

## 들어가며

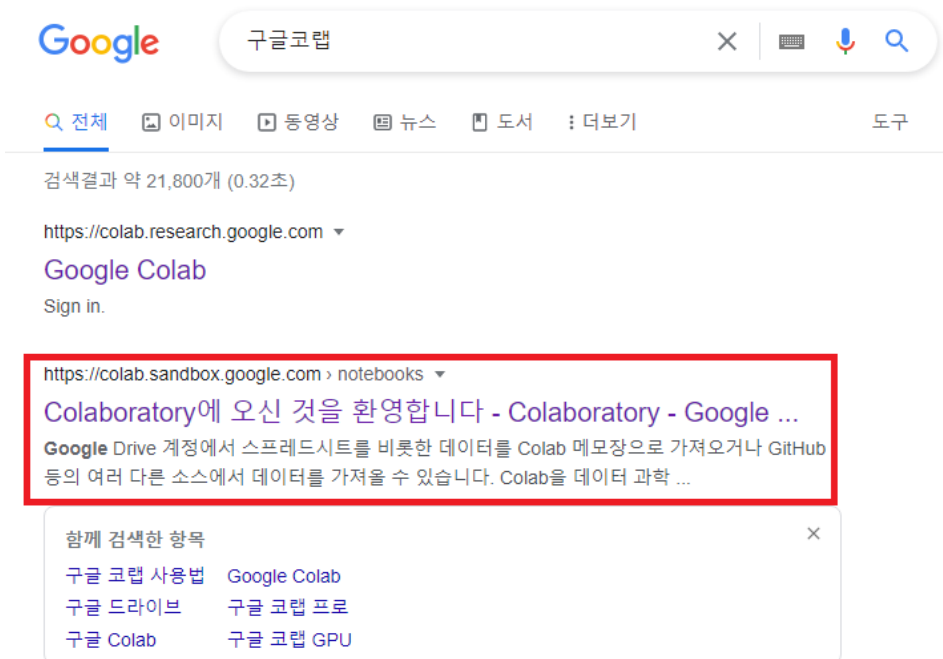
반갑습니다. 조금 까다로워 보이는 강의계획에도 이렇게 모인 것은 인공지능에 대한 관심이 그만큼 지대하기 때문인 것 같습니다. 그래도 내용이 내용이니 만큼 조금 긴장할 필요는 있습니다만 강의를 진행하다가 평균적인 이해 정도에 따라 눈높이를 조절할터이니 크게 걱정할 필요는 없습니다. 주최측에서 학생들에게 특별한 요구사항이 있는 것은 아닌 것으로 보이므로 이번 주 오후에 시간을 내어 편안하게 강의를 들으면서 실습을 진행하면 되겠습니다. 단기에 수업을 마치므로 매일 나오는 것이 조금 부담은 되겠지만, 사흘 정도만 오후 시간에 학교에 다닌다 생각하면 크게 부담이 되는 것은 아니죠? 수업에 들어가기 전에 프로그래밍 언어에 대한 기초적인 내용은 좀 알아야 하겠죠?

수업의 형태는 앞에서 코드를 설명하고, 여러분들이 이 코드를 따라하는 방식입니다. 앞에서 타이핑해주는대로 코드 구성이 잘 되지 않거나 결과가 잘 나오지 않거나, 에러가 나오면 지체없이 질문을 해주세요. 수업의 내용은 자료를 준비하는 방법에 대한 것입니다. 데이터를 인공지능에 제대로 적용하기 위해서는 여러가지 방법을 동원하여 필요한 형식으로 맞추어야 합니다. 이 과정을 전처리라고 합니다. 그리고 전처리를 하려면 코딩을 배우는 것이 좋습니다. 그렇지 않으면 자료의 형태를 원하는대로 탈바꿈시키기가 만만하지 않을테니까요.

