
# Lesson 5: String Manipulation and Time Series Data

**Lecturer:** **Dr. Nguyen Tuan Long**

Email: ntlong@neu.edu.vn

Mobile: 0982 746 235

# CONTENTS



**Topic 1: String Manipulation**

- Why is text data processing important? Text data is one of the richest but also messiest sources of information in business.
- The power of the .str accessor and Regular Expressions (Regex) to extract information from text.

**Topic 2: Working with Time Series Data - Basic**

- The importance of time-based data in economics & business: trend analysis, forecasting, and detecting seasonality.
- Core data structures and techniques: Smart indexing, date range creation, and frequencies.

# Business Context



**Scenario:** "In a customer analytics problem, we have:

- **Text Data:** The Description column in transaction data contains product descriptions (e.g., 'WHITE HANGING HEART T-LIGHT HOLDER'). How can we extract key terms like 'WHITE', 'HEART' to categorize products, or standardize different names for the same product?
- **Time Data:** The InvoiceDate column contains transaction timestamps. How can we analyze revenue by day of the week or by month to find purchasing patterns, or create a complete report for all days in a quarter, including days with no sales?"

**Connection:** Today's session will equip you with techniques to turn these raw data columns into intelligent 'features', helping to uncover customer insights and build more accurate predictive models.



# Vectorized String Operations

# Introducing the .str accessor



- **The Problem:** Using for loops to process strings in Python is slow and cumbersome, especially on large datasets. More importantly, they will raise an error if they encounter a missing value (NaN), a very common issue.
- **The Solution:** Pandas provides the .str accessor, which allows you to apply most of Python's string processing methods to an entire Series quickly (vectorized) and automatically skips missing values. This makes your code not only faster but also safer and more readable.

```
data = pd.Series(['peter', 'Paul', None, 'MARY', 'gUIDO'])
# This would fail with a standard loop
# [s.capitalize() for s in data] # -> Raises an error because of None
# The correct way with Pandas
print(data.str.capitalize())
```

# Common String Methods



| Method | Description |
|---|---|
| **Series.str.split()** | Split strings on delimiter or regular expression |
| **Series.str.strip()** | Trim whitespace from both sides, including newlines |
| **Series.str.lower()** | |
| **Series.str.upper()** | |
| **Series.str.get()** | Index into each element(retrieve i-th element) |
| **Series.str.replace()** | Replace each occurrence of pattern/regex in the Series/Index. |

**Serries+ str+ Method**

# Common String Methods - Normalization



**Objective:** Master basic string functions to clean and normalize data, ensuring consistency.

```
s = pd.Series([' house ', 'kitchen', 'BATHROOM '])
# Case normalization
print("Upper case:\n", s.str.upper())
# Stripping whitespace
print("\nStripped:\n", s.str.strip())
# Replacing strings
print("\nReplaced:\n", s.str.replace(' ', '_'))
```

**Business Thinking:** Normalizing text data (e.g., converting 'Hanoi' and 'hanoi' to the same form, removing extra spaces) is a critical step to ensure that groupby or merge operations work correctly. Otherwise, the system might treat 'Hanoi' and 'hanoi ' as two different locations, leading to incorrect regional sales reports and potentially flawed business decisions.

# Common String Methods - Searching



**Objective:** Learn how to search for simple patterns in text data.

```
s = pd.Series(['apple', 'banana', 'apricot', 'blueberry’])

# Check for substring existence (case-insensitive)
print("Contains 'A' (case-insensitive):\n", s.str.contains('A', case=False))

# Check if string starts with...
print("\nStarts with 'a':\n", s.str.startswith('a’))

# Check if string ends with...
print("\nEnds with 't':\n", s.str.endswith('t'))
```

**Application:** Filter all customer feedback containing the word "complaint", or find all products with a code starting with "SKU-". Using case=False allows for more flexible searching.

