팬시 인덱싱¶

팬시 인덱싱은 단순인덱싱과 비슷하지만 단일 스칼라 대신 인덱스 배열을 전달한다. 이로써 복잡한 배열 값의 하위 집합에 매우 빠르게 접근해 그것을 수정할 수 있다.

팬시 인덱싱 알아보기¶

팬시 인덱싱은 개념적으로 간단하다. 즉, 한 번에 여러 배열 요소에 접근하기 위해 인덱스의 배열을 전달한다.

```
In [1]:import numpy as np
        rand = np.random.RandomState(42)
        x = rand.randint(100, size=10)
        print(x)
[51 92 14 71 60 20 82 86 74 74]
```

세 개의 다른 요소에 접근하고자 하는 경우 :

```
In [2]:[x[3], x[7], x[2]]
Out[2]:[71, 86, 14]
```

아니면 인덱스의 단일 리스트나 배열을 전달해 같은 결과를 얻을 수 있다.

```
In [3]:ind = [3, 7, 4]
        x[ind]
Out[3]:array([71, 86, 60])
```

팬시 인덱싱을 이용하면 결과의 형상이 인덱싱 대상 배열의 형상이 아니라 인덱스 배열의 형상을 반영한다.

```
In [4]:ind = np.array([[3, 7],
           [4, 5]])
        x[ind]
Out[4]:array([[71, 86],
              [60, 20]])
```

팬시 인덱싱은 여러 차원에서도 동작한다. :

```
In [5]:X = np.arange(12).reshape((3, 4))
        X
Out[5]:array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

표준 인덱싱을 사용할 때와 마찬가지로 첫번째 인덱스는 행을 말하며 두 번째 인덱스는 열을 말한다.

```
In [6]:row = np.array([0, 1, 2])
        col = np.array([2, 1, 3])
        X[row, col]
Out[6]:array([ 2,  5, 11])
```

결과의 첫번째 값은 x[0,2], 두 번째 값은 x[1,1], 세번째 값은 x[2,3]이다. 팬시 인덱싱에서 인덱스 쌍을 만드는 것은 “배열 연산: 브로드캐스팅”에서 언급했던 모든 브로드캐스팅 규칙을 따른다. 따라서 안덱스 내의 열 벡터와 향 벡터를 결합하면 2 차원 결과를 얻는다.

```
In [7]:X[row[:, np.newaxis], col]
```

```
Out[7]:array([[ 2,  1,  3],
              [ 6,  5,  7],
              [10,  9, 11]])
```

여기서 각 행의 값은 산술 연산의 브로드캐스팅에서 본 것과 똑같이 각 열 벡터와 일치한다. :

```
In [8]:row[:, np.newaxis] * col
Out[8]:array([[0, 0, 0],
              [2, 1, 3],
              [4, 2, 6]])
```

팬시 인덱싱을 사용하면 반환값은 인덱싱 대상 배열의 향상이 아니라 브로드캐스팅된 인덱스의 형상을 반영한다는 사실을 반드시 기억하자.

결합 인덱스¶

더 강력한 연산을 위해 팬시 인덱싱을 앞에서 본 다른 인덱싱과 결합할 수 있다.

```
In [9]:print(X)
              [[ 0  1  2  3]
              [ 4  5  6  7]
              [ 8  9 10 11]]
```

팬시 인덱스와 단순 인덱스를 결합할 수 있다.

```
In [10]:X[2, [2, 0, 1]]
Out[10]:array([10,  8,  9])
```

또한, 팬시 인덱싱과 슬라이싱을 결합할 수 있다.