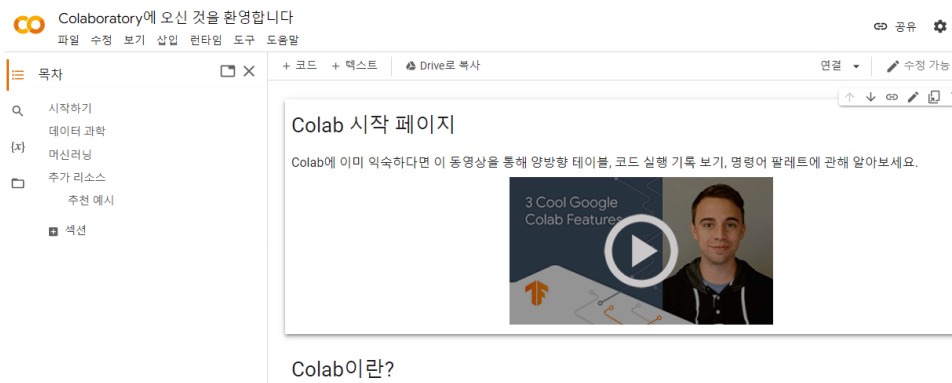
예상했던대로 언어는 파이썬으로 합니다. 파이썬은 1991년에 네덜란드 프로그래머인 귀도 반 로섬이 만든 컴퓨터 언어입니다. 비영리를 표방하고, 연구소나 개인들이 많이 쓰던 언어에서 점차 확산되더니 인공지능용 언어로 채택되면서 세계적으로 유명세를 떨치게 되었습니다. 인공지능에 대해 실습하려면 파이썬을 먼저 떠올릴 정도입니다. 구글 등을 비롯한 많은 영리, 비영리 기관에서 파이썬을 활용한 인공지능 개발용 라이브러리를 앞다투어 지원하고 있습니다.

## 파이썬 첫걸음

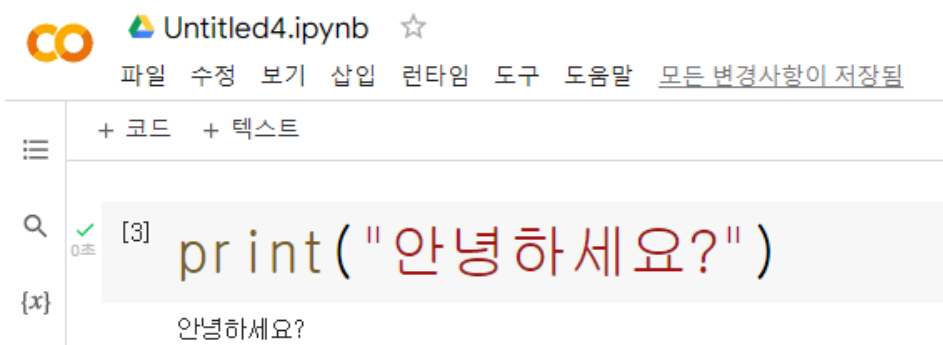
일단, 실습환경이 필요하겠죠? 파이썬 인터프리터를 파이썬 재단에서 다운받고 설치하는 것도 좋겠지만, 이것은 소프트웨어를 공부할 때 더 효율적인 것 같습니다. 이런 방법보다는 인공지능을 위한 실습환경을 미리 체험하는 것이 더 좋을 것 같습니다. 구글 검색창에서 “구글코랩”을 검색하기 바랍니다.



검색되는 첫번째 링크도 좋습지만, 기왕이면 두번째 링크가 더 괜찮습니다. 구글코랩 홈페이지답게 구글코랩에 대한 설명이 나와 있어서 한번 읽어보는 것도 괜찮을 것 같습니다.



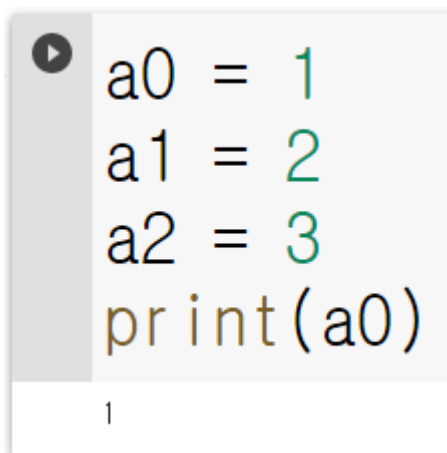
구글 코랩은 인공지능 프로그램 개발을 위한 구글 클라우드 서비스입니다. 본격적인 개발에 앞서서 간단한 코딩테스트를 하는 정도는 무료로 개방되어 있습니다. 구글 계정은 가지고 있나요? 아니면 이번 기회에 한 번 만들어보세요. 로그인을 해야 실습한 자료를 보관할 수 있습니다.



