Import the dataset

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [28]:
```
import sqlite3
db_connection = sqlite3.connect("E:/Data Analyst Bootcamp/Projects/Olist_Dataset
print("✅ Connected successfully!")
```

```
✅ Connected successfully!
```

Created SQLite3 Connection

In [5]:
```
cursor = db_connection.cursor()
cursor.execute("SELECT name FROM sqlite_master WHERE type='table';")
print(cursor.fetchall())
```

```
[('product_category_name_translation',), ('sellers',), ('customers',), ('geolocat
ion',), ('order_items',), ('order_payments',), ('order_reviews',), ('orders',),
('products',), ('leads_qualified',), ('leads_closed',)]
```

In [7]:
```
import pandas as pd

def view_table(table, limit):
    query = f"""
        SELECT *
        FROM {table}
        LIMIT {limit}
    """
    return pd.read_sql_query(query, db_connection)
```

Exploratory Data Analysis EDA¶

NUMBERS OF ORDERS

In [37]:
```
# Table orders, first 3 columns
view_table('orders', 5).iloc[:, :3]
```

Out[37]:

	order_id	customer_id	order_status
0	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered
1	53cdb2fc8bc7dce0b6741e2150273451	b0830fb4747a6c6d20dea0b8c802d7ef	delivered
2	47770eb9100c2d0c44946d9cf07ec65d	41ce2a54c0b03bf3443c3d931a367089	delivered
3	949d5b44dbf5de918fe9c16f97b45f8a	f88197465ea7920adcdbec7375364d82	delivered
4	ad21c59c0840e6cb83a9ceb5573f8159	8ab97904e6daea8866dbdbc4fb7aad2c	delivered

In [39]:
```
# Table orders, first 3 columns
view_table('order_items', 5).iloc[:, :3]
```

Out[39]:

	order_id	order_item_id	product_id
0	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61
1	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8
2	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fc
3	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc
4	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089

Let's count the number of daily orders in the dataset using an SQL GROUP BY clause on the order timestamp and view the first 5 resulting rows:

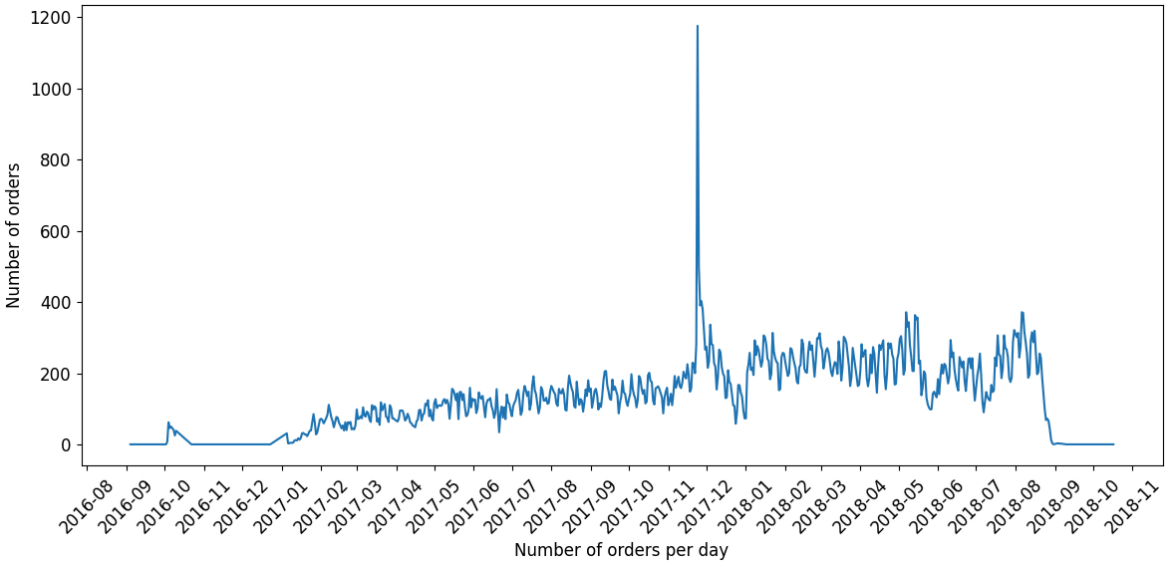
In [43]:
```
orders_per_day = """
select
    Date(order_purchase_timestamp) AS day,
    count(*) AS order_count
from orders
group by day
"""

df = pd.read_sql_query(orders_per_day, db_connection)
df.head()
```

Out[43]:

	day	order_count
0	2016-09-04	1
1	2016-09-05	1
2	2016-09-13	1
3	2016-09-15	1
4	2016-10-02	1

At the beginning of the dataset, order activity seems quite low. To gain insights into how order frequency evolved over time, we'll visualize it using Matplotlib.

In [47]:
```
import matplotlib.pyplot as plt
import matplotlib.dates as mdates


plt.rcParams['font.size']=12
plt.rcParams['axes.titlesize']=16

#Line Plot

plt.figure(figsize=(14,6))
plt.plot(pd.to_datetime(df['day']),df['order_count'])
plt.ylabel('Number of orders')
plt.xlabel('Number of orders per day')
plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
plt.xticks(rotation=45)
plt.show()
```



As we can see from the plot, there is a clear spike in orders around the Christmas period, particularly on December 24th. Overall, the trend shows a steady increase in orders over time, reflecting the growth of Olist's business. At the very beginning and end of the dataset, the number of orders is quite low, so we'll exclude these sparse dates from some of our upcoming analyses to focus on the periods with more meaningful data.

How are orders distributed throughout the week? Are weekends busier than weekdays? What about the time of day — do customers tend to place more orders in the evening? We can explore these questions using a heatmap, which visualizes the frequency of orders across both days of the week and hours of the day. To create this heatmap, we first need a matrix of order counts, where each row corresponds to a day of the week and each column corresponds to an hour of the day. We can extract this information from the order timestamps using the STRFTIME function.

In [79]:
```
order_day_hour="""
select
    --Day of the week abreviated
   CASE STRFTIME('%w', order_purchase_timestamp)
       when '1' then 'Mon'
       when '2' then 'Tue'
       when '3' then 'wed'
       when '4' then 'Thu'
       when '5' then 'Fri'
       when '6' then 'Sat'
       when '0' then 'Sun'
       END AS day_of_week_name,
     --Day of the week as integer (Sunday=7)
     CAST(STRFTIME('%w', order_purchase_timestamp) AS integer) AS day_of_week_in
     --Hour of the day (0-24)
      CAST(STRFTIME('%H', order_purchase_timestamp) AS integer) AS hour
 from orders
 """

pd.read_sql_query(order_day_hour, db_connection)
```

Out[79]:

	day_of_week_name	day_of_week_int	hour
0	Mon	1	10
1	Tue	2	20
2	wed	3	8
3	Sat	6	19
4	Tue	2	21
...	...	...	...
99436	Thu	4	9
99437	Tue	2	12
99438	Sun	0	14
99439	Mon	1	21
99440	Thu	4	20

99441 rows × 3 columns

To efficiently calculate the number of orders for each hour of the day in our SQL query, we can leverage a Python list comprehension. This approach helps us avoid writing 24 repetitive statements for every hour. We will also use the results from our previous query as a Common Table Expression (CTE) to streamline the analysis.

In [80]:
```
count_orders_per_hour = ',\n    '.join([
    f'COUNT(CASE WHEN hour = {i} THEN 1 END) AS "{i}"' \
    for i in range(24)
])

orders_per_day_of_the_week_and_hour = f"""
WITH OrderDayHour AS (
    {order_day_hour}
)
SELECT
    day_of_week_name,
    {count_orders_per_hour}
FROM OrderDayHour
GROUP BY day_of_week_int
ORDER BY day_of_week_int
"""
```

In [81]:
```
# SQL query without the CTE
print(orders_per_day_of_the_week_and_hour[591:])
```

```
T
    day_of_week_name,
    COUNT(CASE WHEN hour = 0 THEN 1 END) AS "0",
    COUNT(CASE WHEN hour = 1 THEN 1 END) AS "1",
    COUNT(CASE WHEN hour = 2 THEN 1 END) AS "2",
    COUNT(CASE WHEN hour = 3 THEN 1 END) AS "3",
    COUNT(CASE WHEN hour = 4 THEN 1 END) AS "4",
    COUNT(CASE WHEN hour = 5 THEN 1 END) AS "5",
    COUNT(CASE WHEN hour = 6 THEN 1 END) AS "6",
    COUNT(CASE WHEN hour = 7 THEN 1 END) AS "7",
    COUNT(CASE WHEN hour = 8 THEN 1 END) AS "8",
    COUNT(CASE WHEN hour = 9 THEN 1 END) AS "9",
    COUNT(CASE WHEN hour = 10 THEN 1 END) AS "10",
    COUNT(CASE WHEN hour = 11 THEN 1 END) AS "11",
    COUNT(CASE WHEN hour = 12 THEN 1 END) AS "12",
    COUNT(CASE WHEN hour = 13 THEN 1 END) AS "13",
    COUNT(CASE WHEN hour = 14 THEN 1 END) AS "14",
    COUNT(CASE WHEN hour = 15 THEN 1 END) AS "15",
    COUNT(CASE WHEN hour = 16 THEN 1 END) AS "16",
    COUNT(CASE WHEN hour = 17 THEN 1 END) AS "17",
    COUNT(CASE WHEN hour = 18 THEN 1 END) AS "18",
    COUNT(CASE WHEN hour = 19 THEN 1 END) AS "19",
    COUNT(CASE WHEN hour = 20 THEN 1 END) AS "20",
    COUNT(CASE WHEN hour = 21 THEN 1 END) AS "21",
    COUNT(CASE WHEN hour = 22 THEN 1 END) AS "22",
    COUNT(CASE WHEN hour = 23 THEN 1 END) AS "23"
FROM OrderDayHour
GROUP BY day_of_week_int
ORDER BY day_of_week_int
```

Once the query has been executed, we can set the day_of_the_week_name column as the index of the dataframe. This step transforms our data into the matrix format required to construct the heatmap, with days as rows and hours as columns.

In [82]:
```
df = pd.read_sql_query(orders_per_day_of_the_week_and_hour, db_connection)
df = df.set_index('day_of_week_name')
df
```

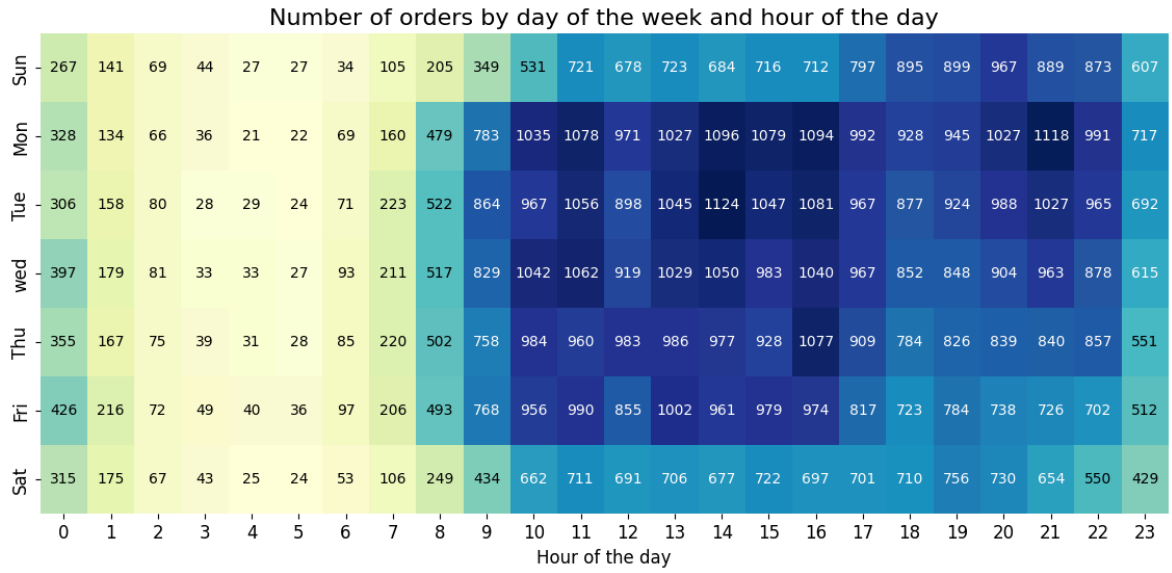
Out[82]:

day_of_week_name	0	1	2	3	4	5	6	7	8	9	...	14	15	16
Sun	267	141	69	44	27	27	34	105	205	349	...	684	716	712
Mon	328	134	66	36	21	22	69	160	479	783	...	1096	1079	1094
Tue	306	158	80	28	29	24	71	223	522	864	...	1124	1047	1081
wed	397	179	81	33	33	27	93	211	517	829	...	1050	983	1040
Thu	355	167	75	39	31	28	85	220	502	758	...	977	928	1077
Fri	426	216	72	49	40	36	97	206	493	768	...	961	979	974
Sat	315	175	67	43	25	24	53	106	249	434	...	677	722	697

7 rows × 24 columns

With the data matrix prepared, we can now use Seaborn to generate the heatmap. This visualization will clearly show how order frequencies vary across days of the week and hours of the day.

In [83]:
```
import seaborn as sns

fig, ax = plt.subplots(figsize=(14, 6))
sns.heatmap(df, cmap='YlGnBu', cbar=False)
mean_orders = df.mean().mean()
for i in range(len(df)):
    for j in range(len(df.columns)):
        text_color = 'white' if df.iloc[i, j] > mean_orders else 'black'
        ax.text(j+0.5, i+0.5, int(df.iloc[i, j]),
            color=text_color, fontsize=10, ha="center", va="center")
plt.title("Number of orders by day of the week and hour of the day")
plt.xlabel("Hour of the day")
plt.ylabel("")
plt.show()
```



From the heatmap, we can observe that the majority of orders are placed on weekdays between 10 AM and 4 PM, with a slight dip around 12 PM, likely due to lunchtime. There is also noticeable evening activity, with customers placing orders around 9 PM from Sunday through Thursday. Saturdays have the lowest number of orders compared to other weekdays, though they still see a fair amount of activity. The quietest period is between 3 AM and 5 AM, although a small number of orders still occur during these hours.

Having explored the temporal distribution of orders, we can now examine their geographic distribution. The customers table provides location information for each customer, including zip code prefix, city, and state. Note that there are two customer ID columns: customer_unique_id, which uniquely identifies each individual customer, and customer_id. For our analysis, we will focus on customer_unique_id to track individual customer activity.

In [85]:
```
view_table('customers',10)
```

Out[85]:

	customer_id	customer_unique_id	customer_zi
0	06b8999e2fba1a1fbc88172c00ba8bc7	861eff4711a542e4b93843c6dd7febb0	
1	18955e83d337fd6b2def6b18a428ac77	290c77bc529b7ac935b93aa66c333dc3	
2	4e7b3e00288586ebd08712fdd0374a03	060e732b5b29e8181a18229c7b0b2b5e	
3	b2b6027bc5c5109e529d4dc6358b12c3	259dac757896d24d7702b9acbbff3f3c	
4	4f2d8ab171c80ec8364f7c12e35b23ad	345ecd01c38d18a9036ed96c73b8d066	
5	879864dab9bc3047522c92c82e1212b8	4c93744516667ad3b8f1fb645a3116a4	
6	fd826e7cf63160e536e0908c76c3f441	addec96d2e059c80c30fe6871d30d177	
7	5e274e7a0c3809e14aba7ad5aae0d407	57b2a98a409812fe9618067b6b8ebe4f	
8	5adf08e34b2e993982a47070956c5c65	1175e95fb47ddff9de6b2b06188f7e0d	
9	4b7139f34592b3a31687243a302fa75b	9afe194fb833f79e300e37e580171f22	

Which cities have the highest number of orders in the dataset? To answer this, we will construct a query to retrieve the top 10 cities by order count. This involves joining the orders table with the customers table to link each order to the corresponding customer's city.