```
In [11]:X[1:, [2, 0, 1]]
Out[11]:array([[ 6,  4,  5],
              [10,  8,  9]])
```

그리고 팬시 인덱싱 방식은 모두 배열값에 접근하고 수정하기에 매우 유연한 연산을 수행하게 해준다.:

```
In [12]:mask = np.array([1, 0, 1, 0], dtype=bool)
              X[row[:, np.newaxis], mask]
Out[12]:array([[ 0,  2],
              [ 4,  6],
              [ 8, 10]])
```

이렇게 결합된 인덱싱 방식은 모두 배열값에 접근하고 수정하기에 매우 유연한 연산을 수행하게 해준다..

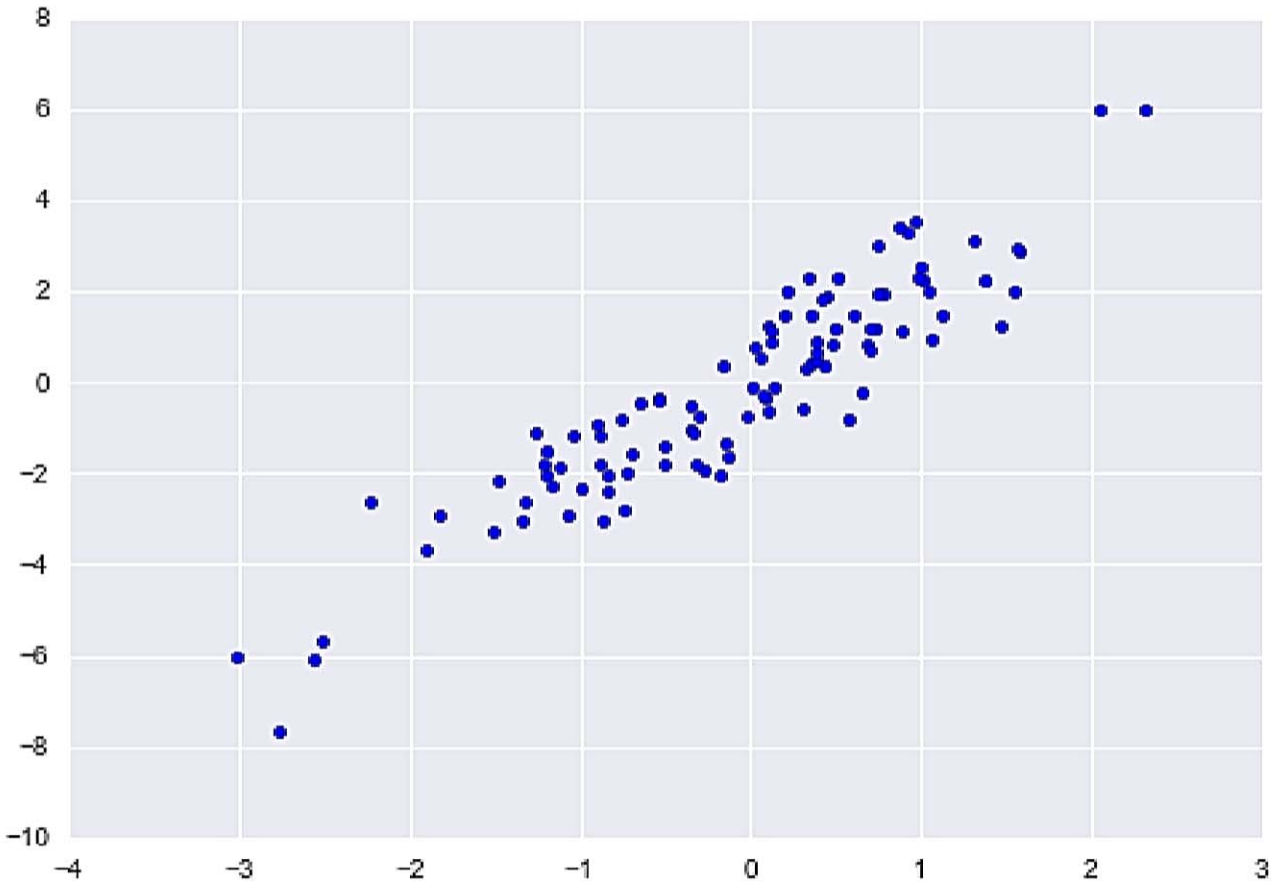
예제: 임의의 점 선택하기¶

팬시 인덱싱의 보편작인 용도는 행렬에서 행의 부분집합을 선택하는 것이다. 예를 들어 2 차원 정규분포에서 뽑아낸 다음 점들처럼 D 차원 N 개의 점을 표시하는 NxD 행렬이 있다고 해보자.

