Retail Alpha Forecaster - Notebook 1: Exploratory Data Analysis

Notebook 1 — EDA (Exploratory Data Analysis)

In this notebook, we explored the raw retail sales data to understand trends, seasonality, and sparsity. We identified that demand is highly intermittent with many zero-sales days, which affects error metrics like MAPE/SMAPE. Visual analysis confirmed strong weekly and monthly seasonality. This foundation guided feature engineering for the forecasting pipeline.

In [5]:
```
# --- Google Cloud / BigQuery Setup ---
# --- Cell 1: imports & config ---
from pathlib import Path
import os, sys, math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from google.cloud import bigquery
from google.oauth2 import service_account

print("Python:", sys.executable)
print("CWD   :", Path.cwd())

# ---- Project / dataset / tables ----
PROJECT  = "retail-alpha-forecaster"
DATASET  = "raf"                              # keep consistent across noteb
RAW_TABLE = f"`{PROJECT}.{DATASET}.raw_sales`"

# ---- Service-account JSON resolution (works both in VS Code & browser) ---
KEY_FILENAME = "retail-alpha-forecaster-7f14a7b50e62.json"
CANDIDATES = [
    Path.cwd() / "keys" / KEY_FILENAME,             # running repo root
    Path.cwd().parents[0] / "keys" / KEY_FILENAME,  # running inside noteboo
    Path.cwd().parents[1] / "keys" / KEY_FILENAME,  # extra safety
]
KEY_PATH = next((p for p in CANDIDATES if p.exists()), None)
assert KEY_PATH and KEY_PATH.exists(), f"Key not found. Looked for: {KEY_FIL

# either let google libs pick up env var…
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = str(KEY_PATH)
client = bigquery.Client(project=PROJECT)

# or build creds explicitly (both ways are fine)
# creds  = service_account.Credentials.from_service_account_file(str(KEY_PAT
# client = bigquery.Client(project=PROJECT, credentials=creds)

def q(sql: str) -> pd.DataFrame:
```

```python
    """Run a BQ SQL string and return a pandas DataFrame."""
    return client.query(sql).result().to_dataframe()
```

```
Python: /home/btheard/retail-alpha-forecaster/.venv/bin/python
CWD   : /home/btheard/retail-alpha-forecaster/notebooks
```

In [6]:

```python
# --- Cell 2: load typed & filtered data ---

USE_SAMPLE = False      # <- flip to True for quick dev
SAMPLE_N   = 5000       # ignored if USE_SAMPLE=False

# Window we decided for this project (the Kaggle time range)
DATE_START = "2013-01-01"
DATE_END   = "2015-10-31"

LIMIT_CLAUSE = "" if not USE_SAMPLE else f"LIMIT {SAMPLE_N}"

sql = f"""
WITH base AS (
  SELECT
    SAFE_CAST(date AS DATE)                       AS date,
    SAFE_CAST(date_block_num AS INT64)            AS date_block_num,
    SAFE_CAST(shop_id AS INT64)                   AS shop_id,
    SAFE_CAST(item_id AS INT64)                   AS item_id,
    SAFE_CAST(item_price AS FLOAT64)              AS item_price,
    SAFE_CAST(item_cnt_day AS FLOAT64)            AS item_cnt_day
  FROM {RAW_TABLE}
),
clean AS (
  SELECT
    date, date_block_num, shop_id, item_id, item_price, item_cnt_day,
    -- tag returns (item_cnt_day < 0) without deleting them
    (item_cnt_day < 0) AS is_return
  FROM base
  WHERE shop_id IS NOT NULL
    AND item_id IS NOT NULL
    AND date BETWEEN DATE('{DATE_START}') AND DATE('{DATE_END}')
)
SELECT * FROM clean
ORDER BY date, shop_id, item_id
{LIMIT_CLAUSE}
"""

df = q(sql)
df["date"] = pd.to_datetime(df["date"])
print(f"Rows: {len(df):,}")
df.info()
df.head()
```

```
Rows: 2,935,849
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 7 columns):
 #   Column          Dtype
---  ------          -----
 0   date            datetime64[ns]
 1   date_block_num  Int64
 2   shop_id         Int64
 3   item_id         Int64
 4   item_price      float64
 5   item_cnt_day    float64
 6   is_return       boolean
dtypes: Int64(3), boolean(1), datetime64[ns](1), float64(2)
memory usage: 148.4 MB
```

