Seaborn & Object-Oriented Programming

🎯 Session Objectives:

- ✅ Understand why Seaborn is important in visualization
- ✅ Recognize common Seaborn plot types.
- ✅ Understand why OOP is needed and the problems it solves
- ✅ Learn the concepts of classes and objects.
- ✅ Implement classes and objects with practical examples.
- ✅ Relate OOP to real-world systems (like a Banking System).
- ✅ Explore inheritance, polymorphism, encapsulation, and abstraction.
- ✅ Appreciate the benefits of OOP: managing complexity, reusability, and extensibility.

```
sns.color_palette("rocket", as_cmap=True)
```

```
sns.color_palette("magma", as_cmap=True)
```

```
sns.color_palette("mako", as_cmap=True)
```

```
sns.color_palette("viridis", as_cmap=True)
```

```
sns.color_palette("flare", as_cmap=True)
```

```
sns.color_palette("crest", as_cmap=True)
```

It is also possible to use the perceptually uniform colormaps f

`"viridis"`:

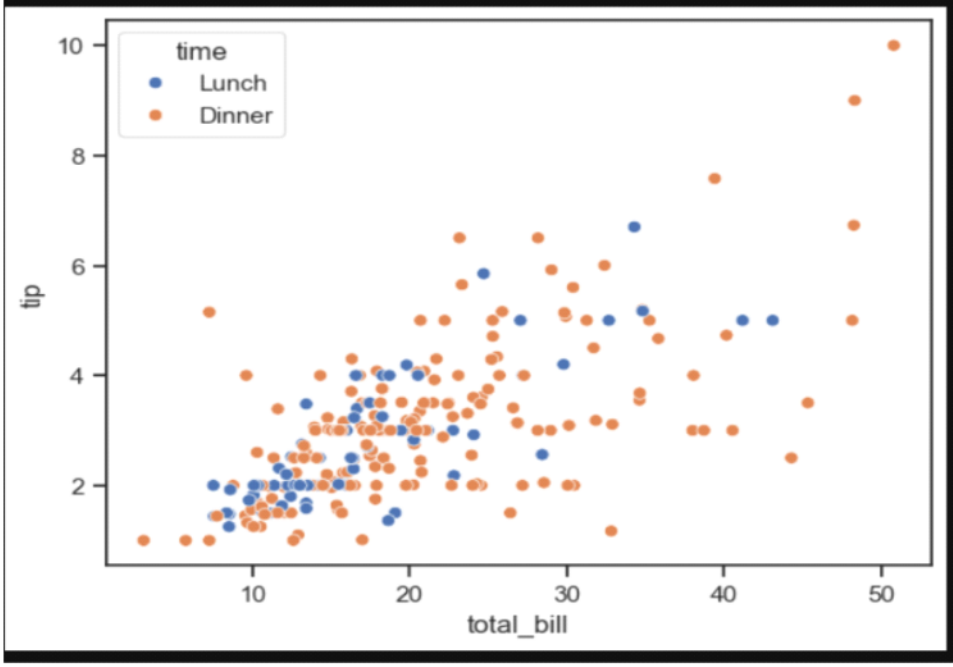
```python
# Scatter Plot
import seaborn as sns
sns.set_theme(style="ticks")

tips = sns.load_dataset("tips")
tips
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

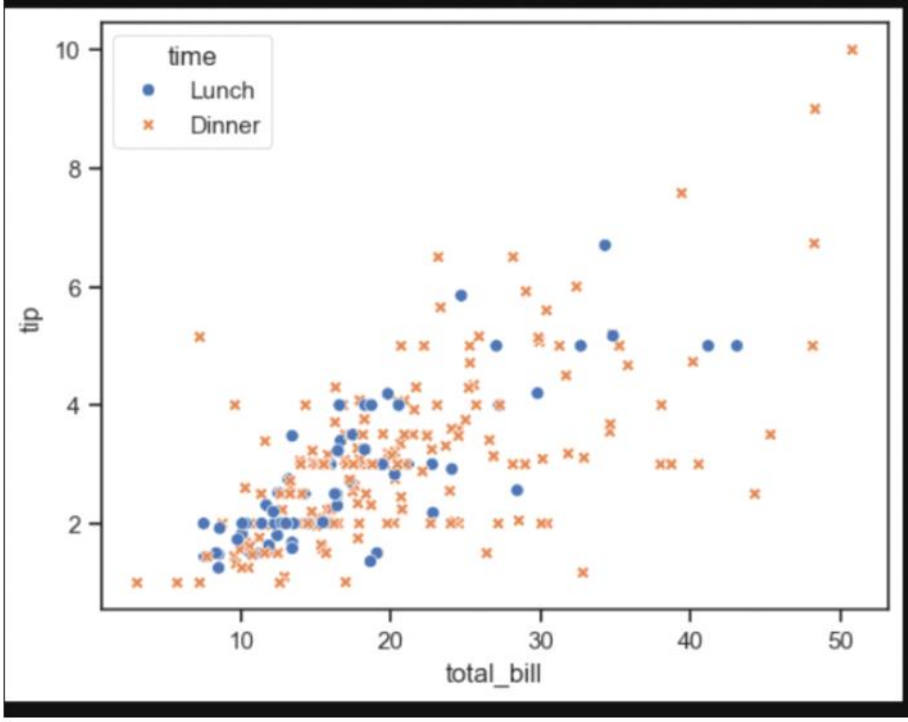
```python
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="time")
```

