Pandas DataFrame - Adding New Columns

Complete Guide to Creating and Modifying Columns

Download This Guide as PDF

Introduction

Adding new columns is one of the most common operations in data analysis with pandas. This guide covers all methods from basic to advanced techniques.

📚 **Sample DataFrame Used in Examples:**
We'll use this employee dataset throughout the guide:

Copy

```
import pandas as pd import numpy as np # Sample DataFrame df = pd.DataFrame({ 'Name': ['Alice', 'Bob', 'Charlie', 'Diana'], 'Age': [25, 30, 35, 28], 'Salary': [50000, 60000, 70000, 55000], 'City': ['New York', 'London', 'Tokyo', 'Paris'], 'Join_Date': ['2020-01-15', '2019-03-20', '2021-07-10', '2022-11-05'] })
```

Basic Methods to Add New Columns

1. Direct Assignment

Simple and most common method

```
# Single value for all rows df['Department'] = 'Engineering' # List of values df['Employee_ID'] = [101, 102, 103, 104] # Calculated from existing column df['Bonus'] = df['Salary'] * 0.1
```

2. Using assign()

Functional approach, returns new DataFrame

```
df = df.assign( Monthly_Salary = df['Salary'] / 12, Tax_Rate = 0.2, Net_Salary = df['Salary'] * 0.8 )
```

3. Using insert()

Add column at specific position

```
# Insert at position 1 (after first column) df.insert(1, 'Employee_Code', ['A001', 'B002', 'C003', 'D004'])
```

Conditional Columns

Using np.where() - Single Condition

```
# Simple if-else logic df['Experience_Level'] = np.where(df['Age'] > 30, 'Senior', 'Junior') df['High_Earner'] =
np.where(df['Salary'] > 60000, 'Yes', 'No') print(df[['Name',
'Age', 'Salary', 'Experience_Level', 'High_Earner']])
```

Output:
Name: Alice, Age: 25, Salary: 50000, Experience_Level: Junior, High_Earner: No
Name: Bob, Age: 30, Salary: 60000, Experience_Level: Junior, High_Earner: No
Name: Charlie, Age: 35, Salary: 70000, Experience_Level: Senior, High_Earner: Yes
Name: Diana, Age: 28, Salary: 55000, Experience_Level: Junior, High_Earner: No

Using np.select() - Multiple Conditions

Copy

```
# Multiple conditions with multiple choices conditions = [
df['Salary'] < 55000, (df['Salary'] >= 55000) & (df['Salary'] <=
65000), df['Salary'] > 65000 ] choices = ['Low', 'Medium',
'High'] df['Salary_Group'] = np.select(conditions, choices,
default='Unknown') print(df[['Name', 'Salary', 'Salary_Group']])
```

Using apply() with Custom Functions

Named Functions

Copy

```
def calculate_annual_bonus(row): """Calculate bonus based on
salary and age""" base_bonus = row['Salary'] * 0.1 if row['Age']
> 30: return base_bonus + 5000 # Extra bonus for experienced
else: return base_bonus df['Annual_Bonus'] =
df.apply(calculate_annual_bonus, axis=1)
```

Lambda Functions

Copy

```
# Simple transformations df['Name_Length'] =
df['Name'].apply(lambda x: len(x)) df['City_Code'] =
df['City'].apply(lambda x: x[:3].upper()) # Multiple column
operations df['Salary_to_Age_Ratio'] = df.apply( lambda row:
row['Salary'] / row['Age'] if row['Age'] > 0 else 0, axis=1 )
print(df[['Name', 'Name_Length', 'City', 'City_Code',
'Salary_to_Age_Ratio']])
```

⚠️ **Performance Warning:**
`apply()` can be slow for large datasets. Use vectorized operations when possible.

