문자열로 처리한 시리즈의 인덱스 검색은 실제 문자열에 정의된 인덱스에 따라 정수형으로 대괄호([ ]) 기호 안에 넣어 조회가 가능합니다.

```
In :  ss.str[0]

Out:  0    가
      1    라
      2   NaN
      3    사
      dtype: object

In :  ss.str[1]

Out:  0    -
      1    -
      2   NaN
      3    -
      dtype: object

In :  ss.str[2]

Out:  0    나
      1    마
      2   NaN
      3    아
      dtype: object
```


Chapter

3

판다스 시리즈와 데이터 프레임 실행하기

지금까지 시리즈와 데이터 프레임 클래스로 객체를 생성하고 객체의 구성 요소를 검색하는 방법을 알아보았습니다. 이번에는 시리즈와 데이터 프레임으로 연산자를 실행해보고 간단한 메소드들을 처리하는 방법을 익힙니다.

- 시리즈와 데이터 프레임 공통 메소드
- 연산자와 메소드 연산 이해하기
- 메소드를 이용한 다양한 처리 알아보기
- 누락 값 처리하기
- 범주형 데이터 처리하기

3.1 메소드

시리즈와 데이터 프레임 안에서 같은 메소드들을 많이 제공합니다. 계산된 결과가 시리즈와 데이터 프레임으로 나오는 것을 제외하면 같은 결과를 출력합니다.

먼저 어떤 메소드들이 같은지를 확인해보고 기본적인 연산을 실행해보겠습니다.

3.1.1 시리즈와 데이터 프레임의 동일 메소드 확인

시리즈와 데이터 프레임 안의 메소드를 확인해 같은 이름의 메소드를 확인합니다.

먼저 시리즈와 데이터 프레임이 갖는 속성과 메소드를 csv 파일로 저장합니다. 이 파일을 읽고 속성과 메소드를 분리해 처리합니다.

파이썬은 인스턴스 메소드를 클래스에서 보관합니다. 클래스에서 인스턴스 메소드를 단순한 함수로 인식합니다. 실제 인스턴스에서 메소드를 바인딩할 때 인스턴스 메소드로 변환되어 처리합니다.

■ 시리즈와 데이터 프레임에 있는 동일 메소드 확인

시리즈와 데이터 프레임에서 같은 이름으로 각 연산을 처리하는 메소드들이 있습니다. 이 메소드들에는 어떤 것이 있는지를 먼저 확인해봅니다.

[예제 3-1] 시리즈와 데이터 프레임 안의 동일 메소드 확인하기

시리즈와 데이터 프레임에 있는 속성과 메소드의 정보를 모아둔 csv 파일을 읽습니다. 이 파일은 기본적으로 영어로만 되어 있어 인코딩을 하지 않아도 데이터 프레임으로 전환이 됩니다.

```
In :  op = pd.read_csv('../data/series_dataframe.csv')
```

```
In :  op.head()
```

Out:

	Member	Series	DataFrame
0	T	T	T
1	abs	abs	abs
2	add	add	add
3	add_prefix	add_prefix	add_prefix
4	add_suffix	add_suffix	add_suffix

데이터 프레임을 읽어오면 2차원 데이터인지를 확인하기 위해 .shape 속성을 확인합니다.

```
In :  op.shape
```

```
Out:  (264, 3)
```

이 데이터 프레임 중 하나의 열은 Series에 대한 정보이고 다른 열은 DataFrame입니다. 이 두 열을 == 연산자로 비교하면 두 열의 각 원소별로 비교한 결과를 새로운 시리즈 객체에 불리언 값으로 반환합니다.

데이터 프레임 안의 .loc 속성으로 시리즈와 데이터 프레임 안의 동일한 이름을 가진 멤버를 검색해서 op_eq 변수에 할당합니다.

```
In :  (op['Series'] == op['DataFrame']).head()
```

```
Out:  0    True
      1    True
      2    True
      3    True
      4    True
      dtype: bool
```

```
In :  op_eq = op.loc[(op['Series'] == op['DataFrame'])]
```

동일한 속성과 메소드는 총 190 정도가 되는 것을 알 수 있습니다.

```
In :  op_eq.shape
```

```
Out:  (190, 3)
```


첫 번째 5개를 .head 메소드로 조회하면 같은 이름이 3개의 열에 다 있는 것을 알 수 있습니다.

```
In :  op_eq.head()
```

Out:

	Member	Series	DataFrame
0	T	T	T
1	abs	abs	abs
2	add	add	add
3	add_prefix	add_prefix	add_prefix
4	add_suffix	add_suffix	add_suffix

하위 5개를 .tail 메소드로 조회합니다.

```
In :  op_eq.tail()
```

Out:

	Member	Series	DataFrame
256	update	update	update
259	values	values	values
260	var	var	var
262	where	where	where
263	xs	xs	xs

동일한 멤버(member)가 들어 있는 하나의 열을 선택해서 리스트로 변환합니다. 이때 .tolist 메소드를 실행하고 이를 op_eq_list 변수에 할당합니다.

```
In :  op_eq_list = op_eq['Member'].tolist()
```

같은 멤버들 중에 함수를 찾아서 확인해봅니다. types 모듈을 임포트합니다.

```
In :  import types
```

시리즈 클래스의 네임스페이스인 __dict__ 안의 속성을 확인하기 위해 문자열을 전달합니다. 매칭되는 객체를 찾아 types.FunctionType과 비교해서 True이면 함수입니다.

파이선 클래스에 있는 인스턴스 메소드는 클래스에 있을 때는 함수로 체크되고 인스턴스 객체와 바인딩(binging)될 때 인스턴스 메소드로 변환됩니다.

```
In :  count = 0
      for i in op_eq_list :
          try :

              if type(pd.Series.__dict__[i])  ==  types.FunctionType :
                  print(i, end=',  ')
                  count += 1
              if count % 7 == 0 :
                  print()
          except Exception as e :
              pass
```

시리즈 내부 속성을 제외한 함수인 인스턴스 메소드의 이름이 출력됩니다. 이 책에 이 메소드들을 가지고 많은 예제를 다루겠습니다.

```
Out:  add,  agg,  aggregate,  align,  all,  any,  append,
      apply,  combine,  combine_first,  compound,  corr,  count,  cov,
      cummax,  cummin,  cumprod,  cumsum,  diff,  div,  divide,
      dot,  drop_duplicates,  dropna,  duplicated,  eq,  ewm,  expanding,
      fillna,  first_valid_index,  floordiv,  ge,  get_value,  get_values,  gt,
      hist,  idxmax,  idxmin,  isin,  isna,  isnull,  keys,
      kurt,  kurtosis,  last_valid_index,  le,  lt,  mad,  max,
      mean,  median,  memory_usage,  min,  mod,  mode,  mul,
      multiply,  ne,  nlargest,  notna,  notnull,  nsmallest,  pow,
      prod,  product,  quantile,  radd,  rdiv,  reindex,  reindex_axis,
      rename,  reorder_levels,  reset_index,  rfloordiv,  rmod,  rmul,  rolling,
      round,  rpow,  rsub,  rtruediv,  sem,  set_value,  shift,
      skew,  sort_index,  sort_values,  sortlevel,  std,  sub,  subtract,
      sum,  swaplevel,  to_csv,  to_dict,  to_excel,  to_period,  to_sparse,
      to_string,  to_timestamp,  transform,  truediv,  unstack,  update,  var,
```

3.1.2 연산자와 메소드 처리

앞 절에서 판다스의 시리즈와 데이터 프레임의 공통된 메소드들의 이름을 알아보았습니다. 이제 이 메소드를 가지고 메소드의 기본적인 처리를 알아봅니다.

먼저 연산자를 처리하고 연산자에 해당하는 메소드를 추가해서 제공합니다. 연산자 계산과 연산자 대용으로 만든 메소드 처리 방법을 비교해보겠습니다.

■ 연산자와 메소드 처리

연산자나 메소드를 이용할 때는 시리즈나 데이터 프레임의 열의 자료형에 맞는 계산을 합니다.

배열 처리를 위해 넘파이 모듈에서 빌려와서 벡터화 연산을 수행하므로 별도의 순환문 처리가 필요 없습니다.

[예제 3-2] 연산자와 메소드 계산하기

대학등록금 파일을 읽어와서 college 변수에 할당합니다. 이 데이터 프레임의 모양을 .shape로 확인합니다.

```
In :  college = pd.read_csv('../data/2017_college_tuition.csv',encoding='euc-kr')
```

```
In :  college.shape
```

```
Out:  (196, 3)
```

이 파일을 .head 메소드로 조회하면 평균등록금이 숫자 자료형입니다.

```
In :  college.shape
```

Out:

	학교명	설립구분	평균등록금
0	강릉원주대학교	국공립	4,262
1	강원대학교	국공립	4,116
2	경남과학기술대학교	국공립	3,771
3	경북대학교	국공립	4,351
4	경상대학교	국공립	3,967

```
In :  college.dtypes
```

```
Out:  학교명          object
      설립구분        object
      평균등록금       int64
      dtype: object
```

이 데이터 프레임에 특정 정수를 더하면 어떻게 될까요? 예외가 발생됩니다. 파이썬 문자열은 문자열 간의 덧셈을 지원하지만 정수와의 덧셈을 처리할 수 없습니다.