# Splitting Strings with .str.split()



**Objective:** Split a single string into multiple parts based on a delimiter, a very common technique for restructuring data.

```
s = pd.Series(['a_b_c', 'c_d_e', 'f_g_h'])
# Split the string, returns a Series of lists
print(s.str.split('_'))
# Access elements of the split lists
print("\nSecond element:\n", s.str.split('_').str.get(1))
# Expand into a DataFrame, with each element in a column
print("\nExpanded DataFrame:\n", s.str.split('_', expand=True))
```

**Business Application:** Splitting a "Full Name" column into "First Name" and "Last Name" columns to personalize marketing emails. Splitting a URL string to extract the domain name or campaign parameters.

# Lab #1: Structuring Customer Data



**Scenario:** You have received a list of new customer sign-ups. The data is messy and needs to be structured before being added to the main database.

```
df_lab1 = pd.DataFrame({
    'full_name': ['  John Smith ', 'Jane Doe', '  peter jones  '],
    'location': ['123 Main St, New York, USA', '456 Oak Ave, London,
UK', '789 Pine Ln, New York, USA']
})
```

**Tasks:**

1. **Normalize Names :** Create a new column cleaned_name where all names from full_name are in title case (e.g., 'John Smith') and have no leading/trailing whitespace. (Hint: Chain methods like .str.strip().str.title()).
2. **Find New York Customers:** Create a new DataFrame ny_customers that contains only the customers from 'New York'.
3. **Split Location Data :** Split the location column into three new columns: street, city, and country. Use , as the delimiter.

# Introduction to Regular Expressions (Regex)



**Concept:** Think of Regex as a super-powered version of 'Find and Replace'. It's a mini-language designed specifically for defining complex text patterns, allowing you to find, validate, and extract information with incredible precision.

**Basic Special Characters:**

- \d: Any digit (0-9).
- \w: Any alphanumeric character (a-z, A-Z, 0-9, _).
- +: One or more occurrences.
- *: Zero or more occurrences.
- (): Groups parts of the pattern together for extraction. This is called a "capture group".

**Example:** The pattern \d+ will find all sequences of numbers in a text. The pattern ([A-Z]+) will find and "capture" sequences of uppercase letters.

# The Pandas Regex methods



| Method | Description |
|---|---|
| **Series.str.match()** | Use re.match with the passed regular expression on each element, returning matched groups as list |
| **Series.str.contains()** | Return Boolean array if each string contains pattern/regex |
| **Series.str.extract()** | Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group |
| **Series.str.findall()** | Compute list of all occurrences of pattern/regex for each string |

**Note:** `import re`

- The pandas regular expression methods accept an optional flags argument. The most common and the most useful is the re.IGNORECASE(re.I) flag.
- All available flags are here

# Regex with .str.match()



- **Objective:** Validate if strings **start with** a specific pattern.
- **Concept:** str.match() checks for a match **only at the beginning** of the string. It returns a boolean Series. It's stricter than contains().

```
s = pd.Series(['Apple', 'Banana', 'apricot', 'Avocado'])
# Without flags (case-sensitive)
# Checks if the string STARTS with 'A'
print("Starts with 'A' (case-sensitive):\n", s.str.match(r'A.*'))

# With flags (case-insensitive)
# Checks if the string STARTS with 'a' or 'A'
print("\nStarts with 'a' (case-insensitive):\n",
s.str.match(r'a.*',flags=re.IGNORECASE))
```

**Business Application:** Validating data entry. For example, ensuring all product SKUs in a column start with the correct prefix (e.g., SKU-) or that all phone numbers start with a country code.

# Regex with .str.contains()



- **Objective:** Find if a pattern exists **anywhere** within a string.
- **Concept:** str.contains() checks if the pattern is present **anywhere** in the string. This is the most common method for filtering text. It returns a boolean Series.

```
s = pd.Series(['Order #123', 'REF:456', 'order
#789'])
# Without flags (case-sensitive)
print("Contains 'Order':\n",
s.str.contains(r'Order'))

# With flags (case-insensitive)
print("\nContains 'order' (case-insensitive):\n",
s.str.contains(r'order', flags=re.IGNORECASE))
```

**Business Application:** The workhorse for filtering text data. Finding all customer reviews that mention "delivery", identifying all transactions that are "refunds", or flagging comments with specific keywords.