In [87]:

```
orders_per_city = """
select
    customer_city AS customer_city,
    UPPER(customer_city) AS city,
    COUNT(orders.order_id) as city_order_count
from
    customers
    JOIN orders USING (customer_id)
GROUP by customer_city
ORDER by city_order_count DESC
LIMIT 10
"""

pd.read_sql_query(orders_per_city, db_connection)
```

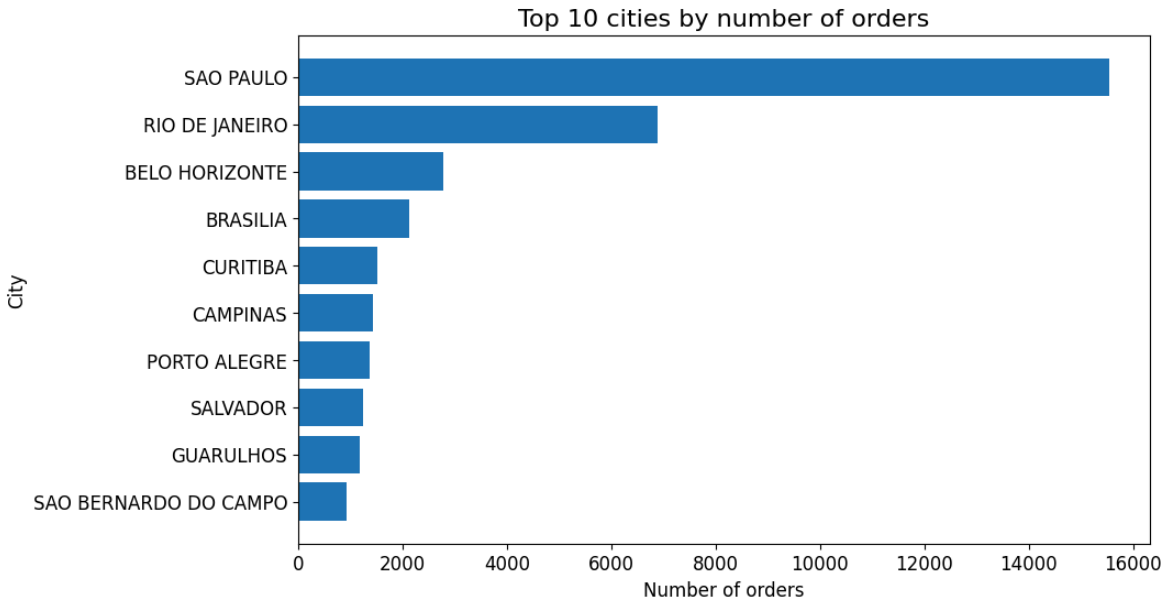
Out[87]:

	customer_city	city	city_order_count
0	sao paulo	SAO PAULO	15540
1	rio de janeiro	RIO DE JANEIRO	6882
2	belo horizonte	BELO HORIZONTE	2773
3	brasilia	BRASILIA	2131
4	curitiba	CURITIBA	1521
5	campinas	CAMPINAS	1444
6	porto alegre	PORTO ALEGRE	1379
7	salvador	SALVADOR	1245
8	guarulhos	GUARULHOS	1189
9	sao bernardo do campo	SAO BERNARDO DO CAMPO	938

Next, we can visualize the top cities using a horizontal bar plot. The barh function in Matplotlib plots data from bottom to top, so to display the cities in descending order, we need to reverse the results. To achieve this, we will use the previous query as a subquery in SQL to reorder the results before plotting.

In [92]:
```
orders_per_city_reversed=f"""
select *
from ({orders_per_city})
Order by city_order_count
"""
```

In [93]:
```
top_cities = pd.read_sql_query(orders_per_city_reversed, db_connection)

plt.figure(figsize=(10,6))
plt.barh(top_cities['city'], top_cities['city_order_count'])
plt.xlabel('Number of orders')
plt.ylabel('City')
plt.title('Top 10 cities by number of orders')
plt.show()
```



From the chart, we can see that São Paulo and Rio de Janeiro account for a significantly higher number of orders compared to other cities. This is expected, as these are the largest and most economically active cities in Brazil. We'll explore the geographic distribution in greater detail later, when we analyze customer lifetime value by zip code prefix.

ORDER PRICE

What is the average value of an order on Olist? Do customers typically purchase low-value items, or does Olist also handle higher-value transactions? Before answering these questions, let's examine the order_items table. This table contains various IDs related to each order, along with the item price and shipping cost. All values are recorded in Brazilian Real (BRL).

In [97]:
```
# Table order_items, first 5 columns
view_table('order_items',5).iloc[:,:5]
```

Out[97]:

	order_id	order_item_id	product_id
0	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61
1	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8
2	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fc
3	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc
4	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089

In [98]:
```
# Table order_items, last 2 columns

view_table('order_items', 5).iloc[:,5:]
```

Out[98]:

	price	freight_value
0	58.90	13.29
1	239.90	19.93
2	199.00	17.87
3	12.99	12.79
4	199.90	18.14

Let's now address our first question: What is the average order value, considering both product prices and shipping costs? To begin, we'll identify the orders with the lowest and highest total costs to understand the range of transaction values in the dataset.

In [100…

```
order_price_stats="""
select
    MIN(order_price) AS min_order_price,
    ROUND(AVG(order_price),2) AS avg_order_price,
    MAX(order_price) AS max_order_price
from(
    select
      orders.order_id,
      SUM(order_items.price + order_items.freight_value) AS order_price
      from orders
          JOIN order_items USING (order_id)
         GROUP by orders.order_id
)
"""

pd.read_sql_query(order_price_stats, db_connection)
```

Out[100…

	min_order_price	avg_order_price	max_order_price
0	9.59	160.58	13664.08

The average order value is 160.58 BRL, about the price of a pair of sports shoes in Brazil in 2017. The highest order is 13,664.08 BRL, nearly 100 times the average, indicating a right-skewed distribution. Next, we'll separate and aggregate product and shipping costs for each order to explore this variation further.

In [18]:

```
order_product_and_shipping_costs = """
select
  orders.order_id,
  SUM(price) AS product_cost,
  SUM(freight_value) AS shipping_cost
from
    orders
    JOIN order_items USING (order_id)
WHERE order_status = 'delivered'
GROUP by orders.order_id
"""

df=pd.read_sql_query(order_product_and_shipping_costs, db_connection)
df
```

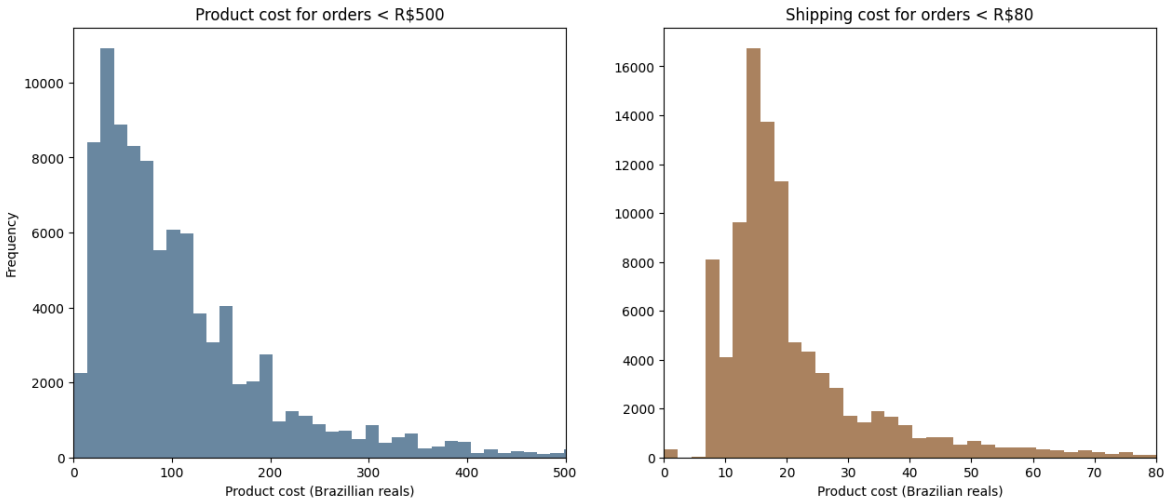
Out[18]:

	order_id	product_cost	shipping_cost
0	00010242fe8c5a6d1ba2dd792cb16214	58.90	13.29
1	00018f77f2f0320c557190d7a144bdd3	239.90	19.93
2	000229ec398224ef6ca0657da4fc703e	199.00	17.87
3	00024acbcdf0a6daa1e931b038114c75	12.99	12.79
4	00042b26cf59d7ce69dfabb4e55b4fd9	199.90	18.14
...	...	...	...
96473	fffc94f6ce00a00581880bf54a75a037	299.99	43.41
96474	fffcd46ef2263f404302a634eb57f7eb	350.00	36.53
96475	fffce4705a9662cd70adb13d4a31832d	99.90	16.95
96476	fffe18544ffabc95dfada21779c9644f	55.99	8.72
96477	fffe41c64501cc87c801fd61db3f6244	43.00	12.79

96478 rows × 3 columns

Let's visualize the distribution of costs using histograms. Since most orders are low in value but a few are much higher, we'll limit the x-axis to 500 BRL for product costs and 80 BRL for shipping costs to better highlight the most common price ranges.

In [19]:
```
plt.figure(figsize=(15,6))

# Histogram for total product cost
plt.subplot(1,2,1)
plt.hist(df['product_cost'], bins=1000, color="#6c87a3")
plt.title('Product cost for orders < R$500')
plt.xlabel('Product cost (Brazillian reals)')
plt.ylabel('Frequency')
plt.xlim([0,500])

# Histogram for total shipping cost

plt.subplot(1, 2, 2)
plt.hist(df['shipping_cost'], bins=800, color='#ad865f')
plt.title('Shipping cost for orders < R$80')
plt.xlabel('Product cost (Brazilian reals)')
plt.xlim([0, 80])
plt.show()
```



Product values vary widely, ranging from very low to extremely high, though most orders total under 200 BRL. Shipping costs typically fall between 7–20 BRL, rarely lower, but can occasionally reach much higher values.

Product categories

Let's explore Olist's product categories based on sales volume. We'll begin with the products table, which includes nine columns, but for this analysis, we'll focus on product category and product weight.

In [23]:
```
import pandas as pd

def view_table(table_name, n=5):
    """
    Displays the first n rows of a table from your SQLite database.
    """
    query = f"SELECT * FROM {table_name} LIMIT {n};"
    return pd.read_sql_query(query, db_connection)
```

The dataset contains 71 unique product categories. To visualize their relative sales, we'll use a treemap, where area size represents sales volume. Displaying all 71 categories would be cluttered, so we'll focus on the top 18 categories and group the rest as "Other categories." First, we'll calculate total sales per category and use the product_category_name_translation table to show category names in English.

In [23]:
```
ranked_categories = """

select
    product_category_name_english AS category,
    SUM(price) AS sales,
    RANK() over (order by SUM (price) DESC) AS rank
from order_items
    JOIN orders USING (order_id)
    JOIN products USING (product_id)
    JOIN product_category_name_translation USING (product_category_name)
WHERE order_status = 'delivered'
GROUP by product_category_name_english
"""
pd.read_sql_query(ranked_categories, db_connection)
```

Out[23]:

	category	sales	rank
0	health_beauty	1233131.72	1
1	watches_gifts	1166176.98	2
2	bed_bath_table	1023434.76	3
3	sports_leisure	954852.55	4
4	computers_accessories	888724.61	5
...	...	...	...
66	flowers	1110.04	67
67	home_comfort_2	760.27	68
68	cds_dvds_musicals	730.00	69
69	fashion_childrens_clothes	519.95	70
70	security_and_services	283.29	71

71 rows × 3 columns

We'll use the previous query as a CTE (Common Table Expression) to calculate the total sales per category and assign a ranking based on sales. Then, we'll select the top 18 categories and group all remaining categories together under "Other categories." Finally, we'll combine both results with a UNION ALL to ensure "Other categories" appears at the end of the result set.

In [15]:
```
category_sales_summary = f"""
WITH RankedCategories AS (
    {ranked_categories}
)
-- Top 18 categories by sales
SELECT
    category,
    sales
FROM RankedCategories
WHERE rank <= 18
-- Other categories, aggregated
UNION ALL
SELECT
    'Other categories' AS category,
    SUM(sales) AS sales
FROM RankedCategories
WHERE rank > 18
"""

df = pd.read_sql_query(category_sales_summary, db_connection)
df
```

Out[15]:

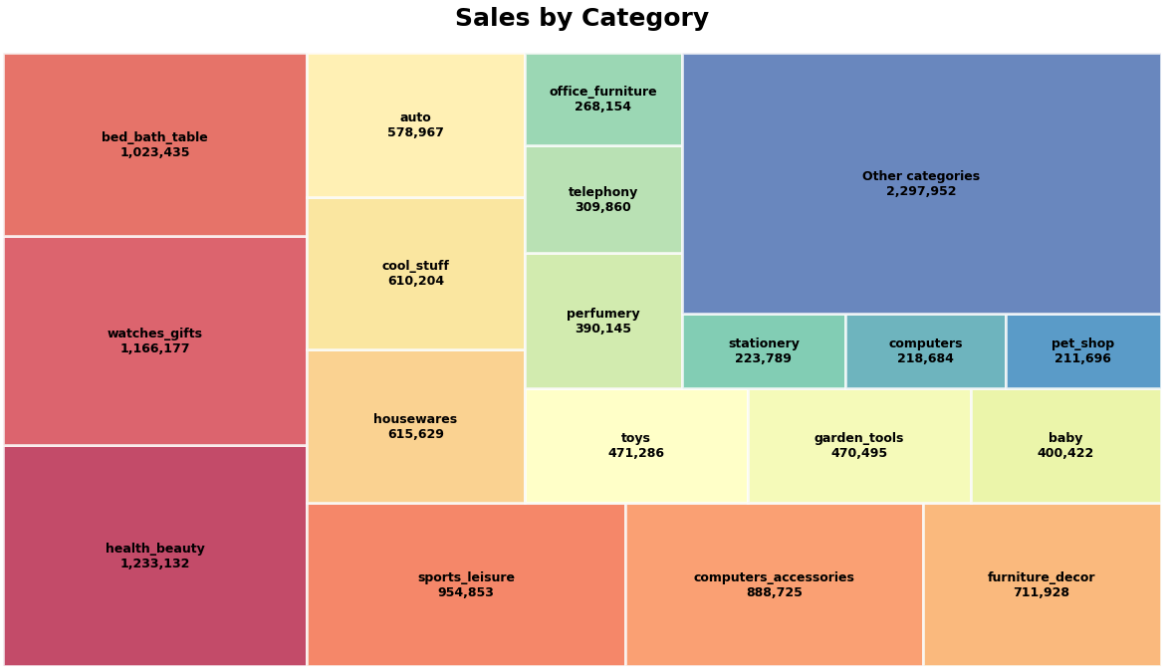
	category	sales
0	health_beauty	1233131.72
1	watches_gifts	1166176.98
2	bed_bath_table	1023434.76
3	sports_leisure	954852.55
4	computers_accessories	888724.61
5	furniture_decor	711927.69
6	housewares	615628.69
7	cool_stuff	610204.10
8	auto	578966.65
9	toys	471286.48
10	garden_tools	470495.28
11	baby	400421.84
12	perfumery	390144.65
13	telephony	309860.23
14	office_furniture	268154.31
15	stationery	223788.69
16	computers	218684.14
17	pet_shop	211695.64
18	Other categories	2297951.89

Let's visualize the category-wise sales distribution using a treemap created with the Squarify library. This will help us compare the relative contribution of each product category at a glance.

In [16]:

```
import squarify
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 8))
plt.title('Sales by Category', fontsize=18, fontweight='bold', pad=20)

# Create formatted labels
labels = [f"{cat}\n{val:,.0f}" for cat, val in zip(df['category'], df['sales'])]

# Color palette
color = sns.color_palette("Spectral", len(df))

# Plot treemap
squarify.plot(
    sizes=df['sales'],
    label=labels,
    alpha=0.8,
```

```
    color=color,
    edgecolor="white",
    linewidth=2,
    text_kwargs={'fontsize': 9, 'weight': 'bold'}
)

plt.axis('off')
plt.show()
```



We can explore product characteristics further by analyzing how product weights vary across categories. Using box plots, we'll visualize the distribution of product weights for each category. To do this, we'll build a query that retrieves product weights while reusing the same list of top categories from our previous DataFrame.

In [20]:
```
top_18_categories = tuple(category for category in df ['category'] if category !
```

It would be useful to order the box plots based on each category's median weight. While we could compute the medians directly in Python, we'll instead handle it in SQL for consistency. Since SQLite doesn't provide a built-in median function, we'll rank the products within each category using row numbers and include the total product count per category. The resulting DataFrame will then serve as the data source for our visualization.

In [27]:
```
ordered_categories = f"""
SELECT
    product_weight_g AS weight,
    product_category_name_english AS category,
    ROW_NUMBER() OVER(PARTITION BY product_category_name_english ORDER BY produc
        AS category_row_n,
    COUNT(*) OVER(PARTITION BY product_category_name_english) AS category_count
FROM
    products
    JOIN order_items USING (product_id)
    JOIN product_category_name_translation USING (product_category_name)
WHERE
    product_category_name_english IN {top_18_categories}
```

```
"""

df = pd.read_sql_query(ordered_categories, db_connection)
df
```

Out[27]:

	weight	category	category_row_n	category_count
0	50.0	auto	1	4235
1	50.0	auto	2	4235
2	50.0	auto	3	4235
3	50.0	auto	4	4235
4	50.0	auto	5	4235
...	...	...	...	...
92419	15267.0	watches_gifts	5987	5991
92420	15267.0	watches_gifts	5988	5991
92421	15267.0	watches_gifts	5989	5991
92422	17175.0	watches_gifts	5990	5991
92423	22175.0	watches_gifts	5991	5991

92424 rows × 4 columns

We can extend our previous query, ordered_categories, to compute the median weight for each product category and then sort the categories based on these median values.

