

B O N U S



Pandas 的效能、 除錯與測試

C.1 轉換資料

在本章，我們將檢視一些程式碼，它們可用來分析 2018 年 Kaggle 使用者的問卷資料，該問卷調查了 Kaggle 使用者的社會及經濟資訊。

本節會展示問卷資料以及用於分析資料的程式碼，讓我們深入研究這些資料，看看可以挖掘出什麼有用的資訊。該原始資料可在 <https://www.kaggle.com/kaggle/kaggle-survey-2018> 上取得。

動手做

01 載入相關資料集：

 In

```
import pandas as pd
import numpy as np
import zipfile
url = 'data/kaggle-survey-2018.zip'

with zipfile.ZipFile(url) as z:
    print(z.namelist()) ← 檢視該壓縮檔中有哪些文件
    kag = pd.read_csv(z.open('multipleChoiceResponses.csv')) ← 讀入相關的 CSV 檔案
    df = kag.iloc[1:] ← 取得第一列（代表具體問題）以外的資料
```

Out

```
['multipleChoiceResponses.csv', 'freeFormResponses.csv', 'SurveySchema.csv']
```

02 查看資料與資料型別：

 In

df.T ← 進行轉置以顯示更多資訊

Out

	1	2	...	23858	23859
Time from Start to Finish (seconds)	710	434	...	36	502
Q1	Female	Male	...	Male	Male
Q1_OTHER_TEXT	-1	-1	...	-1	-1
Q2	45-49	30-34	...	25-29	25-29
Q3	United States of America	Indonesia	...	United Kingdom of Great Britain and Northern I...	Spain
...
Q50_Part_5	NaN	NaN	...	NaN	NaN
Q50_Part_6	NaN	NaN	...	NaN	NaN
Q50_Part_7	NaN	NaN	...	NaN	NaN
Q50_Part_8	NaN	NaN	...	NaN	NaN
Q50_OTHER_TEXT	-1	-1	...	-1	-

 In

df.dtypes

Out

Time from Start to Finish (seconds)	object
Q1	object
Q1_OTHER_TEXT	object
Q2	object
Q3	object
...	...
Q50_Part_5	object
Q50_Part_6	object

```
Q50_Part_7          object
Q50_Part_8          object
Q50_OTHER_TEXT      object
Length: 395, dtype: object
```

 In

```
df.dtypes.value_counts(dropna=False) ← 查看不同資料型別出現的次數
```

Out

```
object    395
dtype: int64
```

03 從輸出可見，所有欄位的型別都是 object。我們可以使用 `value_counts()` 來探究 Q1 欄位中，不同資料出現的次數：

 In

```
df.Q1.value_counts(dropna=False) ← 查看 Q1 欄位的資料出現次數
```

Out

```
Male          19430
Female        4010
Prefer not to say  340
Prefer to self-describe  79
Name: Q1, dtype: int64
```

04 為了方便後續的操作，我們將個別對感興趣的欄位（Q1 至 Q6 欄位，以及 Q8 和 Q9 欄位）先進行處理，然後再以 Series 的形式傳回。對於那些相異資料數很多的欄位，會將其過濾成只有特定的幾種資料。接著，使用 `rename()` 賦予欄位更恰當的名稱。某些欄位中的值為數值範圍，例如 Q2 欄位（存放年齡資訊）存放了類似 55-59 和 60-69 的數值。以下程式會使用 `str.slice()` 來取得前兩個字元（**編註**：以『55-59』為例，會取出『55』的結果），並將資料型別從字串轉換為整數。

對於 Q4 欄位（存有教育程度資訊），此處會將其中的資料轉為**有序數**（ordinal number，詳見以下程式）。在處理完所有感興趣的欄位後，使用 `pd.concat()` 將個別傳回的 Series 整合成一個 DataFrame。

此處，我們將相關欄位的處理工作放到 `tweak_kag()` 函式中（**編註**：建議讀者搭配 `multipleChoiceResponses.csv` 來檢視以下程式碼，這將有助於理解）：

In

```
def tweak_kag(df):
    na_mask = df.Q9.isna()
    hide_mask = df.Q9.str.startswith('I do not').fillna(False)
    df = df[~na_mask & ~hide_mask]

    q1 = (df.Q1.replace({'Prefer not to say': 'Another',
                        'Prefer to self-describe': 'Another'})
          .rename('Gender'))

    q2 = df.Q2.str.slice(0,2).astype(int).rename('Age')

    def limit_countries(val):
        if val in {'United States of America', 'India', 'China'}:
            return val
        return 'Another'

    q3 = df.Q3.apply(limit_countries).rename('Country')

    q4 = (df.Q4.replace({'Master's degree': 18,
                        'Bachelor's degree': 16,
                        'Doctoral degree': 20,
                        'Some college/university study without \
earning a bachelor's degree': 13,
                        'Professional degree': 19,
                        'I prefer not to answer': None,
                        'No formal education past high school': 12})
```

建立 2 個布林陣列，用來篩選出符合要求的列

使用 Another 來取代欄位中的特定資料

將 Q1 欄位改名為 Gender

取出前兩個字元，並轉換成整數型別

將 Q2 欄位改名為 Age

所有不符合以上 3 個國家名稱的資料，統一改為 Another

將 Q3 欄位改名為 Country

定義一個函式，用來限制 Q3 欄位中的分類

用數值來替換 Q4 欄位中的資料

```

        .fillna(11)      ← 用『11』來替換缺失值
        .rename('Edu')  ← 將 Q4 欄位改名為 Edu
    )

def only_cs_stat_val(val):
    if val not in {'cs', 'eng', 'stat'}:
        return 'another'
    return val

```

定義一個函式，用來限制 Q5 欄位中的分類

```

q5 = (df.Q5
      .replace({
          'Computer science (software engineering, etc.)': 'cs',
          'Engineering (non-computer focused)': 'eng',
          'Mathematics or statistics': 'stat'})
      .apply(only_cs_stat_val)
      .rename('Studies')) ← 將 Q5 欄位改名為 Studies

```

定義一個函式，用來限制 Q6 欄位中的資料分類

```

def limit_occupation(val):
    if val in {'Student', 'Data Scientist', 'Software Engineer', 'Not employed',
              'Data Engineer'}:
        return val
    return 'Another'

```

```

q6 = df.Q6.apply(limit_occupation).rename('Occupation')