# Regex with .str.extract()



- **Objective:** Pull out the **first specific piece of information** that matches a pattern.
- **Concept:** str.extract() searches for a pattern and returns the **first match** from the **first capture group ()**. It returns a DataFrame (or Series if only one group).

```
s = pd.Series(['ID: A123', 'id: b456', 'No ID here'])
# Without flags (case-sensitive) - will miss 'id: b456'
print("Extract number after 'ID: ':\n", s.str.extract(r'ID: [A-
Z](\d+)'))

# With flags (case-insensitive)
print("\nExtract number after 'id: ' (case-insensitive):\n",
s.str.extract(r'id: [a-z](\d+)', flags=re.I))
```

**Business Application:** Extracting specific, structured data from messy text, like pulling the zip code from an address string, the user ID from a log entry, or a version number from a software name.

# Regex with .str.findall()



**Objective:** Find **all** occurrences of a pattern within each string.

**Concept:** str.findall() finds **every non-overlapping match** of the pattern in each string and returns a Series of lists.

```
s = pd.Series(['#sale #promo', '#Sale #new', '#shipping'])
# Without flags (case-sensitive) - finds only lowercase hashtags
print("Find all lowercase hashtags:\n", s.str.findall(r'#([a-z]+)'))

# With flags (case-insensitive) - finds all hashtags
print("\nFind all hashtags (case-insensitive):\n", s.str.findall(r'#([a-
z]+)', flags=re.IGNORECASE))
```

**Business Application:** Extracting all hashtags from social media posts for trend analysis, finding all mentioned products in a customer review, or pulling all email addresses from a block of text.

# Lab #2: Advanced Text Parsing with Regex



**Scenario:** You are analyzing a dataset of user comments from a website. You need to validate, categorize, and extract key information from these comments for further analysis.

```
df_comments = pd.DataFrame({
    'comments': [
        'REF-1234: Great product!',
        'Complaint: The item (REF-5678) was broken.',
        'REF-9101, love it! #awesome #greatbuy',
        'This is not a valid comment',
        'complaint: another issue with REF-1122']})
```

**Tasks:**

1. **Validate Comments (.str.match()):** Create a boolean Series is_valid that is True for comments that start with either "REF" or "Complaint". Use a case-insensitive flag. (Hint: r'(REF|Complaint).*').
2. **Categorize Complaints (.str.contains()):** Using the original df_comments, create a new DataFrame complaints_df that contains only comments that include the word "complaint" (case-insensitive).
3. **Extract Reference Numbers (.str.extract()):** Create a new column ref_id in the original DataFrame that extracts only the digits from the reference number (e.g., '1234', '5678').
4. **Extract Hashtags (.str.findall()):** Create a new column hashtags that finds all hashtags (words starting with #) in each comment.

# Times Series - basic

# What is a Time Series?



**Definition:** A time series is a series of data points listed in time order. The key idea is that the sequence is important, as each data point is tied to a specific moment in time.

**Importance:** Economic data (GDP, inflation), financial data (stock prices, interest rates), and sales data (daily/monthly revenue) are all time series. Correctly analyzing this type of data is fundamental for forecasting and strategic decision-making.

**Key Structures in Pandas:**

- **Timestamp:** Represents a single point in time, with nanosecond precision.
- **DatetimeIndex**: An Index containing Timestamp objects. This is the "backbone" of time series analysis, providing powerful and performance-optimized selection and manipulation capabilities.