In [28]:

```
categories_by_median = f"""
WITH OrderedCategories AS (
    {ordered_categories}
)
SELECT category
FROM OrderedCategories
WHERE
    -- Odd number of products: Select the middle row
    (category_count % 2 = 1 AND category_row_n = (category_count + 1) / 2) OR
    -- Even number of products: Select the two middle rows to be averaged
    (category_count % 2 = 0 AND category_row_n IN ((category_count / 2), (catego
GROUP BY category
ORDER BY AVG(weight)
"""

categories_by_median_df = pd.read_sql_query(categories_by_median, db_connection)
categories_by_median_df
```

Out[28]:

	category
0	telephony
1	computers_accessories
2	watches_gifts
3	health_beauty
4	perfumery
5	baby
6	sports_leisure
7	toys
8	pet_shop
9	auto
10	stationery
11	housewares
12	bed_bath_table
13	furniture_decor
14	cool_stuff
15	garden_tools
16	computers
17	office_furniture

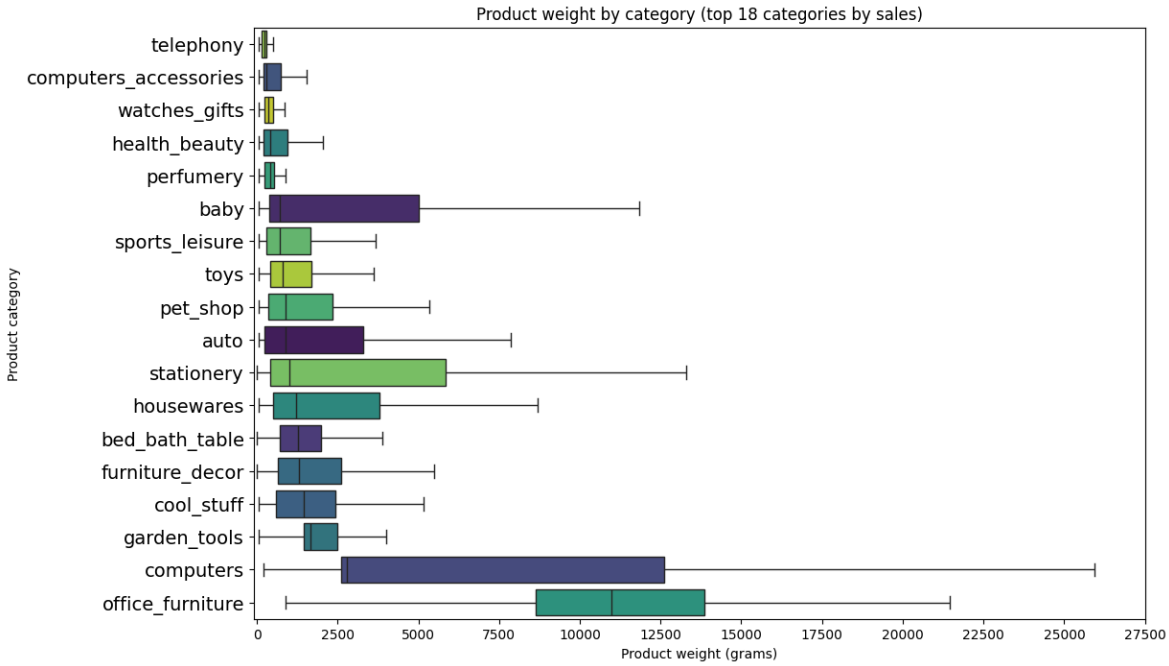
Now we can build the box plots, using the result of the previous query to order them:

In [29]:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Example DataFrame check
print(df.columns)

plt.figure(figsize=(12,8))
order = categories_by_median_df['category'].tolist()

sns.boxplot(
    x='weight',
    y='category',
    data=df,
    order=order,
    showfliers=False,
    palette='viridis',
    hue='category',
    legend=False
)

plt.xlabel('Product weight (grams)')
plt.ylabel('Product category')
```

```
plt.title('Product weight by category (top 18 categories by sales)')
plt.xlim(-100, 2100)
plt.xticks(ticks=range(0, 30000, 2500))
plt.yticks(fontsize=14)
plt.show()
```

```
Index(['weight', 'category', 'category_row_n', 'category_count'], dtype='object')
```



The box plot shows that most items in 'telephony', 'computers_accessories', 'watches_gifts', and 'health_beauty' weigh under 2 kg. In contrast, categories like 'computers' and 'office_furniture' contain significantly heavier products.

Sales Prediction:-

We'll now move forward to analyze the sales trends of a selected set of key product categories.

In [53]:
```
selected_categories = ('health_beauty', 'auto', 'toys', 'electronics', 'fashion_
```

To create a line plot of monthly sales for each category, we first need to aggregate the sales data into a matrix where the columns represent product categories and the rows represent months. This will allow us to visualize trends over time.

In [54]:
```
monthly_sales_selected_categories = f"""
select
    strftime('%Y-%m', order_purchase_timestamp) AS year_month,
    SUM(CASE WHEN product_category_name_english = 'health_beauty' THEN price END
    SUM(CASE WHEN product_category_name_english = 'auto' THEN price END) AS auto
    SUM(CASE WHEN product_category_name_english = 'toys' THEN price END) AS toys
    SUM(CASE WHEN product_category_name_english = 'electronics' THEN price END)
     SUM(CASE WHEN product_category_name_english = 'fashion_shoes' THEN price EN
from orders
    JOIN order_items USING (order_id)
    JOIN products USING (product_id)
    JOIN product_category_name_translation USING (product_category_name)
WHERE order_purchase_timestamp >= '2017-01-01'
    AND product_category_name_english IN {selected_categories}
```

```
    GROUP BY year_month
"""

df=pd.read_sql_query(monthly_sales_selected_categories, db_connection)
df=df.set_index('year_month')
df
```

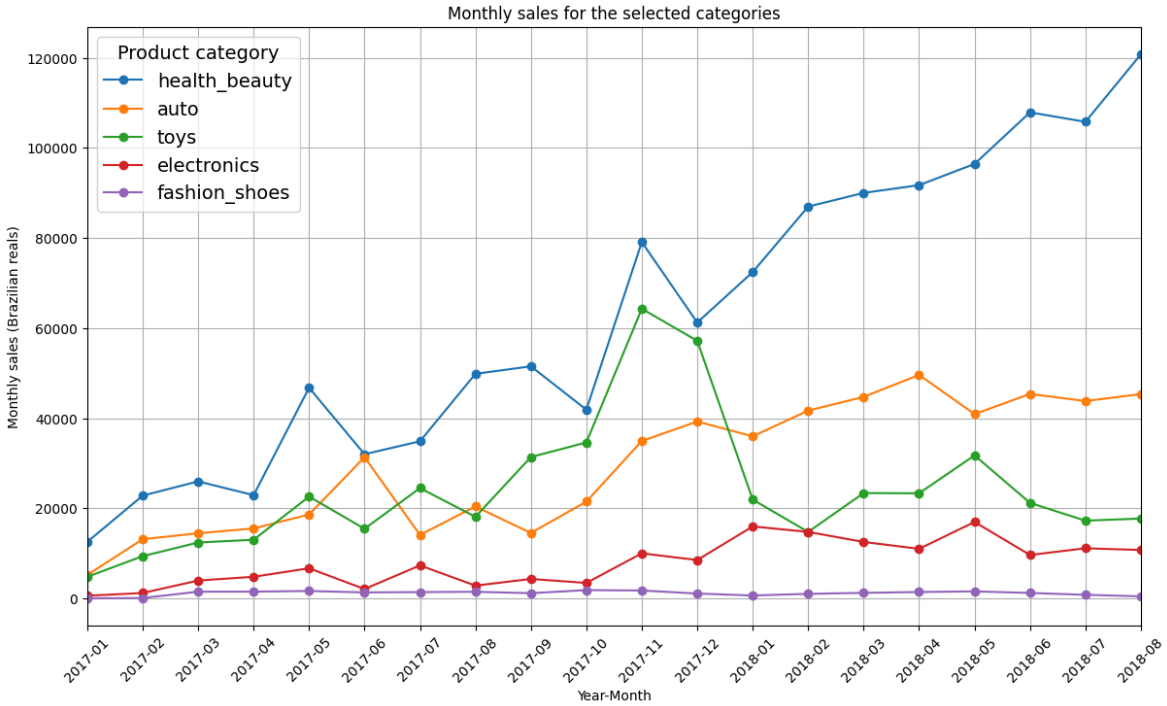
Out[54]:

year_month	health_beauty	auto	toys	electronics	fashion_shoes
2017-01	12561.32	5218.53	4814.09	617.00	34.90
2017-02	22838.79	13162.40	9403.34	1218.85	107.05
2017-03	25995.25	14482.07	12429.47	3986.00	1515.00
2017-04	22935.75	15548.17	13039.43	4822.24	1526.50
2017-05	46786.02	18640.03	22626.95	6709.11	1669.54
2017-06	32029.39	31370.69	15476.35	2100.04	1351.20
2017-07	34896.86	14119.74	24511.72	7346.84	1423.01
2017-08	49873.90	20421.11	18126.87	2854.15	1491.69
2017-09	51537.65	14544.67	31399.22	4338.96	1179.60
2017-10	41915.72	21505.28	34633.01	3454.32	1860.92
2017-11	79120.40	34955.38	64320.89	10026.14	1777.70
2017-12	61264.66	39287.64	57194.14	8537.51	1099.00
2018-01	72470.49	35995.02	21948.23	15996.68	666.70
2018-02	86996.06	41706.89	14778.93	14780.60	1033.70
2018-03	90034.41	44753.24	23407.40	12554.80	1252.80
2018-04	91751.04	49629.19	23352.38	11022.43	1438.70
2018-05	96460.36	40928.14	31767.30	16989.29	1582.70
2018-06	107908.82	45407.49	21221.45	9664.57	1230.58
2018-07	105813.03	43830.29	17277.29	11150.75	820.19
2018-08	120803.94	45380.89	17753.05	10769.47	471.30

Since there's minimal data before January 1, 2017, we'll exclude those early orders from the analysis. Now, let's plot the time series to visualize the sales trends over time.

In [55]:
```
df.index = pd.to_datetime(df.index)
fig, ax= plt.subplots(figsize = (14,8))
df.plot(ax=ax, marker='o', linestyle= '-')
ax.set_xticks(df.index)
ax.set_xticklabels(df.index.strftime('%Y-%m'), rotation = 90)
plt.title('Monthly sales for the selected categories')
plt.xlabel('Year-Month')
plt.ylabel('Monthly sales (Brazilian reals)')
plt.xticks(rotation=45)
```

```
plt.legend(title='Product category', title_fontsize=14, fontsize= 14)
plt.grid(True)
plt.show()
```



As you can see, I picked these categories because they are diverse and each shows a unique trend. At the bottom, in purple there's 'fashion_shoes' which has the lowest-performing sales of the five categories and seems stagnant; across the whole period covered by the dataset, this category hovers around 1000–1500 reals of sales per month. The categories 'electronics' (red), 'auto' (green) and 'health_beauty' (blue), all seem to grow linearly at different rates. The category 'toys' peeks around the holidays, almost tripling in sales on november and december, but quickly goes back to its baseline after that period.

Linear Regression:-

We can actually perform linear regression using only SQL! To do this, we first need to prepare our data so that we can compute the regression line for each category.

:- We'll use two key variables:

1. Days — our independent variable, expressed as the number of days since 2017-01-01.

2. Total sales — our dependent variable, representing daily sales per category.

Once we have these, we can calculate the regression parameters (slope and intercept) directly in SQL.

In [56]:
```
daily_sales_per_category = f"""
select
    DATE(order_purchase_timestamp) AS date,
    --Days since 2017-01-01
    CAST(JULIANDAY(order_purchase_timestamp)- JULIANDAY('2017-01-01') AS INTEGER
```

```
    product_category_name_english AS category,
    SUM(price) AS sales
from orders
    JOIN order_items USING (order_id)
    JOIN products USING (product_id)
    JOIN product_category_name_translation USING (product_category_name)
WHERE
    order_purchase_timestamp BETWEEN '2017-01-01' AND '2018-08-29'
    AND category IN {selected_categories}
GROUP BY day,
    product_category_name_english
"""

pd.read_sql_query(daily_sales_per_category, db_connection)
```

Out[56]:

	date	day	category	sales
0	2017-01-05	4	auto	21.80
1	2017-01-05	4	toys	43.80
2	2017-01-06	5	health_beauty	636.00
3	2017-01-06	5	toys	159.99
4	2017-01-07	6	health_beauty	370.00
...	...	...	...	...
2440	2018-08-27	603	auto	343.38
2441	2018-08-27	603	health_beauty	661.50
2442	2018-08-27	603	toys	257.79
2443	2018-08-28	604	auto	124.00
2444	2018-08-28	604	health_beauty	469.80

2445 rows × 4 columns

We can reuse the previous query as a Common Table Expression (CTE) to calculate the slope and intercept for each category's regression line using the least squares method.

The formulas are as follows:

Slope (m):

slope

$n \sum (day \times sales) - (\sum day)(\sum sales) n \sum (day 2) - (\sum day) 2$ slope= n∑(day 2)−(∑day) 2 n∑(day×sales)−(∑day)(∑sales)

Intercept (b):

intercept

$\sum sales - \text{slope} \times (\sum day)}{n}$ intercept= n ∑sales−slope×(∑day)

Here,

day → number of days since 2017-01-01,

sales → total sales per day, and

n → total number of rows (days) per category.

In [57]:
```
lm_per_category = f"""
WITH DailySalesPerCategory AS (
    {daily_sales_per_category}
)

select
    category,
    --Slope
    (COUNT(*) * SUM(day*sales) - SUM(sales)) /
    (COUNT(*) * SUM(day*day) - SUM(day)*SUM(day))
      AS slope,
      --Intercept
      (SUM(sales) - ((COUNT(*) *SUM(day*sales) - SUM(day)*SUM(sales)) /
      (COUNT(*) *SUM(day*day) - SUM(day)*SUM(day))) *
           SUM(day)) / COUNT(*)
           AS intercept
from
    DailySalesPerCategory
GROUP BY category
"""

df=pd.read_sql_query(lm_per_category, db_connection)
df
```

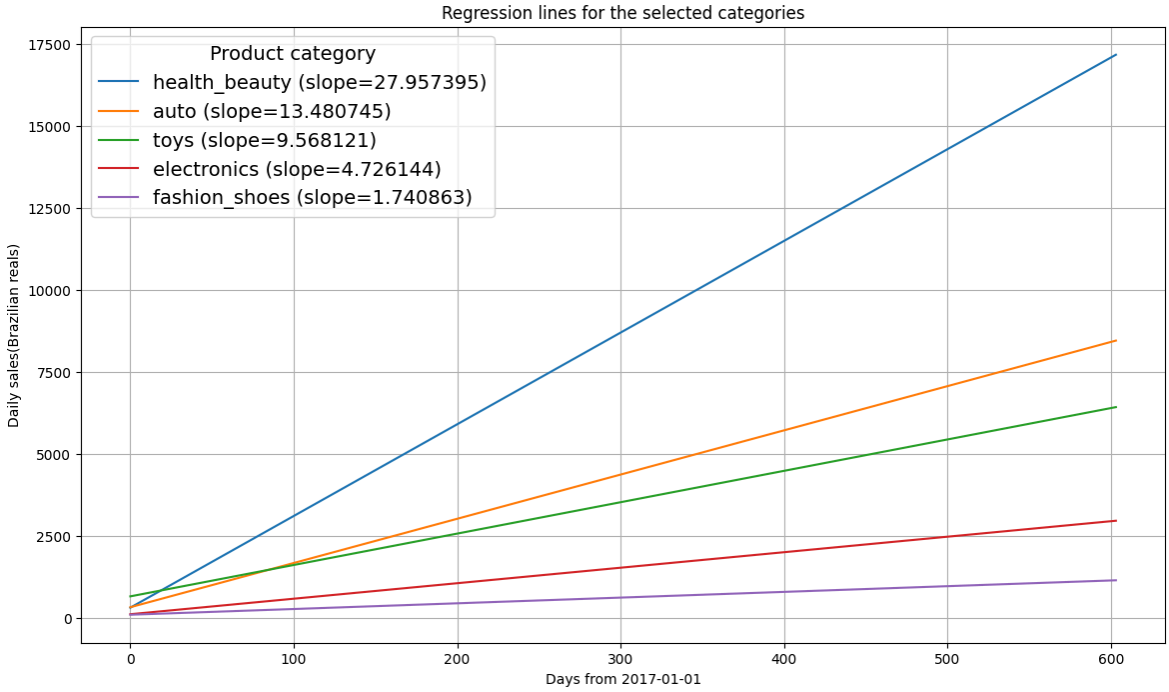
Out[57]:

	category	slope	intercept
0	auto	13.480745	319.055318
1	electronics	4.726144	105.302974
2	fashion_shoes	1.740863	89.637179
3	health_beauty	27.957395	305.451543
4	toys	9.568121	649.648518

Let's see the regression lines for each category:

In [60]:
```
import numpy as np

plt.figure(figsize=(14,8))
days = np.arange(0,604)
for category in selected_categories:
    lm=df[df['category']==category]
    slope= lm['slope'].values[0]
    intercept = lm['intercept'].values[0]
```

