누락된 데이터 처리하기

많은 튜토리얼에서 보는 데이터와 현실 세계 데이터의 차이점은 현실 세계의 데이터는 깨끗하거나 형태가 단일한 경우가 드물다는 것이다. 특히 흥미로운 데이터세트는 데이터가 어느 정도 누락돼 있는 경우가 많다. 더 골치 아픈 것은 데이터 소스가 다르면 전혀 다른 방식으로 데이터가 누락됐다는 뜻일 수도 있다는 데 있다.

이번 절에서는 누락된 데이터에 대해 일반적으로 고려할 사항 몇가지를 논의하고 Pandas 가 그것을 어떻게 표현하는지 알아보고 누락 데이터를 처리하는 파이썬의 내장된 Pandas 도구 몇가지 설명하려고 한다. 누락된 데이터를 null 이나 NaN, NA 값으로 나타낸다.

누락된 데이터 처리 방식의 트레이드오프

표나 DataFrame 의 누락된 데이터 존재를 나타내기 위해 여러가지 방식이 개발됐다. 일반적으로 그 방식은 누락된 값을 전체적으로 가리키는 마스크를 사용하거나 누락된 항목 하나를 가리키는 센티널 값을 선택하는 두 전략 중 하나를 중심으로 한다.

마스킹 방식에서 마스크는 완전히 별개의 부울 배열일 수도 있고 지역적으로 값의 널 상태를 가리키기 위해 데이터 표현에서 1 비트를 전용으로 사용할 수도 있다.

센티널 방식에서 샌티널 값은 누락된 정숫값을 -9999 나 보기 드문 비트 패턴으로 표시하는 등 데이터에 특화된 표시법일 수도 있고 누락된 부동 소수점 값을 IEEE 부동 소수점 표준을 따르는 특수 값인 NaN(Not a Number)으로 표시하는 것과 같은 좀 더 일반적인 표시법일 수도 있다.

이 방식들은 모두 장단점이 있다. 별도의 마스크 배열을 사용하면 추가적인 부울 배열 할당이 필요한데, 이는 스토리지와 연산에 별도의 (대체로 최적화되지 않은) 로직이 필요할 수도 있다. NaN 과 같은 보편적인 특수 값은 모든 데이터 타입에서 사용할 수 있는 것은 아니다.

대부분의 경우 모두를 만족하는 답이 존재하지 않듯이 언어와 시스템에 따라 다른 규칙을 사용한다. 예를 들어, R 언어는 누락된 데이터를 가리키는 센티널 값으로 각 데이터 타입에 예약된 비트 패턴을 사용하고, SciDB 시스템은 NA 상태를 나타내기 위해 모든 셀에 추가 바이트를 더해 사용한다.

Pandas 에서 누락된 데이터

Pandas 에서 누락된 값을 처리하는 방식은 Pandas 의 기반이 되는 NumPy 패키지가 부동 소수점이 아닌 다른 데이터 타입에는 NA 값 표기법이 기본으로 없다는 사실로 인해 제약을 받는다.

Pandas 가 값없음을 표시하기 위해 각 데이터 타입에 비트 패턴을 지정하는 R 의 방식을 따를 수 있었지만, 이 방식은 다루기가 까다롭다. R 은 네개의 기본 데이터 타입을 가지고 있는 반면, NumPy 는 그보다 훨씬 더 많은 데이터 타입을 가지고 있기 때문이다. 예를 들어 R 에는 정수형 하나지만 NumPy 는 인코딩 방식에 있어 정밀도, 부호, 엔디언까지 고려하면 무려 열 네가지의 기본 정수형을 지원한다. NumPy 의 모든 데이터 타입에 대해 특정 비트 패턴을 예약하게 되면 다양한 데이터 타입에 대한 여러 가지 연산으로 많은 오버헤드가 발생해서 이를 해결하기 위해 새로운 유형의 NumPy 패키지가 필요할 수도 있다. 게다가 작은 데이터 타입(8 비트 정수 같은)에서 마스크로 사용하기 위해 1 비트를 별도로 뺀다면 표현할 수 있는 값의 범위가 상당히 줄어든다.