스칼라인 정수가 먼저 브로드캐스팅이 되지만 문자열과 정수 덧셈이 발생하면 처리할 수 없어 예외가 발생합니다.

```
In :
try :
    college + 5
except Exception as e :
    print(e)
```

```
Out:  Could not operate 5 with block values must be str, not int
```

평균등록금 열만 정수이므로 단위를 맞추기 위해 여기에 1000을 곱했습니다. 새로운 시리즈가 발생합니다. 인덱스 검색에 평균등록금 열 이름 넣고 할당 연산자를 통해 갱신된 것을 할당하면 기존 데이터 프레임 안의 평균등록금 열이 갱신된 것을 알 수 있습니다.

```
In : college['평균등록금'] = college['평균등록금'] * 1000
```

```
In : college.head()
```

Out:

	학교명	설립구분	평균등록금
0	강릉원주대학교	국공립	4262000
1	강원대학교	국공립	4116000
2	경남과학기술대학교	국공립	3771000
3	경북대학교	국공립	4351000
4	경상대학교	국공립	3967000

연산자로 처리한 부분과 이 연산자에 해당하는 메소드를 실행한 결과를 확인하면 같은 결과가 나옵니다.

```
In : (college['평균등록금'] + college['평균등록금']).head()
```

```
Out:  0    8524000
      1    8232000
      2    7542000
      3    8702000
      4    7934000
      Name: 평균등록금, dtype: int64
```

```
In : college['평균등록금'].add(college['평균등록금']).head()
```



```
Out:  0    8524000
      1    8232000
      2    7542000
      3    8702000
      4    7934000
      Name: 평균등록금, dtype: int64
```

어떻게 연산자와 이에 상응하는 메소드를 다 처리할 수 있는지를 살펴봅시다. .add를 help 함수로 조회하면 .add 메소드 내부에 여러 개의 매개변수가 있는 것을 알 수 있습니다.

```
In : help(college['평균등록금'].add)
```

```
Out:  Help on method add in module pandas.core.ops:

      add(other, level=None, fill_value=None, axis=0) method of pandas.core.series.Series
      instance
          Addition of series and other, element-wise (binary operator `add`).

          Equivalent to ``series + other``, but with support to substitute a fill_value for
          missing data in one of the inputs.

          Parameters
          ----------
          other : Series or scalar value
          fill_value : None or float value, default None (NaN)
              Fill missing (NaN) values with this value. If both Series are
              missing, the result will be missing
          level : int or name
              Broadcast across a level, matching Index values on the
              passed MultiIndex level

          Returns
          -------
          result : Series

          See also
          --------
          Series.radd
```

3.1.3 데이터 프레임 내의 열의 자료형을 기준으로 데이터 불러오기

데이터 분석을 처리할 때는 행보다 열 기준으로 데이터를 처리하는 경우가 더 많습니다. 판다스에서 데이터 프레임을 처리할 때 기본으로 데이터 프레임의 열을 쉽게 읽는 것을 선택

연산자(operator)에 정의했습니다. 데이터 프레임의 각 열은 시리즈로 구성되고 이 열로 데이터 자료형을 구분해 관리합니다. 또한 특정 자료형으로 검색해 실제 열 단위로 가져와서 데이터 프레임을 재구성합니다.

■ 자료형별로 데이터 검색

데이터 프레임의 열이 많을 경우 실제 계산을 하기 위해서는 계산에 필요한 열을 별도로 구분해서 처리해야 합니다. 이때 어떤 자료형으로 구성되었는지를 메소드를 이용해서 구분하여 처리할 수 있습니다.

[예제 3-3] 특정 자료형을 가지고 열을 구분하기

위에 파일로 읽어온 대학등록금에 대한 데이터 프레임의 각 열로 자료형을 조회하면 두 개는 object이고 하나가 int64로 구성되어 있는 것을 볼 수 있습니다.

```
In : college.dtypes
```

```
Out: 학교명        object
     설립구분      object
     평균등록금     int64
     dtype: object
```

이 대학등록금의 데이터 프레임 자료형을 get_dtype_counts 메소드로 읽어오면 각 자료형으로 만들어진 열의 개수를 확인할 수 있습니다.

```
In : college.get_dtype_counts()
```

```
Out: int64     1
     object    2
     dtype: int64
```

이번에는 데이터 프레임 안의 특정 열을 특정 자료형으로 읽어오는 방법이 있습니다. 특정 데이터 자료형을 추출하기 위해 select_dtypes 메소드의 include 매개변수에 데이터 자료형을 넣고 조회하면 새로운 데이터 프레임이 만들어집니다.

```
In : college.select_dtypes(include=['int64']).head()
```


Out:

	평균등록금
0	4262000
1	4116000
2	3771000
3	4351000
4	3967000

또한 전체 숫자 자료형으로 구성된 열을 가져오려면 number 문자열이나, np.number를 매개변수에 넣고 조회하면 됩니다.

```
In : college.select_dtypes(include=['number']).head()
```

Out:

	평균등록금
0	4262000
1	4116000
2	3771000
3	4351000
4	3967000

문자열로 구성된 열도 object 문자열이나 np.object를 전달하면 열을 전부 검색해 새로운 데이터 프레임을 만드는 것을 알 수 있습니다.

```
In : college.select_dtypes(include=['object']).head()
```

Out:

	학교명	설립구분
0	강릉원주대학교	국공립
1	강원대학교	국공립
2	경남과학기술대학교	국공립
3	경북대학교	국공립
4	경상대학교	국공립

3.1.4 시리즈와 데이터 프레임 내의 자료값 확인

시리즈와 데이터 프레임 클래스로 생성된 객체 인스턴스에는 다양한 값들이 들어가 있습니다. 이 값들에 대한 범주 등을 알아봐야 합니다.

실제 열에 구성된 값들의 빈도나 순위에 대해 알아보면 실제 값의 구성에 대한 기본적인 정보를 알 수 있습니다.

■시리즈 안의 값에 대한 확인

먼저 시리즈 안의 자료값이 공통으로 갖는 개수에 대해 .value_counts 메소드로 확인할 수 있습니다.

[예제 3-4] 시리즈 안의 자료값 확인하기

이번 예제에서 시리즈의 각 원소의 값을 빈도로 표시하기 위해서는 .value_counts 메소드를 사용합니다. 빈도가 레이블로 변하고 그 빈도 수가 값으로 계산된 시리즈 객체를 결과 값으로 반환합니다. 그중에서 상위 5개만 보기 위해 .head 메소드를 이용해서 출력해보았습니다.

```
In :  college['평균등록금'].value_counts().head()

Out:  3424000    2
      7161000    2
      6803000    1
      7535000    1
      6904000    1
      Name: 평균등록금, dtype: int64
```

시리즈의 값들에 대한 빈도를 퍼센트로 표시할 수 있는 normalize 매개변수도 지원합니다.

```
In :  college['평균등록금'].value_counts(normalize=True).head()

Out:  7161    0.010204
      3424    0.010204
      4351    0.005102
      7756    0.005102
      2390    0.005102
      Name: 평균등록금, dtype: float64
```

문자열로 들어온 데이터를 기준으로 .value_counts 메소드를 적용하면 실제 범주 값이 나오는 것을 볼 수 있습니다. 결과를 보면 한국에는 주로 사립대학 형태가 많음을 알 수 있습니다.


```
In :  college['설립구분'].value_counts().head()

Out:  사립      156
      국공립     40
      Name: 설립구분, dtype: int64
```

또한 실제 들어있는 값들이 중복되어 있는지 확인하기 위해 .duplicated 메소드로 실행하면 논리식으로 평가된 결과를 제공합니다.

```
In :  college['설립구분'].head()

Out:  0    국공립
      1    국공립
      2    국공립
      3    국공립
      4    국공립
      Name: 설립구분, dtype: object

In :  college['설립구분'].duplicated().head()

Out:  0    False
      1     True
      2     True
      3     True
      4     True
      Name: 설립구분, dtype: bool
```

유일한 값을 알고 싶을 때에는 .unique 메소드로 실행하면 numpy 배열 값으로 유일한 값을 반환합니다.

```
In :  college['설립구분'].unique()

Out:  array(['국공립', '사립'], dtype=object)
```

■내부의 값 중에 순서로 조회

내부 원소 값들의 순위를 확인해서 큰 것과 작은 것을 가지고 조회할 수 있습니다. 또한 정렬(alignment)을 통해 순차적으로 처리한 후에 조회해도 순서대로 처리됩니다.