```
    line= intercept + slope *days
    plt.plot(days, line, label=f'{category} (slope={slope:2f})')
plt.title('Regression lines for the selected categories')
plt.xlabel('Days from 2017-01-01')
plt.ylabel('Daily sales(Brazilian reals)')
plt.legend(title='Product category', title_fontsize=14, fontsize=14)
plt.grid(True)
plt.show()
```



If we assume that the sales trends identified through our regression lines will persist, we can use them to forecast daily sales for December 2018.

To estimate this, we project the regression line forward by 365 days (one year). The formula for the predicted sales on a given day becomes:

sales $d\ a\ y$ + 365

intercept + slope × ($d\ a\ y$ + 365) sales day+365

=intercept+slope×(day+365)

This expression represents the expected sales for each day of December 2018 based on the linear growth observed since December 2017.

In the ForecastedSales CTE, we apply this formula to generate daily forecasts. Then, in the main query, we smooth out short-term fluctuations by calculating a 5-day moving average, providing a clearer view of overall sales trends.

In [74]:
```
forecasted_sales_dec_2018 = f"""
WITH DailySalesPerCategory AS (
    {daily_sales_per_category}
),
LmPerCategory AS (
    {lm_per_category}
```

```
),
ForecastedSales AS (
    SELECT
        DATE(date, '+1 year') AS date,
        category,
        -- Increase in predicted sales * sales 1 year ago
        (intercept + slope * (day + CAST(JULIANDAY('2018-12-31') - JULIANDAY('20
            / (intercept + slope * day) * sales
            AS forecasted_sales
    FROM DailySalesPerCategory
        JOIN LmPerCategory USING (category)
    -- Filter for days of December 2018
    WHERE day + CAST(JULIANDAY('2018-12-31') - JULIANDAY('2017-12-31') AS INTEGE
        BETWEEN CAST(JULIANDAY('2018-12-01') - JULIANDAY('2017-01-01') AS INTEGE
          AND CAST(JULIANDAY('2018-12-31') - JULIANDAY('2017-01-01') AS INTEGER)
)
SELECT
    CAST(strftime('%d', date) AS INTEGER) AS december_2018_day,
    category,
    -- 5-day moving average
    AVG(forecasted_sales)
        OVER (PARTITION BY category ORDER BY date ROWS BETWEEN 2 PRECEDING AND 2
        AS moving_avg_sales
FROM ForecastedSales
"""

forecast_2018_12_df = pd.read_sql_query(forecasted_sales_dec_2018, db_connection
forecast_2018_12_df
```

Out[74]:

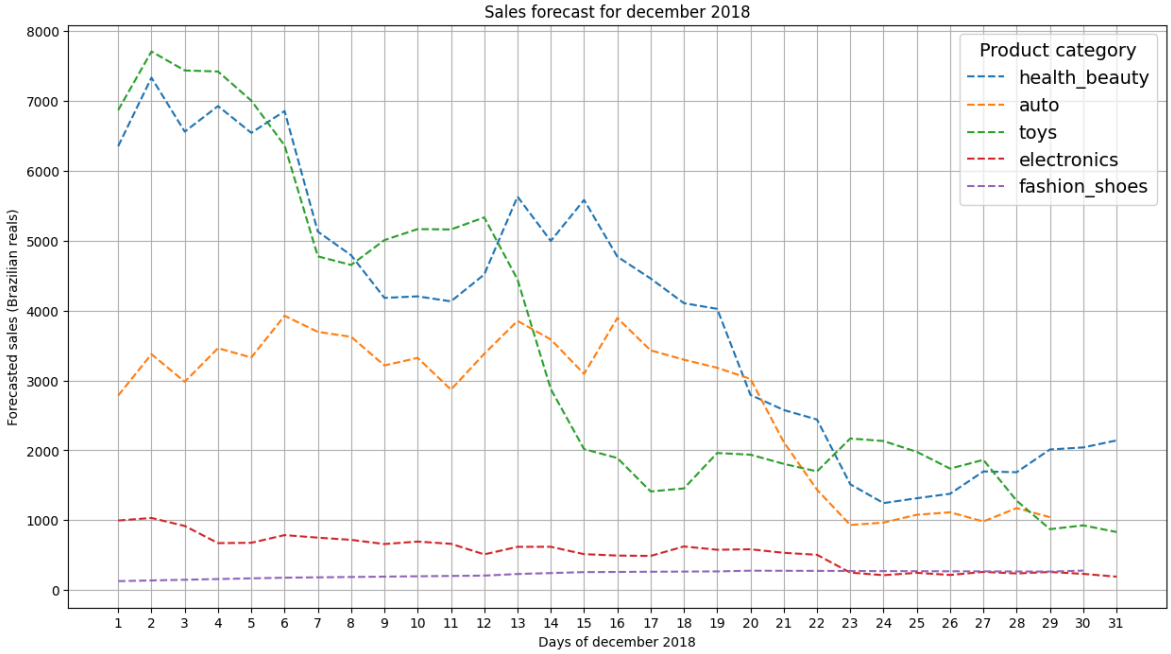
	december_2018_day	category	moving_avg_sales
0	1	auto	2785.527561
1	2	auto	3376.853786
2	3	auto	2982.572110
3	4	auto	3462.852630
4	5	auto	3330.068163
...	...	...	...
126	27	toys	1864.069223
127	28	toys	1279.184894
128	29	toys	871.918464
129	30	toys	926.092115
130	31	toys	831.347380

131 rows × 3 columns

In the next plot, we can see the predicted sales for december 2018 for each category:

In [76]:
```
plt.figure(figsize=(15, 8))
for category in selected_categories:
    category_forecast = forecast_2018_12_df[forecast_2018_12_df['category'] == c
    plt.plot(category_forecast['december_2018_day'], category_forecast['moving_a
```

```
plt.title('Sales forecast for december 2018')
plt.xlabel('Days of december 2018')
plt.ylabel('Forecasted sales (Brazilian reals)')
plt.legend(title='Product category', title_fontsize=14, fontsize=14)
plt.grid(True)
plt.xticks(range(1, 32))
plt.show()
```



Our model predicts that health_beauty will be the top-selling category among the five we analyzed, with sales gradually declining as Christmas approaches. Toys are expected to perform strongly during the first two weeks of December but will see a sharp drop afterward. Auto sales remain steady through the first three weeks before decreasing, while electronics follow a similar pattern. In contrast, fashion_shoes sales appear largely unaffected by holiday trends, showing a slight increase toward the end of the month.

Order Delivery:-

As we saw earlier, the orders table contains several timestamps that track the lifecycle of an order:

. order_purchase_timestamp – when the customer places the order.

. order_approved_at – when Olist approves the order.

. order_delivered_carrier_date – when the order is handed to the shipping company.

. order_delivered_customer_date – when the customer receives the order.

. order_estimated_delivery_date – the estimated delivery date.

Each timestamp represents a key stage in the shipping process. Next, we'll query the data needed to calculate and visualize the average duration of each stage for the top 10 cities by order volume.

In [93]:
```
top_cities_query = """
SELECT customer_city, COUNT(*) AS num_orders
FROM orders
JOIN customers USING (customer_id)
GROUP BY customer_city
ORDER BY num_orders DESC
LIMIT 10
"""

top_cities = pd.read_sql_query(top_cities_query, db_connection)
```

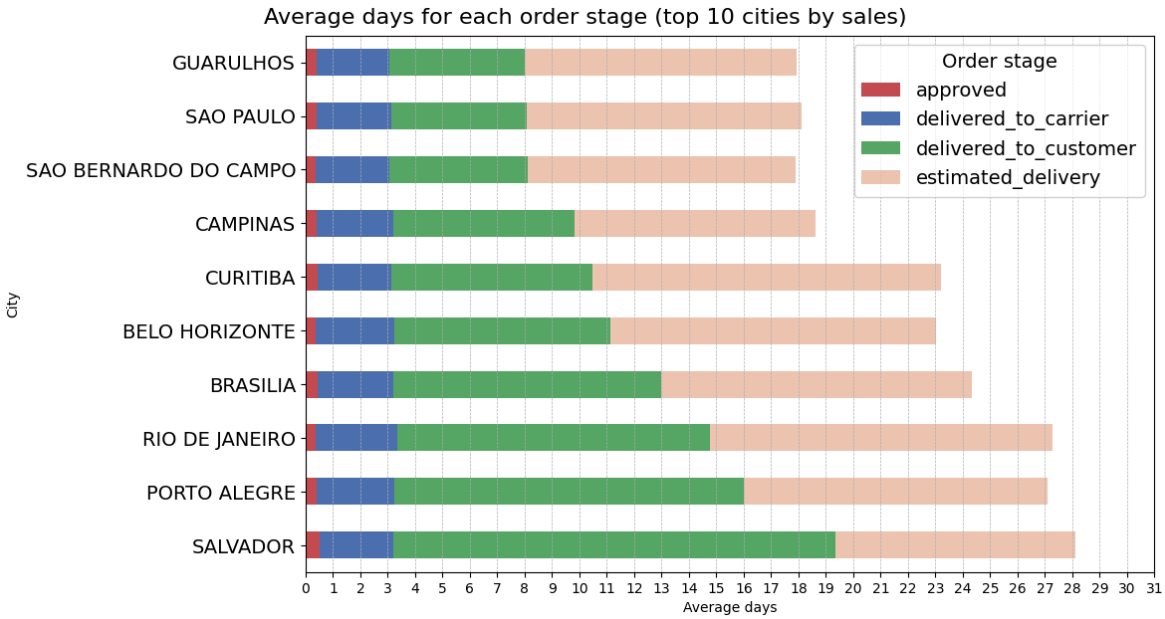
In [98]:
```
order_stage_times_top_10_citites = f"""
SELECT
    UPPER(customer_city)
        AS city,
    AVG(JULIANDAY(order_approved_at) - JULIANDAY(order_purchase_timestamp))
        AS approved,
    AVG(JULIANDAY(order_delivered_carrier_date) - JULIANDAY(order_approved_at))
        AS delivered_to_carrier,
    AVG(JULIANDAY(order_delivered_customer_date) - JULIANDAY(order_delivered_car
        AS delivered_to_customer,
    AVG(JULIANDAY(order_estimated_delivery_date) - JULIANDAY(order_delivered_cus
        AS estimated_delivery
FROM orders
    JOIN customers USING (customer_id)
WHERE  customer_city IN {tuple(top_cities['customer_city'])}
GROUP BY  customer_city
ORDER BY approved + delivered_to_carrier + delivered_to_customer DESC
"""

df = pd.read_sql_query(order_stage_times_top_10_citites, db_connection)
df = df.set_index('city')
df
```

Out[98]:

city	approved	delivered_to_carrier	delivered_to_customer	estimated_delivery
SALVADOR	0.498863	2.700868	16.160566	8.755085
PORTO ALEGRE	0.420697	2.809786	12.786769	11.091915
RIO DE JANEIRO	0.383342	2.979133	11.409813	12.504991
BRASILIA	0.432585	2.762064	9.805472	11.333709
BELO HORIZONTE	0.378608	2.848813	7.900486	11.901479
CURITIBA	0.453929	2.690824	7.316206	12.732135
CAMPINAS	0.420262	2.784653	6.611740	8.821432
SAO BERNARDO DO CAMPO	0.349693	2.702651	5.045101	9.799631
SAO PAULO	0.385928	2.726784	4.961921	10.027349
GUARULHOS	0.420191	2.621652	4.952466	9.944980

Let's visualize the dataframe as a stacked bar plot:

In [99]:
```
fig, ax = plt.subplots(figsize=(11, 7))
df.plot(kind='barh', stacked=True, color=['#c44f53', '#4c72b1', '#55a968', '#dd8
ax.set_xlabel('Average days')
ax.set_ylabel('City')
fig.suptitle('Average days for each order stage (top 10 cities by sales)', fonts
ax.grid(True, linestyle='--', linewidth=0.5, axis='x')
max_bar_length = int(df.sum(axis=1).max())
ax.set_xticks(range(0, max_bar_length + 4))
ax.tick_params(axis='y', labelsize=14)
plt.legend(title='Order stage', title_fontsize=14, fontsize=14)
plt.show()
```



Let's visualize the dataframe as a stacked bar plot:

As we can see, the time from when an order is approved until it is dispatched to the carrier is fairly consistent across most cities, averaging around 3 days. In contrast, the time it takes for an order to reach the customer varies significantly between cities: São Paulo, Guarulhos, and São Bernardo do Campo all show an average shipping time of 5 days, whereas cities at the lower end of the spectrum, such as Rio de Janeiro, Porto Alegre, and Salvador, take more than twice as long on average. Additionally, the average estimated delivery dates are at least a week later than the actual delivery, indicating that these predictions are not precise but rather conservative estimates.

To investigate whether there is any seasonal variation in shipping times, we can create a line plot of daily average shipping times. For this analysis, I will focus on orders from June 2017 to June 2018.

In [100…

```
daily_avg_shipping_time=f"""
select
    DATE(order_purchase_timestamp) AS purchase_date,
    AVG(JULIANDAY(order_delivered_customer_date) - JULIANDAY(order_purchase_time
     AS avg_delivery_time
from orders
WHERE order_purchase_timestamp >= '2017-06-01' AND order_purchase_timestamp <= '
GROUP BY DATE(order_purchase_timestamp)
"""