# Converting to Datetime Objects pd.to_datetime



**Objective:** Learn how to efficiently convert string or numeric columns into a DatetimeIndex.

```
# Convert strings to datetime, Pandas can often infer the format
dates_str = ['2023-01-01', '2023/01/02', '05-Jan-2023']
dates = pd.to_datetime(dates_str,format="mixed")
print(dates)
```

```
# Handle non-standard formats with the `format` parameter
date_custom = '01--2023--15'
print(pd.to_datetime(date_custom, format='%m--%Y--%d'))
```

```
# Handle errors: coerce invalid values to NaT (Not a Time)
bad_dates = ['2023-01-01', 'not a date']
# The errors='coerce' parameter will turn any unparseable date into NaT (Not a Time),
# which is Pandas' equivalent of NaN for datetime objects. This prevents the code from crashing.
print(pd.to_datetime(bad_dates, errors='coerce'))
```

**Mindset:** Always convert your date columns to the datetime type as the first step. This does two critical things: 1) It unlocks a suite of specialized time series functions (like resampling) that don't work on strings, and 2) it dramatically speeds up filtering and aggregation operations.

# Setting a Datetime Column as the Index



**Objective:** Understand why and how to make a datetime column the DataFrame's index.

**Why it's the most important step:**

- **Unlocks Power:** It transforms a regular DataFrame into a time-series-aware object, enabling powerful, intuitive slicing and selection (e.g., df['2023-05']).
- **Enables Advanced Functions:** It is a prerequisite for most advanced time series functions like resample() (for changing frequency, e.g., from daily to monthly) and rolling() (for calculating moving averages).
- **Improves Performance:** Operations on a DatetimeIndex are highly optimized and much faster than on a standard column.

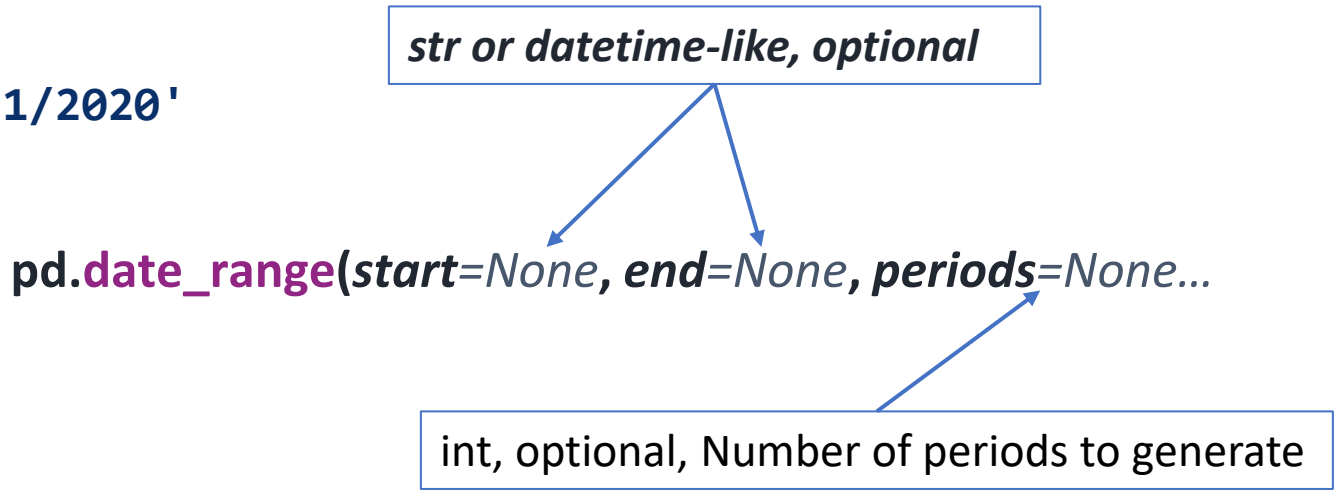
```
df = pd.DataFrame({
    'sale_date':
pd.to_datetime(['2023-01-15', '2023-
01-16']),
    'sales': [100, 150]
})
print("--- Before setting index ---")
print(df.info())

# Set the 'sale_date' column as the
index
df.set_index('sale_date',
inplace=True)

print("\n--- After setting index ---")
print(df.info())
print("\nIndex Type:", type(df.index))
```