[예제 3-5] 내부 구성 값의 크기 비교하기

대학등록금을 확인할 수 있는 평균등록금 열을 읽어서 .nlargest 메소드로 값이 비싼 5개를 출력하면 인덱스 레이블은 변하지 않고 평균등록금이 높은 5개를 출력합니다. 하위 5개를 보기 위해서는 .nsmallest 메소드로 호출해서 처리하면 됩니다. 특정 금액이 0인 곳이 있습니다.

```
In :  college['평균등록금'].nlargest(5)
```

```
Out:  173    9004000
      132    8824000
      128    8649000
      147    8526000
      146    8499000
      Name: 평균등록금, dtype: int64
```

```
In :  college['평균등록금'].nsmallest(5)
```

```
Out:  61          0
      157   1760000
      137   2000000
      20    2390000
      15    3024000
      Name: 평균등록금, dtype: int64
```

데이터 프레임에서 .nlargest, .nsmallest 메소드를 처리할 때에는 순위를 정수로 표시하고 문자열로 열의 이름을 인자로 전달해서 처리합니다. 처리된 결과가 평균등록금 열에 따라 처리됩니다. 한 대학의 등록금이 0으로 들어온 것을 볼 수 있습니다.

```
In :  college.nlargest(5,'평균등록금')
```

Out:

	학교명	설립구분	평균등록금
173	한국산업기술대학교	사립	9004000
132	연세대학교	사립	8824000
128	신한대학교	사립	8649000
147	이화여자대학교	사립	8526000
146	을지대학교	사립	8499000


```
In :  college.nsmallest(5,'평균등록금')
```

Out:

	학교명	설립구분	평균등록금
61	광주가톨릭대학교	사립	0
157	중앙승가대학교	사립	1760000
137	영산선학대학교	사립	2000000
20	서울시립대학교	국공립	2390000
15	부산교육대학교	국공립	3024000

3.1.5 시리즈에만 있는 메소드

시리즈는 1차원 배열이면서 데이터 프레임의 열을 구성하는 기본 단위이고, 시리즈는 행 단위로 처리하므로 행을 기준으로만 처리하는 전용 메소드를 지원합니다.

■ 시리즈 안의 메소드 확인

이번 예제를 통해 시리즈 내에서만 작동하는 메소드들이 무엇이 있는지를 알아봅니다.

[예제 3-6] 시리즈만 가지는 메소드 확인하기

파일을 읽어서 Series와 DataFrame의 열을 인덱스 검색으로 가져와 각 필드가 다른 것을 처리해봅니다. 또한 레이블을 기준으로 검색하는 loc 인덱서를 통해 행 단위로 추출합니다.

```
In :  op = pd.read_csv('../data/series_dataframe.csv')
```

```
In :  op_not = op.loc[(op['Series'] != op['DataFrame'])]
```

행 단위로 추출한 데이터 프레임을 확인하면 Series와 DataFrame에 각각 존재하지 않는 것은 누락 값으로 표시된 것을 알 수 있습니다.

```
In :  op_not.head()
```

Out:

	Member	Series	DataFrame
12	applymap	NaN	applymap
13	argmax	argmax	NaN
14	argmin	argmin	NaN
15	argsort	argsort	NaN
19	asobject	asobject	NaN

```
In :  op_not.shape
```

```
Out:  (74, 3)
```

Series 열에 있는 메소드를 확인하기 위해 누락 값이 있는 요소를 .drapna 메소드로 삭제합니다. Series를 검색한 후에 .tolist 메소드로 리스트로 변환합니다.

```
In :  op_ser = op_not['Series'].dropna()
```

```
In :  op_ser_list = op_ser.tolist()
```

파이썬에서 클래스의 네임스페이스(namespace)에 있는 함수를 확인해서 출력하면 시리즈에 있는 메소드를 알 수 있습니다.

```
In :  count = 0
      for i in op_ser_list :
          try :

              if type(pd.Series.__dict__[i])  ==  types.FunctionType :
                  print(i, end=',  ')
                  count += 1
                  if count % 5 == 0 :
                      print()
          except Exception as e :
              pass
```

```
Out:  argmax,  argmin,  argsort,  autocorr,  between,
      compress,  items,  iteritems,  map,  nonzero,
      ptp,  put,  ravel,  repeat,  reshape,
      searchsorted,  to_frame,  unique,  valid,  view,
```


3.1.6 데이터 프레임만 있는 메소드

판다스의 2차원 배열인 데이터 프레임에 맞는 처리를 위한 기능이 있는 메소드가 있습니다.

■ 데이터 프레임 내의 메소드 확인

앞 절에서 시리즈만 보유한 메소드를 확인한 파일을 공부했습니다. 이번에는 데이터 프레임에만 있는 메소드를 확인해봅니다.

[예제 3-7] 데이터 프레임만 가지는 메소드 확인하기

데이터 프레임의 열을 가져와서 누락 값을 .dropna 메소드로 삭제하고 이를 리스트로 변환합니다.

```
In :  op_df = op_not['DataFrame'].dropna()
```

```
In :  op_df_list = op_df.tolist()
```

시리즈에서 처리한 것과 같이 데이터 프레임이 가진 메소드를 처리합니다.

```
In :  ount = 0
      for i in op_ser_list :
          try :

              if type(pd.DataFrame.__dict__[i])  ==  types.FunctionType :
                  print(i, end=',  ')
                  count += 1
                  if count % 5 == 0 :
                      print()
          except Exception as e :
              pass
```

```
Out:  items,  iteritems,
```

3.2 인덱스 레이블과 누락 값 처리

판다스의 특징은 인덱스의 레이블을 가지고 행과 열로 검색해서 값을 갱신하고 처리한다는 것입니다. 또한 행과 열로 검색했을 때 내부의 빈 값들을 얻어낼 수도 있습니다.

판다스는 모든 데이터를 읽어오면 예외 처리를 하지 않기 위해 실제 값이 없는 경우에도 누락 값을 자동으로 만들어서 객체를 만드는데, 누락 값이 발생할 때 새로운 값으로 대체하거나 혹은 삭제해서 미완성된 데이터를 완성된 데이터로 만들어야 합니다. 이번 절에서는 누락 값이 생겼을 때 이를 다른 값으로 대체하거나 누락 값을 가진 행이나 열을 삭제해서 처리하는 방법에 대해 알아봅니다.

3.2.1 시리즈 인덱스 대체 및 변경

인덱스의 레이블을 만들 때 문자나 숫자일 경우 연속되지 않아도 됩니다. 이런 기준으로 레이블이 처리되므로 실제 연산을 할 때 레이블 매칭이 안 되어 이슈가 발생하기도 합니다.

■ 인덱스에 대한 정보 변경 및 대체

먼저 행의 레이블이 없는 인덱스를 정의해서 시리즈를 만든 후에 시리즈의 인덱스가 연산 등에 어떤 영향을 미치는지를 확인하고 행의 레이블을 변경하는 방법을 알아봅니다.

[예제 3-8] 인덱스 변경 및 대체하기

파이썬 리스트로 숫자만 받았고 index 매개변수에 아무것도 주지 않았습니다. 시리즈를 생성할 때 암묵적으로 행의 레이블이 만들어지는 것을 알 수 있습니다.

행의 레이블 이름을 바꿀 때는 .rename 메소드를 사용합니다. .rename 메소드의 인자에 딕셔너리(dict)의 기존 레이블과 변경 레이블을 넣어서 실행하면 실제 새로운 시리즈가 만들어집니다.

```
In :  s = pd.Series([1,2,3,4])

In :  s
```



```
Out:  0    1
      1    2
      2    3
      3    4
      dtype: int64

In :  s_re = s.rename({0:'a',1:'b',2:'c',3:'d'})

In :  s_re

Out:  a    1
      b    2
      c    3
      d    4
      dtype: int64
```

이번에는 인덱스의 이름을 바꾼 것으로 다시 인덱스를 변경할 수 있습니다. 이때는 .reindex 메소드를 가지고 기존에 만들어진 인덱스 위치나 일부 인덱스를 바꿀 수 있습니다.

.reindex 메소드로 레이블만 바꾸는 것이 아니라 전체 인덱스의 위치와 새로운 인덱스도 추가할 수 있습니다.

```
In :  s_re.reindex(['a','b','c','e'])

Out:  a    1.0
      b    2.0
      c    3.0
      e    NaN
      dtype: float64

In :  s_re.reindex(['a','b','c'])

Out:  a    1
      b    2
      c    3
      dtype: int64

In :  s_re.reindex(['c','b','a'])

Out:  c    3
      b    2
      a    1
      dtype: int64
```

이번에는 index 매개변수에 리스트를 지정해서 행의 인덱스를 만들어봅니다.

먼저 하나의 시리즈를 만들고 index 매개변수에 리스트 리터럴로 숫자를 지정했습니다. .index.dtype으로 확인하면 int64인 것을 알 수 있습니다.

```
In :  s1 = pd.Series([10,20,30], index=[0,1,2])

In :  s1

Out:  0    10
      1    20
      2    30
      dtype: int64

In :  s1.index.dtype

Out:  dtype('int64')
```

시리즈의 index 매개변수 list 생성자를 이용해서 문자열을 지정했습니다. .index.dtype으로 자료형을 확인하면 문자열이므로 판다스 자료형으로는 object 자료형입니다.

```
In :  s2 = pd.Series([1,2,3], index=list('012'))

In :  s2

Out:  0    1
      1    2
      2    3
      dtype: int64

In :  s2.index.dtype

Out:  dtype('O')
```

두 개의 시리즈를 더하면 두 시리즈의 인덱스가 같아 보이지만 서로 다른 자료형이기 때문에 일치되지 않고, 두 개의 시리즈의 길이와 같은 시리즈가 만들어집니다. 매핑되는 행의 레이블이 없으므로 전부 NaN으로 처리합니다.