df=pd.read_sql_query(daily_avg_shipping_time, db_connection)
df
```

Out[100…

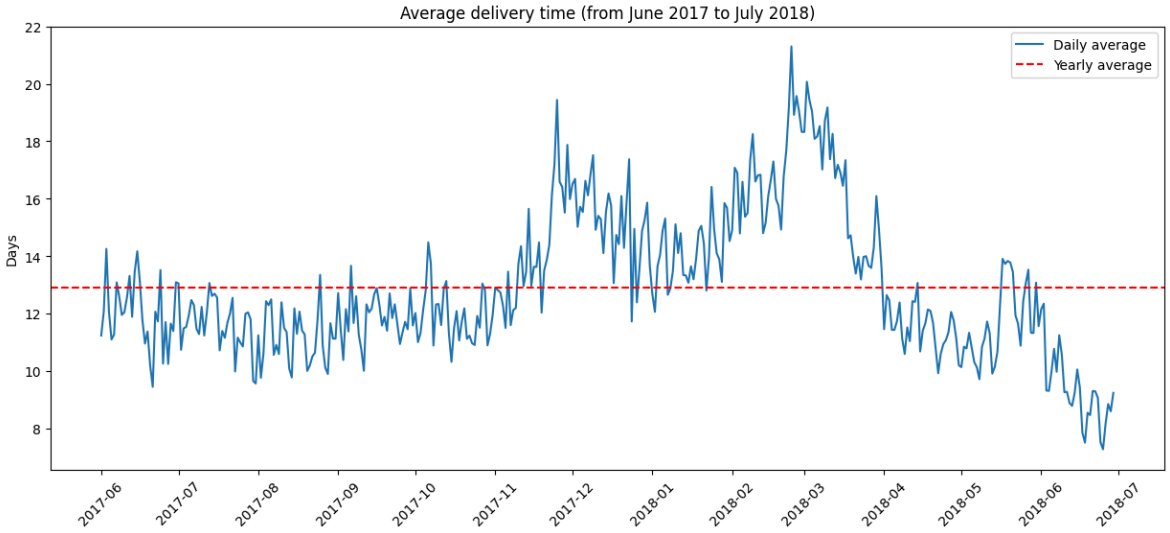
	purchase_date	avg_delivery_time
0	2017-06-01	11.238623
1	2017-06-02	12.079623
2	2017-06-03	14.255051
3	2017-06-04	12.064359
4	2017-06-05	11.101563
...	...	...
389	2018-06-25	7.280438
390	2018-06-26	8.173340
391	2018-06-27	8.854751
392	2018-06-28	8.601704
393	2018-06-29	9.242896

394 rows × 2 columns

Let's plot the time series, with a red line showing the global average shipping time during the period:

In [108…
```
import matplotlib.dates as mdates
```

In [110…
```
plt.figure(figsize=(15,6))
plt.plot ( pd.to_datetime(df['purchase_date']), df['avg_delivery_time'], label='
plt.axhline(y=df['avg_delivery_time'].mean(), color='r', linestyle= '--', label=
plt.ylabel('Days')
plt.title('Average delivery time (from June 2017 to July 2018)')
plt.gca().xaxis.set_major_locator(mdates.MonthLocator())
plt.xticks(rotation=45)
plt.legend()
plt.show()
```



We can observe two clear periods where shipping times rise notably above the average — December 2017 and February–March 2018. The spike in December is likely due to the

holiday season rush, when order volumes typically surge. Meanwhile, the delays in February and March could be attributed to the postal strikes that occurred in parts of Brazil during that period.

Order Reviews:-

The dataset also contains an order_reviews table, where customers can rate their orders on a scale from 1 to 5 and leave written feedback. Let's take a look at the key columns from this table that we'll use for our ana

In [112…
```
view_table('order_reviews', 5).iloc[:, [1, 2, 4]]
```

Out[112…

	order_id	review_score	review_comment_message
0	73fc7af87114b39712e6da79b0a377eb	4	None
1	a548910a1c6147796b98fdf73dbeba33	5	None
2	f9e4b658b201a9f2ecdecbb34bed034b	5	None
3	658677c97b385a9be170737859d3511b	5	Recebi bem antes do prazo estipulado.
4	8e6bfb81e283fa7e4f11123a3fb894f1	5	Parabéns lojas lannister adorei comprar pela I...

Let's count how many orders there are for each review score:

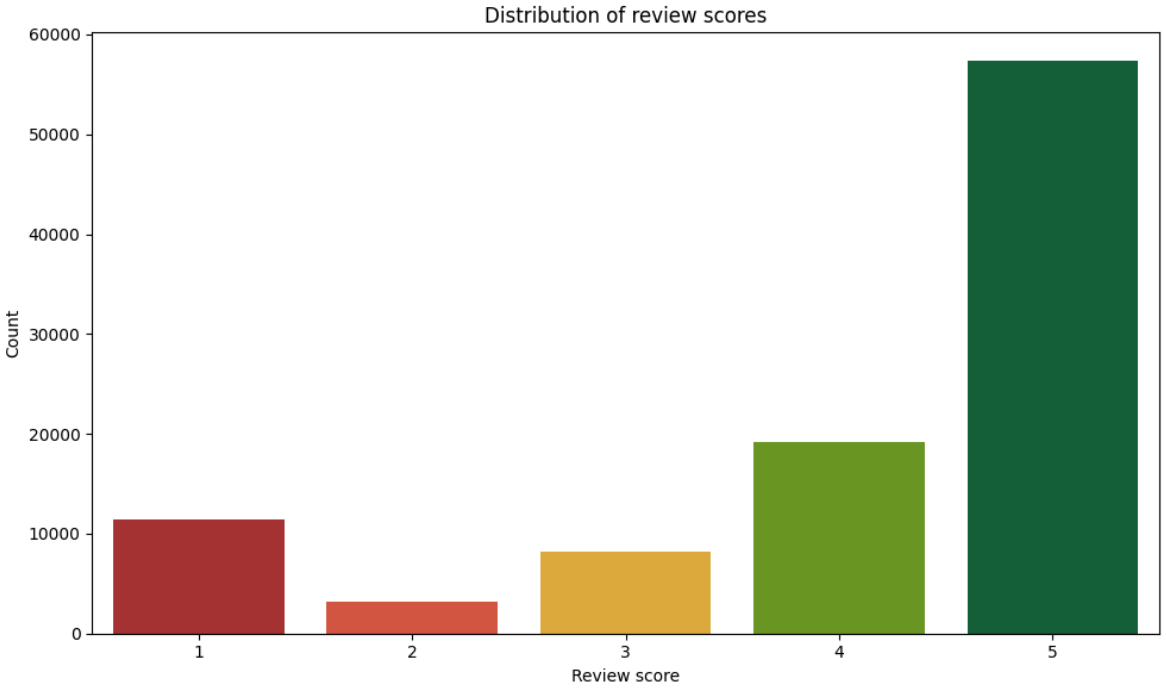
In [9]:
```
review_score_count= """

SELECT
    review_score,
    COUNT(*) AS count
FROM order_reviews
GROUP BY review_score
"""

df=pd.read_sql_query(review_score_count, db_connection)
df
```

Out[9]:

	review_score	count
0	1	11424
1	2	3151
2	3	8179
3	4	19142
4	5	57328

In [10]:
```
plt.figure(figsize=(10,6))
colors = ['#BC2023', '#EB442C', '#F8B324', '#6da814', '#0C6B37']
sns.barplot(x='review_score', y='count', data=df, hue='review_score', palette=co
plt.title('Distribution of review scores')
plt.xlabel('Review score')
```

```
plt.ylabel('Count')
plt.legend().remove()
plt.tight_layout()
plt.show()
```



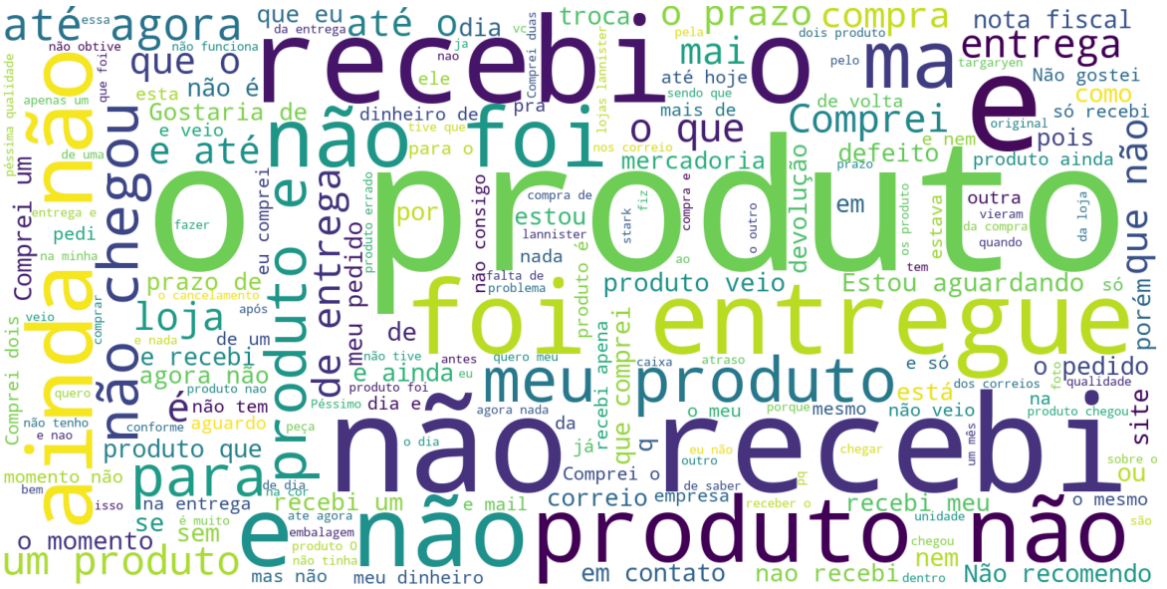
In [11]:
```
negative_comments = """
SELECT GROUP_CONCAT(review_comment_message, ' ') AS comments
FROM order_reviews
WHERE review_score IN(1,2)
"""

negative_comments_df = pd.read_sql(negative_comments, db_connection)['comments']
negative_comments_df[:150]
```

Out[11]:
```
'GOSTARIA DE SABER O QUE HOUVE, SEMPRE RECEBI E ESSA COMPRA AGORA ME DECPCIONOU
Péssimo Não gostei ! Comprei gato por lebre Sempre compro pela Internet'
```

In [2]:
```
!pip install wordcloud
```

```
Requirement already satisfied: wordcloud in e:\anaconda\lib\site-packages (1.9.4)
Requirement already satisfied: numpy>=1.6.1 in e:\anaconda\lib\site-packages (fro
m wordcloud) (2.3.2)
Requirement already satisfied: pillow in e:\anaconda\lib\site-packages (from word
cloud) (11.3.0)
Requirement already satisfied: matplotlib in e:\anaconda\lib\site-packages (from
wordcloud) (3.10.5)
Requirement already satisfied: contourpy>=1.0.1 in e:\anaconda\lib\site-packages
(from matplotlib->wordcloud) (1.3.3)
Requirement already satisfied: cycler>=0.10 in e:\anaconda\lib\site-packages (fro
m matplotlib->wordcloud) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in e:\anaconda\lib\site-packages
(from matplotlib->wordcloud) (4.59.1)
Requirement already satisfied: kiwisolver>=1.3.1 in e:\anaconda\lib\site-packages
(from matplotlib->wordcloud) (1.4.9)
Requirement already satisfied: packaging>=20.0 in e:\anaconda\lib\site-packages
(from matplotlib->wordcloud) (24.2)
Requirement already satisfied: pyparsing>=2.3.1 in e:\anaconda\lib\site-packages
(from matplotlib->wordcloud) (3.2.3)
Requirement already satisfied: python-dateutil>=2.7 in e:\anaconda\lib\site-packa
ges (from matplotlib->wordcloud) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in e:\anaconda\lib\site-packages (from py
thon-dateutil>=2.7->matplotlib->wordcloud) (1.17.0)
```

In [12]:
```
from wordcloud import WordCloud

wordcloud = WordCloud(width=1600, height=800, background_color='white').generate
plt.figure(figsize=(15, 8))
plt.imshow(wordcloud)
plt.axis("off")
plt.show()
```



Customer Segmentation :-

Yes, SQL can definitely be used for customer segmentation! A common and effective approach is the RFM model, which classifies customers based on three key metrics:

. Recency – how recently a customer made their last purchase.

. Frequency – how often they make purchases.

. Monetary value – how much they typically spend.

By calculating and scoring these three factors, we can group customers into segments. This RFM-based segmentation helps businesses personalize marketing efforts, enhance customer retention, and deliver targeted service improvements based on customer behavior patterns.

The next SQL query is slightly longer than the previous ones but still straightforward. We'll divide it into three main steps for clarity:

1. Compute RFM scores – Use the NTILE function across three CTEs to rank each customer from 1 to 5 based on Recency, Frequency, and Monetary Value.

2. Group customers – Assign each customer to one of 11 segments following the method described by
   Mark Rittman's RFM Segmentation Guide

3. Summarize results – Calculate key statistics for each segment, allowing us to visualize their differences in sales, recency, and customer count.

In [27]:
```
rfm_buckets = """
-- 1. Calculate RFM scores
WITH RecencyScore AS (
    SELECT customer_unique_id,
           MAX(order_purchase_timestamp) AS last_purchase,
           NTILE(5) OVER (ORDER BY MAX(order_purchase_timestamp) DESC) AS recenc
    FROM orders
        JOIN customers USING (customer_id)
    WHERE order_status = 'delivered'
    GROUP BY customer_unique_id
),
FrequencyScore AS (
    SELECT customer_unique_id,
           COUNT(order_id) AS total_orders,
           NTILE(5) OVER (ORDER BY COUNT(order_id) DESC) AS frequency
    FROM orders
        JOIN customers USING (customer_id)
    WHERE order_status = 'delivered'

     GROUP BY customer_unique_id
),
MonetaryScore AS (
    SELECT customer_unique_id,
           SUM(price) AS total_spent,
           NTILE(5) OVER (ORDER BY SUM(price) DESC) AS monetary
    FROM orders
        JOIN order_items USING (order_id)
        JOIN customers USING (customer_id)
    WHERE order_status = 'delivered'
    GROUP BY customer_unique_id
),

-- 2. Assign each customer to a group
RFM AS (
    SELECT last_purchase, total_orders, total_spent,
```

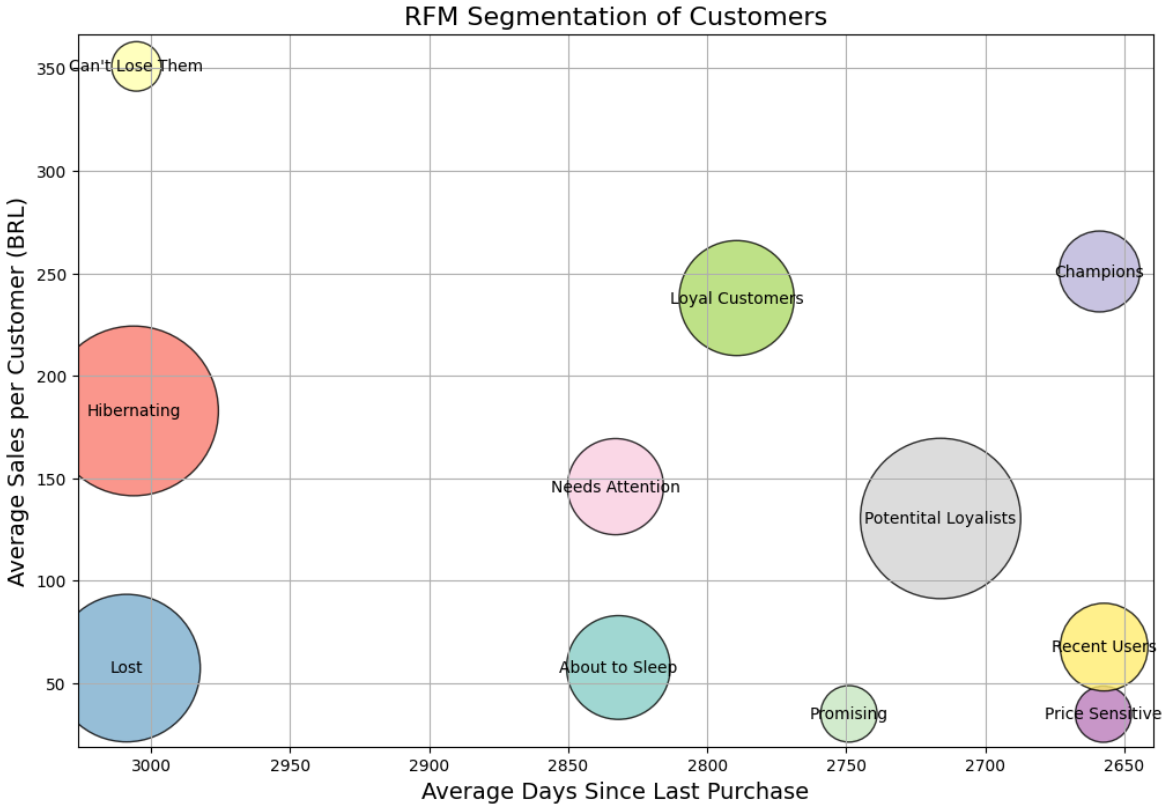
```
        CASE
            WHEN recency = 1 AND frequency + monetary IN (1, 2, 3, 4) THEN "Cham
            WHEN recency IN (4, 5) AND frequency + monetary IN (1, 2) THEN "Can'
            WHEN recency IN (4, 5) AND frequency + monetary IN (3, 4, 5, 6) THEN
            WHEN recency IN (4, 5) AND frequency + monetary IN (7, 8, 9, 10) THE
            WHEN recency IN (2, 3) AND frequency + monetary IN (1, 2, 3, 4) THEN
            WHEN recency = 3 AND frequency + monetary IN (5, 6) THEN "Needs Atte
            WHEN recency = 1 AND frequency + monetary IN (7, 8) THEN "Recent Use
            WHEN recency = 1 AND frequency + monetary IN (5, 6) OR
                recency = 2 AND frequency + monetary IN (5, 6, 7, 8) THEN "Poten
            WHEN recency = 1 AND frequency + monetary IN (9, 10) THEN "Price Sen
            WHEN recency = 2 AND frequency + monetary IN (9, 10) THEN "Promising
            WHEN recency = 3 AND frequency + monetary IN (7, 8, 9, 10) THEN "Abo
        END AS RFM_Bucket
    FROM RecencyScore
        JOIN FrequencyScore USING (customer_unique_id)
        JOIN MonetaryScore USING (customer_unique_id)
)

-- 3. Calculate group statistics for plotting
SELECT RFM_Bucket,
       AVG(JULIANDAY('now') - JULIANDAY(last_purchase)) AS avg_days_since_purcha
       AVG(total_spent / total_orders) AS avg_sales_per_customer,
       COUNT(*) AS customer_count
FROM RFM
GROUP BY RFM_Bucket
"""

df = pd.read_sql(rfm_buckets, db_connection)
df
```

Out[27]:

	RFM_Bucket	avg_days_since_purchase	avg_sales_per_customer	customer_count
0	About to Sleep	2831.920751	57.684959	7584
1	Can't Lose Them	3005.224060	350.886817	1723
2	Champions	2658.951317	250.856821	4607
3	Hibernating	3006.233737	182.845816	20288
4	Lost	3008.762515	57.393210	15331
5	Loyal Customers	2789.416780	237.881257	9315
6	Needs Attention	2832.977831	145.904925	6510
7	Potentital Loyalists	2716.111949	130.374773	18114
8	Price Sensitive	2657.575912	34.909351	2220
9	Promising	2749.065584	35.085359	2245
10	Recent Users	2657.324656	67.642129	5421

Let's bring this to life visually — we'll plot each customer group with average recency on the x-axis, average sales per customer on the y-axis, and use the circle size to represent the number of customers in each group.

In [30]:
```
plt.figure(figsize=(12, 8))