# Time Series Selection and Slicing



**Objective:** Leverage the power of a DatetimeIndex for intelligent and intuitive data selection.

```
ts = pd.Series(np.random.randn(1000),
                index=pd.date_range('1/1/2020'
, periods=1000))
# Select a year
print(ts['2021'].head())
# Select a month
print(ts['2021-05'].head())
# Slice a date range
ts['2022-01-01':'2022-01-31'].head()
```




**Application:** Easily retrieve data from the previous quarter to compare with the current one, or analyze the effectiveness of a marketing campaign that ran during a specific date range.

# Time-based Feature Engineering with the .dt accessor



**Objective:** Learn how to extract time components to create new features for models.

**Concept:** Similar to .str for strings, Pandas provides a .dt accessor for Series with a datetime type. It allows you to easily access properties of the dates.

```
# Create a datetime Series
dates = pd.Series(pd.to_datetime(['2023-01-01', '2023-02-15', '2023-03-30']))

print("Year:", dates.dt.year)
print("Month:", dates.dt.month)
print("Day of Week:", dates.dt.dayofweek) # Monday=0, Sunday=6
print("Day Name:", dates.dt.day_name())
print("Quarter:", dates.dt.quarter)
```

**Business Thinking:** These features directly drive business decisions. For example, dayofweek helps optimize staffing by identifying peak sales days (like weekends). quarter is crucial for financial reporting, and creating combined features like is_weekend can significantly improve predictive models for retail.

# Lab #3



**Task 1:** Using the ts time series created earlier:

1. Select all data from the second quarter of 2021.
2. Create a new column containing the name of the day of the week (e.g., 'Monday', 'Tuesday') from the index.
3. Filter out all Sundays from the dataset.

**Tasks 2:**

```
df_sales = pd.DataFrame({
    'date': ['2023-01-29', '2023-01-30', '2023-02-01', '2023-02-02', '2023-02-03'],
    'sales': [250, 275, 310, 290, 350]})
```

1. **Prepare the Data:** Convert the date column to datetime objects and then set this column as the DataFrame's index.
2. **Monthly Analysis:** Select all sales data for February 2023. Calculate the total sales for that month.
3. **Feature Engineering:** Create a new column called day_of_week that contains the name of the day for each sale date.
4. **Business Insight:** Calculate the average sales for each day of the week to find out which day is the most profitable based on this small sample. (Hint: df.groupby('day_of_week')['sales'].mean()).

# Summary



- **String Manipulation:** Master cleaning and structuring text using the .str accessor for normalization, searching (.contains), and splitting. Employ Regex methods like .str.extract() and .str.findall() for advanced pattern extraction.

- **Time Series:** The core workflow involves converting date columns to a DatetimeIndex using pd.to_datetime and setting it as the index. This enables powerful time-based slicing and the creation of new features (year, month, day of week) with the .dt accessor

# Lab #4



**1. Load and Clean Data**

- Load the OnlineRetail.csv file.
- Remove rows with missing CustomerID.
- Convert the InvoiceDate column to the datetime data type.

**2. Time Analysis**

- Create an Hour column (hour of the day) from InvoiceDate.
- Analyze which hour of the day has the most transactions.

**3. Text Analysis**

- Normalize the Description column by converting it to lowercase.
- Create a boolean column named IsCoffeeRelated which is True if the product description contains the word "coffee".
- Calculate the total revenue (Quantity * UnitPrice) for "coffee" related products and compare it with other products.

**4. Combined Analysis**

- Filter for transactions that occurred in December.
- Within December, which product related to "gift" or "christmas" was sold the most (by quantity)?

# Q&A