```
<Axes: xlabel='total_bill', ylabel='tip'>
```



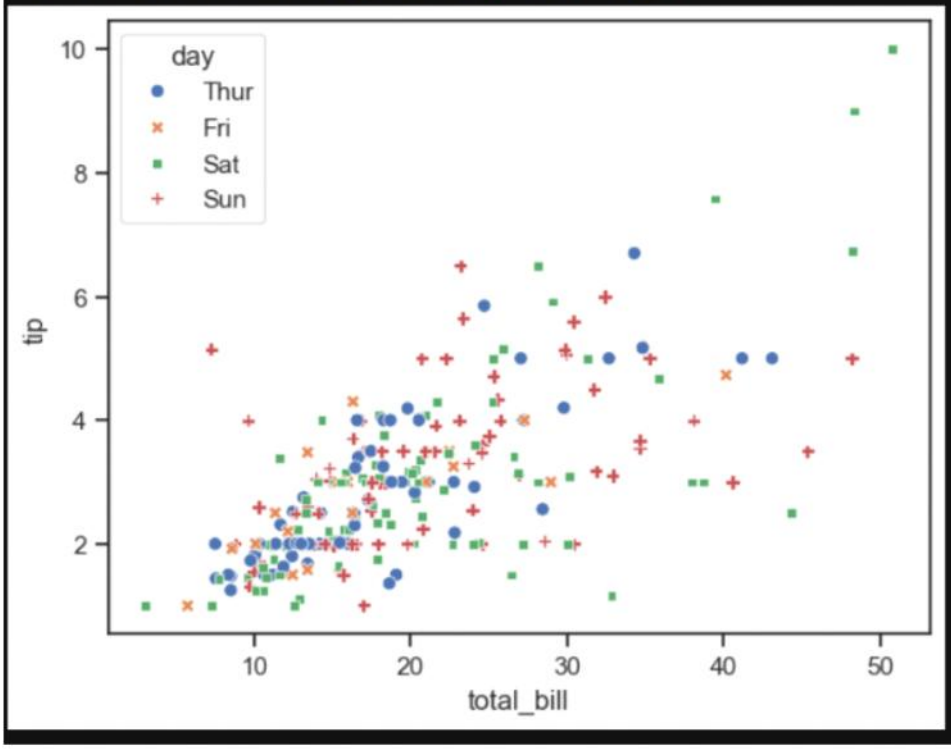
```
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="time", style = 'time')
```