String Operations

Copy

```
# Basic string operations df['Email'] = df['Name'].str.lower() +
'@company.com' df['First_Name'] = df['Name'].str.split().str[0]
df['Last_Name'] = df['Name'].str.split().str[-1] if
df['Name'].str.split().str.len() > 1 else '' # String
manipulation df['Name_Upper'] = df['Name'].str.upper()
df['City_Length'] = df['City'].str.len() df['Contains_York'] =
df['City'].str.contains('York') # Complex string operations
df['Formatted_Info'] = df['Name'] + ' (' + df['City'] + ') - $'
+ df['Salary'].astype(str) print(df[['Name', 'Email',
'First_Name', 'City_Length', 'Formatted_Info']].head())
```

DateTime Operations

Copy

```
# Convert to datetime first df['Join_Date'] =
pd.to_datetime(df['Join_Date']) # Extract date components
df['Join_Year'] = df['Join_Date'].dt.year df['Join_Month'] =
df['Join_Date'].dt.month_name() df['Join_Day'] =
df['Join_Date'].dt.day df['Join_Quarter'] =
df['Join_Date'].dt.quarter # Calculate time differences
current_date = pd.Timestamp.now() df['Years_in_Company'] =
(current_date - df['Join_Date']).dt.days / 365.25
df['Months_in_Company'] = ((current_date -
df['Join_Date']).dt.days / 30).astype(int) # Working with dates
df['Anniversary_This_Year'] = df['Join_Date'] +
pd.offsets.DateOffset(years=5) df['Is_Weekend_Join'] =
df['Join_Date'].dt.dayofweek >= 5 print(df[['Name', 'Join_Date',
'Join_Year', 'Years_in_Company', 'Is_Weekend_Join']])
```

Real-World Example: ATECO Data

Copy

```
# Example with ATECO data processing def
enhance_ateco_data(df_ateco): """Add analytical columns to ATECO
DataFrame""" # Ensure Codice is string df_ateco['Codice'] =
df_ateco['Codice'].astype(str) # Basic string operations
df_ateco['Codice_Lunghezza'] = df_ateco['Codice'].str.len()
df_ateco['Codice_Pulito'] = df_ateco['Codice'].str.replace('.',
'').str.strip() # Categorize by code structure conditions = [
df_ateco['Codice_Lunghezza'] == 1, df_ateco['Codice_Lunghezza']
== 2, df_ateco['Codice_Lunghezza'] == 5,
df_ateco['Codice_Lunghezza'] > 5 ] choices = ['Sezione',
'Divisione', 'Gruppo', 'Sottocategoria'] df_ateco['Livello'] =
np.select(conditions, choices, default='Altro') # Extract
section (first character) df_ateco['Sezione'] =
df_ateco['Codice'].str[0] # Boolean flags
df_ateco['Contiene_Punti'] =
df_ateco['Codice'].str.contains('.', na=False)
```

```
df_ateco['Categoria_Principale'] = df_ateco['Codice_Lunghezza']
<= 2 # Description analysis df_ateco['Descrizione_Lunghezza'] =
df_ateco['Descrizione'].str.len()
df_ateco['Descrizione_Maiuscole'] =
df_ateco['Descrizione'].str.isupper() return df_ateco # Usage
example: # df_ateco_enhanced =
enhance_ateco_data(df_ateco.copy())
```

Handling Missing Data

Copy

```
# Sample data with missing values df_missing = pd.DataFrame({
'Product': ['A', 'B', 'C', 'D'], 'Price': [100, None, 150, 200],
'Quantity': [10, 5, None, 8] }) print("Original data with
missing values:") print(df_missing) # Safe column creation with
fillna() df_missing['Total_Value'] = df_missing['Price'] *
df_missing['Quantity'] # Will have NaN
df_missing['Total_Value_Safe'] = ( df_missing['Price'].fillna(0)
* df_missing['Quantity'].fillna(1) ) # Using conditions with
missing values df_missing['Price_Category'] = np.where(
df_missing['Price'] > 100, 'High', np.where(df_missing['Price']
<= 100, 'Low', 'Unknown') ) # Using notna() for conditional
logic df_missing['Has_Complete_Data'] =
df_missing['Price'].notna() & df_missing['Quantity'].notna()
print("\nAfter handling missing values:") print(df_missing)
```

💡 **Tip:** Always handle missing values before calculations to avoid unexpected NaN results.