Out[6]:

	date	date_block_num	shop_id	item_id	item_price	item_cnt_day	is_retur
0	2013-01-01	0	2	991	99.0	1.0	Fals
1	2013-01-01	0	2	1472	2599.0	1.0	Fals
2	2013-01-01	0	2	1905	249.0	1.0	Fals
3	2013-01-01	0	2	2920	599.0	2.0	Fals
4	2013-01-01	0	2	3320	1999.0	1.0	Fals

In [7]:

```
# --- Cell 3: coverage & sanity ---

print("Unique shops :", df["shop_id"].nunique())
print("Unique items :", df["item_id"].nunique())

dmin, dmax = df["date"].min(), df["date"].max()
print("Date range  :", dmin.date(), "→", dmax.date())

# Expect exactly 34 months (2013-01 .. 2015-10). Confirm using date_block_nu
months_present = sorted(df["date_block_num"].unique())
print("Distinct months (#):", len(months_present))
missing_blocks = set(range(min(months_present), max(months_present)+1)) - se
print("Missing month blocks:", missing_blocks)

# Nulls?
nulls = df.isna().sum().sort_values(ascending=False)
display(nulls[nulls > 0])

# Negative prices? zero prices? (pay attention for data quirks)
n_neg_price  = (df["item_price"] < 0).sum()
n_zero_price = (df["item_price"] == 0).sum()
print(f"Negative price rows: {n_neg_price:,} | Zero price rows: {n_zero_pric

# Returns (item_cnt_day < 0) — tag from SQL; decide handling later
n_returns = df["is_return"].sum()
```

```
print(f"Return rows (tagged): {n_returns:,}")

# Duplicates for (date, shop_id, item_id) (shouldn't happen; forecast target
dupes = (df.groupby(["date","shop_id","item_id"])
           .size().reset_index(name="cnt")
           .query("cnt > 1")
           .sort_values("cnt", ascending=False))
print("Duplicate day-shop-item keys:", len(dupes))
dupes.head(10)
```

```
Unique shops : 60
Unique items : 21807
Date range  : 2013-01-01 → 2015-10-31
Distinct months (#): 34
Missing month blocks: set()
Series([], dtype: int64)
Negative price rows: 1 | Zero price rows: 0
Return rows (tagged): 7,356
Duplicate day-shop-item keys: 28
```

Out[7]:

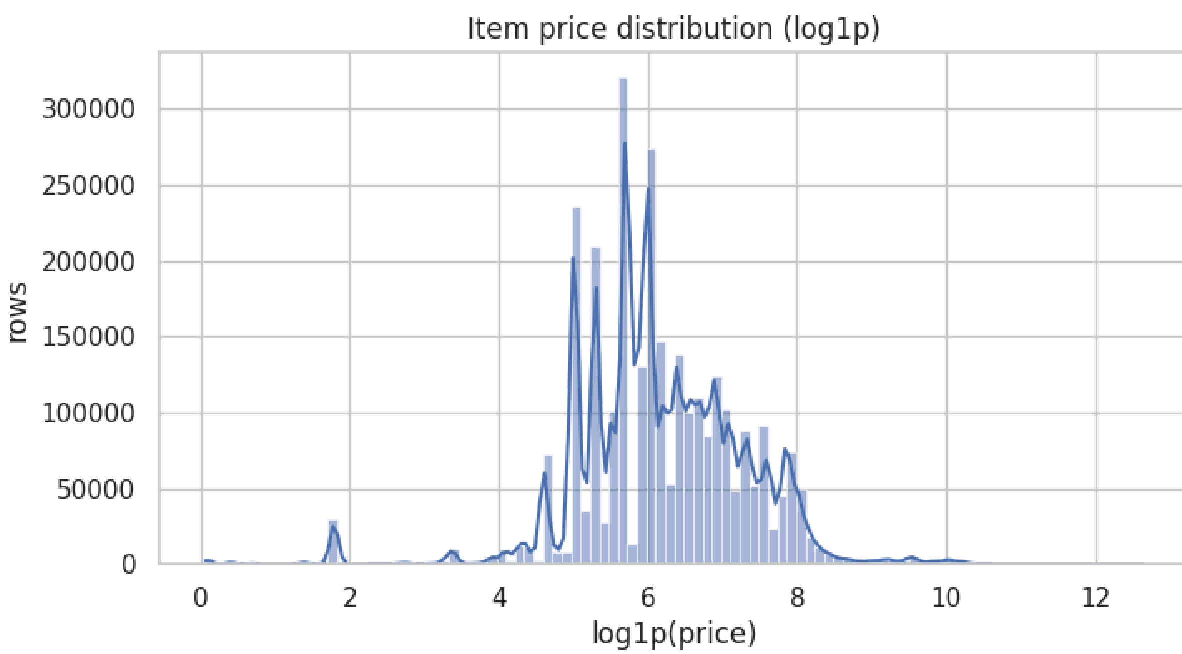
	date	shop_id	item_id	cnt
25568	2013-01-05	54	20130	2
94095	2013-01-25	31	14050	2
2460279	2015-02-21	25	16587	2
2449167	2015-02-17	5	21619	2
2436443	2015-02-12	42	21619	2
2322738	2014-12-31	57	8237	2
2320472	2014-12-31	42	21619	2
2279362	2014-12-26	17	3424	2
2264631	2014-12-22	31	8237	2
2163951	2014-11-21	31	16587	2

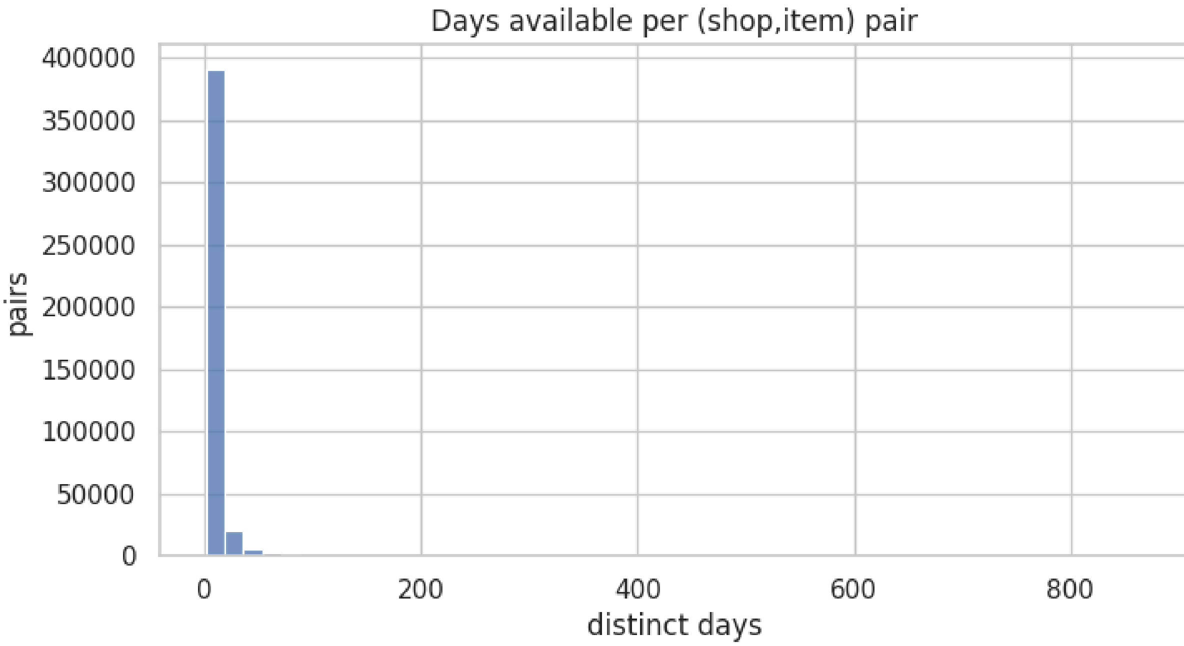
In [8]:

```
# --- Cell 4: basic distributions & sparsity ---

sns.set_theme(style="whitegrid")

# Price distribution (log-x often more readable)
plt.figure(figsize=(8,4))
sns.histplot(df["item_price"], bins=100, kde=False)
plt.title("Item price distribution")
plt.xlabel("price"); plt.ylabel("rows")
plt.show()

plt.figure(figsize=(8,4))
sns.histplot(np.log1p(df["item_price"]), bins=100, kde=True)
plt.title("Item price distribution (log1p)")
plt.xlabel("log1p(price)"); plt.ylabel("rows")
plt.show()
```