문자열인 .index를 숫자 리스트로 대체하고 계산하면 두 시리즈의 .index 속성이 같아져 서로 매핑되어 연산이 되는 것을 알 수 있습니다.

```
In :  s1 + s2

Out:  0   NaN
      1   NaN
      2   NaN
      0   NaN
      1   NaN
      2   NaN
      dtype: float64

In :  s2.index = [0,1,2]

In :  s1 + s2

Out:  0    11
      1    22
      2    33
      dtype: int64
```

새로운 시리즈를 만들고 행의 레이블을 .rename 메소드를 이용해서 정수형으로 변경한 후에 덧셈 연산을 하면 실행되는 것을 알 수 있습니다.

```
In :  s3 = pd.Series([11,12,13], index=list('012'))

In :  s3 = s3.rename({'0':0, '1':1,'2':2})

In :  s1 + s3

Out:  0    21
      1    32
      2    43
      dtype: int64
```

새로운 시리즈를 만들어서 .index.values 속성에 가서 .astype 메소드로 int64로 변환한 후에 덧셈 연산을 처리해도 됩니다.

```
In :  s4 = pd.Series([11,12,13], index=list('012'))
```

```
In :  s4.index.values

Out:  array(['0', '1', '2'], dtype=object)

In :  s4.index = s4.index.values.astype('int64')

In :  s1 + s4

Out:  0    21
      1    32
      2    43
      dtype: int64
```

3.2.2 열 이름을 이용한 필터링

데이터 프레임에는 다양한 열이 있습니다. 이 열들의 이름이 초기 데이터일 때는 정제되지 않아 열의 이름이 잘 작성되어 있지 않을 수도 있습니다. 그러나 이런 열의 이름을 가지고 검색할 줄 알면 좋습니다.

특정 열에 대한 이름을 임의의 기준으로 필터링해서 가져와야 하는 경우 .filtering 메소드를 사용합니다.

정규표현식이나 특정 문자열의 포함 여부를 가지고 열을 추출하는 방법을 알아봅니다.

■ 데이터 프레임에 대한 열 정보 검색

데이터 프레임 안의 .filter 메소드를 이용해 like, regex 매개변수로 열의 이름을 확인한 후 그에 해당되는 것만 추출하여 새로운 데이터 프레임으로 만들어서 반환합니다.

[예제 3-9] 특정 열에 대한 정보를 가져오기

엑셀로 만들어져 있는 파일은 utf-8이나 euc-kr 인코딩 처리하면 예외가 발생할 수 있습니다. 이런 예외가 발생하면 마이크로소프트가 제공하는 cp949로 인코딩을 이용해 처리하면 됩니다.

```
In :  flight_route_info = pd.read_csv('../data/flight_route_info.csv',encoding='cp949')
```

데이터 프레임에 .columns 속성에서 관리하는 열의 정보를 확인합니다.


```
In :  flight_route_info.columns

Out:  Index(['노선명', '항공사', '출발지', '출발국', '도착지', '도착국', '검수', '2016.6', '2018.5',
             '상태', '비고', '운항 횟수', '운항_2016', '운항_2017', '운항_2018', '운항_2016.12',
             '운항_2017.1', '운항_2017.2', '운항_2017.12', '운항_2018.1', '운항_2018.2',
             '운항_2016겨울', '운항_2017겨울', '운항_겨울 총합', '운항_2017.3', '운항_2017.4',
             '운항_2017.5', '운항_2018.3', '운항_2018.4', '운항_2018.5', '운항_2017봄',
             '운항_2018봄', '운항_봄 총합', '운항_2016.6', '운항_2016.7', '운항_2016.8',
             '운항_2017.6', '운항_2017.7', '운항_2017.8', '운항_2016여름', '운항_2017여름',
             '운항_여름 총합', '운항_2016.9', '운항_2016.10', '운항_2016.11', '운항_2017.9',
             '운항_2017.10', '운항_2017.11', '운항_2016가을', '운항_2017가을', '운항_가을 총합', '운송량',
             '운송량_2016', '운송량_2017', '운송량_2018', '운송량_2016.12', '운송량_2017.1',
             '운송량_2017.2', '운송량_2017.12', '운송량_2018.1', '운송량_2018.2', '운송량_2016겨울',
             '운송량_2017겨울', '운송량_겨울 총합', '운송량_2017.3', '운송량_2017.4', '운송량_2017.5',
             '운송량_2018.3', '운송량_2018.4', '운송량_2018.5', '운송량_2017봄', '운송량_2018봄',
             '운송량_봄총합', '운송량_2016.6', '운송량_2016.7', '운송량_2016.8', '운송량_2017.6',
             '운송량_2017.7', '운송량_2017.8', '운송량_2016여름', '운송량_2017여름', '운송량_여름 총합',
             '운송량_2016.9', '운송량_2016.10', '운송량_2016.11', '운송량_2017.9', '운송량_2017.10',
             '운송량_2017.11', '운송량_2016가을', '운송량_2017가을', '운송량_가을 총합'],
            dtype='object')
```

열 이름 중에 특정 값이 들어간 것을 검색하려면 .filter 메소드를 이용합니다. 열의 이름 중에 특정 문자열이 있는 것을 like 매개변수를 이용해서 처리하면 이 메소드에 처리된 결과만 열을 가져옵니다.

```
In :  flight_route_info.filter(like='2016.6').head()
```

Out:

	2016.6	운항_2016.6	운송량_2016.6
0	0	4	692
1	0	4	692
2	0	4	596
3	0	0	0
4	0	5	985

.filter 메소드의 regex 매개변수는 정규표현식(regular expression)을 이용해서 열의 이름을 선택할 수 있습니다. 열 이름에 숫자가 들어간 것을 선택하면 새로운 데이터 프레임으로 반환합니다.

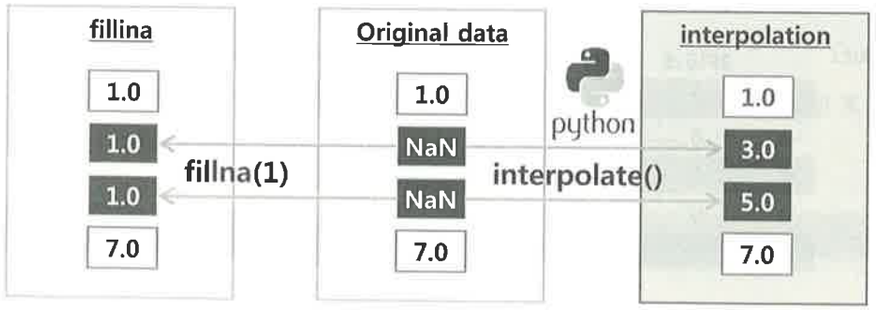
```
In :  flight_regex = flight_route_info.filter(regex='\d')

In :  flight_regex.columns
```

```
Out:  Index(['2016.6', '2018.5', '운항_2016', '운항_2017', '운항_2018', '운항_2016.12',
       '운항_2017.1', '운항_2017.2', '운항_2017.12', '운항_2018.1', '운항_2018.2',
       '운항_2016겨울', '운항_2017겨울', '운항_2017.3', '운항_2017.4', '운항_2017.5',
       '운항_2018.3', '운항_2018.4', '운항_2018.5', '운항_2017봄', '운항_2018봄',
       '운항_2016.6', '운항_2016.7', '운항_2016.8', '운항_2017.6', '운항_2017.7',
       '운항_2017.8', '운항_2016여름', '운항_2017여름', '운항_2016.9', '운항_2016.10',
       '운항_2016.11', '운항_2017.9', '운항_2017.10', '운항_2017.11', '운항_2016가을',
       '운항_2017가을', '운송량_2016', '운송량_2017', '운송량_2018', '운송량_2016.12',
       '운송량_2017.1', '운송량_2017.2', '운송량_2017.12', '운송량_2018.1', '운송량_2018.2',
       '운송량_2016겨울', '운송량_2017겨울', '운송량_2017.3', '운송량_2017.4', '운송량_2017.5',
       '운송량_2018.3', '운송량_2018.4', '운송량_2018.5', '운송량_2017봄', '운송량_2018봄',
       '운송량_2016.6', '운송량_2016.7', '운송량_2016.8', '운송량_2017.6', '운송량_2017.7',
       '운송량_2017.8', '운송량_2016여름', '운송량_2017여름', '운송량_2016.9', '운송량_2016.10',
       '운송량_2016.11', '운송량_2017.9', '운송량_2017.10', '운송량_2017.11', '운송량_2016가을',
       '운송량_2017가을'],
      dtype='object')
```

3.2.3 누락 값 변경

실제 데이터가 들어오지 않아서 생기는 누락 값을 특정 목적에 대해 바꿔야 합니다. 특정 값 하나로 열의 모든 누락 값을 동일하게 처리하는 방법과 그 열에 있는 값을 보간하는 방법이 있습니다.

두 개의 .fillna, .interpolate 메소드를 이용하는 방법을 알아봅니다.



[그림 3-1] 누락 값 처리

■누락 값 변경

시리즈나 데이터 프레임 안에 누락 값이 있을 때 일부 연산이나 메소드에서 누락 값을 제외해서 계산할 수 있습니다. 먼저 누락 값을 필요한 값으로 바꾸는 방법을 알아봅니다.