# Create scatter plot
plt.scatter(
    df['avg_days_since_purchase'],
    df['avg_sales_per_customer'],
    s=df['customer_count'] * 0.55,
    c=range(len(df)),  # Use a color index
    cmap='Set3',       # Set color map
    alpha=0.8,
    edgecolors='black'
)

# Labels and title
plt.xlabel('Average Days Since Last Purchase', fontsize=14)
plt.ylabel('Average Sales per Customer (BRL)', fontsize=14)
plt.title('RFM Segmentation of Customers', fontsize=16)
plt.grid(True)

# Add group labels
for i in range(len(df)):
    plt.annotate(df['RFM_Bucket'].iloc[i],
                 (df['avg_days_since_purchase'].iloc[i],
                  df['avg_sales_per_customer'].iloc[i]),
                 ha='center', va='center', fontsize=10)

# Reverse x-axis to show recent customers on left
plt.gca().invert_xaxis()

plt.show()
```



The previous chart highlights how each customer segment differs in terms of recency and spending. For instance, the 'Champions' group (top right) consists of customers who buy frequently and spend more, whereas the 'Lost' group (bottom left) includes customers with low spending and long inactivity.

To deepen our understanding of customer loyalty, let's visualize the ratio of one-time buyers to repeat customers and see how many users continue purchasing through Olist.

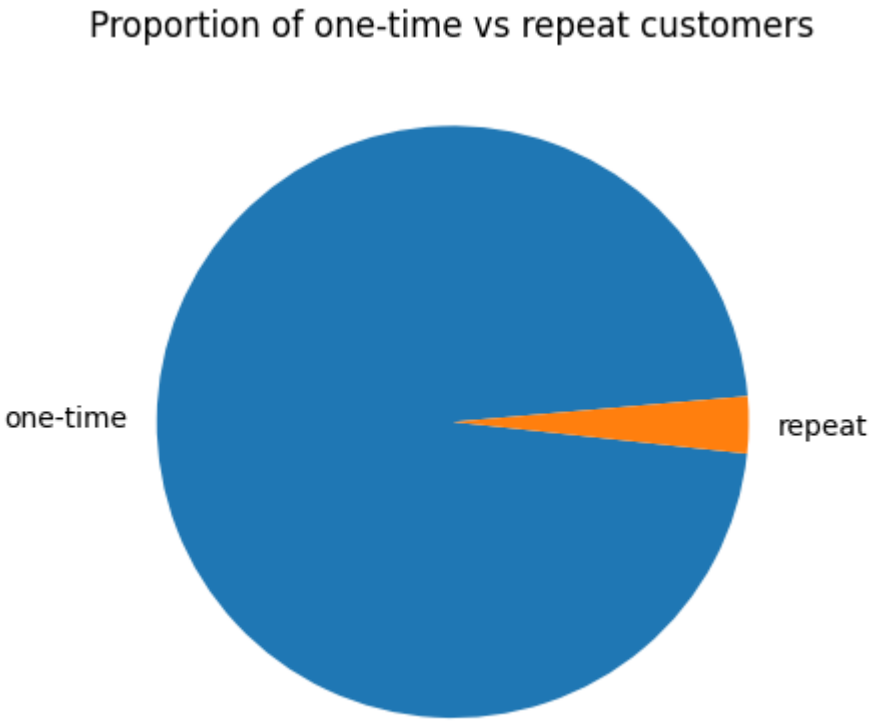
In [34]:
```
repeat_customer_proportion = """
WITH CustomerOrders AS (
    SELECT COUNT (orders.order_id) AS order_count
    FROM orders JOIN customers USING (customer_id)
    GROUP BY customers.customer_unique_id
)
SELECT
    CASE WHEN order_count > 1 THEN 'repeat' ELSE 'one-time' END AS order_type,
    ROUND(100.0*COUNT(*) / (SELECT COUNT(*) FROM CustomerOrders),1) AS proportio
FROM CustomerOrders
GROUP BY order_type
"""

df=pd.read_sql_query(repeat_customer_proportion, db_connection)
df
```

Out[34]:

	order_type	proportion
0	one-time	96.9
1	repeat	3.1

In [35]:
```
fig, ax=plt.subplots()
ax.pie(df['proportion'], labels=df['order_type'],startangle=5)
```

```
ax.set_title('Proportion of one-time vs repeat customers')
fig.set_facecolor('white')
plt.show()
```



As we can see most customers only place a single order through Olist.

Customer Lifetime Value:-

Customer Lifetime Value (CLV) is a key business metric that estimates the total revenue a company can expect from a customer over the entire duration of their relationship. To calculate CLV for our dataset, we'll use the order_payments table, which provides details about the payments made and the payment methods used.

In [38]:
```
import pandas as pd

query = "SELECT * FROM order_payments LIMIT 5;"
df = pd.read_sql_query(query, db_connection)
df
```

Out[38]:

	order_id	payment_sequential	payment_type	payment_ins
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	

In [39]:
```
def view_table(table_name, n=5):
    query = f"SELECT * FROM {table_name} LIMIT {n};"
    return pd.read_sql_query(query, db_connection)
```

In [40]:
```
view_table('order_payments', 5)
```

Out[40]:

	order_id	payment_sequential	payment_type	payment_ins
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	

To begin, we'll calculate the key components of Customer Lifetime Value (CLV) for each customer:

1. Purchase Frequency (PF): The total number of orders placed by each customer.
2. Average Order Value (AOV): The total amount spent divided by the number of orders.
3. Average Customer Lifespan (ACL): The duration (in weeks) between the first and last purchase, with a minimum of 1 week.

For better readability, we'll first collect the required data using a CTE, and then compute each CLV component in the main query.

In [29]:
```
clv = """
WITH CustomerData AS (
    SELECT
        customer_unique_id,
        customer_zip_code_prefix AS zip_code_prefix,
        COUNT(DISTINCT orders.order_id) AS order_count,
        SUM(payment_value) AS total_payment,
        JULIANDAY(MIN(order_purchase_timestamp)) AS first_order_day,
        JULIANDAY(MAX(order_purchase_timestamp)) AS last_order_day
    FROM customers
        JOIN orders USING (customer_id)
        JOIN order_payments USING (order_id)
    GROUP BY customer_unique_id
)
SELECT
    customer_unique_id,
    zip_code_prefix,
    order_count AS PF,
    total_payment / order_count AS AOV,
      CASE
        WHEN (last_order_day - first_order_day) < 7 THEN
            1
        ELSE
            (last_order_day - first_order_day) / 7
```

```
        END AS ACL
FROM CustomerData
"""

pd.read_sql(clv, db_connection)
```

Out[29]:

	customer_unique_id	zip_code_prefix	PF	AOV	ACL
0	0000366f3b9a7992bf8c76cfdf3221e2	7787	1	141.90	1.0
1	0000b849f77a49e4a4ce2b2a4ca5be3f	6053	1	27.19	1.0
2	0000f46a3911fa3c0805444483337064	88115	1	86.22	1.0
3	0000f6ccb0745a6a4b88665a16c9f078	66812	1	43.62	1.0
4	0004aac84e0df4da2b147fca70cf8255	18040	1	196.89	1.0
...	...	...	...	...	...
96090	fffcf5a5ff07b0908bd4e2dbc735a684	55250	1	2067.42	1.0
96091	fffea47cd6d3cc0a88bd621562a9d061	44054	1	84.58	1.0
96092	ffff371b4d645b6ecea244b27531430a	78552	1	112.46	1.0
96093	ffff5962728ec6157033ef9805bacc48	29460	1	133.69	1.0
96094	ffffd2657e2aad2907e67c3e9daecbeb	83608	1	71.56	1.0

96095 rows × 5 columns

Since our goal is to visualize the geographic distribution of CLV across Brazil, we'll include the latitude and longitude for each zip code prefix. This information is available in the geolocation table.

In [33]:
```
import pandas as pd

query = "SELECT * FROM geolocation LIMIT 5;"
df = pd.read_sql_query(query, db_connection)
df
```

Out[33]:

	geolocation_zip_code_prefix	geolocation_lat	geolocation_lng	geolocation_city	geolo
0	1037	-23.545621	-46.639292	sao paulo	
1	1046	-23.546081	-46.644820	sao paulo	
2	1046	-23.546129	-46.642951	sao paulo	
3	1041	-23.544392	-46.639499	sao paulo	
4	1035	-23.541578	-46.641607	sao paulo	

In [34]:
```
def view_table(table_name, n=5):
    query = f"SELECT * FROM {table_name} LIMIT {n};"
    return pd.read_sql_query(query, db_connection)
```

In [35]:
```
view_table('geolocation', 5)
```

Out[35]:

	geolocation_zip_code_prefix	geolocation_lat	geolocation_lng	geolocation_city	geolo
0	1037	-23.545621	-46.639292	sao paulo	
1	1046	-23.546081	-46.644820	sao paulo	
2	1046	-23.546129	-46.642951	sao paulo	
3	1041	-23.544392	-46.639499	sao paulo	
4	1035	-23.541578	-46.641607	sao paulo	

Building on the previous query as a CTE, we can now compute the average CLV and customer count for each zip code prefix. To determine the CLV, we'll simply multiply its three components — Purchase Frequency (PF), Average Order Value (AOV), and Average Customer Lifespan (ACL).

$$CLV = PF.AOV.ACL$$

In [36]:
```
clv = """
SELECT
    c.customer_unique_id,
    c.customer_zip_code_prefix AS zip_code_prefix,
    COUNT(DISTINCT o.order_id) AS PF,  -- Purchase Frequency
    AVG(oi.price) AS AOV,              -- Average Order Value
    COUNT(DISTINCT o.order_id) / COUNT(DISTINCT o.customer_id) AS ACL
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
WHERE o.order_status = 'delivered'
GROUP BY c.customer_unique_id, c.customer_zip_code_prefix
"""
```

In [37]:
```
avg_clv_per_zip_prefix = f"""
WITH CLV AS (
    {clv}
)

SELECT
    zip_code_prefix AS zip_prefix,
    AVG(PF*AOV*ACL) AS avg_clv,
    COUNT(customer_unique_id) AS customer_count,
    geolocation_lat AS latitude,
    geolocation_lng AS longitude
FROM CLV
    JOIN geolocation ON clv.zip_code_prefix = geolocation.geolocation_zip_code_p
GROUP BY zip_code_prefix
"""

df=pd.read_sql(avg_clv_per_zip_prefix, db_connection)
df
```

Out[37]:

	zip_prefix	avg_clv	customer_count	latitude	longitude
0	1003	89.990	17	-23.549032	-46.635313
1	1004	89.495	44	-23.550116	-46.635122
2	1005	98.118	125	-23.549819	-46.635606
3	1006	417.400	18	-23.550524	-46.636694
4	1007	113.870	104	-23.550393	-46.637302
...	...	...	...	...	...
14732	99955	46.880	36	-28.107588	-52.144019
14733	99960	125.000	5	-27.953797	-52.029641
14734	99965	81.950	12	-28.173892	-52.038447
14735	99970	229.000	21	-28.345143	-51.876926
14736	99980	59.445	52	-28.389218	-51.846012

14737 rows × 5 columns

Using this data, we can generate an interactive map with the Folium library. Each zip code prefix will be represented by a circle, where the circle's opacity reflects the average CLV, and the circle's size represents the number of customers in that area.

In [38]:
```
print(df.columns)
print(df.head())
```

```
Index(['zip_prefix', 'avg_clv', 'customer_count', 'latitude', 'longitude'], dtype
='object')
   zip_prefix  avg_clv  customer_count   latitude  longitude
0        1003   89.990              17 -23.549032 -46.635313
1        1004   89.495              44 -23.550116 -46.635122
2        1005   98.118             125 -23.549819 -46.635606
3        1006  417.400              18 -23.550524 -46.636694
4        1007  113.870             104 -23.550393 -46.637302
```

In [39]:
```
import folium
import numpy as np

# Create base map centered on Brazil
m = folium.Map(location=[-14.2350, -51.9253], zoom_start=4)

# Add circle markers for each location
for i, row in df.iterrows():
    folium.CircleMarker(
        location=[row['latitude'], row['longitude']],   # ✅ now correctly rena
        radius=0.1 * np.sqrt(row['customer_count']),
        color=None,
        fill_color='#85001d',
        fill_opacity=0.1 + 0.1 * np.sqrt(row['avg_clv'] / df['avg_clv'].max()),
        popup=(
            f"<b>Zip Code Prefix:</b> {int(row['zip_prefix'])}<br>"
            f"<b>Average CLV:</b> {int(row['avg_clv'])}<br>"
            f"<b>Customers:</b> {int(row['customer_count'])}"
```

```
        )
    ).add_to(m)

m
```

Out[39]:

Product Affinity:-

Which products are typically purchased together? While the Olist dataset doesn't include product names for privacy reasons, we do have information on product categories. Let's gather the data needed to build a graph showing which categories frequently appear in the same order. To start, we'll create a list of products that appear in more than five orders:

In [32]:

```
min_orders = 5

most_ordered_products = f"""
SELECT
    products.product_id,
    product_category_name_english AS category,
    COUNT(order_id) AS orders_count
FROM order_items
    JOIN products USING (product_id)
    JOIN product_category_name_translation AS tr
        ON products.product_category_name = tr.product_category_name
GROUP BY products.product_id
    HAVING COUNT(order_id) > {min_orders}
"""

most_ordered_products_df = pd.read_sql(most_ordered_products, db_connection)
most_ordered_products_df
```

Out[32]:

	product_id	category	orders_count
0	001795ec6f1b187d37335e1c4704762e	consoles_games	9
1	001b72dfd63e9833e8c02742adf472e3	furniture_decor	14
2	00210e41887c2a8ef9f791ebc780cc36	health_beauty	7
3	002159fe700ed3521f46cfcf6e941c76	fashion_shoes	8
4	00250175f79f584c14ab5cecd80553cd	housewares	11
...	...	...	...
3798	ffb97eb64c6fe1baada2410288c04457	perfumery	8
3799	ffc0b406806006602c5853b00ab5f7fd	christmas_supplies	6
3800	ffc9caf33e2d1e9f44e3e06da19085f7	health_beauty	27
3801	ffd4bf4306745865e5692f69bd237893	fashion_bags_accessories	8
3802	fff0a542c3c62682f23305214eaeaa24	stationery	8

3803 rows × 3 columns

Next, we need to capture the relationships between products. We can do this by counting how many orders each pair of products appears in together. This can be achieved with a self-join on the orders table to generate all product combinations within the same order, and then grouping by product pairs:

In [35]:
```
most_ordered_product_ids = tuple(most_ordered_products_df['product_id'])

products_often_ordered_together = f"""
SELECT
    oi1.product_id AS product_id1,
    oi2.product_id AS product_id2,
    COUNT(DISTINCT oi1.order_id) AS common_orders_count
FROM order_items AS oi1
    JOIN order_items AS oi2
        ON oi1.order_id = oi2.order_id -- Same order
        AND oi1.product_id < oi2.product_id -- Avoid permutations
WHERE
    oi1.product_id IN {most_ordered_product_ids} AND
    oi2.product_id IN {most_ordered_product_ids}
GROUP BY
    oi1.product_id,
    oi2.product_id
    HAVING COUNT(DISTINCT oi1.order_id) > {min_orders}
"""