NumPy 는 마스킹된 배열, 즉 '좋음' 또는 '나쁨'으로 데이터를 마스킹하기 위해 부착된 별도의 부울 마스크배열을 가진 배열을 지원한다. Pandas 가 이로부터 파생됐을 수는 있지만 저장과 연산, 코드 유지보수에서의 오버헤드는 그 방법의 매력을 떨어뜨린다.

이같은 제약을 고려해 Pandas 는 누락된 데이터에 대해 썬티널을 사용하고, 아울러 기존의 두 가지 파이써 널값인 특수 부동 소수점 NaN 값과 파이썬 None 객체를 사용한다. 이 방식은 몇가지 부작용이 있지만 실무에서 이해관계가 충돌하는 대부분의 경우에 좋은 절충안이 된다.

None: 파이썬의 누락된 데이터

Pandas 가 사용한 첫번째 센티널 값은 None 이다. 이는 파이썬의 싱글턴 객체로, 종종 파이썬 코드에서 누락된 데이터를 위해 사용된다. None 은 파이썬 객체이므로 임의의 NumPy/Pandas 배열에서 사용할 수 없고 데이터 타입이 'object'인 배열(즉, 파이썬 객체의 배열)에서만 사용할 수 있다.

```
In [1]: import numpy as np
        import pandas as pd
In [2]: vals1 = np.array([1, None, 3, 4])
        vals1
Out[2]: array([1, None, 3, 4], dtype=object)
```

이 코드에서 `dtype=object` 는 NumPy 가 배열의 내용에 대해 추론할 수 있는 가장 일반적인 타입 표현이 파이썬 객체라는 것을 의미한다. 이러한 객체 배열이 몇가지 목적에서는 유용하지만, 데이터에 대한 연산은 파이썬 수준에서 이뤄지며 기본 데이터 타입의 배열에서 볼 수 있는 전형적으로 빠른 연산보다 훨씬 더 많은 오버헤드가 발생한다.

```
In [3]: for dtype in ['object', 'int']:
            print("dtype =", dtype)
            %timeit np.arange(1E6, dtype=dtype).sum()
        print()
        dtype = object
        10 loops, best of 3: 78.2 ms per loop
        dtype = int
        100 loops, best of 3: 3.06 ms per loop
```

배열에서 파이썬 객체를 사용한다는 것은 None 값을 가진 배열에서 sum()이나 min()같은 집계 연산을 하면 일반적으로 오류가 발생할 것이라는 뜻이기도 하다.

```
In [4]:vals1.sum()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()

/Users/jakevdp/anaconda/lib/python3.5/site-packages/numpy/core/_methods.py      in
_sum(a, axis, dtype, out, keepdims)
     30
     31 def _sum(a, axis=None, dtype=None, out=None, keepdims=False):
---> 32     return umr_sum(a, axis, dtype, out, keepdims)
     33
     34 def _prod(a, axis=None, dtype=None, out=None, keepdims=False):

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
```

이 코드는 정수와 None 의 덧셈이 정의돼 있지 않다는 사실을 보여준다.

NaN: 누락된 숫자 데이터

다른 누락된 데이터 표현인 NaN(Not a Number)은 다르다. 이것은 표준 IEEE 부동 소수점 표기를 사용하는 모든 시스템이 인식하는 특수 부동 소수점이다.

```
In [5]:vals2 = np.array([1, np.nan, 3, 4])
       vals2.dtype
Out[5]:dtype('float64')
```

NumPy 가 이 배열에 대해 기본 부동 소수점 타입을 선택했다는 사실에 주목하자. 이 말은 곧 앞에서 본 객체 배열과는 달리 이 배열은 컴파일된 코드에 삽입된 빠른 연산을 지원한다는 뜻이다. NaN 은 데이터 바이러스와 약간 비슷하다는 사실을 알아야한다. 말하자면 접촉한 모든 객체를 감염시킨다. 어떤 연산이든 상관 없이 NaN 이 포함된 산술 연산의 결과는 또 다른 NaN 이다.

```
In [6]:1 + np.nan
Out[6]:nan
In [7]:0 *  np.nan
Out[7]:nan
```