[예제 3-10] 누락 값 처리하기

시리즈를 만들어서 간단한 누락 값을 처리하기 위해 넘파이 모듈의 np.nan을 가지고 누락 값을 시리즈에 넣어 생성합니다.

```
In :  import numpy as np
```

```
In :  s = pd.Series([0, 1, np.nan, 3])
```

매개변수 인자로 ffill, pad를 주면 누락 값이 발생하기 전의 값으로 누락 값을 대체합니다.

```
In :  s.fillna(method='ffill')
```

```
Out:  0    0.0
      1    1.0
      2    1.0
      3    3.0
      dtype: float64
```

```
In :  s.fillna(method='pad')
```

```
Out:  0    0.0
      1    1.0
      2    1.0
      3    3.0
      dtype: float64
```

매개변수 인자로 bfill, backfill을 주면 누락 값이 발생한 후의 값을 가지고 누락 값을 대체합니다.

```
In :  s.fillna(method='bfill')
```

```
Out:  0    0.0
      1    1.0
      2    3.0
      3    3.0
      dtype: float64
```

```
In :  s.fillna(method='backfill')
```

```
Out:  0    0.0
      1    1.0
      2    3.0
      3    3.0
      dtype: float64
```

앞의 [예제 3-9]에서 읽은 파일을 가지고 실제 열별로 누락 값이 있는지를 .isnull 메소드로 실행하고 .sum 메소드로 처리하면 열로 누락 값의 개수를 알 수 있습니다.

```
In :  flight_route_info.isnull().sum().head(13)
```

```
Out:  노선명          0
      항공사          0
      출발지          0
      출발국          0
      도착지          0
      도착국          0
      검수            0
      2016.6          0
      2018.5          0
      상태            0
      비고          412
      운항 횟수       0
      운항_2016       0
      dtype: int64
```

데이터 프레임 전체의 누락 값을 알기 위해서는 .isnull.sum.sum 메소드를 연결해서 처리합니다. 그 결과 정수값으로 이 데이터 프레임 전체의 누락 값을 알려줍니다. 특정 열 '비고'의 값들의 빈도를 .value_counts 메소드로 알아볼 때 매개변수 dropna=False를 넣어 처리해야 누락 값의 빈도도 같이 나옵니다.

```
In :  flight_route_info.isnull().sum().sum()
```

```
Out:  412
```

```
In :  flight_route_info['비고'].value_counts(dropna=False).head()
```

```
Out:  NaN                        412
      2016.11.05. 이후 폐선        4
      2018.04.24. 이후 기록 없음    3
      2018.04.26. 이후 기록 없음    2
```



```
      2017.03.24. 이후 폐선        2
      Name: 비고, dtype: int64
```

이 데이터 프레임 안의 비고 열에 대한 자료형을 확인하면 object이므로 문자열입니다.

.fillna 메소드에 매개변수 axis=1을 넣으면 열에 있는 모든 원소를 처리합니다. 또한 기존 데이터 프레임을 유지하려면 매개변수 inplace=True를 넣습니다.

누락 값을 없앤 후에 데이터 프레임 전체에 누락 값이 있는지를 확인하면 정수 값이 0으로 처리되어 누락 값이 전부 대체된 것을 볼 수 있습니다.

```
In :  flight_route_info['비고'].dtype
```

```
Out:  dtype('O')
```

```
In :  flight_route_info.fillna(" ",axis=1, inplace=True)
```

```
In :  flight_route_info.isnull().sum().sum()
```

```
Out:  0
```

'비고' 열의 값의 빈도를 확인하기 위해 .value_counts 메소드를 실행하면 빈 문자열로 바뀐 것을 볼 수 있습니다.

```
In :  flight_route_info['비고'].value_counts(dropna=False).head()
```

```
Out:                             412
      2016.11.05. 이후 폐선        4
      2018.04.24. 이후 기록 없음    3
      2017.09.16  이후 폐선        2
      2017.03.24. 이후 폐선        2
      Name: 비고, dtype: int64
```

위의 누락 값이 같은지를 확인하기 위해 다시 파일을 읽고 비고 열의 값의 빈도를 확인하면 누락 값이 있음을 확인할 수 있습니다. 위처럼 같은 값으로 누락 값이 바뀝니다.

```
In :  flight_route_info_2 = pd.read_csv('../data/flight_route_info.csv',encoding='cp949')
```

```
In :  flight_route_info_2['비고'].value_counts(dropna=False).head()
```

```
Out:  NaN                          412
      2016.11.05. 이후 폐선          4
      2018.04.24. 이후 기록 없음     3
      2018.04.26. 이후 기록 없음     2
      2017.03.24. 이후 폐선          2
      Name: 비고, dtype: int64
```

누락 값을 간단히 처리하기 위해 메소드 체인을 이용할 수 있습니다. 먼저 특정 자료형의 열을 가져오는 .select_dtypes 메소드에 object의 문자열을 가진 리스트를 넣으면 데이터 프레임 안의 object 자료형을 가진 열이 전부 검색됩니다.

다음 메소드인 .fillna를 연결해서 처리한 결과를 이번에는 다른 변수에 할당합니다.

누락 값이 없어진 것을 .value_counts 메소드로 조회하면 빈 문자열로 처리된 것을 알 수 있습니다.

```
In :  flight_route_info_3 = flight_route_info_2.select_dtypes(['object']).fillna('')
```

```
In :  flight_route_info_3['비고'].value_counts(dropna=False).head()
```

```
Out:                               412
      2016.11.05. 이후 폐선          4
      2018.04.24. 이후 기록 없음     3
      2018.04.26. 이후 기록 없음     2
      2017.03.24. 이후 폐선          2
      Name: 비고, dtype: int64
```

■누락 값 보간

누락 값이 발생할 때 임의의 값을 하나로 확정하는 것보다 특정 간격의 값을 넣어야 할 때가 있습니다. 이럴 때 보간 방식을 이용해서 처리하는 방법을 알아봅니다.

[예제 3-11] 누락 값을 보간 처리하기

누락 값을 가진 시리즈를 하나 만듭니다. 이 시리즈의 값을 확인하기 위해 .interpolate 메소드를 실행하면 누락 값에 두 사이의 숫자의 중간 값이 할당된 것을 알 수 있습니다.


```
In :  import numpy as np
```

```
In :  s = pd.Series([0, 1, np.nan, 3])
```

```
In :  s.interpolate()
```

```
Out:  0    0.0
      1    1.0
      2    2.0
      3    3.0
      dtype: float64
```

데이터 프레임을 가지고 보간법을 처리하는 방식을 알아보기 위해서 하나의 csv 파일을 읽습니다. 누락 값이 발생했는지를 .isnull.sum으로 확인하면 총정원 열에 누락 값이 있습니다.

```
In :  airplane_info = pd.read_csv('../data/airplane_info.csv',encoding='cp949')
```

```
In :  airplane_info.isnull().sum()
```

```
Out:  기체번호        0
      항공사          0
      기종            0
      정원_F          0
      정원_C          0
      정원_W          0
      정원_Y          0
      총정원         71
      dtype: int64
```

데이터 프레임 인덱스 검색에 인자로 열의 이름이 들어간 리스트를 넣어서 조회하고 .tail 메소드를 실행하면 누락 값을 가진 기체번호가 조회됩니다.

```
In :  airplane_info[['기체번호','총정원']].tail()
```

Out:

	기체번호	총정원
2173	VP-BWW	NaN
2174	VP-BRM	NaN
2175	VP-BWZ	NaN
2176	VP-BWX	NaN
2177	VP-BRB	NaN

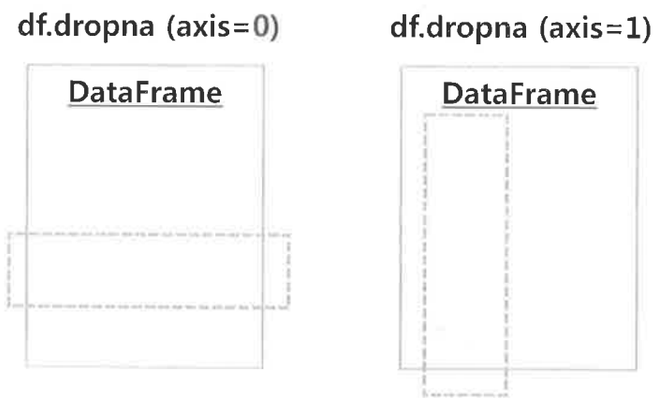
총정원 열을 검색한 후에 .interpolate 메소드를 실행하면 내부에 들어간 값들에 대해서 보간을 처리합니다.

```
In :  a = airplane_info['총정원'].interpolate()

In :  a.isnull().sum()

Out:  0

In :  a.tail()

Out:  2173    180.0
      2174    180.0
      2175    180.0
      2176    180.0
      2177    180.0
      Name: 총정원, dtype: float64
```

3.2.4 누락 값 삭제

누락 값을 삭제하기 위해 .dropna나 .drop 메소드를 이용해 처리할 수 있습니다. 실제 데이터 값을 변경하지 않아도 된다고 생각하면 누락 값을 전부 제거하고 처리하는 것이 좋습니다.

누락 값은 행과 열 단위로 생기므로 삭제가 필요한 경우 행과 열 단위로 삭제해야 합니다.



[그림 3-2] 누락 값 삭제

■ 열 삭제

열에 누락 값이 존재할 경우 열 전체를 삭제할 필요가 있다면 .dropna, .drop 메소드로 열을 삭제할 수 있습니다.