```
In [13]:mean = [0, 0]
              cov = [[1, 2],
                   [2, 5]]
              X = rand.multivariate_normal(mean, cov, 100)
              X.shape
Out[13]:(100, 2)
```

프로팅도구를 사용해 산점도로 이 점들을 시각화할 수 있다.

```
In [14]:%matplotlib inline
        import matplotlib.pyplot as plt
        import seaborn; seaborn.set()  # 플롯 스타일 설정
        plt.scatter(X[:, 0], X[:, 1]);
```



팬시 인덱싱을 이용해 임의의 점 20개를 선택하자 우선 반복 없이 임의로 인덱스를 20개 선택하고 그 인덱스를 사용해 원본 배열의 일부를 선택한다.

```
In [15]:indices = np.random.choice(X.shape[0], 20, replace=False)
        indices
Out[15]:array([93, 45, 73, 81, 50, 10, 98, 94,  4, 64, 65, 89, 47, 84, 82, 80, 25,90, 63, 20])
In [16]:selection = X[indices]  # fancy indexing here
        selection.shape
Out[16]:(20, 2)
```

이제 어느 점이 선택됐는지 보기 위해 선택된 점 위에 큰 동그라미를 표시하자

```
In [17]:plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
        plt.scatter(selection[:, 0], selection[:, 1],
        facecolor='none', s=200);
```

이러한 전략은 통계 모델 검증을 위해 훈련 / 테스트 집단을 분할하거나 통계적 질문에 답하기 위해 샘플링할 때 종종 필요한 데이터를 신속하게 분할하는데 자주 사용된다..

팬시 인덱싱으로 값을 변경하기¶

팬시 인덱싱이 배열의 일부에 접근하느 데 사용되는 것과 마찬가지로 배열의 일부를 수정하는 데도 사용될 수 있다.
예를 들어 인덱스 배열이 있고 배열에서 그 인덱스 배열에 해당하는 항목에 특정 값을 설정하고 싶다고 하자.

```
In [18]:x = np.arange(10)
         i = np.array([2, 1, 8, 4])
         x[i] = 99
         print(x)
[ 0 99 99  3 99  5  6  7 99  9]
```

이를 위해 할당 유형의 연산자는 모두 사용할 수 있다. :

```
In [19]:x[i] -= 10
         print(x)
[ 0 89 89  3 89  5  6  7 89  9]
```

그렇지만 이 연산에서 반복되는 인덱스는 예상하지 못한 결과를 초래할 수도 있다.:

```
In [20]:x = np.zeros(10)
         x[[0, 0]] = [4, 6]
         print(x)
```

[6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

4 는 어디로 갔을까? 이 연산의 결과는 먼저 x[0]=4 가 할당되고 그 다음에 x[0]=6 이 할당됐다. 물론 그 결과는 x[0]이 값 6 을 갖는 것이다.

```
In [21]:i = [2, 3, 3, 4, 4, 4]
         x[i] += 1
         x
Out[21]:array([ 6.,  0.,  1.,  1.,  1.,  0.,  0.,  0.,  0.,  0.])
```

이 경우 각 인덱스가 반복되는 횟수에 따라 x[3]이 값 2 를 포함하고 x[4]가 값을 3 을 포함할 거라고 예상할 것이다. 그런데 왜 그렇지 않을까? 개념적으로 이것은 x[i] += 1 이 x[i] = x[i] + 1 의 축약형을 의미하기 때문이다.
X[i] + 1 이 평가되고 나면 결과가 x 의 인덱스에 할당된다. 이 점을 생각하면 그것은 여러차례 일어나는 증가가 아니라 할당이므로 보기와는 다른 결과를 가져온다.

그렇다면 연산이 반복되는 곳에 다른 행동을 원한다면 어떻게 될까? 이러한 경우에는 유니버설 함수의 at() 메서드(NumPy 1.8 부터)를 사용해 다음과 같이 하면된다.

