Q1. 본격적으로 Numpy와 친해지기 위해서 다양한 연산을 연습해볼 예정입니다. 무작위의 데이터를 가 진 5x3의 행렬을 가지는 numpy array와 3x2 행렬을 가지는 numpy array를 만든 후 행열곱 연산을 진행해보세요.

핵심: **무작위의 데이터를 갖는 행렬** 2개를 생성한 후, 두 행렬의 **행렬곱 연산(내적 연산)**을 계산하는 것이 목표이다.

1. **무작위 데이터 행렬**: numpy모듈의 난수 생성 메소드인 random.random()를 사용해서, 파라미터 값으로 요구하는 크기를 전달해서 생성한다.

* numpy.random 모듈의 대표적인 메소드 3가지:
 - np.random.rand(): 0~1사이 값의 균일분포에 해당하는 값을 반환한다.
 - np.random.random(): 0~1사이 값의 연속균일분포에 해당하는 값을 반환한다.
 - np.random.randn(): -1~1사이 값인 정규분포에 해당하는 값을 반환한다.

참조:https://stackoverflow.com/questions/47231852/np-random-rand-vs-np-random-random

2. **행렬곱 연산**: 행렬곱 연산은 두 행렬간의 내적 연산(dot product)를 구하는 것이다 - '행렬1@행렬2'

소스코드:

```
>>> import numpy as np
>>>
>>> arr1 = np.random.random((5,3))
>>> arr2 = np.random.random((3,2))
>>>
>>> dot = arr1@arr2
>>>
>>> print(dot, dot.shape)
[[1.40872894 0.75038566]
 [1.45327472 0.86703592]
 [1.3815716  0.83416785]
 [1.99963846 0.88173935]
 [0.83125986 0.53278745]] (5, 2)
>>>
```

Q2. 두번째로는 numpy에서 자주 사용하는 concatenate 연산에 대한 미션을 수행해보겠습니다. 이제 Numpy에서 사용하는 2개의 numpy array가 있을때, 두 numpy array의 concatenate 연산을 구해보세요.

핵심: 주어진 두개의 행렬에 대해서 **concatenate연산**을 수행하는 것이 목표이다.

1. **concatenate의 의미와 연산**: concatenate란 사전적인 의미로 '사슬 같이 연결하다'라는 의미로, 주어진 두개의 행렬을 원하는 방향으로 접합하는 것입니다.

numpy모듈의 concatenate메소드를 사용하여 연산을 합니다. concatenate메소드는 파라미터 값으로 2개의 값이 들어가게 되는데, 첫번째는 연산을 진행할 두개의 행렬을 하나의 튜플로 묶어 들어갑니다. 두번째는, 연산을 진행할 축인 axis가 들어갑니다.

소스코드:

```
>>> import numpy as np
>>>
>>> arr1 = np.array([[5,7],[9,11]], float)
>>> arr2 = np.array([[2,4],[6,8]],float)
>>>
>>> concat_1 = np.concatenate((arr1,arr2), axis=0)
>>> concat_2 = np.concatenate((arr1,arr2), axis=1)
>>>
>>> print(concat_1)
[[ 5.  7.]
 [ 9. 11.]
 [ 2.  4.]
 [ 6.  8.]]
>>> print(concat_2)
[[ 5.  7.  2.  4.]
 [ 9. 11.  6.  8.]]
>>>
```

Q3. 3번부터 5번까지의 미션는 Numpy를 이용해서 정답값을 예측해보는 linear regression을 구현해 보는 미션입니다. 첫번째 단계로 데이터를 준비해보도록 하겠습니다. 아래와 같이 데이터가 주어져있을 때, 경사하강법을 위한 데이터를 분리해보세요.

핵심: 주어진 데이터에서 경사하강법을 위한 **데이터 분리**(두개의 행을 분리)를 하는것이 목표입니다.

1. **데이터 분리**: 예제로 주어진 데이터의 모양이 매우 간결하며, 1행과 2행을 분리하는 작업만 하면 됩니다. 이때, xy행렬의 각 행 인덱스를 사용해서 데이터를 분리 및 저장할 수 있다 - xy[0] (1행 데이터/ 1번 데이터), xy[1] (2행 데이터/ 2번 데이터)

소스코드:

```
>>> import numpy as np
>>>
>>> xy = np.array([[1.,2.,3.,4.,5.,6.,],[10.,20.,30.,40.,50.,60.]])
>>>
>>> x_train =xy[0]
>>> y_train = xy[1]
>>>
>>> print(x_train, x_train.shape)
[1. 2. 3. 4. 5. 6.] (6,)
>>> print(y_train, y_train.shape)
[10. 20. 30. 40. 50. 60.] (6,)
>>>
```

Q4. 경사 하강법 구현을 위해서 위에서 분리한 x_train 데이터와 계산될 weight, bias 값을 정의해보세요. 여기서 구현한 weight와 bias 는 linear regression 계산을 진행할 시 직선의 기울기와 y 절편의 값이 됩니다.

핵심: linear regression에 사용할 weight / bias값을 **무작위로 할당**하는 것이 목표입니다.

1. **무작위 할당**: 앞서 1번에서 했던 것과 동일하게 numpy 모듈의 random메소드를 사용합니다. 다양한 메소드를 사용해보기 위해서, 이번에는 rand()메소드를 사용해봅니다.