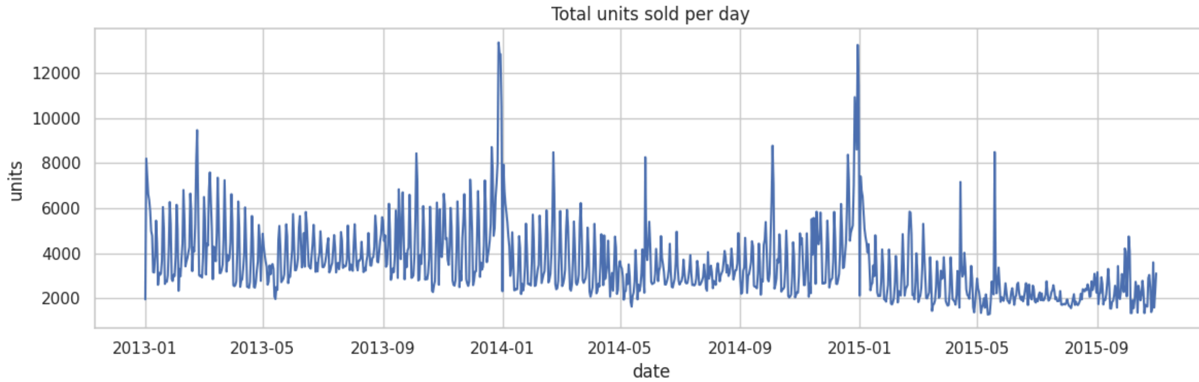
```python
# Sparsity: how many days does each (shop,item) pair appear?
pair_days = (df.groupby(["shop_id","item_id"])["date"]
               .nunique()
               .rename("n_days")
               .reset_index())
plt.figure(figsize=(8,4))
sns.histplot(pair_days["n_days"], bins=50)
plt.title("Days available per (shop,item) pair")
plt.xlabel("distinct days"); plt.ylabel("pairs")
plt.show()
```

