@armantunga

#datascience

7 Tips for Effective Pandas

Arman Tunga

1. Chaining

Chaining is a powerful feature in Pandas that allows you to perform multiple operations in a single line of code.

```
df_updated = (df
              .query("release_year>2018")
              .loc[:, ["title", "release_year", "duration"]]
              .assign(over_three_hours=lambda dataframe:
np.where(dataframe["duration"] > 180, "Yes", "No"))
              .groupby(by=["release_year", "over_three_hours"])
              .count()
              )
df_updated

########## RESULT ##########
                              title  duration
release_year over_three_hours
2019         No                 995       995
             Yes                  1         1
2020         No                 867       867
             Yes                  1         1
2021         No                  31        31
```

2. nlargest and nsmallest

These functions are faster and more memory-efficient than sort_values(), making them a great choice for large datasets.

```
df.nsmallest(3, "age")  # Youngest 3 passengers
df.nlargest(3, "age")   # Oldest 3 passengers

########## nsmallest ##########
     survived  pclass     sex   age  ...  deck  embark_town  alive  alone
803         1       3    male  0.42  ...   NaN    Cherbourg    yes  False
755         1       2    male  0.67  ...   NaN  Southampton    yes  False
469         1       3  female  0.75  ...   NaN    Cherbourg    yes  False

########## nlargest ##########
     survived  pclass   sex   age  ...  deck  embark_town  alive alone
630         1       1  male  80.0  ...     A  Southampton    yes  True
851         0       3  male  74.0  ...   NaN  Southampton     no  True
96          0       1  male  71.0  ...     A    Cherbourg     no  True
```

3. Filtering Data With .query() Method

It allows you to filter your data using logical expressions. You can also use @ symbols to refer to variables in your query, making it a convenient and powerful tool for filtering data.

```
df["embark_town"].unique()  # ['Southampton', 'Cherbourg', 'Queenstown', nan]

embark_towns = ["Southampton", "Queenstown"]  # Only want to select these towns
df.query("age>21 & fare>250 & embark_town==@embark_towns").head()

########## RESULT ##########
     survived  pclass     sex   age  ...  deck  embark_town  alive  alone
88          1       1  female  23.0  ...     C  Southampton    yes  False
341         1       1  female  24.0  ...     C  Southampton    yes  False
438         0       1    male  64.0  ...     C  Southampton     no  False
[3 rows x 15 columns]
```

4. Using df.cut Method

The cut function is a useful tool for binning your data into discrete categories.

```
# Child - 0 to 9 years
# Teen - 10-19 years
# Young - 19 to 24 years
# Adult - 25 to 59
# Elderly > 59

# bins = [-float('inf'), 10, 19, 24, 59, float('inf')]
bins = [0, 10, 19, 24, 59, float('inf')]
labels = ["Child", "Teen", "Young", "Adult", "Elderly"]
df["age"].hist()
plt.show()
df["age_category"] = pd.cut(df["age"], bins=bins, labels=labels)
sorted_df = df.sort_values(by="age_category")
sorted_df["age_category"].hist()
plt.show()
```

5. Conversion of Data Types To Save Memory and Speed Up

Pandas uses the int128 data type by default for integers, but in some cases, it may be necessary to convert them to int64, int32... in order to save memory.

```
# Creates a dataframe of random generated values
from create_random_data import create_random_data

df = create_random_data(1_000_000) #  Create a df with 1M rows
df.head()

# Check the memory usage of the DataFrame before conversion
print("Before Conversion:")
print(df.info(memory_usage='deep'))  # memory usage: 84.9 MB

columns_to_convert = ["prob", "previous_races_count", "next_round"]
df[columns_to_convert] = df[columns_to_convert].astype("int8")
df["height", "weight"] = df["height"].astype("int16")

# Check the memory usage of the DataFrame after conversion
print("\nAfter Conversion:")
print(df.info(memory_usage='deep'))  # memory usage: 72.5 MB
```

6. Avoid using inplace

The inplace parameter in Pandas allows you to perform operations directly on your dataframe, but it can be dangerous to use, as it can make your code harder to read and debug. Instead, try to use the standard method of assigning the result of your operation to a new object.

```
# using inplace to remove the first row of the DataFrame directly

# DON'T
# df.drop(0, inplace=True)

# DO
df = df.drop(0)
```

7. Avoid unnecessary .apply()

The apply function can be a powerful tool, but it can also be slow and memory-intensive. Try to avoid using apply when there are direct, faster and more efficient ways to accomplish your goal.

```
columns = ['space_ship', 'galaxy', 'speed',
           'maneuverability', 'pilot_skill', 'resource_management']

# Calculate the win probability element-wise for
# each row using the specified formula
df['win_prob'] = (df['speed'] * df['maneuverability']
                  * df['pilot_skill']) / df['resource_management']

# ------------------
# Using .apply()
# df['win_prob'] = df.apply(lambda row:
#                           (row['speed'] * row['maneuverability'] *
#row['pilot_skill']) / row['resource_management'], axis=1)
```

8. BONUS! Master Aggregation

We need to master aggregation to gain deeper insights into our data by summarizing it in meaningful ways. Aggregations allow for efficient and effective analysis of large datasets, enabling us to answer complex questions quickly. This can help us make informed decisions based on the data and drive more successful outcomes.

Was This Helpful ?

Don't forget to like and share
Write in the comment area
about your opinion!

/armantunga