```

將 Q6 欄位改名為 Occupation

```

q8 = (df.Q8 ← 該欄位存有相關工作年資
      .str.replace('+', '')
      .str.split('-', expand=True)
      .iloc[:,0]
      .fillna(-1)
      .astype(int)
      .rename('Experience') ← 將 Q8 欄位改名為 Experience
    )

```

```

q9 = (df.Q9 ← 該欄位存有年薪資料
      .str.replace('+', '')
      .str.replace(',', '')
      .str.replace('500000', '500')
      .str.replace('I do not wish to disclose my approximate yearly compensation', '')
    )

```

過濾掉沒有提供年薪資訊的受訪者

```

.str.split('-', expand=True)
.iloc[:,0]
.astype(int)
.mul(1000)
.rename('Salary')) ← 將 Q9 欄位改名為 Salary

return pd.concat([q1, q2, q3, q4, q5, q6, q8, q9], axis=1) ←
                                                    沿著欄位軸方向合併處理後的 Series
tweak_kag(df) ← 整理原始的資料集

```

Out

	Gender	Age	Country	Edu	Studies	Occupation	Experience	Salary
2	Male	30	Another	16.0	eng	Another	5	10000
3	Female	30	United States of America	18.0	cs	Data Scientist	0	0
5	Male	22	India	18.0	stat	Another	0	0
7	Male	35	Another	20.0	another	Another	10	10000
8	Male	18	India	18.0	another	Another	0	0
...
23844	Male	30	Another	18.0	cs	Software Engineer	10	90000
23845	Male	22	Another	18.0	stat	Student	0	0
23854	Male	30	Another	20.0	cs	Another	5	10000
23855	Male	45	Another	20.0	cs	Another	5	250000
23857	Male	22	Another	18.0	cs	Software Engineer	0	10000

 In

```
tweak_kag(df).dtypes
```

Out

```
Gender      object
Age          int32
Country      object
Edu          float64
Studies      object
Occupation   object
Experience   int32
Salary       int32
dtype: object
```

了解更多

此處使用的問卷資料有很豐富的資訊，但分析起來有點困難。因此，我們將一些欄位（Age、Edu、Experience 和 Salary）轉換為數值欄位以便於量化，並降低其餘分類欄位內的**基數**（cardinality，即相異資料數）。在清理資料後，分析工作會變得更容易。舉例來說，我們可以輕鬆地按國家分組，並計算年薪和工作年資的相關性：

 In

```
kag = tweak_kag(df)    傳回 Salary 和 Experience 的相關性
(kag.groupby('Country')
 .apply(lambda g: g.Salary.corr(g.Experience)))
```

Out

```
Country
Another      0.289827
China         0.252974
India         0.167335
United States of America  0.354125
dtype: float64
```


C.2 apply() 方法的效能

在 Series 和 DataFrame 上使用 `apply()` 方法，是 Pandas 中最慢的運算之一。在下面的例子中，我們將探索其速度並嘗試其它做法。

動手做

01 讓我們使用 `%%timeit` 計算 `apply()` 方法的用時。以下程式碼來自上一節的 `tweak_kag()` 函式，它可用來限制 country 欄位（即原始資料集中的 Q3 欄位）的基數：

In

```
def limit_countries(val):
    if val in {'United States of America', 'India', 'China'}:
        return val
    return 'Another'
```

```
%%timeit
q3 = df.Q3.apply(limit_countries) ] ← 測試 apply() 的用時
```

Out

4.91 ms ± 1.22 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)

02 讓我們看看改用 `replace()` 是否可以提高效能：

In

```
%%timeit
other_values = df.Q3.value_counts().iloc[3:].index ← 取出 Q3 欄位中最常出現
q3_2 = df.Q3.replace(other_values, 'Another') ← States of America、India 和 China) 以外的資料
                                              將 other_values 內的資料統一改為 Another
```

Out

27.7 ms ± 535 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

03 結果顯示，使用 `replace()` 的速度比 `apply()` 還要慢！現在，嘗試使用 `isin()` 結合 `where()` 來完成同樣的工作。你會發現，它的運算速度比 `apply()` 來得快：

 In

```
%%timeit
values = {'United States of America', 'India', 'China'}
q3_3 = df.Q3.where(df.Q3.isin(values), 'Another')
```

Out

3.39 ms ± 570 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

04 最後這個步驟使用的 `np.where()` 函式並不屬於 Pandas（而是一個 NumPy 函式）。從結果可見，該做法的速度是最快的。若我們使用了 NumPy 函式，就要將結果轉回為 Series（並且賦予跟原始 DataFrame 相同的索引），以取得與步驟 1 至步驟 3 相同的輸出：

 In

```
%%timeit
values = {'United States of America', 'India', 'China'}
q3_4 = pd.Series(np.where(df.Q3.isin(values), df.Q3, 'Another'), index=df.index)
```

↑
若 Q3 中的某筆資料不屬於 values 串列中的任一個，則改為 Another

Out

2.75 ms ± 345 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

05 讓我們檢查結果是否相同：

 In

```
q3.equals(q3_2)
```

Out

True

 In

```
q3.equals(q3_3)
```

Out

True

 In

```
q3.equals(q3_4)
```

Out

True

了解更多

`apply()` 的說明文件指出，如果傳遞給它的是一個 NumPy 函式，它會選擇較快速的做法，即將整個 Series 傳遞給該函式（**編註**：只需呼叫一次函式）。但如果傳入的是 Python 函式，則 Series 中的每個值都會呼叫該函式。由於 `apply()` 方法的行為取決於傳遞給它的參數（NumPy 函式或 Python 函式），因此有時會讓人感到搞不清楚。

如果你需要將一個函式傳遞給 `apply()`（或是已經完成 `groupby()` 運算並呼叫 `agg()`、`transform()` 或其他把函式作為參數的方法），但忘了什麼參數會被傳遞到函式時，可以使用以下程式碼來進行協助（當然，你也可以參考說明文件，甚至去查看 `apply()` 的程式碼）：

 In

```
def limit_countries(val):  
    if val in {'United States of America', 'India', 'China'}:  
        return val  
    return 'Another'  
q3 = df.Q3.apply(limit_countries).rename('Country')
```

和之前一樣，先整理 Q3 欄位中的資料，並更改欄位名稱

```
def debug(something):  
    print(type(something), something)  
    1/0 ← 由於 1/0 (1 除以 0) 會傳回錯誤訊息，因此在印出第一筆傳入  
        debug() 的資料及其型別後，便會停止運行程式  
q3.apply(debug)
```

Out

```
<class 'str'> United States of America  
Traceback (most recent call last):  
...  
ZeroDivisionError: division by zero
```