products_often_ordered_together_df = pd.read_sql(products_often_ordered_together
products_often_ordered_together_df
```

Out[35]:

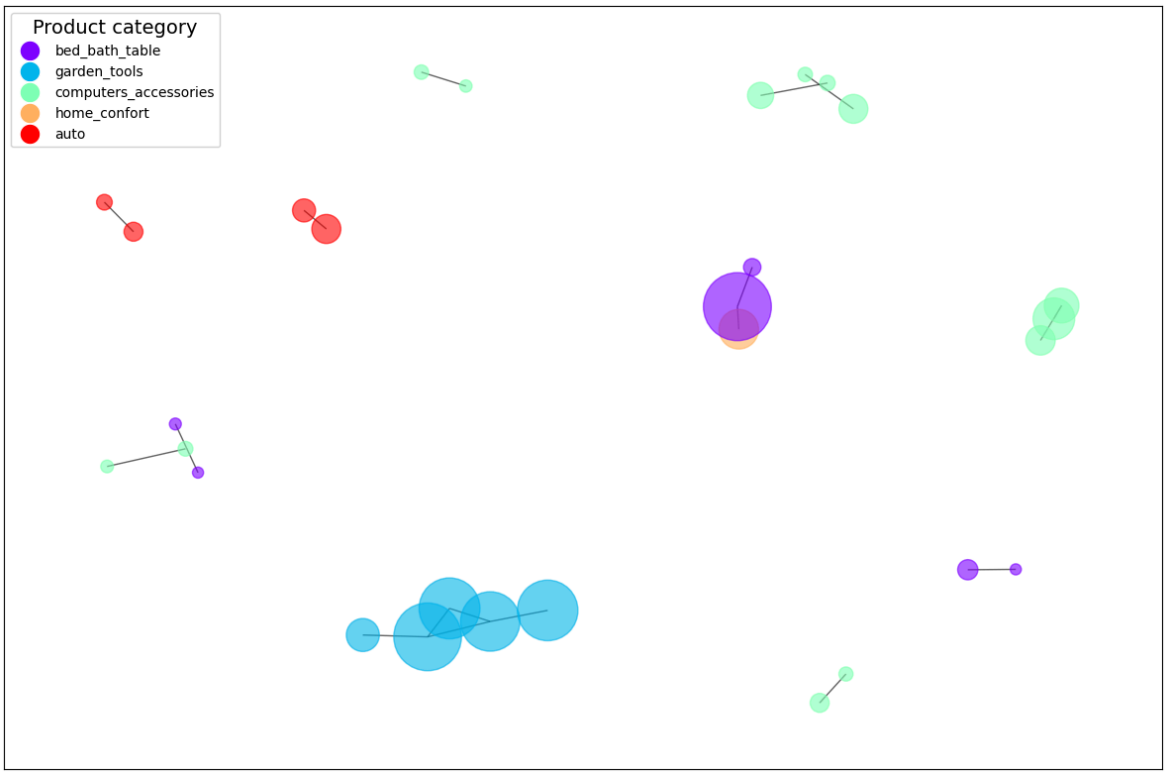
	product_id1	product_id2	common_o
0	060cb19345d90064d1015407193c233d	98d61056e0568ba048e5d78038790e77	
1	0bcc3eeca39e1064258aa1e932269894	422879e10f46682990de24d770e7f83d	
2	0d85c435fd60b277ffb9e9b0f88f927a	ee57070aa3b24a06fdd0e02efd2d757d	
3	18486698933fbb64af6c0a255f7dd64c	dbb67791e405873b259e4656bf971246	
4	35afc973633aaeb6b877ff57b2793310	99a4788cb24856965c36a24e339b6058	
5	368c6c730842d78016ad823897a372db	53759a2ecddad2bb87a079a1f1519f73	
6	36f60d45225e60c7da4558b070ce4b60	3f14d740544f37ece8a9e7bc8349797e	
7	36f60d45225e60c7da4558b070ce4b60	e53e557d5a159f5aa2c5e995dfdf244b	
8	389d119b48cf3043d311335e499d9c6b	422879e10f46682990de24d770e7f83d	
9	389d119b48cf3043d311335e499d9c6b	53759a2ecddad2bb87a079a1f1519f73	
10	422879e10f46682990de24d770e7f83d	53759a2ecddad2bb87a079a1f1519f73	
11	4d0ec1e9b95fb62f9a1fbe21808bf3b1	9ad75bd7267e5c724cb42c71ac56ca72	
12	4fcb3d9a5f4871e8362dfedbdb02b064	f4f67ccaece962d013a4e1d7dc3a61f7	
13	5b8a5a9417210b1b84b67b9a7aefb935	e5ae72c62ebfa708624f5029d609b160	
14	5d790355cbeded0cd60e25cbc4c527a2	5fc3e6a4b52b0c414458104ed4037f1c	
15	946344697156947d846d27fe0d503033	ad0a798e7941f3a5a2fb8139cb62ad78	
16	99a4788cb24856965c36a24e339b6058	f2e53dd1670f3c376518263b3f71424d	
17	ad4b5def91ac7c575dbdf65b5be311f4	e6b314a2236c162ede1a879f1075430f	

Using the two datasets—the list of frequently ordered products and the counts of product pairs appearing together—we can now build a graph that visualizes which products are most often purchased together:

In [40]:

```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

G = nx.Graph()

# Add nodes
for _, product in most_ordered_products_df.iterrows():
    G.add_node(product['product_id'], category=product['category'], orders_count

# Add edges
for _, pair in products_often_ordered_together_df.iterrows():
    G.add_edge(pair['product_id1'], pair['product_id2'], weight=pair['common_ord

# Remove isolated nodes
G.remove_nodes_from(list(nx.isolates(G)))

# Node colors by category
categories = list(set(nx.get_node_attributes(G, 'category').values()))
```

```
colors = plt.cm.rainbow(np.linspace(0, 1, len(categories)))
category_colors = dict(zip(categories, colors))

# Draw graph
plt.figure(figsize=(12, 8))
pos = nx.spring_layout(G, k=0.5, iterations=50)
node_sizes = [G.nodes[n]['orders_count'] * 5 for n in G.nodes]
node_colors = [category_colors[G.nodes[n]['category']] for n in G.nodes]

nx.draw_networkx(G, pos, node_color=node_colors, node_size=node_sizes, alpha=0.6

# Legend
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label=cat, marker
                   for cat, color in category_colors.items()]
plt.legend(handles=legend_elements, title="Product category", loc='upper left',

plt.tight_layout()
plt.show()
```



In this graph, each circle represents a product, with its color showing the product category. A line connecting two products indicates that they are frequently purchased together in the same order. The size of each circle reflects the product's importance, measured by the number of orders it appears in across the dataset.

We can see that five products in the Garden Tools category are often bought together. Additionally, a product from the Bed, Bath & Table category is frequently purchased alongside an item from the Home Comfort category. Similarly, several products in the Auto and Computer Accessories categories are commonly bought together within their respective categories.

Sellers:-

The table sellers contains 3095 sellers that use Olist's services. Let's have a look at the first few rows:

In [5]:
```
import pandas as pd

query = "SELECT * FROM sellers LIMIT 5;"
df = pd.read_sql_query(query, db_connection)
df
```

Out[5]:

	seller_id	seller_zip_code_prefix	seller_city	seller_state
0	3442f8959a84dea7ee197c632cb2df15	13023	campinas	SP
1	d1b65fc7debc3361ea86b5f14c68d2e2	13844	mogi guacu	SP
2	ce3ad9de960102d0677a81f5d0bb7b2d	20031	rio de janeiro	RJ
3	c0f3eea2e14555b6faeea3dd58c1b1c3	4195	sao paulo	SP
4	51a04a8a6bdcb23deccc82b0b80742cf	12914	braganca paulista	SP

In [6]:
```
def view_table(table_name, n=5):
    query = f"SELECT * FROM {table_name} LIMIT {n};"
    return pd.read_sql_query(query, db_connection)
```

In [7]:
```
view_table('sellers', 5)
```

Out[7]:

	seller_id	seller_zip_code_prefix	seller_city	seller_state
0	3442f8959a84dea7ee197c632cb2df15	13023	campinas	SP
1	d1b65fc7debc3361ea86b5f14c68d2e2	13844	mogi guacu	SP
2	ce3ad9de960102d0677a81f5d0bb7b2d	20031	rio de janeiro	RJ
3	c0f3eea2e14555b6faeea3dd58c1b1c3	4195	sao paulo	SP
4	51a04a8a6bdcb23deccc82b0b80742cf	12914	braganca paulista	SP

Do most sellers have only a few sales, or are they large companies with high sales volumes? And how do review scores differ across sellers? We can explore both questions using a scatterplot. First, let's extract the necessary data from the sellers table:

In [47]:
```
seller_review_scores_and_sales = '''
SELECT
    sellers.seller_id,
    AVG(order_reviews.review_score) AS avg_review_score,
    SUM(order_items.price) AS total_sales,
    COUNT(orders.order_id) AS num_orders
FROM
    sellers
```

```
    LEFT JOIN order_items ON sellers.seller_id = order_items.seller_id
    LEFT JOIN orders ON order_items.order_id = orders.order_id
    LEFT JOIN order_reviews ON orders.order_id = order_reviews.order_id
GROUP BY
    sellers.seller_id
HAVING
    COUNT(orders.order_id) > 10
'''


df=pd.read_sql_query(seller_review_scores_and_sales, db_connection)
df
```

Out[47]:

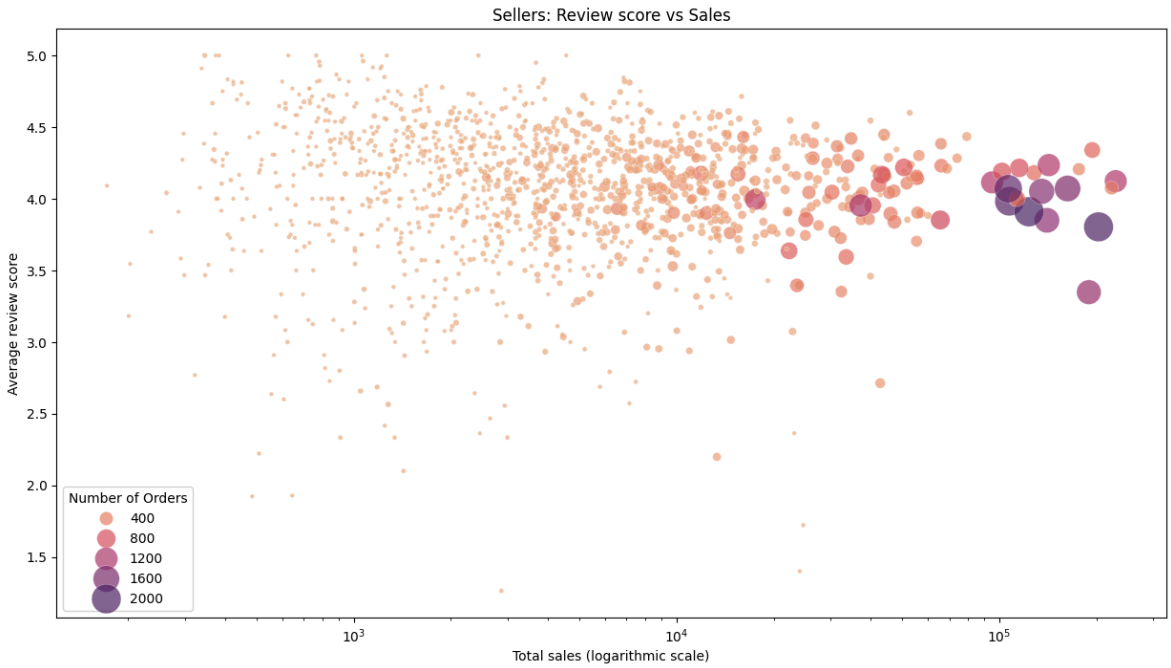
	seller_id	avg_review_score	total_sales	num_orders
0	001cca7ae9ae17fb1caed9dfb1094831	3.902542	25080.03	239
1	002100f778ceb8431b7a1020ff7ab48f	3.982143	1254.40	56
2	004c9cd9d87a3c30c522c48c4fc07416	4.132948	20467.18	176
3	00720abe85ba0859807595bbf045a33b	3.653846	1007.50	26
4	00ee68308b45bc5e2660cd833c3f81cc	4.331395	20345.00	173
...	...	...	...	...
1310	ffc470761de7d0232558ba5e786e57b7	4.300000	1649.01	31
1311	ffdd9f82b9a447f6f8d4b91554cc7dd3	4.250000	2101.20	20
1312	ffeee66ac5d5a62fe688b9d26f83f534	4.214286	1839.86	14
1313	fffd5413c0700ac820c7069d66d98c89	3.866667	9062.30	61
1314	ffff564a4f9085cd26170f4732393726	2.100000	1426.30	20

1315 rows × 4 columns

We'll build a scatterplot where each point represents a seller. The y-axis will show the seller's average order review score, and the x-axis will display total sales on a logarithmic scale—this helps visualize sellers with low sales. Additionally, we'll use color and size to reflect the number of orders per seller:

In [49]:

```
plt. figure(figsize=(15,8))
sns.scatterplot(data=df, x='total_sales',
                y='avg_review_score',
                size='num_orders',
                sizes=(10,500),
               hue='num_orders',
               palette = 'flare',
               alpha = 0.7)
plt.xscale('log')
plt.xlabel('Total sales (logarithmic scale)')
plt.ylabel('Average review score')
plt.title('Sellers: Review score vs Sales')
plt.legend(title='Number of Orders')
plt.show()
```



Most of Olist's sellers are small, handling only a few orders. As expected, larger sellers show less variation in their average order review scores, typically falling between 3.5 and 4.5. Interestingly, among the large sellers on the right of the plot, there's one with noticeably lower scores compared to its peers. We can also see that sellers with similar total sales can have very different numbers of orders (bubbles of different sizes at the same x-position), indicating that some sellers have a higher average order value than others.

To gain further insight, let's create another plot showing the distribution of sellers by order volume. We'll divide sellers into four groups based on the number of orders they dispatched:

. Group 1: 1–9 orders

. Group 2: 10–99 orders

. Group 3: 100–199 orders

. Group 4: 1000 or more orders

In [13]:
```
bucketed_sellers="""
SELECT
    seller_id,
    CASE
    WHEN COUNT(order_id) BETWEEN 1 AND 9 THEN '1-9 orders'
    WHEN COUNT(order_id) BETWEEN 10 AND 99 THEN '10-99 orders'
    WHEN COUNT(order_id) BETWEEN 100 AND 999 THEN '100-999 orders'
        ELSE '1000+ orders'
    END AS bucket
FROM order_items
GROUP BY  seller_id
"""

pd.read_sql_query(bucketed_sellers, db_connection).head(5)
```

Out[13]:

	seller_id	bucket
0	0015a82c2db000af6aaaf3ae2ecb0532	1-9 orders
1	001cca7ae9ae17fb1caed9dfb1094831	100-999 orders
2	001e6ad469a905060d959994f1b41e4f	1-9 orders
3	002100f778ceb8431b7a1020ff7ab48f	10-99 orders
4	003554e2dce176b5555353e4f3555ac8	1-9 orders

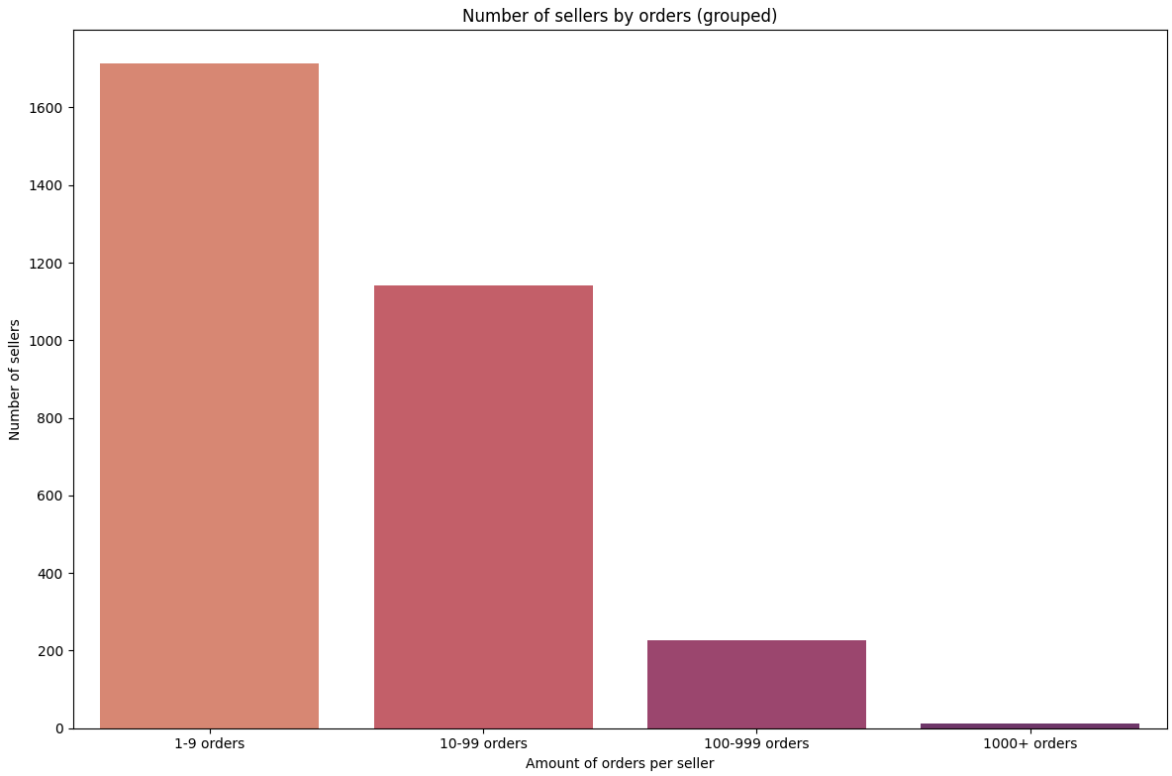
In [18]:
```
sellers_per_bucket= f"""
WITH BucketedSellers AS (
   {bucketed_sellers}
)
SELECT
    bucket,
    COUNT(seller_id) AS seller_count
FROM BucketedSellers
GROUP BY bucket
"""