```
In [22]:x = np.zeros(10)
         np.add.at(x, i, 1)
         print(x)
[ 0.  0.  1.  2.  3.  0.  0.  0.  0.  0.]
```

at()메서드는 지정한 값(여기서는 1)을 가진 특정 인덱스(여기서는 I)에 해당 연산자를 즉시 적용한다. 이것과 개념상 비슷한 다른 메서드로는 유니버설 함수의 reduceat()메서드가 있다.

예제: 데이터 구간화¶

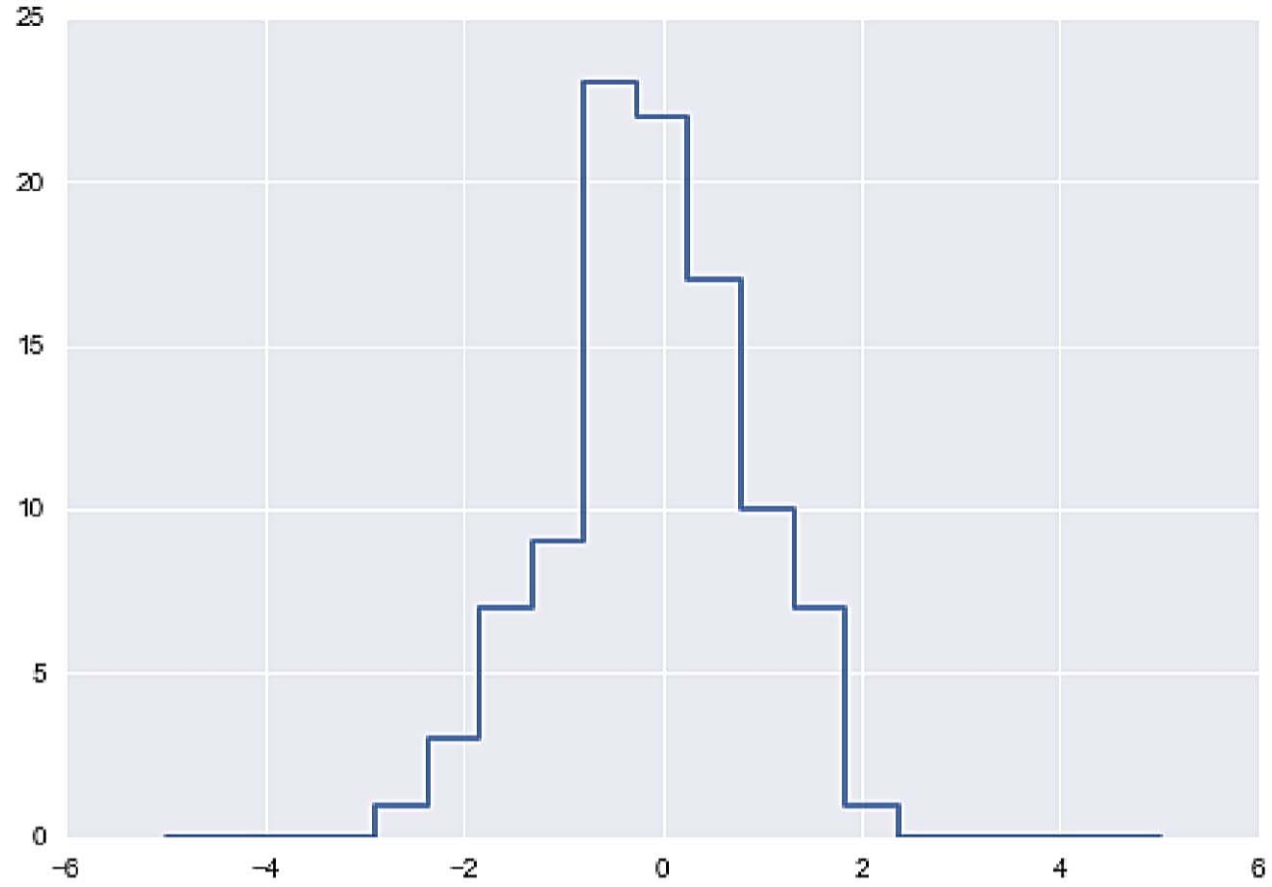
이 아이디어를 이용하면 데이터를 효율적으로 구간화해서 직접 히토그램을 생성할 수 있다. 가령 1/1000 개 값이 있고 그 값들이 구간 배열에서 어디에 속하는 지 빠르게 찾고 싶다고 하자. 다음과 같이 ufunc.at 을 이용해 그것을 계산할 수 있다..