[예제 3-12] 열 삭제하기

한글이 들어 있는 csv 파일을 읽습니다. 인코딩은 cp949로 처리해 읽습니다. 파일이 생성될 때 임의의 열이 들어가 있습니다.

```
In :  central_asia_flight_info = pd.read_csv('../data/central_asia_flight_info.csv',encoding='cp949')

In :  central_asia_flight_info.head()
```

Out:

	노선명	출발공항	출발국가	도착공항	도착국가	기체번호	기종	정원(전체)	연도	월	일	Unnamed: 11
0	7J105	DYU	TJK	IKA	IRI	EY-444	B737-3L9	149	2016	6	5	NaN
1	7J105	DYU	TJK	IKA	IRI	EY-444	B737-3L9	149	2016	6	12	NaN
2	7J105	DYU	TJK	IKA	IRI	EY-757	B757-231	197	2016	6	19	NaN
3	7J105	DYU	TJK	IKA	IRI	EY-757	B757-231	197	2016	6	26	NaN
4	7J105	DYU	TJK	IKA	IRI	EY-757	B757-231	197	2016	7	3	NaN

데이터 프레임 안에 실제 누락 값이 있는지를 확인합니다. 임의의 열만 전부 누락 값이 들어가 있습니다.

```
In :  central_asia_flight_info.count()

Out:  노선명          95015
      출발공항        95015
      출발국가        95015
      도착공항        95015
      도착국가        95015
      기체번호        95015
      기종            95015
      정원(전체)      95015
      연도            95015
      월              95015
      일              95015
      Unnamed: 11         0
      dtype: int64
```

실제 누락 값을 확인하기 위해 .isnull, .sum 메소드를 이용해 확인해봅니다.

```
In :  central_asia_flight_info.isnull().sum()
```

```
Out:  노선명              0
      출발공항            0
      출발국가            0
      도착공항            0
      도착국가            0
      기체번호            0
      기종                0
      정원(전체)          0
      연도                0
      월                  0
      일                  0
      Unnamed: 11     95015
      dtype: int64
```

열 전체를 삭제하기 위해 .dropna 메소드를 실행해 누락 값이 있는 열을 삭제했습니다.

```
In :  central = central_asia_flight_info.dropna(axis=1)
```

```
In :  central.count()
```

```
Out:  노선명              95015
      출발공항            95015
      출발국가            95015
      도착공항            95015
      도착국가            95015
      기체번호            95015
      기종                95015
      정원(전체)          95015
      연도                95015
      월                  95015
      일                  95015
      dtype: int64
```

특정 열을 전부 삭제할 때는 .drop 메소드를 이용해도 삭제할 수 있습니다.

```
In :  central_drop = central_asia_flight_info.drop('Unnamed: 11', axis=1)
```

```
In :  central_drop.count()
```



```
Out:  노선명              95015
      출발공항            95015
      출발국가            95015
      도착공항            95015
      도착국가            95015
      기체번호            95015
      기종                95015
      정원(전체)          95015
      연도                95015
      월                  95015
      일                  95015
      dtype: int64
```

■행 삭제

행을 삭제하려면 누락 값이 들어있는 행에 대한 정보를 조회한 후에 행을 삭제해야 합니다. 행에 있는 누락 값을 비교한 후에 실제 행의 레이블을 추출하고 메소드를 실행해서 삭제해야 합니다.

[예제 3-13] 행 삭제하기

하나의 한글 파일을 읽고 cp949로 인코딩해 처리합니다. 열에 누락 값이 있는지를 count 메소드로 확인합니다.

```
In :  airplane_info = pd.read_csv('../data/airplane_info.csv',encoding='cp949')
```

```
In :  airplane_info.count()
```

```
Out:  기체번호            2178
      항공사              2178
      기종                2178
      정원_F              2178
      정원_C              2178
      정원_W              2178
      정원_Y              2178
      총정원              2107
      dtype: int64
```

넘파이 모듈에서 배열 안에 누락 값을 점검하기 위한 .isnan 함수가 있습니다. 먼저 하나의 배열을 만들어서 이 함수를 실행하면 불리언 값으로 처리되는 것을 볼 수 있습니다.

```
In :  import numpy as np

In :  np.isnan(np.array([1,2,np.nan]))

Out:  array([False, False,  True])
```

데이터 프레임 안의 총정원 열에 누락 값이 있는지를 확인하려면 .apply 메소드를 이용해서 인자로 np.isnan 함수를 넣어 실행하여 누락 값이 있으면 True로 표시합니다.

```
In :  airplane_info['총정원'].apply(np.isnan).tail()

Out:  2173    True
      2174    True
      2175    True
      2176    True
      2177    True
      Name: 총정원, dtype: bool
```

하나의 열을 읽어와 .index 속성을 확인합니다. RangeIndex가 나오고 .index 속성을 펜시 검색으로 조회하면 Int64Index로 표시합니다.

```
In :  airplane_info['총정원'].index

Out:  RangeIndex(start=0, stop=2178, step=1)

In :  airplane_info['총정원'].index[[2107, 2108]]

Out:  Int64Index([2107, 2108], dtype='int64')
```

실제 .index 속성 안의 특정 인덱스 레이블을 가져올 때는 apply 메소드와 np.isnan 함수가 실행된 결과를 이용해서도 인덱스 레이블을 가져올 수 있습니다.

```
In :  index_nan = airplane_info['총정원'].index[airplane_info['총정원'].apply(np.isnan)]

In :  index_nan

Out:  Int64Index([2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117,
                  2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128,
```



```
                  2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139,
                  2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150,
                  2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161,
                  2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172,
                  2173, 2174, 2175, 2176, 2177],
                 dtype='int64')
```

위에서 만들어진 인덱스 정보를 가지고 데이터 프레임의 행을 삭제하는 .drop 메소드의 인자로 전달해서 실행하면 누락 값이 삭제됩니다.

```
In :  airplane_info_nan = airplane_info.drop(index=index_nan)

In :  airplane_info_nan.shape

Out:  (2107, 8)

In :  airplane_info_nan.isnull().sum()

Out:  기체번호      0
      항공사        0
      기종         0
      정원_F       0
      정원_C       0
      정원_W       0
      정원_Y       0
      총정원        0
      dtype: int64
```

데이터 프레임의 행을 삭제할 경우 .dropna 메소드에 매개변수 axis=0을 이용해도 가능합니다.

```
In :  airplane_info_dropna = airplane_info.dropna(axis=0)

In :  airplane_info_dropna.shape

Out:  (2107, 8)

In :  airplane_info_dropna.isnull().sum()
```

```
Out:  기체번호          0
      항공사            0
      기종              0
      정원_F            0
      정원_C            0
      정원_W            0
      정원_Y            0
      총정원            0
      dtype: int64
```

하나의 열을 검색해서 시리즈로 만든 후에 이 시리즈의 누락 값을 .dropna 메소드에 axis=0을 주고 삭제할 수 있습니다.

```
In :  ai = airplane_info['총정원']

In :  ai.isnull().sum()

Out:  71

In :  ai_d = ai.dropna()

In :  ai_d.isnull().sum()

Out:  0
```

3.2.5 중복 값 확인 및 삭제

판다스는 누락 값만 관리하는 것이 아니라 중복된 값들도 시리즈나 데이터 프레임에서 관리합니다. 시리즈의 중복 값은 1차원이라 행에 속한 값들에 중복이 가능하고 데이터 프레임은 2차원이라 행과 열로 중복 값을 가질 수 있습니다.
이번 절에서는 중복 값을 확인하고 삭제하는 방법을 알아봅니다.

■중복 값 삭제

알고리즘 분석에 예제로 많이 사용되는 seaborn 모듈에서 제공하는 데이터셋 중에 iris를 가져와서 중복 값을 삭제하는 방법을 알아봅니다.

[예제 3-14] 중복 값 삭제하기

데이터 프레임을 만들기 위해 seaborn 모듈을 임포트하고 load_dataset 함수에 iris 문자열을 넣으면 csv 파일을 가져옵니다. 변수명을 iris로 주고 할당하면 하나의 데이터 프레임이 생깁니다. .columns 속성을 보면 열의 이름을 확인할 수 있습니다.

```
In :  import seaborn as sns

In :  iris = sns.load_dataset('iris')

In :  iris.columns

Out:  Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
             'species'],
            dtype='object')
```

실제 iris 안의 데이터 정보를 조회합니다.

```
In :  iris.head()
```

Out:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
In :  iris.shape

Out:  (150, 5)
```

데이터 프레임의 중복된 데이터는 행에 포함된 모든 열이 같은 값을 가졌는지를 확인합니다. 데이터 프레임에 .duplicated 메소드를 실행하면 True/False 값으로 표시합니다. 이를 .sum 메소드로 확인하면 하나가 중복된 것을 알 수 있습니다.

또한 species 열을 조회해 하나의 시리즈를 만들고 .duplicated 메소드를 실행하면 중복된 값은 True로 표시됩니다.

```
In :  iris.duplicated().sum()

Out:  1

In :  iris['species'].duplicated().head()

Out:  0    False
      1     True
      2     True
      3     True
      4     True
      Name: species, dtype: bool
```

특정 열에 대한 정보를 보고 유일한 값을 확인하기 위해서는 .unique 메소드를 실행합니다.

중복된 열의 정보를 삭제하기 위해 .drop_duplicate 메소드를 실행하면 .duplicate 메소드 실행 결과 중에 True로 표시된 것을 삭제합니다.

```
In :  iris['species'].unique()

Out:  array(['setosa', 'versicolor', 'virginica'], dtype=object)

In :  iris['species'].drop_duplicates()

Out:  0          setosa
      50     versicolor
      100     virginica
      Name: species, dtype: object
```

중복된 값을 삭제하기 위해 새로운 변수에 .copy 메소드를 사용해서 변수에 할당합니다.