Performance Tips

Vectorized vs Apply Performance

Copy

```
import time # Create large dataset for testing large_df =
pd.DataFrame({ 'value1': np.random.randint(1, 100, 100000),
'value2': np.random.randint(1, 100, 100000) }) # SLOW: Using
apply start_time = time.time() large_df['result_slow'] =
large_df.apply( lambda row: row['value1'] * row['value2'],
axis=1 ) slow_time = time.time() - start_time # FAST: Vectorized
operation start_time = time.time() large_df['result_fast'] =
large_df['value1'] * large_df['value2'] fast_time = time.time()
- start_time print(f"Apply method: {slow_time:.4f} seconds")
print(f"Vectorized method: {fast_time:.4f} seconds")
print(f"Speed improvement: {slow_time/fast_time:.1f}x faster")
```

✅ DO - Vectorized Operations

Copy

```
# Fast - vectorized df['new_col'] = df['col1'] + df['col2']
df['squared'] = df['value'] ** 2 df['combined'] = df['str1']
+ '_' + df['str2']
```

❌ DON'T - Slow Methods

Copy

```
# Slow - avoid these results = [] for i in range(len(df)):
results.append(df.iloc[i]['col1'] * 2) df['new_col'] =
results # Also slow df['new_col'] = df['col1'].apply(lambda
x: x * 2)
```

Best Practices Summary

✅ DO These

- Use direct assignment for simple cases
- Use `assign()` for method chaining
- Prefer vectorized operations
- Use `np.where()` for simple conditions
- Handle missing values explicitly
- Use string methods for text operations

❌ AVOID These

- Using `apply()` for simple math
- Iterating with loops for calculations
- Ignoring missing values
- Modifying DataFrames in place without copies
- Using `apply()` when vectorized works

Complete Example

Copy

```
def create_comprehensive_employee_profile(df): """Add multiple
calculated columns to employee data""" # Basic calculations
(vectorized) df = df.assign( Monthly_Salary = df['Salary'] / 12,
Tax_Amount = df['Salary'] * 0.2, Net_Salary = df['Salary'] * 0.8
) # Conditional columns df['Experience_Level'] =
np.where(df['Age'] > 30, 'Senior', 'Junior') df['Salary_Group']
= np.select( [df['Salary'] < 55000, df['Salary'] <= 65000,
df['Salary'] > 65000], ['Low', 'Medium', 'High'],
default='Unknown' ) # String operations df['Email'] =
df['Name'].str.lower().str.replace(' ', '.') + '@company.com'
df['Name_Initials'] = df['Name'].str.split().str[0].str[0] +
df['Name'].str.split().str[-1].str[0] # DateTime operations
df['Join_Date'] = pd.to_datetime(df['Join_Date'])
df['Years_in_Company'] = (pd.Timestamp.now() -
df['Join_Date']).dt.days / 365.25 # Complex logic with apply def
career_stage(row): if row['Age'] < 25: return 'Entry' elif
row['Age'] < 35: return 'Mid-Career' else: return 'Experienced'
df['Career_Stage'] = df.apply(career_stage, axis=1) # Boolean
columns df['Eligible_for_Bonus'] = (df['Salary'] > 50000) &
(df['Years_in_Company'] > 1) return df # Usage # enhanced_df =
create_comprehensive_employee_profile(df.copy())
```

Quick Reference

Method	Use Case	Example	Performance
Direct Assignment	Simple calculations, single values	`df['new'] = df['a'] + df['b']`	⭐⭐⭐⭐⭐
assign()	Multiple columns, method chaining	`df.assign(x=1, y=df['a']*2)`	⭐⭐⭐⭐⭐
insert()	Specific position	`df.insert(1, 'new', values)`	⭐⭐⭐⭐⭐
np.where()	Single condition	`np.where(cond, val1, val2)`	⭐⭐⭐⭐⭐
np.select()	Multiple conditions	`np.select([cond1, cond2], [val1, val2])`	⭐⭐⭐⭐
apply()	Complex row-wise logic	`df.apply(func, axis=1)`	⭐⭐

String Methods	Text manipulation	`df['str'].str.upper()`	⭐⭐⭐⭐⭐

Hello World!