```
In [23]:np.random.seed(42)
         x = np.random.randn(100)
         # 직접 히스토그램 계산하기
         bins = np.linspace(-5, 5, 20)
         counts = np.zeros_like(bins)
# 각 x 에 대한 적절한 구간 찾기
i = np.searchsorted(bins, x)

# 각 구간에 1 더하기
np.add.at(counts, i, 1)
```

이제 집계값인 counts 는 각 구간 내에 포함된 점의 개수, 즉 히스토그램을 나타낸다.

```
In [24]:# 결과 프로팅하기
plt.plot(bins, counts, linestyle='steps');
```



물론 히스토그램을 그릴 때마다 이 과정을 직접 수행하는 것은 바보 같은 짓이다. Matplotlib 이 한줄로 동일한 결과를 내는 plt.hist()루틴을 제공하는 이유가 바로 여기에 있다.

```
In [25]:print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)

        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
NumPy routine:
10000 loops, best of 3: 97.6 µs per loop
Custom routine:
10000 loops, best of 3: 19.5 µs per loop
```

직접 작성한 한 줄짜리 알고리즘이 NumPy 의 최적화된 알고리즘보다 몇 배 더 빠르다. 어떻게 이것이 가능할까? Np.histogram 소스코드를 들여다보면(IPython 에서 np.histgram??을 입력하면 소스코드를 확인할 수 있다) 루틴이 간단히 검색하고 세는 것보다 더 많은 일을 한다는 것을 알 수 있다. 그 이유는 NumPy 알고리즘이 더 유연하고, 특히 데이터 포인트 개수가 많아질 때 성능이 좋아지도록 설계돼 있기 때문이다.

```
In [26]:x = np.random.randn(1000000)
        print("NumPy routine:")
        %timeit counts, edges = np.histogram(x, bins)
        print("Custom routine:")
        %timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

NumPy routine:
10 loops, best of 3: 68.7 ms per loop
Custom routine:
10 loops, best of 3: 135 ms per loop

이 비교는 알고리즘 효율성은 결코 간단한 문제가 아님을 보여준다. 대규모 데이터에서 효율적인 알고리즘이 소규모 데이터에서는 최선의 방식이 아닐 수 있으며 그 반대도 마찬가지다. 하지만 이 알고리즘을 직접 코딩하는 것의 장점은 이러한 기본적인 메서드를 이해하면 그러한 기본 구성요소를 이용해 매우 ㅎㅇ미로운 사용자 정의 행위를 하도록 확장할 수 있다는 점이다. 데이터 집약적인 애플리케이션에서 파이썬을 효율적으로 사용하려면 np.histogram 같이 편리한 일번적인 루틴과 언제 그것들이 필요한지 아는 것은 물론이고 좀 더 세밀한 행위가 필요할 때 저수준 기능을 사용하는 방법을 알아야 한다.