Direct Runner vs Dataflow Runner

- Direct Runner: process a small amount of data; executes pipelines on your local machine.
- Dataflow Runner: process a large amount of data; run your pipeline with the Cloud Dataflow service, the runner uploads your executable code and dependencies to a Google Cloud Storage bucket and creates a Cloud Dataflow job, which executes your pipeline on managed resources in Google Cloud Platform. There will be overhead for setting up cluster.

ParDo vs DoFun

- **ParDo** is the computational pattern of per-element computation. The **DoFn** , here I called it fn , is the logic that is applied to each element.**Pardo** takes a **DoFn subclass** argument

ParDo

- Similar to WHERE in SQL
- Input elements are processed independently and in parallel.
- Output are bundled into a new PCollection.

DoFn

- DoFn is an argument to ParDo providing the code to use to process elements of the input PCollection.The function to use to process each element is specified by a DoFn<InputT, OutputT>, primarily via its ProcessElement method.

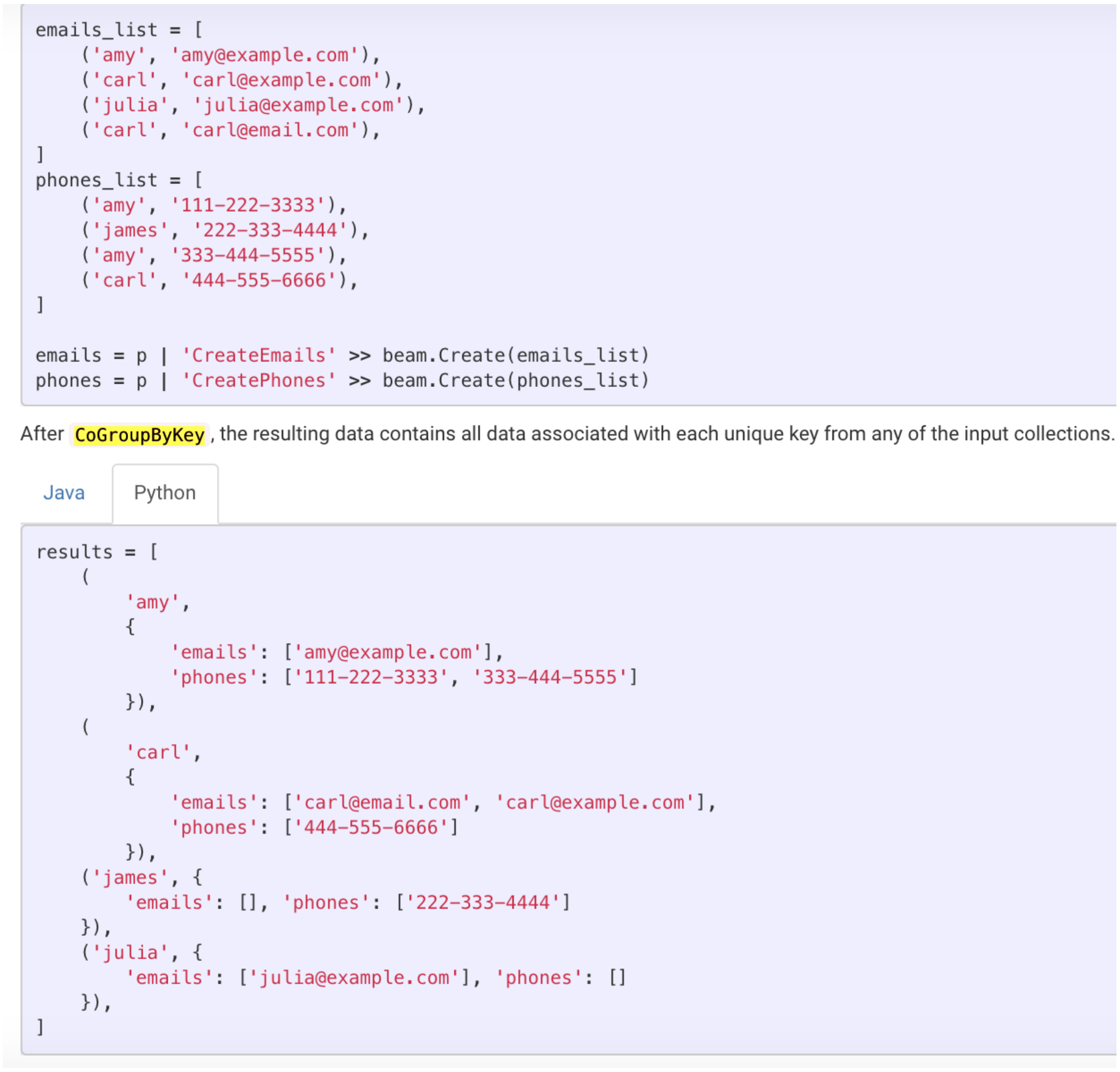
GroupByKey

- Similar to GROUP BY in SQL
- Takes a PCollection as input where each element is a (k,v) pair, which is why you may return a tuple for this transformation.
- Produces a PCollection as output where each element is a (k, **list** of v) pair

CoGroupByKey

- Similar to FULL OUTER JOIN in SQL
- Takes >= 2 PCollections as input
- Every element in the input is tuple (k,v) pair
- Produces a PCollection as output where each element is a (k, v) pair

- As a result, the result for each key is a list of dictionaries containing all data associated with that key in each input collection.



```
emails_list = [
    ('amy', 'amy@example.com'),
    ('carl', 'carl@example.com'),
    ('julia', 'julia@example.com'),
    ('carl', 'carl@email.com'),
]
phones_list = [
    ('amy', '111-222-3333'),
    ('james', '222-333-4444'),
    ('amy', '333-444-5555'),
    ('carl', '444-555-6666'),
]

emails = p | 'CreateEmails' >> beam.Create(emails_list)
phones = p | 'CreatePhones' >> beam.Create(phones_list)
```

After `CoGroupByKey`, the resulting data contains all data associated with each unique key from any of the input collections.

Java Python

```
results = [
    (
        'amy',
        {
            'emails': ['amy@example.com'],
            'phones': ['111-222-3333', '333-444-5555']
        }),
    (
        'carl',
        {
            'emails': ['carl@email.com', 'carl@example.com'],
            'phones': ['444-555-6666']
        }),
    ('james', {
        'emails': [], 'phones': ['222-333-4444']
    }),
    ('julia', {
        'emails': ['julia@example.com'], 'phones': []
    }),
]
```

Side Input

- Ordinary values or entire PCollection
- Optional arg
- Extra arg to process() method
- Basic: process(self, input)
- Single: process(self, input, side_input1)
- Multiples: process(self, input, side_input1, side_input2, side_input3...)

*Helpful link:
https://www.waitingforcode.com/apache-beam/side-input-apache-beam/read

For Milestone 6:

Be sure to include `[DIR_PATH = BUCKET + '/output/' + datetime.datetime.now().strftime('%Y_%m_%d_%H_%M_%S') + '/']` in your run(), since you are no longer run on your local machine. Explicitly specify the output location in `WriteToText(DIR_PATH + '<file_name>.txt')`