导语

Python正迅速成为数据科学家偏爱的语言，这合情合理。它拥有作为一种编程语言广阔的生态环境以及众多优秀的科学计算库。如果你刚开始学习Python，可以先了解一下Python的学习路线。
在众多的科学计算库中，我认为Pandas对数据科学运算最有用。Pandas，加上Scikit-learn几乎能构成了数据科学家所需的全部工具。 本文旨在提供Python数据处理的12种方法。文中也分享了一些会让你的工作更加便捷的小技巧。
在继续推进之前，我推荐读者阅览一些关于数据探索 (data exploration)的代码。
为了帮助理解，本文用一个具体的数据集进行运算和操作。本文使用了贷款预测(loan prediction) 问题数据集，下载数据集请到http://datahack.analyticsvidhya.com/contest/practice-problem-loan-prediction。(http://datahack.analyticsvidhya.com/contest/practice-problem-loan-prediction。)

开始工作

首先我要导入要用的模块，并把数据集载入Python环境。

```
import pandas as pd

import numpy as np

data = pd.read_csv("train.csv", index_col="Loan_ID")
```

1.布尔索引(Boolean Indexing)

如何你想用基于某些列的条件筛选另一列的值，你会怎么做？例如，我们想要一个全部无大学学历但有贷款的女性列表。这里可以使用布尔索引。代码如下：

```
data.loc[(data["Gender"]=="Female") & (data["Education"]=="Not Graduate") & (data["Loan_Status"
]=="Y"), ["Gender","Education","Loan_Status"]]
```

	Gender	Education	Loan_Status
Loan_ID			
LP001155	Female	Not Graduate	Y
LP001669	Female	Not Graduate	Y
LP001692	Female	Not Graduate	Y
LP001908	Female	Not Graduate	Y
LP002300	Female	Not Graduate	Y
LP002314	Female	Not Graduate	Y
LP002407	Female	Not Graduate	Y
LP002489	Female	Not Graduate	Y
LP002502	Female	Not Graduate	Y
LP002534	Female	Not Graduate	Y
LP002582	Female	Not Graduate	Y
LP002731	Female	Not Graduate	Y
LP002757	Female	Not Graduate	Y
LP002917	Female	Not Graduate	Y

想了解更多请阅读 Pandas Selecting and Indexing

2.Apply函数

Apply是摆弄数据和创造新变量时常用的一个函数。Apply把函数应用于数据框的特定行/列之后返回一些值。这里的函数既可以是系统自带的也可以是用户定义的。例如，此处可以用它来寻找每行每列的缺失值个数：

```
#创建一个新函数:

def num_missing(x):

  return sum(x.isnull())

#Apply到每一列:

print "Missing values per column:"

print data.apply(num_missing, axis=0) #axis=0代表函数应用于每一列

#Apply到每一行:

print "\nMissing values per row:"

print data.apply(num_missing, axis=1).head() #axis=1代表函数应用于每一行
```

输出结果：

```
Missing values per column:
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64

Missing values per row:
Loan_ID
LP001002    1
LP001003    0
LP001005    0
LP001006    0
LP001008    0
dtype: int64
```

由此我们得到了想要的结果。
注意：第二个输出使用了head()函数，因为数据包含太多行。
想了解更多请阅读 Pandas Reference (apply)

3.替换缺失值

‘fillna()’ 可以一次解决这个问题。它被用来把缺失值替换为所在列的平均值/众数/中位数。

```
#首先导入一个寻找众数的函数：

from scipy.stats import mode

mode(data['Gender'])
```

输出: ModeResult(mode=array([‘Male’], dtype=object), count=array([489]))
返回了众数及其出现次数。记住，众数可以是个数组，因为高频的值可能不只一个。我们通常默认使用第一个：

```
mode(data['Gender']).mode[0]
```

```
'Male'
```

现在可以填补缺失值，并用上一步的技巧来检验。

```
#值替换:

data['Gender'].fillna(mode(data['Gender']).mode[0], inplace=True)

data['Married'].fillna(mode(data['Married']).mode[0], inplace=True)

data['Self_Employed'].fillna(mode(data['Self_Employed']).mode[0], inplace=True)

#再次检查缺失值以确认:

print data.apply(num_missing, axis=0)
```

```
Gender              0
Married             0
Dependents         15
Education           0
Self_Employed       0
ApplicantIncome     0
CoapplicantIncome   0
LoanAmount         22
Loan_Amount_Term   14
Credit_History     50
Property_Area       0
Loan_Status         0
dtype: int64
```