```
<Axes: xlabel='total_bill', ylabel='tip'>
```



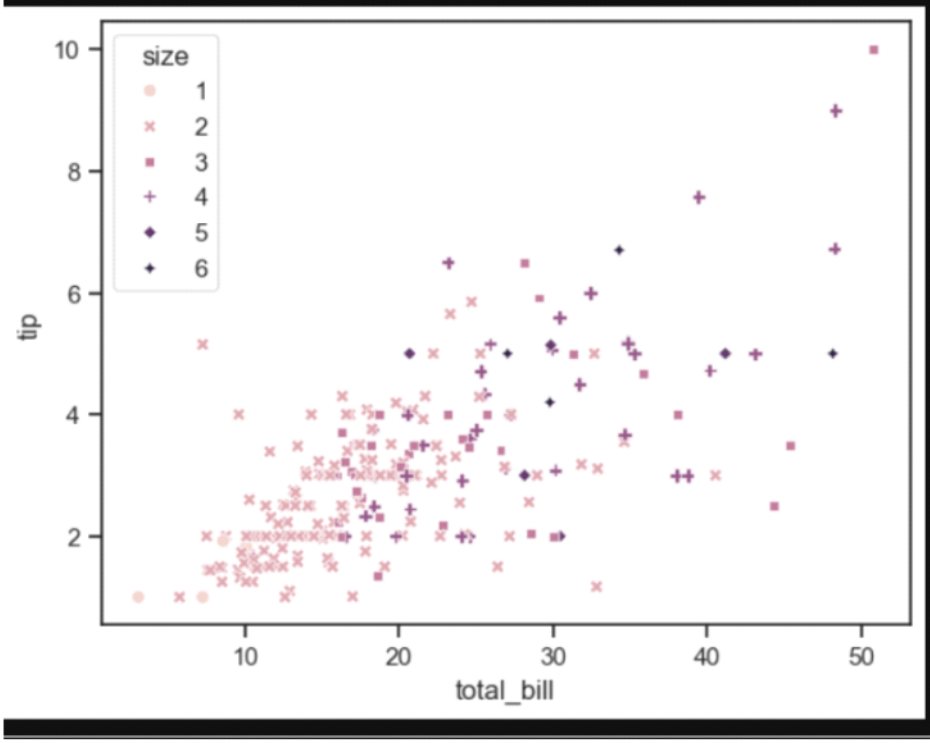

```
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="day", style="day")
```

```
<Axes: xlabel='total_bill', ylabel='tip'>
```



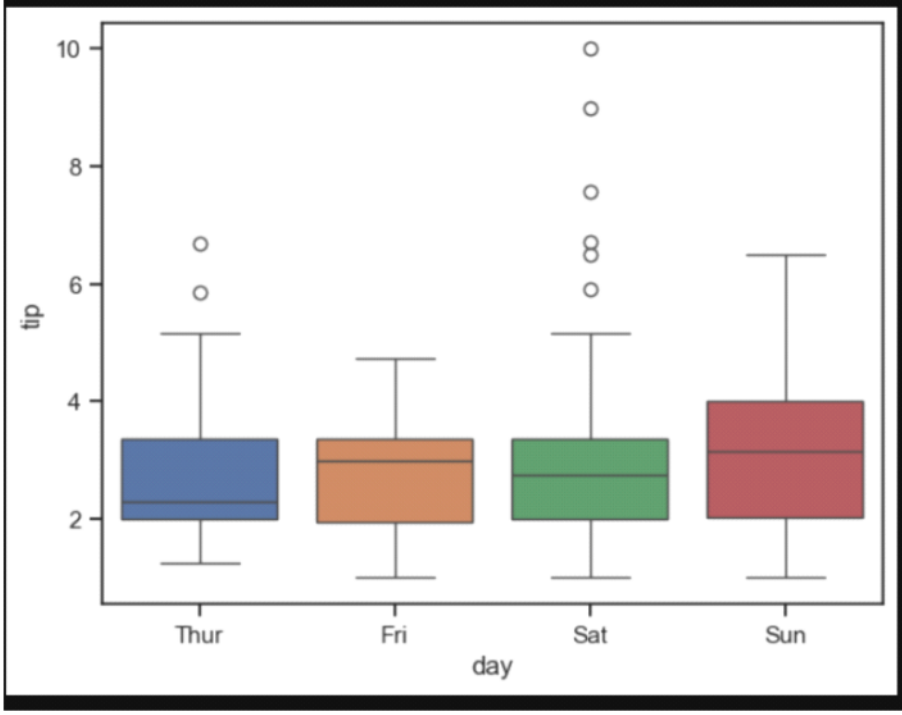
```
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="size", style="size")
```