以上輸出顯示有一個字串 (United States of America) 被傳遞到 debug() 函式中。如果不想跑出異常警告，可以設定一個全域變數來保存傳入函式的參數：

 In

```
the_item = None  
def debug(something):  
    global the_item ← 定義名為 the_item 的全域變數  
    the_item = something  
    return something  
  
_ = q3.apply(debug)  
the_item
```

Out

```
'Another'
```

請記住，Series 中的項目會個別呼叫我們傳遞給 `apply()` 的 Python 函式。由於對單一項目進行運算是緩慢的做法，因此應該盡可能避免這樣做。在下一節，我們將示範另一個提升 `apply()` 速度的選項。

C.3 提高 `apply()` 的效能

`apply()` 可搭配各種函式庫來實現平行運算。若想實現平行運算，最簡單的方法是嘗試進行**向量化**（vectorization）。數學運算在 Pandas 中是向量化的，如果將一個數字（5）加到數值 Series 中，Pandas 不會分別為 Series 中的每個值加上 5。相反的，它會利用現代多核心 CPU 的功能，一次性完成運算。

如果某個操作無法向量化（就像之前用來處理字串資料的 `limit_countries()` 函式），你還有其他選擇。本節將示範其中的一些做法。請注意，此處使用的函式庫並沒有包含在 Pandas 中，你需要額外進行安裝。

動手做

01 匯入並初始化 **Pandarallel 函式庫**，它使用了標準函式庫中的 `multiprocessing` 函式庫，會以平行的方式來進行運算。請注意，這個函式庫在 Linux 和 Mac 上可以成功運行。然而，它利用了共享記憶體技術，因此無法在 Windows 上執行（除非你使用 Linux 的 Windows 子系統來執行 Python）：

In

```
from pandarallel import pandarallel
pandarallel.initialize() ←
```

在初始化函式庫時，可以透過 `nb_workers` 參數來指定欲使用的 CPU 數量（若不指定則預設為使用所有 CPU）

 此處我們將 Pandarallel 函式庫用在 DataFrame，不過它也適用於 groupby 物件和 Series 物件。

02 該函式庫為 DataFrame 擴增了一些額外的方法，例如以下所使用的 `parallel_apply()` 方法：

 In

```
def limit_countries(val):  
    if val in {'United States of America', 'India', 'China'}:  
        return val  
    return 'Another'  
  
%%timeit  
res_p = df.Q3.parallel_apply(limit_countries).rename('Country')
```

Out

133 ms ± 11.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

⚠ 平行化運算要額外做一些事，因此會產生 overhead。對比本節的各種平行化做法，直接使用 `apply()` 方法反而可以運行得比較快。過了某個臨界點後，這些額外的 overhead 才是有意義的。Pandarallel 函式庫通常用在有至少 100 萬筆樣本的資料集上，而我們的資料集遠小於這個數字。因此，直接使用 `apply()` 方法的用時反而少了許多。

03 讓我們匯入另一個 **swifter** 函式庫：

 In

```
import swifter
```

04 該函式庫可為 DataFrame 和 Series 增加 **swifter** 屬性，其採用不同做法來提升程式運行速度。它會先檢查能否進行向量化運算，若無法則會查看 Pandas（在一小部分樣本上）的運算時間。接下來，便可決定要使用 Dask 函式庫（詳見步驟 5 和 6）或沿用 Pandas 就好。同樣的，決定運行方式的過程會產生 overhead，因此盲目地使用該函式庫無法最大化程式碼的效率：

In

```
%%timeit  
res_s = df.Q3.swifter.apply(limit_countries).rename('Country')
```

Out

187 ms \pm 31.4 ms per loop (mean \pm std. dev. of 7 runs, 10 loops each)

05 匯入 Dask 函式庫：

In

```
import dask
```

06 使用 Dask 的 `map_partitions()` 函式：

In

```
%%timeit  
res_d = (dask.dataframe.from_pandas(df, npartitions=4)  
         .map_partitions(lambda df: df.Q3.apply(limit_countries))  
         .rename('Countries'))
```

Out

29.1 s \pm 1.75 s per loop (mean \pm std. dev. of 7 runs, 1 loop each)



請注意，載入資料和利用函式庫提供的平行化運算會產生 overhead。許多 Dask 使用者完全放棄了 Pandas，因為 Dask 可提供與 Pandas 類似的功能，而且允許將處理工作擴展到大數據上（也可在電腦叢集上執行）

07 現在，我們來使用 NumPy 的 **vectorize()** 函式。它可將任意的 Python 函式轉換成一個 NumPy 的 ufunc（一個對 NumPy 陣列進行運算的 universal 函式），並嘗試利用 NumPy 的**擴張規則**（broadcasting rules）：


 In

```
np_fn = np.vectorize(limit_countries)

%%timeit
res_v = df.Q3.apply(np_fn).rename('Country')
```

Out

```
643 ms ± 86.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

 Swifter 網站有一個 notebook，其中包含 Swifter、np.vectorize、Dask 和 Pandas 之間的比較（對不同類型的函式都進行了廣泛的基準測試）。以非向量化函式來說（例如我們的 limit_countries() 函式，因為它具有一般的 Python 邏輯），在資料列數達到 100 萬列以上後，原始的 Pandas apply() 方法才會開始在效能上落後。

08 匯入 Numba 函式庫並用 **jit 修飾器**（decorator）來修飾 limit_countries() 函式：

 In

```
from numba import jit

@jit
def limit_countries2(val):
    if val in ['United States of America', 'India', 'China']:
        return val
    return 'Another'
```


09 使用修飾過的函式：

 In

```
%%timeit  
res_n = df.Q3.apply(limit_countries2).rename('Country')
```

Out

```
158 ms ± 16.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

本節展示的選項可以幫助較大的資料集提升效能。在我們的例子中，盲目地應用它們反而會增加程式碼的運行時間。

C.4 快速檢視程式碼的技巧

Jupyter 環境有一個延伸模組，可讓你快速找出類別、方法及函式的說明文件或程式碼，強烈建議你要養成使用這些模組的習慣。如果可以在 Jupyter 環境中解決可能出現的問題，將會提高你的生產力。

接下來，我們將示範如何查看 `apply()` 的原始碼。在本書中，我們一直強烈推薦對 Pandas 物件進行**串連** (chaining) 運算。遺憾的是，Jupyter (以及任何其它的編輯環境) 無法對『以串連方法呼叫』所傳回的中介物件，實現程式碼自動完成 (code completion) 或查看說明文件。因此，建議你直接在未串連的方法上進行查詢。