위 그림에서 `print`라는 명령어를 쓴 곳이 셀이라고 하는 코딩 장소입니다. 한 셀에 모든 코드를 다 집어넣을 수도 있고, 적당히 셀단위로 나누어서 구성할 수도 있습니다. 지금은 `print`문에 인사말을 넣어서 출력하는 모습을 보여줍니다. 방금 언급했지만 자료를 인쇄할 때는 `print`라는 명령어를 씁니다. `print` 함수라고 합니다. `print` 다음에 괄호를 표시하고 인쇄할 내용을 집어넣으면 됩니다. 문자열을 인쇄할

경우는 따옴표를 사용하여 구분할 수 있습니다. 이렇게 실행할 코드가 만들어지면 결과를 볼 수 있습니다. 실행하는 방법은 **Shift + Enter**키를 치면 됩니다.

하나의 자료가 있다고 합시다. 시간 중에 초를 보관한다고 해보죠. 초기값을 0초라고 할까요? 1초가 지날 때마다 값을 1만큼 증가시켜야 할 것입니다. 이렇게 하려면 초를 보관할 장소가 필요합니다. 특정 자료를 보관하는 장소로는 변수가 적합합니다. **sec = 0**과 같은 형식으로 자료 보관이 가능합니다. '=' 기호를 중심으로 왼쪽에는 변수명을 쓰고 오른쪽에는 보관할 값을 넣습니다(실제로는 상수가 있는 장소를 변수가 가리키고 있다고 해야 맞는 얘기이지만 지금은 그냥 넘어가는 것이 좋겠습니다). 변수명은 영문 알파벳과 숫자 또는 기호를 조합하여 작성합니다. 위의 코드에서는 **print** 함수에 상수 문자열을 그냥 적용했지만, **str = "안녕하세요?"**라고 미리 변수 **str**에 문자열을 보관했다면 **print(str)**이라고 하여 인쇄하는 것도 가능합니다.



```
a0 = 1
a1 = 2
a2 = 3
print(a0)
```

1

위 셀에서는 **a0**부터 **a2**까지 1에서 3에 이르는 숫자를 대입하였습니다. 그리고 **a0**를 인쇄하고 있는 모습입니다. 그런데, 변수 3개의 값을 동시에 인쇄하려면 어떻게 하면 될까요? **print** 함수에는 항목을 1개만

넣을 필요는 없습니다. 필요에 따라 여러 항목을 넣을 수 있지요. 다만 항목과 항목 사이는 ‘,’로 구분해야 합니다.

```
[2] print(a0, a1, a2)
```

```
1 2 3
```

그런데, 하나의 변수에는 이렇게 단일값만 입력할 수 있는 걸까요? 그게 아니라면 여러개의 자료를 보관할 수 있는 걸까요? 여러분은 배열이라고 들어본 적이 있습니까? 변수의 입장에서 배열을 생각해 보면, 변수 하나가 여러개의 값을 가지는 것으로 정의하면 될 것 같습니다. 그런데 파이썬에서 한 변수가 여러개의 값을 가지려면 어떻게 표현해야 할까요?

```
[5] a = [1, 2, 3]
    print(a)
```

```
[1, 2, 3]
```

```
[6] print(a[0], a[1], a[2])
```

```
1 2 3
```

위의 5번 셀과 같이 **a**라는 변수에 1부터 3을 넣습니다. 그런데 대괄호로 둘러싸여 있죠?(사실 이러한 자료구조를 리스트라고 합니다) 그래서 **a**를 인쇄하면 여러값을 한꺼번에 출력할 수 있습니다. 물론 **a**의 원소

하나하나를 개별적으로 출력하는 것도 가능하죠. 6번 셀에서 확인할 수 있습니다.

```
[7] print(a[0])
     print(a[1])
     print(a[2])
```

1  
2  
3

7번 셀에서는 **a** 배열의 원소 하나하나를 명시하여 차례대로 인쇄하는 모습을 나타냅니다. 그런데, 배열의 원소의 갯수가 클 때도 이렇게 하는게 좋을까요? 반복 작업이 너무 심하지 않나요? 컴퓨터 프로그래밍 언어에서는 반복을 해야 할 때 루프 기능을 제공합니다. 같은 행위를 일정한 횟수만큼 되풀이할 수 있게 해줍니다. 파이썬에서도 마찬가지입니다.

```
[8] for b in a:
     print(b)
```

1  
2  
3

결과적으로 7번셀과 8번셀은 같은 결과를 나타냅니다. 배열의 원소가 많을 때 어떤 방법이 더 효율적일까요? 루프를 돌기 위해서는 **for**라는