由此可见，缺失值确定被替换了。请注意这是最基本的替换方式，其他更复杂的技术，如为缺失值建模、用分组平均数（平均值/众数/中位数）填充，会在今后的文章提到。
想了解更多请阅读 Pandas Reference (fillna)

4.透视表

Pandas可以用来创建 Excel式的透视表。例如，“LoanAmount”这个重要的列有缺失值。我们可以用根据‘Gender’、‘Married’、‘Self_Employed’分组后的各组的均值来替换缺失值。每个组的 ‘LoanAmount’可以用如下方法确定：

```
#Determine pivot table

impute_grps = data.pivot_table(values=["LoanAmount"], index=["Gender","Married","Self_Employed"
], aggfunc=np.mean)

print impute_grps
```

```
                           LoanAmount
Gender Married Self_Employed
Female No      No            114.691176
               Yes           125.800000
       Yes     No            134.222222
               Yes           282.250000
Male   No      No            129.936937
               Yes           180.588235
       Yes     No            153.882736
               Yes           169.395833
```

想了解更多请阅读 Pandas Reference (Pivot Table)

5.多重索引

你可能注意到上一步骤的输出有个奇怪的性质。每个索引都是由三个值组合而成。这叫做多重索引。它可以帮助运算快速进行。
延续上面的例子，现在我们有了每个分组的值，但还没有替换。这个任务可以用现在学过的多个技巧共同完成。

```
#只在带有缺失值的行中迭代：

for i,row in data.loc[data['LoanAmount'].isnull(),:].iterrows():

  ind = tuple([row['Gender'],row['Married'],row['Self_Employed']])

  data.loc[i,'LoanAmount'] = impute_grps.loc[ind].values[0]

#再次检查缺失值以确认：

print data.apply(num_missing, axis=0)
```

```
Gender              0
Married             0
Dependents         15
Education           0
Self_Employed       0
ApplicantIncome     0
CoapplicantIncome   0
LoanAmount          0
Loan_Amount_Term    14
Credit_History      50
Property_Area       0
Loan_Status         0
dtype: int64
```

注：

多重索引需要在loc中用到定义分组group的元组(tuple)。这个元组会在函数中使用。
需要使用.values[0]后缀。因为默认情况下元素返回的顺序与原数据库不匹配。在这种情况下，直接指派会返回错误。

6. 二维表

这个功能可被用来获取关于数据的初始“印象”（观察）。这里我们可以验证一些基本假设。例如，本例中“Credit_History” 被认为对欠款状态有显著影响。可以用下面这个二维表进行验证：

```
pd.crosstab(data["Credit_History"],data["Loan_Status"],margins=True)
```

Loan_Status	N	Y	All
Credit_History			
0.0	82	7	89
1.0	97	378	475
All	192	422	614

这些数字是绝对数值。不过，百分比数字更有助于快速了解数据。我们可以用apply函数达到目的：

```
def percConvert(ser):

  return ser/float(ser[-1])

  pd.crosstab(data["Credit_History"],data["Loan_Status"],margins=True).apply(percConvert, axis=
1)
```

Loan_Status	N	Y	All
Credit_History			
0.0	0.921348	0.078652	1
1.0	0.204211	0.795789	1
All	0.312704	0.687296	1

现在可以很明显地看出，有信用记录的人获得贷款的可能性更高：有信用记录的人有80% 获得了贷款，没有信用记录的人只有 9% 获得了贷款。
但不仅仅是这样，其中还包含着更多信息。由于我现在知道了有信用记录与否非常重要，如果用信用记录来预测是否会获得贷款会怎样？令人惊讶的是，在614次试验中我们能预测正确460次，足足有75%！
如果此刻你在纳闷，我们要统计模型有什么用，我不会怪你。但相信我，在此基础上提高0.001%的准确率都是充满挑战性的。你是否愿意接受这个挑战？
注：对训练集而言是75% 。在测试集上有些不同，但结果相近。同时，我希望这个例子能让人明白，为什么提高0.05% 的正确率就能在Kaggle排行榜上跳升500个名次。
想了解更多请阅读Pandas Reference (crosstab)

感谢您阅读到这里，在下一篇文章中将继续为您介绍其余六个实用技巧，请持续关注数据工匠。
原作者：AARSHAY JAIN
翻译：王鹏宇
原文地址：
http://www.analyticsvidhya.com/blog/2016/01/12-pandas-techniques-python-data-manipulation/
(http://www.analyticsvidhya.com/blog/2016/01/12-pandas-techniques-python-data-manipulation/)

用 Python 做数据处理必看：12 个使效率倍增的 Pandas 技巧（上）http://datartisan.com/article/detail/80.html (http://datartisan.com/article/detail/80.html)