01 讀入 Kaggle 的問卷資料集：

 In

```
import zipfile
url = 'data/kaggle-survey-2018.zip'

with zipfile.ZipFile(url) as z:
    kag = pd.read_csv(z.open('multipleChoiceResponses.csv'))
    df = kag.iloc[1:]
```

02 Jupyter 能夠同時查看 Python 物件的 docstrings 和原始碼。標準 Python REPL 可以利用內建的 **help 函式** 來查看 docstrings，但無法顯示原始碼。在 Jupyter 中，只要把問號 (?) 加在函式或方法的後面，就能顯示其說明文件。請注意，這不是有效的 Python 語法，而是 Jupyter 所提供的一個功能。如果後面加的是兩個問號 (??)，那麼 Jupyter 就會顯示函式或方法的原始碼。

現在，讓我們用 Jupyter 的『?』延伸模組來查詢 `apply()` 的說明文件（你也可以按 4 次 `[Shift]+[Tab]` 鍵以在 Jupyter 中得到該文件）：

 In

```
df.Q3.apply?
```

Out

```
Signature: df.Q3.apply(func, convert_dtype=True, args=(), **kwargs)
Docstring:
Invoke function on values of Series.
Can be ufunc (a NumPy function that applies to the entire Series)
or a Python function that only works on single values.
Parameters
-----
func : function
```

```

    Python function or NumPy ufunc to apply.
convert_dtype : bool, default True
    Try to find better dtype for elementwise function results. If
    False, leave as dtype=object.
args : tuple
    Positional arguments passed to func after the series value.
**kwds
    Additional keyword arguments passed to func.
Returns
-----
Series or DataFrame
    If func returns a Series object the result will be a DataFrame.
See Also
-----
Series.map: For element-wise operations.
Series.agg: Only perform aggregating type operations.
Series.transform: Only perform transforming type operations.
Examples
...
File: c:\users\admin\anaconda3\lib\site-packages\pandas\core\series.py
Type: method

```

03 讓我們用 ?? 來查看原始碼（此處無法透過 `Shift` + `Tab` 快捷鍵來完成）。我們還可以看到字串函式是如何運作的。getattr 函式從 DataFrame 中取得相對應的方法。接下來，程式碼會檢查是否正在處理 NumPy 函式。最後，如果它是 np.ufunc() 的一個實例，它會呼叫該函式，否則它會在底層的 _values 屬性上呼叫 map()，或者呼叫 lib.map_infer()：

In

```
df.Q3.apply??
```

Out

```
Signature: df.Q3.apply(func, convert_dtype=True, args=(), **kwds)
```

```
Source:
```

```
def apply(self, func, convert_dtype=True, args=(), **kwds):
```

```
...
```

```

if len(self) == 0:
    return self._constructor(dtype=self.dtype, index=self.index).__finalize__(
        self
    )
# dispatch to agg
if isinstance(func, (list, dict)):
    return self.agg(func, *args, **kwargs)
# if we are a string, try to dispatch
if isinstance(func, str):
    return self._try_agg_string_function(func, *args, **kwargs)
# handle ufuncs and lambdas
if kwargs or args and not isinstance(func, np.ufunc):
    def f(x):
        return func(x, *args, **kwargs)
else:
    f = func
with np.errstate(all="ignore"):
    if isinstance(f, np.ufunc):
        return f(self)
    # row-wise access
    if is_extension_type(self.dtype):
        mapped = self._values.map(f)
    else:
        values = self.astype(object).values
        mapped = lib.map_infer(values, f, convert=convert_dtype)
if len(mapped) and isinstance(mapped[0], Series):
    # GH 25959 use pd.array instead of tolist
    # so extension arrays can be used
    return self._constructor_expanddim(pd.array(mapped), index=self.index)
else:
    return self._constructor(mapped, index=self.index).__finalize__(self)
File: c:\users\admin\anaconda3\lib\site-packages\pandas\core\series.py
Type: method

```

04 從輸出可見，`apply()` 方法會試圖找出合適的程式碼來呼叫。如果這些都失敗了，它最終會計算 `mapped` 變數。讓我們試著瞭解 `lib.map_infer()` 做了什麼：

 In

```
import pandas.core.series
pandas.core.series.lib
```

Out


```
<module 'pandas._libs.lib' from '.env/364/lib/python3.6/sitepackages/ pandas/_
libs/lib.cpython-36m-darwin.so'>
```

 In

```
pandas.core.series.lib.map_infer??
```

Out

```
Docstring:
Substitute for np.vectorize with pandas-friendly dtype inference
Parameters
-----
arr : ndarray
f : function
Returns
-----
mapped : ndarray
Type: builtin_function_or_method
```

 `lib.map_infer()` 是一個 **so 檔** (在 Windows 上則是 **pyd 檔**)。這是一個編譯後的檔案，通常是用 C 編寫 Python 或使用 Cython 的結果，Jupyter 無法顯示編譯後檔案的原始碼。

了解更多

在查看函式或方法的原始碼時，Jupyter 會在視窗底部顯示它所屬的檔案。如果真的要深入研究原始碼，你可以在 Jupyter 以外的編輯器來開啟它。然後，便可以用編輯器來瀏覽該程式碼和任何相應的程式碼（大多數編輯器會比 Jupyter 具有更好的程式碼瀏覽功能）。

C.5 在 Jupyter 中除錯

前面的例子已經示範了如何解讀 Pandas 程式碼，並在 Jupyter 中檢視它們。在本節，我們會說明如何在 Jupyter 中使用 IPython 除錯工具 (ipdb)。

接下來，我們將建立與 `apply()` 一起使用時會引發錯誤的函式，並使用 `ipdb` 來對這個函式進行除錯。

動手做

01 讀入 Kaggle 的問卷資料集：

 In

```
import zipfile
url = 'data/kaggle-survey-2018.zip'

with zipfile.ZipFile(url) as z:
    kag = pd.read_csv(z.open('multipleChoiceResponses.csv'))
    df = kag.iloc[1:]
```

02 嘗試運行可為 Series 加上 1 的函式：

 In

```
def add1(x): ← x 為一 Series
    return x + 1
```

```
df.Q3.apply(add1)
```