키워드를 써야 합니다(다른 키워드도 있습니다만 일단 이거 하나로도 충분합니다). 그리고 **for**의 구조는 변수가 하나 오고 다음에 **in**이라는 키워드를 씁니다. **in** 다음에는 배열 변수를 적습니다. **for** 문의 끝에는 반드시 콜론을 넣어야 합니다. 콜론 다음 줄부터는 **for** 문의 범위에 들어가게 됩니다. 지금은 **print** 문 하나지만 언제까지라도 **for** 문의 범위를 유지할 수 있죠. 그러면 **for** 문을 끝내고 싶을 때는 어떻게 하면 될까요? 위 그림에서 **for** 문의 **f**가 쓰여진 위치와 **print**의 **p**가 쓰여진 위치를 잘 보세요. 위치가 같지 않죠? **f**가 1열이라면 **p**는 3열쯤 되어 보입니다. 이와 같이 **for**의 범위 내에 있는지 그렇지 않은지는 들여쓰기에 따라 좌우됩니다. **indentation**이라고 합니다.

다음 그림을 보세요. 배열이 독특하게 표현되어 있죠? **[1, 2, 3]**이라는 배열과 **[4, 5, 6]**이라는 배열이 하나의 배열 **arr**에 포함된 모습을 하고 있습니다. **a**가 1차원 배열이라면 **arr**은 2차원 배열이라고 할 수 있습니다. 그 모습도 표모양을 연상하도록 일부러 2줄에 걸쳐 표현하였습니다.

```
[9] arr = [[1, 2, 3],
          [4, 5, 6]]
print(arr)
```

**[[1, 2, 3], [4, 5, 6]]**

일부러 표모양을 하도록 타이핑했지만, 행 속에 들어가야 열이 보입니다.

```
[10] print(arr[0], arr[1])
```

```
[1, 2, 3] [4, 5, 6]
```

```
[11] print(arr[0][0], arr[0][1], arr[0][2],
          arr[1][0], arr[1][1], arr[1][2])
```

```
1 2 3 4 5 6
```

2차원 배열의 원소는 11번 셀처럼 표현하여 읽을 수 있습니다.

```
[12] for row in arr:
      print(row)
```

```
[1, 2, 3]
[4, 5, 6]
```

2차원 배열을 하나의 **for** 루프로 인쇄한다면 12번 셀처럼 나올 것입니다. 즉 행별로는 출력이 되지만, 열 각각의 원소로는 출력이 되지 않습니다. 그러니 루프안에 루프를 또 한번 돌릴 필요가 있습니다.



```
[13] for row in arr:
      for col in row:
        print(col)
```

```
1
2
3
4
5
6
```

(연습) `arr[row][col]` 형태로 루프를 구성하려면 어떻게 하면 될까요?

힌트: 이 문제도 1차원 배열에 대해서 만들어 본다음 2차원으로 넘어가기 바랍니다.

여러분, 선생님이 다양한 분야의 도서를 통해 안목을 넓히라고 권하죠? 그런 다양한 분야의 책들은 도서관에 가면 찾을 수 있습니다. 파이썬에서도 라이브러리가 있어서 다양한 형태의 기능을 활용할 수 있도록 도와줍니다. 그 중 넘파이(NumPy)는 대규모 다차원 배열을 처리하는데 도움을 주는 파이썬 라이브러리입니다. `print` 함수처럼 기본적으로 사용이 되지를 않아서 도서관에서 대출받는 형식을 취합니다.

넘파이를 대출받을 때는 아래 19번 셀처럼 `import` 문을 씁니다. `as`는 넘파일을 사용할 때 약자를 뭘로 할 것인지를 나타냅니다. `np.array(a)`는 1차원 배열 `a`를 넘파이의 배열로 취한다는 의미를 내포하고 있습니다. `anp`의 인쇄된 값을 보면 `a`와 같은 값이 나오지만, `anp`를 찍어보면 `array`

```
[19] import numpy as np
      anp = np.array(a)
      print(anp)
      anp
```

```
[1 2 3]
array([1, 2, 3])
```

라고 표시되는 것을 확인할 수 있습니다. 이와 같이 셀의 맨 마지막 문장에 변수명을 나타내면 꼭 `print`와 유사하게 그 값을 나타냅니다. 그런데 이렇게 `array`가 붙고 안붙고 하는 것이 무슨 차이가 있을까요?

```
[20] anp2 = np.array(arr)
```

```
[21] arr[0][2]
```

```
3
```

```
[22] anp2[0,2]
```

```
3
```

셀 20에서 2차원 배열 `arr`을 넘파이의 배열로 삼고 `anp2`에 보관하였습니다. 원래 `arr`은 2차원 배열로, 대괄호를 연속으로 표기하여 특정 원소를 나타내었습니다. 그런데 넘파이의 배열이 되면 셀 22처럼 대괄호와 콤마로 원소 번호를 표기할 수 있습니다. 사실 리스트

자료구조의 경우는 지원되는 함수가 몇개 안되는데, 넘파이의 배열이 되면 지원되는 함수가 매우 다양해집니다.