값을 집계하는 방법은 잘 정의돼 있지만(즉, 오류를 내지 않는다) 그것은 항상 유용하지는 않다.

```
In [8]:vals2.sum(), vals2.min(), vals2.max()
Out[8]:(nan, nan, nan)
```

NumPy 는 이 누락된 값을 무시하는 몇가지 특별한 집계 연산을 제공한다.

```
In [9]:np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[9]:(8.0, 1.0, 4.0)
```

특히 NaN 은 부동 소수점 값이라는 것을 유념하자. 정수나 문자열 등 다른 타입에는 NaN 에 해당하는 값이 없다.

Pandas 에서 NaN 과 None

NaN 과 None 은 각자가 맡은 역할이 있으며 Pandas 는 이 둘을 거의 호환성 있게 처리하고 적절한 경우에는 서로 변환할 수 있게 했다:

```
In [10]:pd.Series([1, np.nan, 2, None])
Out[10]:0    1.0
        1    NaN
        2    2.0
        3    NaN
        dtype: float64
```

사용할 수 있는 센티널 값이 없는 타입의 경우, Na 값이 있으면 Pandas 가 자동으로 타입을 반환한다. 가령 정수 배열의 값을 np.nan 으로 설정하면 NA 를 수용할 수 있도록 부동 소숫점 타입으로 자동 상황 변환한다.

```
In [11]:x = pd.Series(range(2), dtype=int)
        x
Out[11]:0    0
        1    1
        dtype: int64
In [12]:x[0] = None
        x
Out[12]:0    NaN
        1    1.0
        dtype: float64
```

Pandas 는 정수 배열을 부동 소수점으로 변환하는 것 외에도 자동으로 None 을 NaN 값으로 변환한다.

이런 유형의 매직 함수가 R 과 같이 영역 특화된 언어에서 NA 값에 접근하는 단일 방식에 비해 다소 독창적이라고 느낄 수 있지만, 사실상 Pandas 의 센티널/타입 변환 방식은 상당히 잘 동작하며 개인적 경험을 바탕으로 했을 때 문제가 거의 발생하지 않는다.

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Pandas 에서 문자열 데이터는 항상 object dtype 으로 저장된다는 사실을 기억하자.

Null 값 연산하기

Pandas 는 None 과 NaN 을 근본적으로 누락된 값이나 널 값을 가리키기 위해 호환되는 값으로 처리한다. 이 규칙을 이용할 수 있게 Pandas 데이터 구조의 널 값을 감지하고 삭제하고 대체하는 몇가지 유용한 메서드가 있다.:

- `isnull()`: 누락 값을 가리키는 부울 마스크를 생성
- `notnull()`: isnull()의 역
- `dropna()`: 데이터에 필터를 적용한 버전을 반환
- `fillna()`: 누락 값을 채우거나 전가된 데이터 사본을 반환

null 값 탐지

Pandas 데이터 구조에는 널 데이터를 탐지하기 위한 메서드로 isnull()과 notnull()의 두가지가 있다. 둘 다 데이터에 대한 부울 마스크를 반환한다. :

```
In [13]: data = pd.Series([1, np.nan, 'hello', None])
In [14]: data.isnull()
Out[14]: 0    False
         1     True
         2    False
```

```
3     True
dtype: bool
```

부울 마스크는 Series 나 DataFrame 인덱스로 직접 사용될 수 있다.

```
In [15]: data[data.notnull()]
Out[15]: 0        1
         2    hello
         dtype: object
```

isnull()과 notnull() 메서드는 `DataFrames` 와 비슷한 결과를 만들어 낸다.

null 값 제거하기

앞에서 사용했던 마스킹 외에도 편리하게 사용할 수 있 는 dropna()(NA 값을 제거하기)와 fillna()(NA 값 채우기)메서드가 있다. Series 의 경우, 그 결과는 간단하다.:

```
In [16]: data.dropna()
Out[16]: 0        1
         2    hello
         dtype: object
```

`DataFrame` 의 경우에는 더 다양한 방식이 있다. 다음의 DataFrame 을 생각해 보자.

```
In [17]: df = pd.DataFrame([[1,      np.nan, 2],
                            [2,      3,      5],
                            [np.nan, 4,      6]])
         df
```

Out[17]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

DataFrame 에서는 단일 값만 삭제할 수 없으며, 전체 행이나 전체 열을 삭제하는 것만 가능하다. 적용분야에 따라 어느 하나의 방식이 필요하기 때문에 dropna()는 DataFrame 에 대한 다양한 옵션을 제공한다.