Out

TypeError

Traceback (most recent call last)

```

<ipython-input-76-c52fc69777f1> in <module>
----> 1 df.Q3.apply(add1)

~\anaconda3\lib\site-packages\pandas\core\series.py in apply(self, func,
convert_dtype, args, **kwargs)
    3846         else:
    3847             values = self.astype(object).values
-> 3848             mapped = lib.map_infer(values, f, convert=convert_dtype)
    3849
    3850             if len(mapped) and isinstance(mapped[0], Series):

pandas\_libs\lib.pyx in pandas._libs.lib.map_infer()

<ipython-input-75-f4e0c9584bd4> in add1(x)
      1 def add1(x):
----> 2     return x + 1

TypeError: can only concatenate str (not "int") to str

```

03 在出現異常後，立即用 **%debug** 進入除錯視窗。這會在出現異常的地方開啟除錯工具。

你可以用除錯工具的指令在堆疊中進行瀏覽：輸入『u』會把堆疊 pop 到呼叫當前行的函式。此外，使用列印指令 (p) 能檢視物件：

```
In [*]: %debug

> <ipython-input-9-6ce28d2fea57>(2)add1()
   1 def add1(x):
→  2     return x + 1
   3
   4 df.Q3.apply(add1)

ipdb> p x
'United States of America' ← 內容為字串，無法做 +1 的運算
ipdb> u
> /Users/matt/.env/364/lib/python3.6/site-packages/pandas/core/series.py(4045)apply()
   4043         else:
   4044             values = self.astype(object).values
→  4045             mapped = lib.map_infer(values, f, convert=convert_dtype)
   4046
   4047             if len(mapped) and isinstance(mapped[0], Series):

ipdb> p self
1      United S...
2      Indonesia
3      United S...
4      United S...
5      India
...
23855   France
23856   Turkey
23857   Turkey
23858   United K...
23859   Spain
Name: Q3, Length: 23859, dtype: object

ipdb> 
```

圖 C.1 在 Jupyter 中使用 %debug 可進入除錯視窗。



小編補充

輸入『exit』即可關閉除錯視窗。

04 如果想在不出現異常警告的情況下檢視程式碼，可以使用 IPython 除錯工具中的 `set_trace()` 函式立即進入除錯視窗：

```
> <ipython-input-11-cb997d0cb281>(5)add1()
   3 def add1(x):
   4     set_trace()
→  5     return x + 1
   6
   7 df.Q3.apply(add1)

ipdb> 
```

圖 C.2 使用 set_trace() 函式直接進入除錯視窗。

了解更多

Jupyter (源自 IPython) 帶有 IPython 除錯工具。它複製了標準函式庫中 **pdb** 模組的功能，而且具有**語法突顯** (syntax highlighting) 等細部功能 (pdb 還具有以 **tab** 鍵自動完成字串的功能，但這在 Jupyter 中不起作用，僅適用於 IPython 主控台)。

如果你對除錯工具並不熟悉，這裡可以提供一個小竅門：指令『h』會印出所有能在除錯工具中執行的指令：

```
ipdb> h
Documented commands (type help <topic>):
=====
EOF      cl          disable  interact  next      psource   rv        undisplay
a         clear       display  j          p          q          s          unt
alias    commands   down     jump      pdef      quit      skip_hidden until
args     condition  enable   l          pdoc      r          source    up
b         cont       exit     list      pfile     restart   step      w
break    continue  h        ll         pinfo     return    tbreak    whatis
bt       d          help     longlist  pinfo2    retval    u          where
c         debug     ignore   n          pp         run       unalias
```

作者最常用的指令是 s、n、l、u、d 和 c。如果想知道 s 的功能，請輸入 h s：

```
ipdb> h s
s(tep)

Execute the current line, stop at the first possible occasion
(either in a function that is called or in the current
function).
```

這個指令告訴除錯工具印出 step(s) 的 help(h) 說明文件。一般來說，我們在 Jupyter 中是拆成多個小步驟來編寫程式碼，因此使用除錯工具有點大材小用。不過，知道如何使用除錯工具還是很有用的，特別是在你想檢視 Pandas 的原始碼並了解內部運作原理的時候。

C.6 以 Great Expectations 來管理資料完整性

Great Expectations 函式庫是第三方工具，可以捕捉及定義資料集的特性。我們可以利用這些特性來驗證資料，確保資料符合預期。這在建構機器學習模型時非常有用，因為新建立的分類資料容易出錯，或離群值可能會導致沒預期到的問題，這些都會影響模型運作（**編註**：所謂的特性也可視為使用者對資料的限制，如資料型別、數值大小範圍、有無缺失值等。在此假設的限制下，再去建構處理資料的程式碼。為了確保程式可以順利執行，因此透過 Great Expectations 函式庫來進行驗證）。

該函式庫拓展了 DataFrame 的功能，我們可以用它來驗證原始或經調整過的資料。在本節，我們將使用 Kaggle 資料集並製作一整套的 expectations 來測試和驗證資料。

動手做

01 使用第 1 節定義的 `tweak_kag()` 函式來讀取資料：

 In

```
kag = tweak_kag(df)
```

02 使用 Great Expectations 的 `from_pandas()` 函式讀入一個 Great Expectations DataFrame (DataFrame 的子類別，具有一些額外的方法)：

 In

```
import great_expectations as ge
kag_ge = ge.from_pandas(kag)
```

03 檢查該 DataFrame 的額外方法：

 In

```
sorted([x for x in set(dir(kag_ge)) - set(dir(kag))
        if not x.startswith('_')])
```

Out

```
['autoinspect',
 'batch_fingerprint',
 'batch_id',
 'batch_kwargs',
 'column_aggregate_expectation',
 ...
 'set_evaluation_parameter',
 'test_column_aggregate_expectation_function',
 'test_column_map_expectation_function',
 'test_expectation_function',
 'validate']
```

04 Great Expectations 對表格形狀 (shape)、缺失值、型別、範圍、字串、日期、聚合函式、欄位對 (column pairs)、分佈和檔案特性都具有 expectations，讓我們使用其中的一些 expectations。在過程中，函式庫會追蹤我們所使用的 expectations，稍後便可以將它們保存成一整套的 expectations (suite of expectations)：

 In

```
kag_ge.expect_column_to_exist('Salary') ← 預期 kag_ge 存在一個 Salary 欄位
```

Out

```
{
  "success": true, ← 顯示 true，代表符合預期 (若不符合預期會顯示 false)
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
  }
}
```

```
"exception_message": null
},
"result": {}
}
```

 In

```
kag_ge.expect_column_mean_to_be_between(
    'Salary', min_value=10_000, max_value=100_000)
```