## 스칼라(scalar)와 벡터(vector)

우리가 사용하는 정수나 실수를 스칼라라고 하는데, 스칼라는 크기만을 가지고 있는 물리량을 의미합니다. 벡터는 크기뿐만 아니라 방향을 가진 수의 집합을 의미합니다. 넘파이 배열에서는 1차원 행으로 만듭니다.

```
[4] vec = np.arange(10)
vec
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

np의 `arange` 함수를 사용하여 벡터 `vec`을 만들었습니다. `arange`의 매개변수 `10`은 종료값을 의미합니다. `arange`는 시작점부터 종료 지점 직전까지 수를 나열해줍니다. 시작점을 표시하지 않으면 `0`부터 나타냅니다.

```
[5] vec2 = np.arange(3, 10)
vec2
array([3, 4, 5, 6, 7, 8, 9])
```

위 5번 셀의 경우는 시작값을 표시한 것입니다. 이 때는 3에서 출발하여 10이전인 9까지 벡터를 만듭니다.

```
[7] vec3 = np.arange(3, 10, 2)
vec3
array([3, 5, 7, 9])
```

위 그림과 같은 7번 셀의 경우 시작점, 종료지점, 간격을 모두 표시하였습니다. 3부터 출발하여 2만큼씩 증가됨을 확인할 수 있습니다.

## 벡터 크기 계산

벡터의 크기는 피타고리스 정리인 직각 삼각형의 빗변을 구하여 얻을 수 있습니다. 예를 들어 3과 4를 가지는 벡터  $v$ 가 있다고 해봅시다.

```
[9] v = np.arange(3, 5)
v
array([3, 4])
```

이 벡터에 대한 제곱값을 구합니다.

```
[11] vsum = np.sum(vsquare)
      vsum
```

25

그리고 두 제곱수를 더해줍니다.

```
[12] vroot = np.sqrt(vsum)
      vroot
```

5.0

더해진 값에 대해 루트를 구한 결과를 셀 12에 나타내었습니다.

```
[13] np.hypot(v[0], v[1])
```

5.0

셀 13에서는 지금까지 구한 계산 과정을 원스텝으로 표현한 것입니다. `hypot` 함수로 피타고라스 정리를 한 번에 구현할 수 있습니다.

```
[14] np.linalg.norm(v)
```

5.0

셀 14에서도 같은 결과를 얻지만 `hypot` 함수보다는 다소 복잡해 보이죠? 그런데 엄밀히 말해서 `hypot` 함수는 직삼각형의 빗변을 구하기 위한

함수이지 벡터의 크기를 구하는 함수는 아닙니다. 벡터의 원소가 2개일 경우 `hypot`을 이용할 수 있지만, 3개 이상일 경우는 쓸 수 없기 때문이죠. 그래서 일반적으로는 `norm` 함수를 사용해야 합니다.

## 단위 벡터의 계산

벡터의 크기가 1인 벡터를 단위 벡터(`unit vector`)라고 합니다. 단위 벡터 중에 표준 단위벡터(`standard unit vector`)는 원소 중 하나가 1이고 나머지는 0 값을 가집니다.

$$\mathbf{e}_1 = (1, 0, 0, \dots, 0), \mathbf{e}_2 = (0, 1, 0, \dots, 0), \dots, \mathbf{e}_n = (0, 0, 0, \dots, 1), \dots$$

표준 단위 벡터  $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ 를 1차원 배열로 만들고 `norm` 함수로 크기를 계산해보세요(원소를 너무 많이 대입하기보다는 3개 정도로 하여 구해봅니다).

```
[15] e1 = np.array([1, 0, 0])  
      np.linalg.norm(e1)
```

1.0

```
[16] e2 = np.array([0, 1, 0])  
      np.linalg.norm(e2)
```

1.0

```
[17] e3 = np.array([0, 0, 1])  
      np.linalg.norm(e3)
```

1.0

일반적으로 단위 벡터를 구하려면 벡터의 크기를 구하고 이 값을 벡터의 원소에 각각 나누어 계산합니다. 예를 들어 1, 2, 3의 원소를 가지는 1차원 배열을 만들고 벡터의 크기를 구해봅시다.