기본적으로 dropna()는 널 값이 있는 모든 행을 삭제할 것이다.

```
In [18]: df.dropna()
```

Out[18]:

	0	1	2
1	2.0	3.0	5

또 다른 방식으로 다른 축에 따라 NA 값을 삭제할 수 있다. axis= 1 은 널 값을 포함하는 모든 열을 삭제한다.:

```
In [19]: df.dropna(axis='columns')
```

Out[19]:

	2
0	2
1	5
2	6

하지만 이 방식은 일부 유효한 데이터도 삭제한다. 모두 NA 값으로 채워져 있거나 NA 값으로 채워져 있거나 NA 값이 대부분을 차지하는 행이나 열을 삭제하고 싶을 때도 있을 것이다. 이것은 how 나 thresh 매개변수를 통해 지정할 수 있는데 이 매개변수가 통과할 수 있는 널 값의 개수를 세밀하게 조절하게 해준다.

기본 설정값은 how = 'any'로, 널값을 포함하는 행이나 열(axis 키워드에 따라 정해짐)을 모두 삭제한다. 또한 how='all'로 지정해 모두 널 값인 행이나 열만 삭제할 수 있다.

```
In [20]: df[3] = np.nan
         df
```

Out[20]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
In [21]: df.dropna(axis='columns', how='all')
```

Out[21]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

좀 더 세부적으로 제어하기 위해 thresh 매개변수로 행이나 열에서 널이 아닌 값이 최소 몇개가 있어야 하는지 지정할 수 있다.

```
In [22]: df.dropna(axis='rows', thresh=3)
```

Out[22]:	0	1	2	3
1	2.0	3.0	5	NaN

여기서는 첫번째와 마지막 행이 삭제되는데, 거기에서 단 두 개의 값만이 널 값이 아니기 때문이다..

null 값 채우기

때때로 NA 값을 삭제하지 않고 유효한 값으로 대체해야 할 때도 있다. 그 값은 0 과 같은 단일 숫자일 수도 있고 유효한 값으로부터 전가 혹은 보간된 값일 수도 있다. isnull()메서드를 마스크로 직접 사용할 수도 있지만, Pandas 는 이러한 연산을 위해 널 값을 대체한 배열의 사본을 변환하는 fillna()메서드를 제공한다.

다음 Series 를 보자:

```
In [23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
         data
Out[23]: a    1.0
         b    NaN
         c    2.0
         d    NaN
         e    3.0
         dtype: float64
```

0 과 같은 단일 값으로 NA 항목을 채울 수 있다.

```
In [24]: data.fillna(0)
Out[24]: a    1.0
         b    0.0
         c    2.0
         d    0.0
         e    3.0
         dtype: float64
```

이전값으로 채우도록 지정할 수도 있다.

```
In [25]: # forward-fill (이전 값으로 채우기)
         data.fillna(method='ffill')
Out[25]: a    1.0
```

```
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

또는 뒤에 있는 값을 앞으로 전달하도록 지정할 수 있다.

```
In [26]: # back-fill(다음에 오는 값으로 채우기)
         data.fillna(method='bfill')
Out[26]: a    1.0
         b    2.0
         c    2.0
         d    3.0
         e    3.0
         dtype: float64
```

DataFrame 의 경우 옵션은 유사하지만, 값을 어느 축에 따라 채울것인지 `axis` 를 이용해 지정할 수 있다.

```
In [27]: df
```

Out[27]:	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

```
In [28]: df.fillna(method='ffill', axis=1)
```

Out[28]:	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

뒤의 값을 채울 때 이전 값을 사용할 수 없다면 NA 값은 그대로 남는다는 점을 알아두자.