```
<Axes: xlabel='total_bill', ylabel='tip'>
```



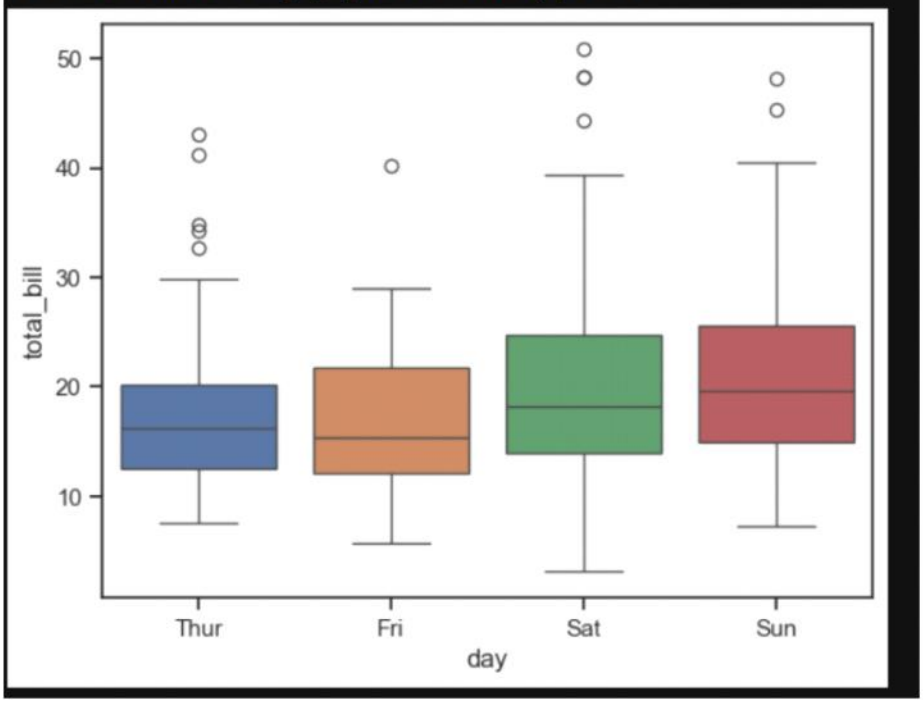

```
# Box Plot -> [Univariate Analysis]
sns.boxplot(data=tips, x="day", y="tip", hue="day")
```