← 預期 Salary 欄位的平均數
在 10000 到 100000 之間

Out

```
{
  "success": true,
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "observed_value": 43869.66102793441, ← Salary 欄位的平均值
    "element_count": 15429, ← Salary 欄位內的項目數量
    "missing_count": null,
    "missing_percent": null
  }
}
```

 In

```
kag_ge.expect_column_values_to_be_between(
    'Salary', min_value=0, max_value=500_000)
```

← 預期 Salary 欄位的資料值
在 0 到 500000 之間

Out

```
{
  "success": true,
  "meta": {},
  "exception_info": {
    "raised_exception": false,
```

```

    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "element_count": 15429,
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0,
    "partial_unexpected_list": []
  }
}

```

 In

```
kag.ge.expect_column_values_to_not_be_null('Salary') ←
```

預期 Salary 欄位中沒有缺失值

Out

```

{
  "success": true, ← 結果顯示，Salary 欄位中的確沒有缺失值
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "element_count": 15429,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "partial_unexpected_list": []
  }
}

```

In

```
kag_ge.expect_column_values_to_match_regex(
    'Country', r'America|India|Another|China')
```

預期 Country 欄位中只會有『America』、『India』、『Another』和『China』這 4 種字串

Out

```
{
  "success": true,
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  },
  "result": {
    "element_count": 15429,
    "missing_count": 0,
    "missing_percent": 0.0,
    "unexpected_count": 0,
    "unexpected_percent": 0.0,
    "unexpected_percent_total": 0.0,
    "unexpected_percent_nonmissing": 0.0,
    "partial_unexpected_list": []
  }
}
```

In

```
kag_ge.expect_column_values_to_be_of_type(
    'Salary', type_='int')
```

預期 Salary 欄位的型別為 int

Out

```
{
  "success": true,
  "meta": {},
  "exception_info": {
    "raised_exception": false,
    "exception_traceback": null,
    "exception_message": null
  }
}
```

```
    },  
    "result": {  
        "observed_value": "int32"  
    }  
}
```

05 將以上的 expectations 存成名為 kaggle_expectations 的 JSON 檔案中：

 In

```
kag_ge.save_expectation_suite('kaggle_expectations.json')
```

JSON 檔案會長成以下的樣子：

```
{  
  "data_asset_name": null,  
  "expectation_suite_name": "default",  
  "meta": {  
    "great_expectations.__version__": "0.8.6"  
  },  
  "expectations": [  
    {  
      "expectation_type": "expect_column_to_exist",  
      "kwargs": {  
        "column": "Salary"  
      }  
    },  
    {  
      "expectation_type": "expect_column_mean_to_be_between",  
      "kwargs": {  
        "column": "Salary",  
        "min_value": 10000,  
        "max_value": 100000  
      }  
    },  
    ...  
    {  
      "expectation_type": "expect_column_values_to_be_of_type",  
      "kwargs": {
```

```

        "column": "Salary",
        "type_": "int"
    }
}
],
"data_asset_type": "Dataset"
}

```

06 利用 `kaggle_expectations.json` 評估 CSV 檔案。我們把步驟 2 得到的 `kag_ge` 儲存至一個 CSV 檔案並測試它（**編註**：此處會一次過測試步驟 4 的所有 expectations）：

In

```

kag_ge.to_csv('kag.csv')
import json
ge.validate(ge.read_csv('kag.csv'),
            expectation_suite=json.load(
                open('kaggle_expectations.json')))

```

Out

```

{
  "results": [
    {
      "success": true,
      "expectation_config": {
        "kwargs": {
          "column": "Salary"
        },
        "meta": {},
        "expectation_type": "expect_column_to_exist"
      },
      "meta": {},
      "exception_info": {
        "raised_exception": false,
        "exception_message": null,
        "exception_traceback": null
      },
      "result": {}
    }
  ]
}

```



```

    },
... ← 省略部分輸出
    "statistics": {
        "evaluated_expectations": 6,
        "successful_expectations": 6,
        "unsuccessful_expectations": 0,
        "success_percent": 100.0
    },
    "meta": {
        "great_expectations_version": "0.13.33",
        "expectation_suite_name": "default",
        "run_id": {
            "run_name": null,
            "run_time": "2021-09-29T04:07:02.564281+00:00"
        },
        "batch_kwargs": {
            "ge_batch_id": "b35a2a4f-20da-11ec-ba49-d6f8d231edf4"
        },
        "batch_markers": {},
        "batch_parameters": {},
        "validation_time": "20210929T040702.564281Z",
        "expectation_suite_meta": {
            "great_expectations_version": "0.13.33"
        }
    }
}

```

C.7 使用 pytest 來測試 Pandas

在本節，我們將透過測試程式碼的產物（artifacts），進而檢驗你的 Pandas 程式碼。我們會使用第三方的 **pytest 函式庫** 來進行此測試。

在以下的例子中，我們將改用命令列來運行程式。

01 Pytest 函式庫支援不同風格的專案佈局 (layout)，我們將建立如下所示的資料夾結構 (編註：也就是前述的佈局，此處的 kag-demo-pytest 資料夾可由本書封面所示的網址進行下載)：

```
kag-demo-pytest/  
├── data  
│   └── kaggle-survey-2018.zip  
├── kag.py  
├── test  
│   └── test_kag.py
```

