & (car_df["year"] >= 2000)
    idx_car_lt_2020 = (car_df["year"] < 2020) & (car_df["year"] >= 2015)
    idx_car_eq_red = car_df["color"] == "Red"
    idx_car_eq_blue = car_df["color"] == "Blue"

    result[idx_car_lt_2000] = result[idx_car_lt_2000] * 4
    result[idx_car_lt_2015] = result[idx_car_lt_2015] * 3
    result[idx_car_lt_2020] = result[idx_car_lt_2020] * 2
    result[idx_car_eq_red] = result[idx_car_eq_red] * 2
    result[idx_car_eq_blue] = result[idx_car_eq_blue] * 1.5

    mileage = car_df["mileage"].copy()
    mileage = (car_df["mileage"] - 100000) // 1000
    mileage = np.array(np.maximum(mileage, 1))
    ret = pd.Series((result + mileage), dtype=float)
    return result + mileage
```

ANITA B.ORG 2024 GRACE HOPPER CELEBRATION

Thank you!

Justine Wezenaar
in/justine-wezenaar-62357857

Tamara Rosenberg
/in/tamarawarton/

Learn more:
www.TechAtBloomberg.com/Python

We are hiring: bloomberg.com/engineering

me +we

BONUS CONTENT

Building Blocks: Leveling Up

Building Blocks: Leveling Up

Broadcasting & Matrix Multiplication

Vectorized hashmap

Sort & np.argsort()

Time Series & np.roll np.nansum

Optimizing w/ Set Theory

Get data into np.array, pd.Series and/or pd.DataFrame

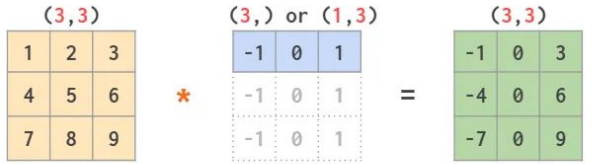
Use library functions with np.array & pd.Series args

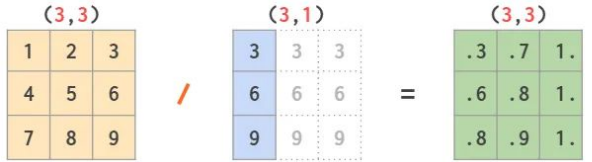
Replace if/else using set theory np.isin() np.where() np.any()

Broadcasting & Matrix Multiplication

Leveling Up: Broadcasting & Matrix Multiplication

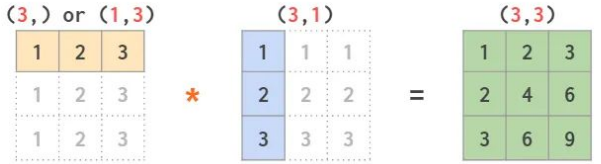
```python
def broadcasting_vec(X, c: list):
    # c is a list, numpy infers its 3x1
    # so nx3 by 3x1 multiplication works
    Y = X * c
    return Y
```





Performance on 1M rows

```
tests/koans/vectorization.py::test_broadcasting
for n=1000000: time_vec: 0.0107, time_normal: 1.5
improvement: 140.972x
```



Vectorized hashmap

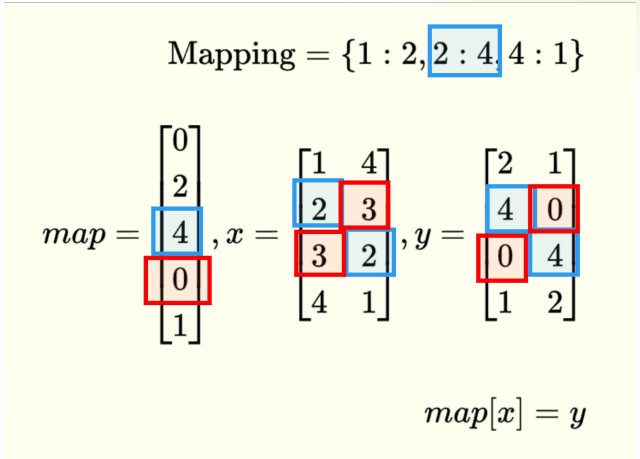
Leveling Up: Hash Lookup

Replace this:

```
def hashmap(x, params) -> list[float]:
    y = [ params.get(x_i, 0.0) for x_i in x ]
    return y
```

With this:

```
def hashmap_vec(x, params):
    # setup array of size max(dict value) + 1
    array_map = np.full(np.max(list(params.keys()))+1,0, dtype=float)
    # assign dictionary elements to array
    for key, value in params.items():
        array_map[key] = value
    # NOTE: using the array values as the index!
    y = array_map[x]
    return y
```



Performance on 10M rows

```
tests/koans/vectorization.py::test_hashmap
for n=10000000:
time_before: 4.2014, time_vectorized: 0.0309.
improvement: 135.9676x
```

Optimizing w/ Set Theory

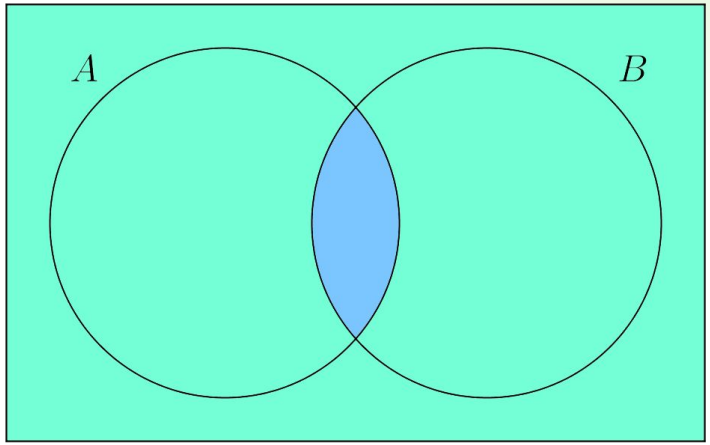
Leveling up: Optimizing Logic with Set Theory

- Use **De Morgan's Laws** (pictured: Law of Intersection) to combine & simplify all bool conditions to a single condition

- Streamline conditional logic
  - Goal 1: Refactor parentheses
  - Goal 2: One operation type (AND or OR)

- Example: (~ is NOT)
  - No error condition (A, B) occurs and input exists (C)
  - ~(A OR B) AND C
  - = (~A AND ~B) AND C
  - = ~A AND ~B AND C
  - np.any([~A, ~B, C], axis=0)

- Drawback: O(kn) where k = # conditions



$A \cap B$

$(A \cap B)^c = A^c \cup B^c$

https://brilliant.org/wiki/de-morgans-laws/

Sort & np.argsort()

Leveling Up: Vectorized Sort (np.argsort)

Replace this:

```
def argsort(x):
    y = sorted(x, key=lambda row: -row )
    return y
```

With this:

```
def argsort_vec(x):
    # make index
    idx = np.argsort(-x)
    # order vec by index
    y = x[idx]
    return y
```

Trick:
recover index!!

```
def argsort_recover_x(x):
    # lookup and order x by index
    idx = np.argsort(-x)
    y = x[idx]
    idx_rev = np.argsort(idx)
    # x = x[idx][idx_rev]
    x_again = y[idx_rev]
    return x_again
```

Performance on 1M rows

```
tests/koans/vectorization.py::test_argsort
for n=1000000:
time_before: 0.4157, time_vectorized: 0.0077.
improvement: 53.987x
```