```
[18] v_3 = np.array([1, 2, 3])
      v_3n = np.linalg.norm(v_3)
      v_3n
```

```
3.7416573867739413
```

```
[19] v_3u = v_3 / v_3n
      v_3u
```

```
array([0.26726124, 0.53452248, 0.80178373])
```

```
[20] np.linalg.norm(v_3u)
```

```
1.0
```

셀 19에서는 벡터 원소를 벡터의 크기로 나누고 있습니다. 이 값으로 다시 벡터의 크기를 구하면 셀 20처럼 1이 나옵니다.

## 두 벡터의 거리 구하기

두 벡터의 거리는 두 벡터 간의 차를 구하고 제곱을 한 후에 모든 제곱값을 더하고 루트를 취합니다. 즉 피타고라스 정리와 같습니다. 간단하게 두 벡터를 만들고, 거리를 구해보도록 합시다. 셀 21에서는 임의의 두 벡터를 만들고, 두 벡터간 차이에 대해 피타고라스 정리를 적용하였습니다. 또한 셀 22에서는 `norm` 함수로 셀 21의 결과와 동일한 값이 나오는 것을 확인하였습니다.



```
[21] v1 = np.arange(1, 4)
      v2 = np.arange(4, 7)
      np.sqrt(np.sum(np.square(v1-v2)))
```

5.196152422706632

```
[22] np.linalg.norm(v1-v2)
```

5.196152422706632

## 행렬

행렬은 2차원이죠? 차원의 수평 방향을 행(row)이라고 하고 수직 방향을 열(column)이라고 합니다.

```
[23] mat = np.array([[1], [2]])
      mat
```

```
array([[1],
       [2]])
```

위 그림은 2행 1열의 행렬을 나타냅니다. 벡터와는 무슨 차이가 있는지 구별이 되나요?

## 행렬의 랭크(rank)

행렬은 행 또는 열을 중심으로 벡터를 쌓아올리는 구조를 가지고 있습니다. 이 행렬 내의 벡터 조합으로 다른 벡터를 만들 수 있는 경우가 있고 없는 경우가 있습니다. 바로 후자의 경우로 다른 벡터를 만들 수 없는 벡터를 행렬의 랭크라고 합니다. 그러면 벡터부터 시작하여 행렬의 랭크를 구해보도록 하죠.

```
[24] rank1 = np.array([1, 2, 3, 4])
      np.linalg.matrix_rank(rank1)
```

1

```
[25] rank2 = np.array([[1, 2, 4], [2, 4, 8]])
      np.linalg.matrix_rank(rank2)
```

1

```
[26] rank3 = np.array([[1, 2, 4], [2, 4, 8], [3, 4, 5]])
      np.linalg.matrix_rank(rank3)
```

2

셀 24의 벡터는 자신만이 유일한 벡터이므로 다른 벡터를 생각할 필요조차 없습니다. 아무튼 다른 벡터를 구성하지 못하므로 랭크는 1이 됩니다. 셀 25의 행렬은 첫 행의 원소에 각각 2를 곱하여 두번째 행을 만들 수 있습니다. 즉 두번째 행인  $[2, 4, 8]$ 은 첫번째 행에 선형종속이라고 합니다.  $[1, 2, 4]$ 는 유일합니다. 이를 선형 독립이라고 하며 랭크는 1이 됩니다. **rank3**는 첫 행과 세번째 행이 유일하게 선형 독립이므로 랭크는 2가 됩니다.

```
[28] np.linalg.det(rank3)
```

0.0

그런데, **rank3**의 행렬식을 구해보면 0이 나옵니다. 행렬식이 0이 나올 경우 역행렬을 구할 수 없습니다. 아래 행렬은 모든 원소가 0인 벡터를 추가한 것입니다. 첫번째와 두번째 벡터에 0을 곱하면 세번째 벡터를 얻을 수 있으므로 랭크는 2가 됩니다. 그리고 행렬식의 값은 0이 되는데, 중요한 의미가 있죠? 딥러닝을 준비할 때 행렬을 확장할 때가 있는데요, 이 때 요긴하게 쓰입니다.

```
[30] adzvec = np.array([[1, 2, 3], [4, 5, 6], [0, 0, 0]])
      adzvec
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [0, 0, 0]])
```

```
[31] np.linalg.matrix_rank(adzvec)
```

```
2
```

```
[32] np.linalg.det(adzvec)
```

```
0.0
```