소스코드:

```
>>> import numpy as np
>>>
>>> xy = np.array([[1., 2., 3., 4., 5., 6.], [10., 20., 30., 40., 50., 60.]])
>>>
>>> x_train = xy[0]
>>> y_train = xy[1]
>>>
>>> beta_gd = np.random.rand(1)
>>> bias = np.random.rand(1)
>>>
>>> print(beta_gd, bias)
[0.90522587] [0.86903265]
>>>
```

Q5. 이제 최종적으로 linear regression을 경사하강법으로 학습하는 코드를 구현해봅시다. 경사하강법 구현을 위한 학습 Loop를 구현해보세요. 최종적으로 1000회 반복했을 시에 결과를 출력하세요.

핵심: linear regression을 **경사하강법으로 학습**하는 것이 목표입니다. 이때, 경사하강법을 정의하기 위해서는, '**목적함수/그래디언트/파라미터 최적화**'을 정의해야한다.

1. 경사하강법:

정의 - 경사하강법은 비용(cost)를 최소화 하기 위한 최적화 알고리즘입니다. 주어진 함수의 최소값 위치를 찾기 위해서 목적함수(비용함수/손실함수 등 같은 개념)의 경사의 반대 방향으로, step_size(learning rate)만큼의 속도로 조금씩 움직여가며 최적의 파라미터를 업데이트하는 방법입니다.

손실값(cost/error/gradient)는 다음과 같이 정의할 수 있습니다.

$$cost(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

일차원 함수(y = Wx + b)에서 최적의 W(weight)와 b(bias)값을 찾기 위해서는, 매 반복(iteration)마다 gradient방향으로 learning rate만큼 이동을 하면서 파라미터 값을 업데이트합니다.

$$W := W - \alpha \frac{\partial}{\partial W} \cos t(W,b) \qquad \text{※ } \alpha = \text{학습률}(learning\ rate)$$

(참조: https://23min.tistory.com/4)

2. **목적함수**: 최적화의 대상이 되는 목적함수를 정의합니다
 - pred = beta_gd * x_train + bias
3. **그레디언트**: 최적화의 방향이 되는 그레디언트를 정의합니다
 - gd_w = sum(np.multiply(pred - y_train, 2 * x_train)) / len(y_train)
 - gd_b = sum(np.multiply(pred - y_train, 2)) / len(y_train)
4. **파라미터 업데이트**: 목적함수를 최적화하기 위해, beta_gd와 bias를 learning_rate (0.01)만큼 업데이트 합니다.

- **beta_gd -= learning_rate * gd_w**
- **bias -= learning_rate * gd_b**

5. **오차(비용/cost) 확인**: 100번마다 예측값과 실제값의 차이, 즉 오차값을 확인합니다
   - error = sum(np.square(pred - y_train)) / len(y_train)

소스코드:

```python
import numpy as np

xy = np.array([[1., 2., 3., 4., 5., 6.], [10., 20., 30., 40., 50., 60.]])

# 각 변수에 x축과 y축의 주소를 저장한다.
x_train = xy[0]
y_train = xy[1]

beta_gd = np.random.rand(1) # 임시 기울기
bias = np.random.rand(1)    # 임시 절편

learning_rate = 0.01        # 학습률

for i in range(1000):
    pred = beta_gd * x_train + bias                         # 예측값 wx+b
    error = sum(np.square(pred - y_train)) / len(y_train)   # MSE mean
    #예측값과 실제 값의 차이

    gd_w = sum(np.multiply(pred - y_train, 2 * x_train)) / len(y_train) # MSE 미분값 / gradient Descent Weight
    gd_b = sum(np.multiply(pred - y_train, 2)) / len(y_train)           # MSE 미분값 / gradient Descent Bias

    beta_gd -= learning_rate * gd_w # 기울기 조정
    bias -= learning_rate * gd_b    # 절편 조정

    if i % 100 == 0: # 100번마다 출력
        print(f'Epoch ({i:10d}/1000) cost: {error:10f}, w: {beta_gd.item(0):10f}\,b: {bias.item(0):10f}')
```

```
(base) youngjoooh@oyeongjuui-MacBookAir week3 % python3 Q5.py
Epoch (         0/1000) cost: 1351.743713, w:   3.260681\,b:   1.362548
Epoch (       100/1000) cost:    0.723532, w:   9.548724\,b:   1.932002
Epoch (       200/1000) cost:    0.348251, w:   9.686917\,b:   1.340370
Epoch (       300/1000) cost:    0.167620, w:   9.782792\,b:   0.929911
Epoch (       400/1000) cost:    0.080679, w:   9.849307\,b:   0.645147
Epoch (       500/1000) cost:    0.038832, w:   9.895453\,b:   0.447585
Epoch (       600/1000) cost:    0.018691, w:   9.927468\,b:   0.310522
Epoch (       700/1000) cost:    0.008996, w:   9.949680\,b:   0.215432
Epoch (       800/1000) cost:    0.004330, w:   9.965089\,b:   0.149461
Epoch (       900/1000) cost:    0.002084, w:   9.975780\,b:   0.103692
(base) youngjoooh@oyeongjuui-MacBookAir week3 %
```