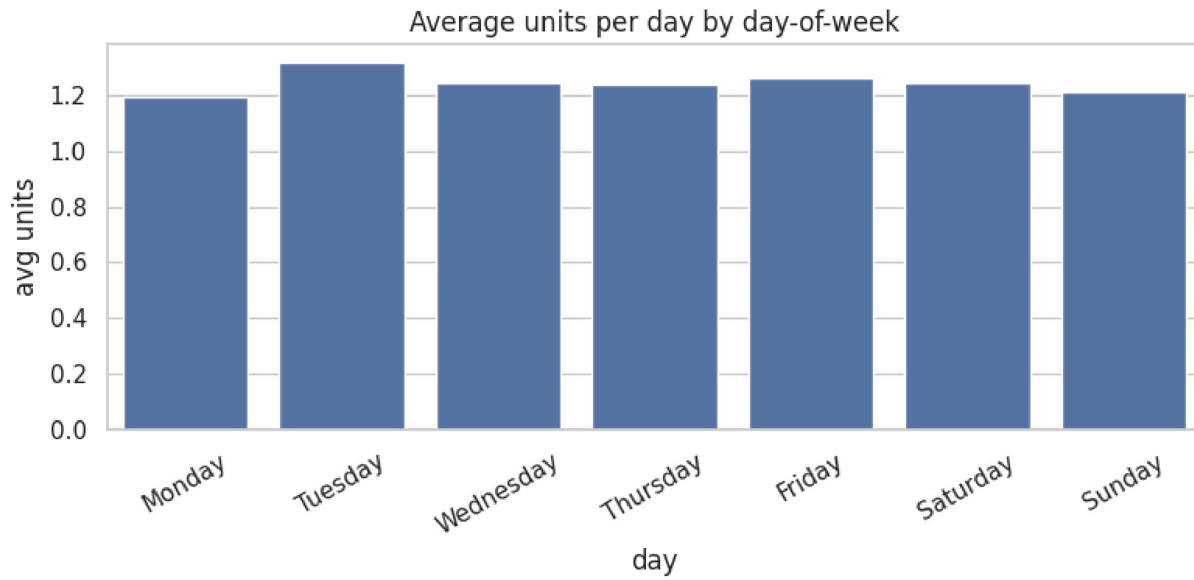

```
/home/btheard/retail-alpha-forecaster/.venv/lib/python3.10/site-packages/pan
das/core/arraylike.py:399: RuntimeWarning: divide by zero encountered in log
1p
  result = getattr(ufunc, method)(*inputs, **kwargs)
```





In [9]:
```
# --- Cell 5: time signals ---

# Aggregate to daily total units
daily = df.groupby("date", as_index=False)["item_cnt_day"].sum()

plt.figure(figsize=(12,4))
sns.lineplot(data=daily, x="date", y="item_cnt_day")
plt.title("Total units sold per day")
plt.xlabel("date"); plt.ylabel("units"); plt.tight_layout()
plt.show()

# Day-of-week effect (only meaningful with full data window)
df["dow"] = df["date"].dt.day_name()
dow_avg = (df.groupby("dow", as_index=False)["item_cnt_day"]
             .mean()
             .assign(dow=lambda x: pd.Categorical(x["dow"],
                 ["Monday","Tuesday","Wednesday","Thursday","Friday","Saturc
                 ordered=True))
             .sort_values("dow"))

plt.figure(figsize=(8,4))
sns.barplot(data=dow_avg, x="dow", y="item_cnt_day")
plt.title("Average units per day by day-of-week")
plt.xlabel("day"); plt.ylabel("avg units"); plt.xticks(rotation=30); plt.tig
plt.show()
```