seller_buckets = pd.read_sql_query(sellers_per_bucket, db_connection)
seller_buckets
```

Out[18]:

	bucket	seller_count
0	1-9 orders	1714
1	10-99 orders	1142
2	100-999 orders	226
3	1000+ orders	13

Let's visualize the dataframe:

In [19]:
```
plt.figure(figsize=(12, 8))
sns.barplot(x='bucket', y='seller_count', data=seller_buckets, hue='bucket', pal
plt.title('Number of sellers by orders (grouped)')
plt.xlabel('Amount of orders per seller')
plt.ylabel('Number of sellers')
plt.tight_layout()
plt.show()
```



We can confirm that the majority of sellers are small-scale, with only a few handling more than 1,000 orders.

To explore whether larger sellers benefit from faster delivery times, let's visualize the distribution of shipping durations across the four seller groups defined earlier. We'll reuse the previous CTE to extract the required data for this comparison.

In [20]:
```
seller_shipping_times = f"""
WITH BucketedSellers AS (
    {bucketed_sellers}
)
SELECT
    bucket,
    BucketedSellers.seller_id,
    JULIANDAY(order_delivered_customer_date) - JULIANDAY(order_purchase_timestam
        AS delivery_time
FROM orders
    JOIN order_items USING (order_id)
    JOIN BucketedSellers USING (seller_id)
WHERE order_status = 'delivered'
"""

df = pd.read_sql_query(seller_shipping_times, db_connection)
df
```

Out[20]:

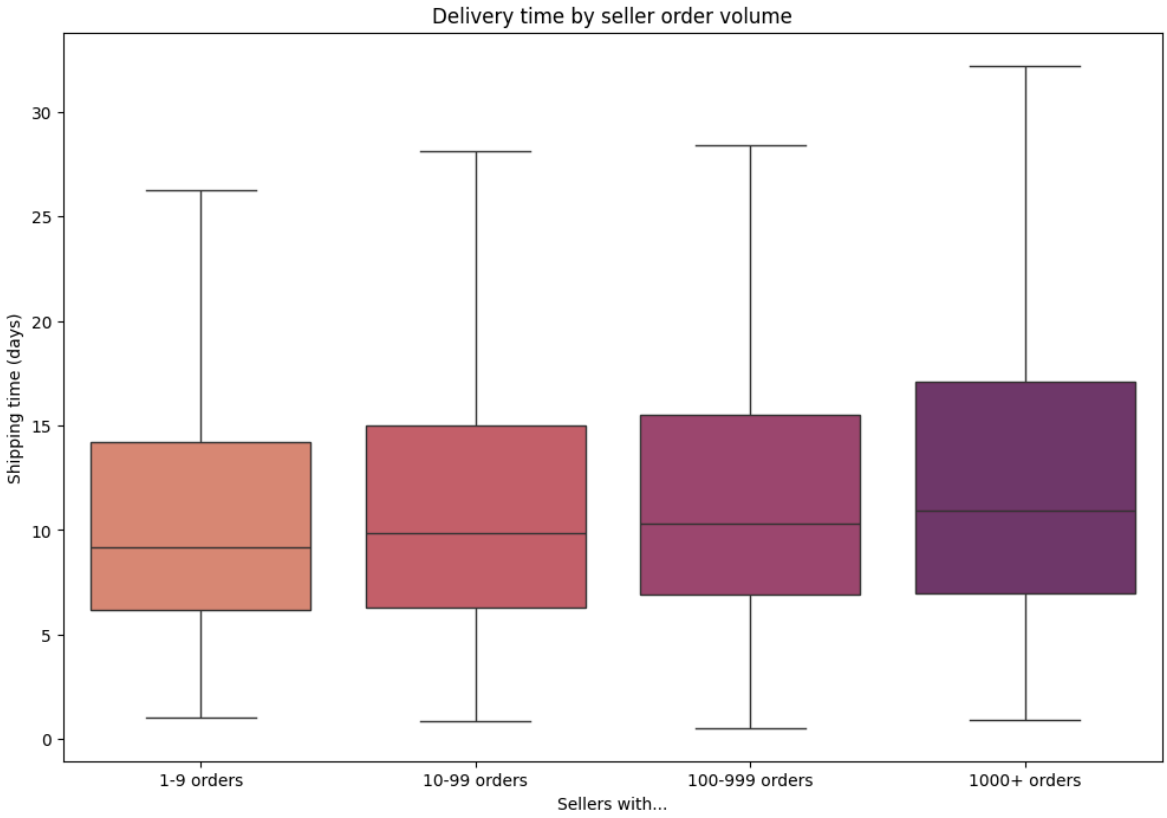
	bucket	seller_id	delivery_time
0	10-99 orders	3504c0cb71d7fa48d967e0e4c94d59d9	8.436574
1	100-999 orders	289cdb325fb7e7f891c38608bf9e0962	13.782037
2	1000+ orders	4869f7a5dfa277a7dca6462dcf3b52b2	9.394213
3	100-999 orders	66922902710d126a0e7d26b0e3805106	13.208750
4	100-999 orders	2c9e548be18521d1c43cde1c582c6de8	2.873877
...	...	...	...
110192	10-99 orders	1f9ab4708f3056ede07124aad39a2554	22.193727
110193	10-99 orders	d50d79cb34e38265a8649c383dcffd48	24.859421
110194	100-999 orders	a1043bafd471dff536d0c462352beb48	17.086424
110195	100-999 orders	a1043bafd471dff536d0c462352beb48	17.086424
110196	100-999 orders	ececbfcff9804a2d6b40f589df8eef2b	7.674306

110197 rows × 3 columns

Let's draw four boxplots, corresponding to each bar in the previous plot:

In [21]:
```
plt.figure(figsize=(12, 8))
palette = sns.color_palette('flare', len(seller_buckets['bucket']))
sns.boxplot(x='bucket', y='delivery_time', data=df, order=seller_buckets['bucket
            hue='bucket', palette=palette[1:]+palette[:1], dodge=False)
plt.title('Delivery time by seller order volume')
plt.xlabel('Sellers with...')
plt.ylabel('Shipping time (days)')
plt.legend().remove()
plt.show()
```

```
C:\Users\intel\AppData\Local\Temp\ipykernel_8320\1530691160.py:8: UserWarning: No
artists with labels found to put in legend.  Note that artists whose label start
with an underscore are ignored when legend() is called with no argument.
  plt.legend().remove()
```



Interestingly, the data suggests that larger sellers tend to have longer delivery times. This raises an important question — what might be causing this delay? Could it be that smaller sellers hand over packages to the shipping company more quickly after order approval? I'll leave it to you to further explore the dataset and uncover the factors behind this trend.

Lead Conversion:-

To conclude our exploration of the Olist database, let's examine the final two tables. The first one, leads_qualified, contains information about sellers who submitted a contact request form on Olist's website and were subsequently filtered and approved by the sales team — meaning they became qualified leads.

In [24]:
```
import pandas as pd

query = "SELECT * FROM leads_qualified LIMIT 5;"
df = pd.read_sql_query(query, db_connection)
df
```

Out[24]:

	mql_id	first_contact_date	landing_pa
0	dac32acd4db4c29c230538b72f8dd87d	2018-02-01	88740e65d5d6b056e0cda098e1e
1	8c18d1de7f67e60dbd64e3c07d7e9d5d	2017-10-20	007f9098284a86ee80ddeb25d53
2	b4bc852d233dfefc5131f593b538befa	2018-03-22	a7982125ff7aa3b2054c6e44f9d2
3	6be030b81c75970747525b843c1ef4f8	2018-01-22	d45d558f0daeecf3cccdffe3c59
4	5420aad7fec3549a85876ba1c529bd84	2018-02-21	b48ec5f3b04e9068441002a19df

In [25]:
```
def view_table(table_name, n=5):
    query = f"SELECT * FROM {table_name} LIMIT {n};"
    return pd.read_sql_query(query, db_connection)
```

In [26]:
```
view_table('leads_qualified', 5)
```

Out[26]:

	mql_id	first_contact_date	landing_pa
0	dac32acd4db4c29c230538b72f8dd87d	2018-02-01	88740e65d5d6b056e0cda098e1e
1	8c18d1de7f67e60dbd64e3c07d7e9d5d	2017-10-20	007f9098284a86ee80ddeb25d53
2	b4bc852d233dfefc5131f593b538befa	2018-03-22	a7982125ff7aa3b2054c6e44f9d2
3	6be030b81c75970747525b843c1ef4f8	2018-01-22	d45d558f0daeecf3cccdffe3c59
4	5420aad7fec3549a85876ba1c529bd84	2018-02-21	b48ec5f3b04e9068441002a19df

We'll create a plot to visualize the number of leads by their origin and how many of those eventually signed up with Olist. The origin field in the leads_qualified table includes an 'unknown' category as well as some null values. To handle this, we'll use the COALESCE function to group null values under the 'unknown' category.

In [28]:
```
lead_conversion = """
SELECT
    COALESCE(origin, 'unknown') AS origin,
    COUNT(DISTINCT leads_qualified.mql_id) AS qualified_leads,
    COUNT(DISTINCT leads_closed.mql_id) AS closed_leads,
    COUNT(DISTINCT leads_closed.mql_id) * 100.0 / COUNT(DISTINCT leads_qualified
        AS conversion_rate
FROM leads_qualified
    LEFT JOIN leads_closed USING (mql_id)
GROUP BY COALESCE(origin, 'unknown')
ORDER BY COUNT(leads_qualified.mql_id) DESC
"""

df = pd.read_sql_query(lead_conversion, db_connection)
df
```

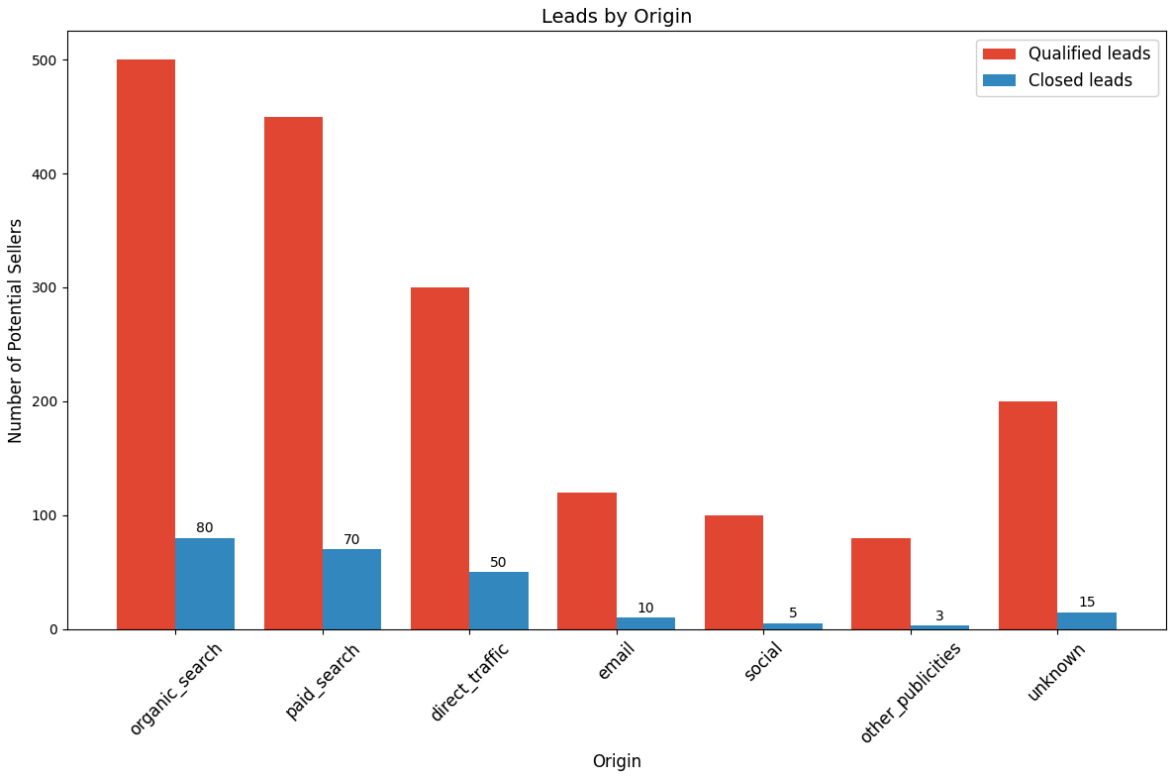
Out[28]:

	origin	qualified_leads	closed_leads	conversion_rate
0	organic_search	2296	271	11.803136
1	paid_search	1586	195	12.295082
2	social	1350	75	5.555556
3	unknown	1159	193	16.652286
4	direct_traffic	499	56	11.222445
5	email	493	15	3.042596
6	referral	284	24	8.450704
7	other	150	4	2.666667
8	display	118	6	5.084746
9	other_publicities	65	3	4.615385

Let's visualize the data using a grouped bar plot:

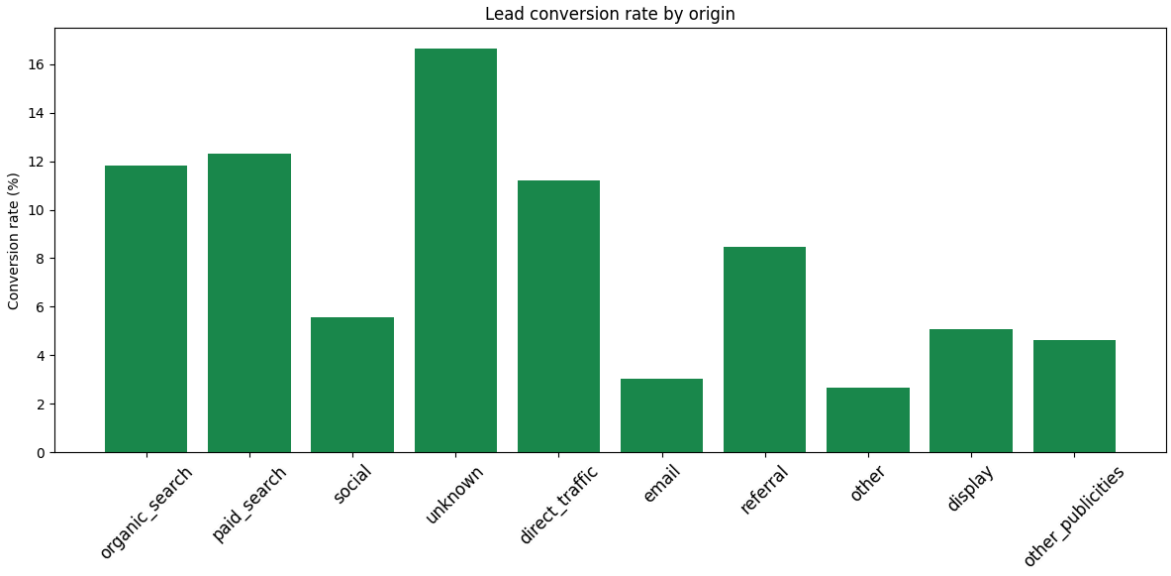
In [5]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Example dataset
data = {
    'origin': ['organic_search', 'paid_search', 'direct_traffic', 'email', 'soci
    'qualified_leads': [500, 450, 300, 120, 100, 80, 200],
    'closed_leads': [80, 70, 50, 10, 5, 3, 15]
}

# Create DataFrame
df = pd.DataFrame(data)
```

In [6]:
```
plt.figure(figsize=(12, 8))
bar_width = 0.4
r1 = range(len(df))
r2 = [x + bar_width for x in r1]

plt.bar(r1, df['qualified_leads'], color='#e24a33', width=bar_width, label='Qual
plt.bar(r2, df['closed_leads'], color='#348abe', width=bar_width, label='Closed

for i, v in enumerate(df['closed_leads']):
    plt.text(i + bar_width, v + 5, str(v), ha='center', fontsize=10)

plt.xlabel('Origin', fontsize=12)
plt.ylabel('Number of Potential Sellers', fontsize=12)
plt.title('Leads by Origin', fontsize=14)
plt.xticks([r + bar_width / 2 for r in range(len(df))], df['origin'], rotation=4
plt.legend(fontsize=12)
plt.tight_layout()
plt.show()
```



As observed, only a small fraction of qualified leads successfully convert into Olist sellers. The main sources of these closed leads are 'organic_search' and 'paid_search', followed by leads with unknown origins. To better understand which channels are most effective in driving seller conversions, let's visualize the conversion rate across different lead sources.

In [31]:
```
plt.figure(figsize=(12, 6))
plt.bar(df['origin'], df['conversion_rate'], color='#1c8a4f')
plt.ylabel('Conversion rate (%)')
plt.title('Lead conversion rate by origin')
plt.xticks(rotation=45, color='black', fontsize=12)
plt.tight_layout()
plt.show()
```



In this plot, we observe the conversion rates of leads into Olist sellers across different channels of origin, displayed in the same order as the previous grouped bar chart.

It’s clear that ‘paid_search’, ‘organic_search’, and ‘direct_traffic’ show the highest conversion rates, while channels like ‘email’, ‘social’, ‘display’, and ‘other_publicities’ demonstrate comparatively lower performance in converting leads into active sellers.

In [8]:
```
! pip install nbconvert
```