```
In :  iris.shape

Out:  (150, 5)

In :  iris_dup =  iris.copy()
```

기존 데이터의 첫 번째 행부터 모든 것을 사본으로 만들어진 데이터 프레임에 두 번째 행부터 넣습니다. 데이터 프레임의 .iloc 속성에 할당해도 기존 데이터 프레임이 변경되는 것을 알 수 있습니다.

.head 메소드로 조회하면 첫째 행과 둘째 행이 같습니다.

```
In :  iris.iloc[[0]]
```

Out:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa

```
In :  iris_dup.iloc[1, : ] = iris.iloc[0, :]

In :  iris_dup.head(5)
```

Out:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	5.1	3.5	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

기존에 조회했던 것보다 중복된 행을 하나 더 만든 것을 알 수 있습니다.

중복된 것을 .drop_duplicates 메소드에 매개변수 inplace=True로 지정한 후 실행하면 데이터 프레임 내부에 중복이 제거됩니다.

```
In :  iris_dup.duplicated().sum()

Out:  2

In :  iris_dup.drop_duplicates(inplace=True)

In :  iris_dup.shape

Out:  (148, 5)
```

.head 메소드로 데이터 프레임을 조회하면 레이블 1인 행이 삭제된 것을 알 수 있습니다. .duplicated 메소드로 확인하면 중복이 없는 것을 알 수 있습니다.

```
In :  iris_dup.head()
```

Out:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
5	5.4	3.9	1.7	0.4	setosa

```
In :  iris_dup.duplicated().sum()
```

```
Out:  0
```

3.3 범주형 데이터 처리

연속적인 숫자 데이터를 제외하고는 대부분 문자열 데이터입니다. 문자열 데이터를 구조화해서 범주형(category) 데이터를 만들어야 할 때가 많습니다.

모든 문자열 데이터가 범주형으로 변환이 되는 것은 아닙니다. 분명 변환이 되지 않는 문자열 데이터도 존재하지만, 이번 절에서는 범주 처리가 가능한 문자열 데이터를 어떻게 변환하는지부터 알아봅니다.

3.3.1 범주형 데이터 생성

자료형을 'category' 문자열로 범주형으로 지정해서 시리즈나 데이터 프레임을 생성합니다. 별도로 범주형 자료형을 Categorical 클래스로 만들어 이 객체를 시리즈나 데이터 프레임의 각 열에 자료형으로 지정하여 처리할 수 있습니다.

■ 범주형 데이터 생성

범주형 데이터를 사용하기 위해 실제 범주형 자료형을 만들어보고 문자열로 생성된 것과의 차이점을 알아봅니다.

[예제 3-15] 범주형 데이터 생성하기

시리즈 생성자에는 리스트가 있고, 리스트 내에는 문자열의 값이 들어 있습니다.

매개변수 dtype에 문자열로 category를 지정하면 자료형이 object에서 category로 변경됩니다. 범주형 자료형의 정보는 3개의 원소, 내부는 문자열이라는 것을 알 수 있습니다.

```
In :  s = pd.Series(["a","b","c","a"], dtype="category")
```

```
In :  s
```

```
Out:  0    a
      1    b
      2    c
      3    a
      dtype: category
      Categories (3, object): [a, b, c]
```

만들어진 시리즈의 .values 속성 안에 .categories 속성이 추가되어 있는 것을 확인할 수 있습니다. 그 결과로 만들어진 객체가 Index 클래스의 객체입니다. .dtype 속성으로 확인해보면 CategoricalDtype 클래스의 객체 정보라는 것을 표시합니다.

```
In :  s.shape
```

```
Out:  (4,)
```

```
In :  s.values.categories
```

```
Out:  Index(['a', 'b', 'c'], dtype='object')
```

```
In :  s.index
```

```
Out:  RangeIndex(start=0, stop=4, step=1)
```

```
In :  s.dtype
```

```
Out:  CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)
```

범주형 자료형은 범주형에 속하지 않은 데이터가 들어오면 갱신되지 않는다는 특징이 있습니다.

위에서 만들어진 시리즈에 특정 원소를 추가하면 범주의 값에 이 문자열이(string) 없어 예외를 발생시킵니다.

```
In :  try :
          s[2] = 'd'
      except Exception as e :
          print(e)
```

```
Out:  Cannot setitem on a Categorical with a new category, set the categories first
```

기존 행의 레이블 범위 안을 범주형에 포함된 데이터로 갱신하면 변경됩니다.

새로운 레이블을 인덱스 검색에 넣고 처리하면 인덱스 레이블 범위가 넘기 때문에 예외를 발생시킵니다.

```
In :  s[2] = 'b'
```

```
In :  try :
          s[4] = 'c'
      except Exception as e :
          print(e)
```

```
Out:  [4] not contained in the index
```

시리즈에서 레이블의 범위를 추가하려면 .loc 인덱서 속성을 이용하여 범주 값을 넣어서 처리합니다. 이렇게 시리즈의 행이 추가되는 것을 볼 수 있습니다.

```
In :  s.loc[4] ='c'
```

```
In :  s
```

```
Out:  0    a
      1    b
      2    b
      3    a
      4    c
      dtype: object
```

[예제 3-16] 범주형 데이터 클래스 이해하기

[예제 3-15]에서는 시리즈를 생성할 때 dtype에 범주형을 지정해 처리했습니다. 실제 범주형

자료형을 이해하기 위해서는 Categorical 클래스로 범주형 데이터를 만들어보는 것도 필요합니다.

Categorical 생성자에 문자열을 원소로 하는 리스트를 넣어서 실행하면 하나의 범주형 객체가 만들어집니다.

```
In :  cat = pd.Categorical(['a','b','c'])
```

```
In :  type(cat)
```

```
Out:  pandas.core.categorical.Categorical
```

범주형 데이터 안의 멤버와 시리즈 안의 멤버를 비교해 범주형 데이터에만 존재하는 멤버들을 확인해봅니다.

```
In :  c = set(dir(cat))
```

```
In :  ss = set(dir(pd.Series))
```

```
In :  c_d =  c - ss
```

시리즈와 범주형의 차집합을 처리하고 그 안의 범주형 데이터에만 있는 함수들을 확인해봅니다.

```
In :  count = 0
      for i in c_d :
          if not i.startswith("_") :
              count += 1
              print(i, end=", ")
              if count % 5 == 0 :
                  print()
```

```
Out:  codes, check_for_ordered, add_categories, as_unordered, reorder_categories,
      from_codes, is_dtype_equal, ordered, take_nd, categories,
      remove_categories, rename_categories, set_ordered, set_categories, remove_
      unused_categories,
      as_ordered,
```

위에서 생성된 범주형 안의 자료형에 대한 정보는 .categories 속성을 확인합니다.

```
In :  cat.categories
```

```
Out:  Index(['a', 'b', 'c'], dtype='object')
```

```
In :  cat.dtype
```

```
Out:  CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)
```

만들어진 범주형 자료형으로 새로운 시리즈를 만들 때 매개변수 dtype에 넣습니다. 문자열로 categoriy로 지정한 것과 같음을 알 수 있습니다.

```
In :  s1 = pd.Series(["a","b","c","a"], dtype=cat.dtype)
```

```
In :  s1
```

```
Out:  0    a
      1    b
      2    c
      3    a
      dtype: category
      Categories (3, object): [a, b, c]
```

[예제 3-17] 데이터 프레임에서 범주형 자료형 처리하기

범주형을 만들 때 실제 들어가는 값에 누락 값을 넣었습니다.
Categorical 클래스의 categories 매개변수에 범주 데이터를 넣고 실행하면 하나의 범주형 인스턴스가 만들어집니다.

```
In :  import numpy as np
```

```
In :  cat_1 = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])
```

```
In :  cat_1.dtype
```

```
Out:  CategoricalDtype(categories=['b', 'a', 'c'], ordered=False)
```

데이터 프레임을 만들 때 하나의 열은 범주형으로, 하나는 문자열로 처리했습니다. 이 데이터 프레임의 열에 누락 값이 있음을 볼 수 있습니다.


```
In :  df = pd.DataFrame({"cat":cat_1, "s":["a", "c", "c", np.nan]})
```

```
In :  df
```

```
Out:
```

	cat	s
0	a	a
1	c	c
2	c	c
3	NaN	NaN

데이터 프레임의 특정 자료형으로 열을 검색하는 .select_dtypes 메소드로 exclude 매개변수에 object가 아닌 것을 검색하면 범주형 열만 가진 데이터 프레임이 조회됩니다.

```
In :  df.select_dtypes(exclude=['object'])
```

```
Out:
```

	cat
0	a
1	c
2	c
3	NaN

이 데이터 프레임 안의 범주형 열을 가지고 누락 값이 있는 행에 범주형 범위의 값을 넣어서 갱신하면 변경됩니다.

```
In :  df['cat'][3] = 'b'
```

```
In :  df
```

```
Out:
```

	cat	s
0	a	a
1	c	c
2	c	c
3	b	NaN

이번에는 .loc 인덱서 속성을 이용해서 특정 행의 두 열에 값을 추가합니다. 범주형 자료형일 때는 항상 범주형 자료형에 맞는 값을 넣어야 합니다.