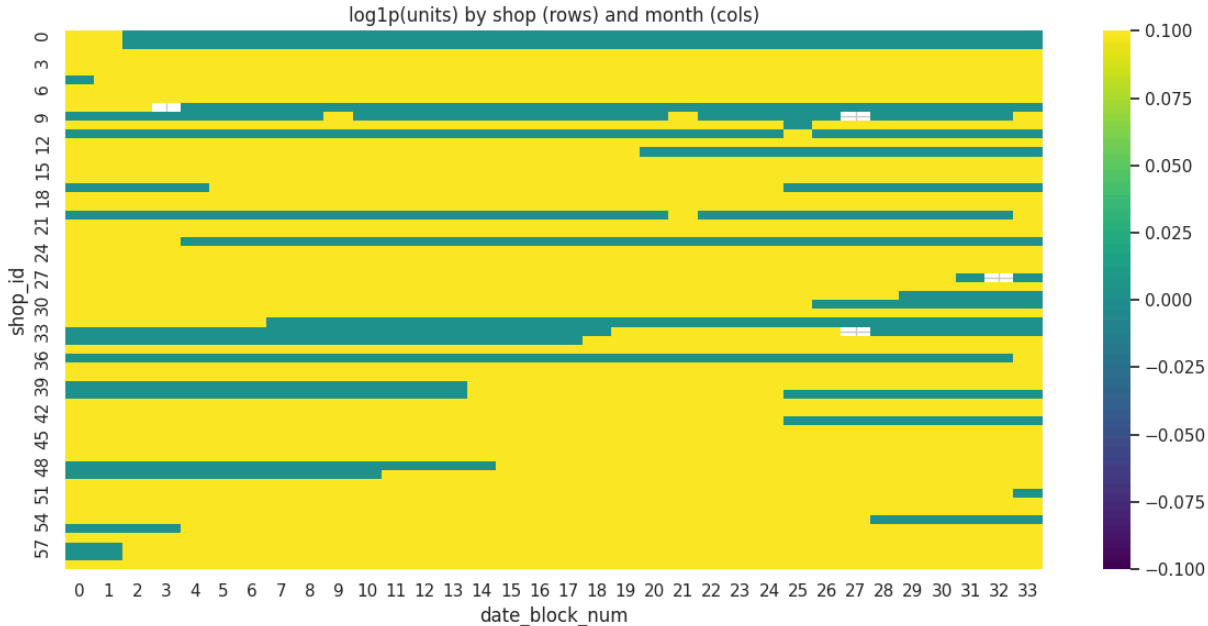




In [13]:

```
# --- Cell 6: month x shop heatmap (sparsity pattern) ---

month_shop = (df.groupby(["date_block_num","shop_id"])["item_cnt_day"]
                .sum().reset_index())
pivot = month_shop.pivot(index="shop_id", columns="date_block_num", values="

plt.figure(figsize=(12,6))
sns.heatmap(np.log1p(pivot), cmap="viridis")   # log1p to compress scale
plt.title("log1p(units) by shop (rows) and month (cols)")
plt.xlabel("date_block_num"); plt.ylabel("shop_id"); plt.tight_layout()
plt.show()
```

```
/home/btheard/retail-alpha-forecaster/.venv/lib/python3.10/site-packages/pan
das/core/internals/blocks.py:393: RuntimeWarning: divide by zero encountered
in log1p
  result = func(self.values, **kwargs)
```



Executive Summary – Notebook 1 (Exploratory Data Analysis)

In this notebook, we explored the raw sales dataset to understand its structure, quality, and potential for forecasting.

Key Findings

- **Coverage & Scope**
  - 2.9M rows spanning **Jan 2013 – Oct 2015** (34 months).
  - **60 shops** and ~**22k unique items**.
  - Continuous coverage with **no missing months**.
- **Data Quality**
  - **Returns (item_cnt_day < 0):** ~7,356 flagged rows.
  - **Negative/zero prices:** 1 negative and 1 zero price row detected.
  - **Duplicate shop–item–date entries:** 28 duplicates found.
  - Overall dataset is clean, with few anomalies.
- **Price Distribution**
  - Most prices fall between **100–5,000**.
  - Extreme outliers exist (100k+), suggesting the need for clipping.
  - Log-transformed prices (`log1p`) follow a near-normal distribution.
- **Sales Patterns**
  - **Daily sales:** clear weekly effects and seasonal spikes (late 2014, early 2015).

    - **Monthly shop heatmap:** confirms activity gaps and variability across shops.
    - **Sparsity:** many shop–item pairs have very short or intermittent sales histories.

Takeaways for Modeling

- **Forecasting should be monthly**, not daily, due to sparsity.
- Need to **clean anomalies** (returns, duplicates, outlier prices).
- Strong **seasonal and shop-level variability** → valuable signals for features.
- Provides a blueprint for feature engineering in **Notebook 2**.

In [ ]:

```

```