7 – 数据框合并

当我们有收集自不同来源的数据时，合并数据框就变得至关重要。假设对于不同的房产类型，我们有不同的房屋均价数据。让我们定义这样一个数据框：

```
prop_rates = pd.DataFrame([1000, 5000, 12000], index=['Rural','Semiurban','Urban'],columns=['rates'])

prop_rates
```

	rates
Rural	1000
Semiurban	5000
Urban	12000

现在可以把它与原始数据框合并：

```
data_merged = data.merge(right=prop_rates, how='inner',left_on='Property_Area',right_index=True, sort=False)

data_merged.pivot_table(values='Credit_History',index=['Property_Area','rates'], aggfunc=len)
```

```
Property_Area  rates
Rural          1000     179
Semiurban      5000     233
Urban          12000    202
Name: Credit_History, dtype: float64
```

这张透视表验证了合并成功。注意这里的 'values'无关紧要，因为我们只是单纯计数。
想了解更多请阅读Pandas Reference (merge)

8 – 给数据框排序

Pandas可以轻松基于多列排序。方法如下：

```
data_sorted = data.sort_values(['ApplicantIncome','CoapplicantIncome'], ascending=False)

data_sorted[['ApplicantIncome','CoapplicantIncome']].head(10)
```

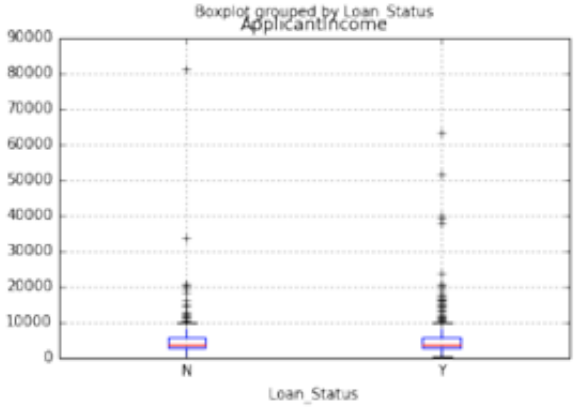
	ApplicantIncome	CoapplicantIncome
Loan_ID		
LP002317	81000	0
LP002101	63337	0
LP001585	51763	0
LP001536	39999	0
LP001640	39147	4750
LP002422	37719	0
LP001637	33846	0
LP001448	23803	0
LP002624	20833	6667
LP001922	20667	0

注：Pandas 的"sort"函数现在已经不推荐使用，我们用 "sort_values"函数代替。
想了解更多请阅读Pandas Reference (sort_values)

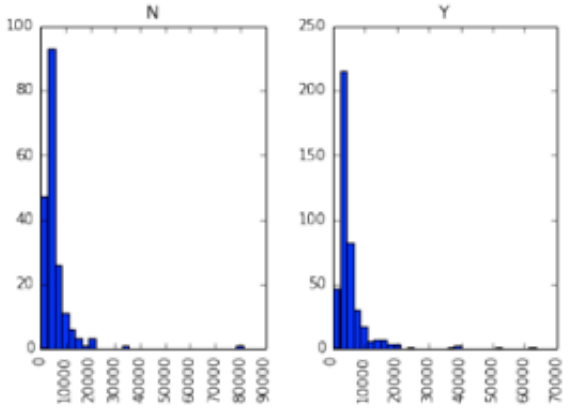
9 – 绘图（箱型图&直方图）

许多人可能没意识到Pandas可以直接绘制箱型图和直方图，不必单独调用matplotlib。只需要一行代码。举例来说，如果我们想根据贷款状态Loan_Status来比较申请者收入ApplicantIncome：

```
data.boxplot(column="ApplicantIncome",by="Loan_Status")
```




```
data.hist(column="ApplicantIncome",by="Loan_Status",bins=30)
```



可以看出获得/未获得贷款的人没有明显的收入差异，即收入不是决定性因素。
想了解更多请阅读Pandas Reference (hist) | Pandas Reference (boxplot)