kag.py 檔案包含載入原始資料，以及調整原始資料的程式碼。其內容如下：

 In

```
import pandas as pd  
import zipfile  
  
def load_raw(zip_fname): ← 載入原始資料的函式  
    with zipfile.ZipFile(zip_fname) as z:  
        kag = pd.read_csv(z.open('multipleChoiceResponses.csv'))  
        df = kag.iloc[1:]  
    return df  
  
def tweak_kag(df): ← 調整原始資料的函式 (詳情可參考本章第 1 節)  
    na_mask = df.Q9.isna()  
    hide_mask = df.Q9.str.startswith('I do not').fillna(False)  
    df = df[~na_mask & ~hide_mask]  
  
    q1 = (df.Q1  
        .replace({'Prefer not to say': 'Another',  
                  'Prefer to self-describe': 'Another'})  
        .rename('Gender')  
    )  
  
    q2 = df.Q2.str.slice(0,2).astype(int).rename('Age')
```

```
def limit_countries(val):
    if val in {'United States of America', 'India', 'China'}:
        return val
    return 'Another'
q3 = df.Q3.apply(limit_countries).rename('Country')

q4 = (df.Q4
      .replace({'Master' s degree': 18,
                'Bachelor' s degree': 16,
                'Doctoral degree': 20,
                'Some college/university study without earning a bachelor' s degree': 13,
                'Professional degree': 19,
                'I prefer not to answer': None,
                'No formal education past high school': 12})
      .fillna(11)
      .rename('Edu')
      )

def only_cs_stat_val(val):
    if val not in {'cs', 'eng', 'stat'}:
        return 'another'
    return val

q5 = (df.Q5
      .replace({
          'Computer science (software engineering, etc.)': 'cs',
          'Engineering (non-computer focused)': 'eng',
          'Mathematics or statistics': 'stat'})
      .apply(only_cs_stat_val)
      .rename('Studies'))

def limit_occupation(val):
    if val in {'Student', 'Data Scientist', 'Software Engineer', 'Not employed',
              'Data Engineer'}:
        return val
    return 'Another'

q6 = df.Q6.apply(limit_occupation).rename('Occupation')
```

```

q8 = (df.Q8
      .str.replace('+', '')
      .str.split('-', expand=True)
      .iloc[:,0]
      .fillna(-1)
      .astype(int)
      .rename('Experience')
      )

q9 = (df.Q9
      .str.replace('+', '')
      .str.replace(',', '')
      .str.replace('500000', '500')
      .str.replace('I do not wish to disclose my approximate yearly compensation', '')
      .str.split('-', expand=True)
      .iloc[:,0]
      .astype(int)
      .mul(1000)
      .rename('Salary'))
return pd.concat([q1, q2, q3, q4, q5, q6, q8, q9], axis=1)

```

test_kag.py 檔案中包含了數個指定的測試。只要某函式的名稱以『test_』開頭，就代表是一個測試函式，它們的參數（df 函式）稱為一個 fixture：



```

import pytest
import kag

```

```

@pytest.fixture(scope='session')

```

```

def df():

```

```

    df = kag.load_raw('data/kaggle-survey-2018.zip')

```

```

    return kag.tweak_kag(df)

```

任何參數為 df 的測試函式都會接收該輸出

在測試階段開始時，
會呼叫一次該函式



df() 的範圍 (scope) 被指定為 session，因此（在整個測試期間）只會載入一次資料。如果我們沒有指定範圍，fixture 的作用範圍預設是函式層級。若處於函式層級範圍，每個使用 fixture 為參數的測試函式都會執行一次 fixture，造成測試的執行時間為原先的 4 倍左右。

```
def test_salary_mean(df):
    assert 10_000 < df.Salary.mean() < 100_000

def test_salary_between(df):
    assert df.Salary.min() >= 0
    assert df.Salary.max() <= 500_000

def test_salary_not_null(df):
    assert not df.Salary.isna().any()

def test_country_values(df):
    assert set(df.Country.unique()) == {'Another', 'United States of America',
                                         'India', 'China'}

def test_salary_dtype(df):
    assert df.Salary.dtype == int
```

02 在 kag-demo-pytest 的所在目錄執行測試。如果你安裝了 pytest 函式庫，會有一個可執行的『pytest』指令。執行該指令後，將會出現錯誤：

```
(env)$ pytest
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-3.10.1, py-1.7.0,
pluggy-0.8.0
rootdir: /Users/matt/pandas-cookbook/kag-demo, inifile:
plugins: asyncio-0.10.0
collected 0 items / 1 errors
===== ERRORS =====
_____ ERROR collecting test/test_kag.py _____
ImportError while importing test module '/Users/matt/pandascookbook/
kag
demo/test/test_kag.py'.
Hint: make sure your test modules/packages have valid Python
names.
Traceback:
test/test_kag.py:3: in <module>
import kag
E ModuleNotFoundError: No module named 'kag'
!!!!!!! Interrupted: 1 errors during collection !!!!!!!
===== 1 error in 0.15 seconds =====
```

以上錯誤是因為 pytest 想要使用已安裝的程式碼來執行測試。由於我們沒有使用過 pip（或其他機制）來安裝 kag.py，因此在安裝程式碼的位置找不到該模組。

03 讓 pytest 找到 kag.py 檔案的一個方法是將 pytest 作為模組來引用。讓我們改成執行以下指令：

```
$ python -m pytest
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-3.10.1, py-1.7.0,
pluggy-0.8.0
rootdir: /Users/matt/pandas-cookbook/kag-demo, inifile:
collected 5 items
test/test_kag.py ..... [100%]
...
===== 5 passed, 1 warnings in 3.51 seconds =====
```

用這種方式引用 pytest，便可把當前目錄增加到 PYTHONPATH，進而成功匯入 kag 模組。

了解更多

你也可以從 pytest 執行 Great Expectations 測試，只需把以下函式加到 test_kag.py 中（**編註**：需先將上一節的 kaggle_expectations.json 放至 kag-demo-pytest 的資料夾中）：

 In

```
def test_ge(df):
    import json
    import great_expectations as ge
    res = ge.validate(ge.from_pandas(df),
                      expectation_suite=json.load(open('kaggle_expectations.json')))
    failures = []
    for exp in res['results']:
```

```
if not exp['success']:
    failures.append(json.dumps(exp, indent=2))
if failures:
    assert False, '\n'.join(failures)
else:
    assert True
```

C.8 使用 Hypothesis 產生測試

Hypothesis 函式庫可以產生測試（**編註**：可理解為產生測試資料，比較好懂），或執行基於特性測試（property-based testing）的第三方函式庫。我們會建立一個 strategy 物件（可產生測試用的樣本資料），然後以其輸入的資料來跑程式碼，確認是否執行狀態。通常會用來測試一個不變量（invariant，程式中不該改變的數值）或是資料中應該一直成立的條件等（**編註**：也就是第 6 節所提符合某個資料特性）。Hypothesis 的功能繁複，本節我們只會用一個小例子大致展示其用途。

我們會展示如何生成 Kaggle 問卷資料，然後將 tweak_kag() 函式套用在生成的資料上，進而驗證該函式是否適用於**新資料**。

本節將沿用上一節的測試碼，而 Hypothesis 函式庫與 pytest 是彼此相容的，所以我們可使用相同的佈局（**編註**：讀者別忘了要先安裝 Hypothesis 函式庫）。

動手做

01 建立一個專案資料夾佈局。與上一節相比，此處多了一個 test_hypot.py 檔案和 conftest.py 檔案（**編註**：kag-demo-hypo 資料夾同樣可由本書封面所示的網址進行下載）：

```
kag-demo-hypo/
├── data
│   └── kaggle-survey-2018.zip
├── kag.py
├── test
│   ├── conftest.py
│   ├── test_hypot.py
│   └── test_kag.py
```