```
In :  df.loc[4,:] = ['a','a']
```

```
In :  df
```

Out:

	cat	s
0	a	a
1	c	c
2	c	c
3	b	NaN
4	a	a

3.3.2 범주형 데이터 활용

문자열된 데이터를 분석해보면 한정된 값이 반복적으로 나옵니다. 이런 특정 범위만을 사용하는 데이터를 문자열로 처리하기보다 특정 범주형으로 지정해 처리하면 메모리 사용도 줄일 수 있고 범위가 지정되므로 임의로 값을 추가하는 것을 제약할 수 있습니다

■범주형 데이터를 변경해 처리

데이터 프레임의 열의 값들을 검토해 범주형으로 변환하는 것이 좋다고 판단했을 때, 그 데이터를 범주형으로 바꾸는 방법을 알아봅니다.

[예제 3-18] 파일을 읽어서 범주형 자료형으로 변경하기

하나의 파일을 읽어 와서 데이터 프레임의 객체로 만듭니다.

```
In :  airplane = pd.read_csv('../data/airplane_info.csv',encoding='cp949')
```

```
In :  airplane.head()
```

Out:

	기체번호	항공사	기종	정원_F	정원_C	정원_W	정원_Y	총정원
0	VP-BDK	Aeroflot	A320-214	0	20	0	120	140.0
1	VP-BWD	Aeroflot	A320-214	0	20	0	120	140.0
2	VP-BWE	Aeroflot	A320-214	0	20	0	120	140.0
3	VP-BWF	Aeroflot	A320-214	0	20	0	120	140.0
4	VP-BRZ	Aeroflot	A320-214	0	20	0	120	140.0

이 데이터 프레임 중 항공사의 열을 .dtype으로 확인합니다. Object 자료형이므로 문자열로 처리됩니다.

```
In :  airplane['항공사'].dtype
```

```
Out:  dtype('O')
```

항공사 열을 Categorical 클래스의 데이터로 넣어서 하나의 범주 자료형을 만듭니다. .categories 속성으로 범주 안의 데이터를 확인합니다.

```
In :  air_cat = pd.Categorical(airplane['항공사'])
```

```
In :  air_cat.categories
```

```
Out:  Index(['AZAL Azerbaijan Airlines', 'Aeroflot', 'Air Arabia', 'Air Astana',
       'Air Baltic', 'Air China', 'Air Kyrgyzstan', 'Air Manas',
       'Airzena Georgian Airways', 'Asiana Airlines', 'AtlasGlobal',
       'Avia Traffic Company', 'Belavia', 'China Southern Airlines',
       'Ellinair', 'Etihad Airways', 'Finnair', 'Globus Airlines',
       'Hainan Airlines', 'Iran Aseman Airlines', 'KLM Royal Dutch Airlines',
       'Kam Air', 'Korean Air', 'LOT - Polish Airlines', 'Lufthansa',
       'MIAT - Monglian Airlines', 'Mahan Air', 'NordStar', 'Pegas Fly',
       'Pegasus Airlines', 'Pobeda', 'Red Wings', 'Rusline', 'S7 Airlines',
       'SCAT', 'Somon Air', 'Sunday Airlines', 'Tajik Air', 'Turkish Airlines',
       'Turkmenistan Airlines', 'Ukraine International Airlines',
       'Ural Airlines', 'Urumqi Air', 'Utair', 'Uzbekistan Airways',
       'VIM Airlines', 'Wizz Air', 'Yakutia Airlines', 'Yamal Airlines',
       'flydubai'],
      dtype='object')
```

생성된 범주 자료형의 변수가 어떤 정보를 가지고 있는지 확인해봅시다. Categoris 클래스의 객체로서 실제 전체 데이터와 내부의 범주형 값들도 보여줍니다.

```
In :  air_cat
```

```
Out:  [Aeroflot, Aeroflot, Aeroflot, Aeroflot, Aeroflot, ..., Red Wings, Red Wings, Red Wings, Red Wings, Red Wings]
      Length: 2178
      Categories (50, object): [AZAL Azerbaijan Airlines, Aeroflot, Air Arabia, Air Astana, ..., Wizz Air, Yakutia
      Airlines, Yamal Airlines, flydubai]
```

```
In :  air_cat.shape
```

```
Out:  (2178,)
```

데이터 프레임 원본인 airplane 안의 항공사 열에 자료형을 조회하고 기존 열의 자료형을 만들어진 범주형 자료형으로 변경합니다.

```
In :  airplane['항공사'].dtype
```

```
Out:  dtype('O')
```

```
In :  airplane['항공사'] = airplane['항공사'].astype(air_cat)
```

변경된 열의 자료형을 확인하면 범주형 자료형이라는 것을 알 수 있습니다.

```
In :  airplane['항공사'].dtype
```

```
Out:  CategoricalDtype(categories=['AZAL Azerbaijan Airlines', 'Aeroflot', 'Air Arabia',
                                   'Air Astana', 'Air Baltic', 'Air China', 'Air Kyrgyzstan',
                                   'Air Manas', 'Airzena Georgian Airways', 'Asiana Airlines',
                                   'AtlasGlobal', 'Avia Traffic Company', 'Belavia',
                                   'China Southern Airlines', 'Ellinair', 'Etihad Airways',
                                   'Finnair', 'Globus Airlines', 'Hainan Airlines',
                                   'Iran Aseman Airlines', 'KLM Royal Dutch Airlines',
                                   'Kam Air', 'Korean Air', 'LOT - Polish Airlines',
                                   'Lufthansa', 'MIAT - Monglian Airlines', 'Mahan Air',
                                   'NordStar', 'Pegas Fly', 'Pegasus Airlines', 'Pobeda',
                                   'Red Wings', 'Rusline', 'S7 Airlines', 'SCAT', 'Somon Air',
                                   'Sunday Airlines', 'Tajik Air', 'Turkish Airlines',
                                   'Turkmenistan Airlines', 'Ukraine International Airlines',
                                   'Ural Airlines', 'Urumqi Air', 'Utair', 'Uzbekistan Airways',
                                   'VIM Airlines', 'Wizz Air', 'Yakutia Airlines',
                                   'Yamal Airlines', 'flydubai'],
                  ordered=False)
```

범주형으로 바뀐 항공사 열에 범주 값이 범위에 없는 것을 입력하면 예외가 발생하는 것을 알 수 있습니다.

```
In :  try :
          airplane['항공사'][0] = "Asia Airline"
      except Exception as e :
          print(e)
```

```
Out:  Cannot setitem on a Categorical with a new category, set the categories first
```


누락 값이 있는지를 확인하면 총정원 열에 71개가 나옵니다.

```
In :  airplane.isnull().sum()
```

```
Out:  기체번호      0
      항공사        0
      기종          0
      정원_F        0
      정원_C        0
      정원_W        0
      정원_Y        0
      총정원       71
      dtype: int64
```

총정원 열에만 누락 값이 있는지 또 한번 확인해봅니다.

```
In :  airplane.columns
```

```
Out:  Index(['기체번호', '항공사', '기종', '정원_F', '정원_C', '정원_W', '정원_Y', '총정원'], dtype='object')
```

```
In :  airplane['총정원'].isnull().sum()
```

```
Out:  71
```

총정원의 열에서 누락 값을 .loc 행으로 조회하고 정원_Y의 값으로 대체해서 할당하면 실제 데이터 프레임 내부의 값이 변경되어 누락 값이 전부 사라진 것을 알 수 있습니다.

```
In :  airplane.loc[airplane['총정원'].isnull(), '총정원'] = airplane.loc[airplane['총정원'].isnull(), '정원_Y']
```

```
Out:  airplane.isnull().sum()
```

```
In :  기체번호      0
      항공사        0
      기종          0
      정원_F        0
      정원_C        0
      정원_W        0
      정원_Y        0
      총정원        0
      dtype: int64
```

이번에는 총정원을 이용해서 다른 범주의 열을 추가하기 위해 특정 값의 사이에 있는 것을 확인합니다. 이때 .between 메소드로 특정 범주에 속한 데이터를 확인하면 결과가 True/False로 나오는 것을 볼 수 있습니다.

```
In :  airplane['총정원'].between(1,120).sum()

Out:  235

In :  mask = airplane['총정원'].between(1,120)
```

항공기를 소형과 중형으로 범주를 분리하기 위해서 넘파이 모듈의 where 함수를 이용해 논리값을 기준으로 소형과 중형의 값을 처리하도록 하고 이를 범주형인 Catergorical로 변환해 새로운 열인 category_R로 저장합니다.

```
In :  import numpy as np

In :  airplane['category_R'] = ""

In :  airplane['category_R']  = pd.Categorical(np.where(mask, '소형', '중형'))
```

데이터 프레임을 확인하면 새로운 열이 추가된 것을 알 수 있습니다.

```
In :  airplane.head()
```

Out:

	기체번호	항공사	기종	정원_F	정원_C	정원_W	정원_Y	총정원	category_R
0	VP-BDK	Aeroflot	A320-214	0	20	0	120	140.0	중형
1	VP-BWD	Aeroflot	A320-214	0	20	0	120	140.0	중형
2	VP-BWE	Aeroflot	A320-214	0	20	0	120	140.0	중형
3	VP-BWF	Aeroflot	A320-214	0	20	0	120	140.0	중형
4	VP-BRZ	Aeroflot	A320-214	0	20	0	120	140.0	중형

새로 추가된 열의 값을 .value_counts 메소드로 확인합니다. 범주형 처리가 되었다는 사실을 알 수 있습니다. 그리고 이 열의 .dtype을 확인합니다.


```
In :  airplane['category_R'].value_counts()

Out:  중형    1943
      소형     235
      Name: category_R, dtype: int64

In :  airplane['category_R'].dtype

Out:  CategoricalDtype(categories=['소형', '중형'], ordered=False)
```