10 – 用Cut函数分箱

有时把数值聚集在一起更有意义。例如，如果我们要为交通状况（路上的汽车数量）根据时间（分钟数据）建模。具体的分钟可能不重要，而时段如“上午”“下午”“傍晚”“夜间”“深夜”更有利于预测。如此建模更直观，也能避免过度拟合。
这里我们定义一个简单的、可复用的函数，轻松为任意变量分箱。

```
#分箱:

def binning(col, cut_points, labels=None):

  #Define min and max values:

  minval = col.min()

  maxval = col.max()

  #利用最大值和最小值创建分箱点的列表

  break_points = [minval] + cut_points + [maxval]

  #如果没有标签，则使用默认标签0 ... (n-1)

  if not labels:

    labels = range(len(cut_points)+1)

  #使用pandas的cut功能分箱

  colBin = pd.cut(col,bins=break_points,labels=labels,include_lowest=True)

  return colBin


#为年龄分箱:

cut_points = [90,140,190]

labels = ["low","medium","high","very high"]

data["LoanAmount_Bin"] = binning(data["LoanAmount"], cut_points, labels)

print pd.value_counts(data["LoanAmount_Bin"], sort=False)
```

```
low          104
medium       273
high         146
very high     91
dtype: int64
```

想了解更多请阅读 Pandas Reference (cut)

11 – 为分类变量编码

有时，我们会面对要改动分类变量的情况。原因可能是：

有些算法（如罗吉斯回归）要求所有输入项目是数字形式。所以分类变量常被编码为0, 1….(n-1)
有时同一个分类变量可能会有两种表现方式。如，温度可能被标记为“High”， “Medium”， “Low”，“H”，“low”。这里 “High” 和 “H”都代表同一类别。同理， “Low” 和“low”也是同一类别。但Python会把它们当作不同的类别。
一些类别的频数非常低，把它们归为一类是个好主意。

这里我们定义了一个函数，以字典的方式输入数值，用‘replace’函数进行编码。

```
#使用Pandas replace函数定义新函数：

def coding(col, codeDict):

  colCoded = pd.Series(col, copy=True)

  for key, value in codeDict.items():

    colCoded.replace(key, value, inplace=True)

  return colCoded


#把贷款状态LoanStatus编码为Y=1, N=0:

print 'Before Coding:'

print pd.value_counts(data["Loan_Status"])

data["Loan_Status_Coded"] = coding(data["Loan_Status"], {'N':0,'Y':1})

print '\nAfter Coding:'

print pd.value_counts(data["Loan_Status_Coded"])
```

```
Before Coding:
Y    422
N    192
Name: Loan_Status, dtype: int64

After Coding:
1    422
0    192
Name: Loan_Status_Coded, dtype: int64
```

编码前后计数不变，证明编码成功。
想了解更多请阅读 Pandas Reference (replace)

12 – 在一个数据框的各行循环迭代

这不是一个常见的操作。但你总不想卡在这里吧？有时你会需要用一个for循环来处理每行。例如，一个常见的问题是变量处置不当。通常见于以下情况：

带数字的分类变量被当做数值。
（由于出错）带文字的数值变量被当做分类变量。

所以通常来说手动定义变量类型是个好主意。如我们检查各列的数据类型：

```
#检查当前数据类型：

data.dtypes
```

```
Gender               object
Married              object
Dependents           object
Education            object
Self_Employed        object
ApplicantIncome       int64
CoapplicantIncome   float64
LoanAmount          float64
Loan_Amount_Term    float64
Credit_History      float64
Property_Area        object
Loan_Status          object
dtype: object
```

这里可以看到分类变量Credit_History被当作浮点数。对付这个问题的一个好办法是创建一个包含变量名和类型的csv文件。通过这种方法，我们可以定义一个函数来读取文件，并为每列指派数据类型。举例来说，我们创建了csv文件datatypes.csv (http://www.analyticsvidhya.com/wp-content/uploads/2015/12/datatypes.csv)。

```
#载入文件：

colTypes = pd.read_csv('datatypes.csv')

print colTypes
```

```
             feature         type
0            Loan_ID  categorical
1             Gender  categorical
2            Married  categorical
3         Dependents  categorical
4          Education  categorical
5      Self_Employed  categorical
6    ApplicantIncome   continuous
7  CoapplicantIncome   continuous
8         LoanAmount   continuous
9   Loan_Amount_Term   continuous
10    Credit_History  categorical
11     Property_Area  categorical
12       Loan_Status  categorical
```

载入这个文件之后，我们能对每行迭代，把用‘type’列把数据类型指派到‘feature’ 列对应的项目。

```
#迭代每行，指派变量类型。
#注，astype用来指定变量类型。
for i, row in colTypes.iterrows(): #i: dataframe索引; row: 连续的每行
  if row['feature']=="categorical":
    data[row['feature']]=data[row['feature']].astype(np.object)
  elif row['feature']=="continuous":
    data[row['feature']]=data[row['feature']].astype(np.float)
  print data.dtypes
```

现在信用记录这一列的类型已经成了‘object’，这在Pandas中代表分类变量。
想了解更多请阅读Pandas Reference (iterrows)

结语