02 我們把共享的 fixture 放到 `conftest.py` 檔案中，該檔案是 `pytest` 在查找 fixture 時所需要的特殊檔案。我們不需匯入 `conftest.py`，但在其中定義的任何 fixture 都可供其他測試檔案使用。

把 fixture 程式碼從 `test_kag.py` 移到 `conftest.py`。我們會稍微重構程式碼，進而建立 `raw_` 函式。該函式並非我們可在測試函式之外呼叫的 fixture：



```
import pytest
import kag

@pytest.fixture(scope='session')
def raw():
    return raw_()

def raw_():
    return kag.load_raw('data/kaggle-survey-2018.zip')

@pytest.fixture(scope='session')
def df(raw):
    return kag.tweak_kag(raw)
```


把以下程式放到 test_hypot.py 中：

```
from hypothesis import given, strategies
from hypothesis.extra.pandas import column, data_frames

from conftest import raw_

import kag

def hypot_df_generator():
    df = raw_()
    cols = []
    for col in ['Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q8', 'Q9']:
        cols.append(column(col, elements=strategies.sampled_from(df[col].unique())))
    return data_frames(columns=cols)

@given(hypot_df_generator())
def test_countries(gen_df):
    if gen_df.shape[0] == 0:
        return
    kag_ = kag.tweak_kag(gen_df)
    assert len(kag_.Country.unique()) <= 4
```

hypot_df_generator() 函式會建構 Hypothesis 的 strategy 物件。您可以指定讓 strategy 物件產生不同類型的資料（**編註**：除了各種型別外，還包括 Email、分數、IP 位址…等），也可以不指定類型，由使用者自行提供各種資料給這些 strategy。在這個例子中，我們使用現有的 CSV 檔案，透過 strategy 物件把各種不同的值放入欄位中。

test_countries() 函式是一個用 @given(hypot_df_generator()) 修飾器進行修飾的 pytest 測試。修飾器會把 gen_df 物件傳遞給測試函式，該物件將是一個符合 strategy 規範的 DataFrame，我們會用這個 DataFrame 來測試程式中的不變量，也就是 Country 欄位中的國家數量。以下我們將執行 tweak_kag() 函式來確保 Country 欄位中的國家數量是小於或等於 4。

03 到 kag_demo-hypo 目錄並執行測試。以下指令只會執行 test_countries 的測試：

```
$ python -m pytest -k test_countries
```

輸出如下：

```
===== test session starts =====
platform darwin -- Python 3.6.4, pytest-5.3.2, py-1.7.0,
pluggy-0.13.1
rootdir: /Users/matt/kag-demo
plugins: asyncio-0.10.0, hypothesis-5.1.2
collected 6 items / 5 deselected / 1 selected
test/test_hypot.py F [100%]
===== FAILURES =====
_____ test_countries _____
@given(hypot_df_generator())
> def test_countries(gen_df):
test/test_hypot.py:19:
-----
test/test_hypot.py:23: in test_countries
    kag_ = kag.tweak_kag(gen_df)
kag.py:63: in tweak_kag
    q8 = (df.Q8
/Users/matt/.env/364/lib/python3.6/site-packages/pandas/core/
generic.py:5175: in
    __getattr__
return object.__getattr__(self, name)
/Users/matt/.env/364/lib/python3.6/site-packages/pandas/core/
accessor.py:175: in
    __get__
accessor_obj = self._accessor(obj)
/Users/matt/.env/364/lib/python3.6/site-packages/pandas/core/
strings.py:1917: in __init__
    self._inferred_dtype = self._validate(data)
-----
data = Series([], Name: Q8, dtype: float64)
@staticmethod
```

```

def _validate(data):
    """
    Auxiliary function for StringMethods, infers and checks
    dtype of data.
    This is a "first line of defence" at the creation of the
    StringMethodsobject
    (see _make_accessor), and just checks that the
    dtype is in the
    *union* of the allowed types over all string methods
    below; this
    restriction is then refined on a per-method basis using
    the decorator
    @forbid_nonstring_types (more info in the corresponding
    docstring).
    This really should exclude all series/index with any nonstring
    values,
    but that isn't practical for performance reasons until we
    have a str
    dtype (GH 9343 / 13877)
    Parameters
    -----
    data : The content of the Series
    Returns
    -----
    dtype : inferred dtype of data
    """
    if isinstance(data, ABCMultiIndex):
        raise AttributeError(
            "Can only use .str accessor with Index, " "not
            MultiIndex"
        )
    # see _libs/lib.pyx for list of inferred types
    allowed_types = ["string", "empty", "bytes", "mixed",
                     "mixed-integer"]
    values = getattr(data, "values", data) # Series / Index
    values = getattr(values, "categories", values) #
    categorical / normal
    try:
        inferred_dtype = lib.infer_dtype(values, skipna=True)
    except ValueError:

```

```
# GH#27571 mostly occurs with ExtensionArray
inferred_dtype = None
if inferred_dtype not in allowed_types:
> raise AttributeError("Can only use .str accessor with
string " "values!")
E AttributeError: Can only use .str accessor with string
values!
/Users/matt/.env/364/lib/python3.6/site-packages/pandas/core/
strings.py:1967: AttributeError
----- Hypothesis -----
Falsifying example: test_countries(
gen_df= Q1 Q2 Q3 ...
Q6 Q8 Q9
0 Female 45-49 United States of America ... Consultant
NaN NaN
[1 rows x 8 columns],
)
===== 1 failed, 5 deselected, 1 warning in 2.23s =====
```

輸出中有很多混亂的訊息，仔細觀察後會發現，問題出在產生 Q8 欄位資料的程式碼。原因是 Q8 欄位中有一筆 NaN（缺失值，型別為 float）的資料。如果我們在該 DataFrame 上執行 `tweak_kag()`，Pandas 會推斷 Q8 欄位的型別為 float，進而在使用 str 存取器時發生錯誤（欄位值中至少有一個字串值時，才能使用該存取器）。

這是一個 bug 嗎？我們很難給出明確的答案，不過可以確定的是，只要我們的原始資料含有缺失值，那麼程式碼就無法運作。

了解更多

在編寫程式時，我們可能會見樹不見林。有時，我們需要退後一步，從不同的角度來看待事物。使用 Hypothesis 是實現此目標的一種方法。