```
<Axes: xlabel='day', ylabel='tip'>
```



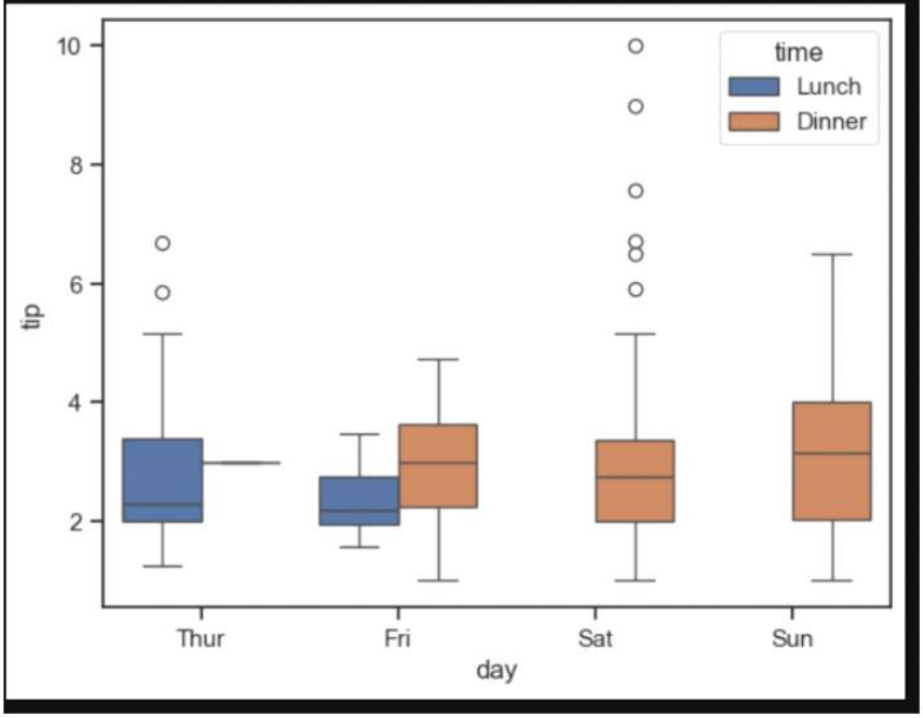
```
sns.boxplot(data=tips, x="day", y="total_bill", hue="day")
```

```
<Axes: xlabel='day', ylabel='total_bill'>
```



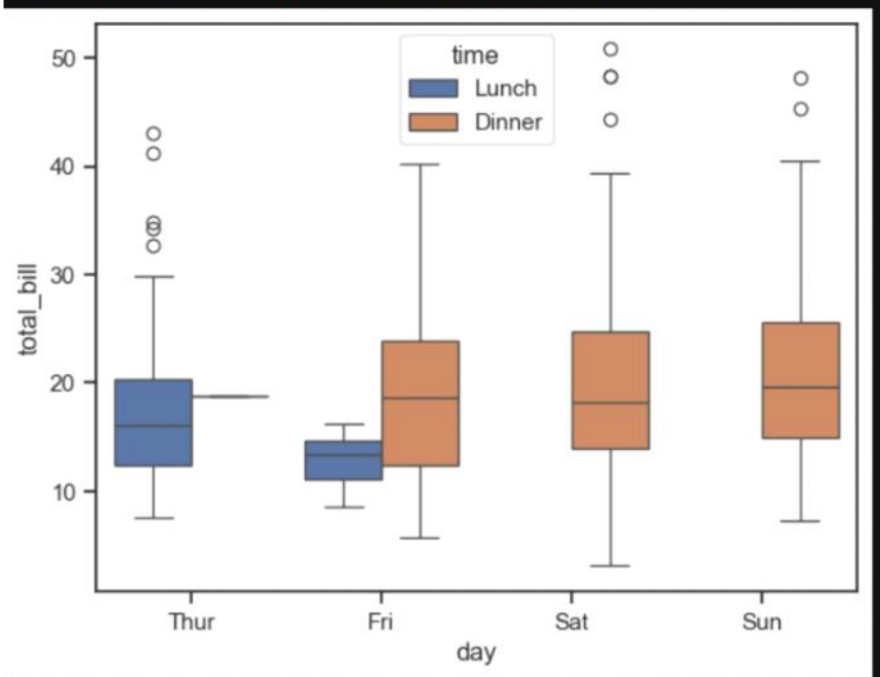

```
sns.boxplot(data=tips, x="day", y="tip", hue="time")
```

```
<Axes: xlabel='day', ylabel='tip'>
```



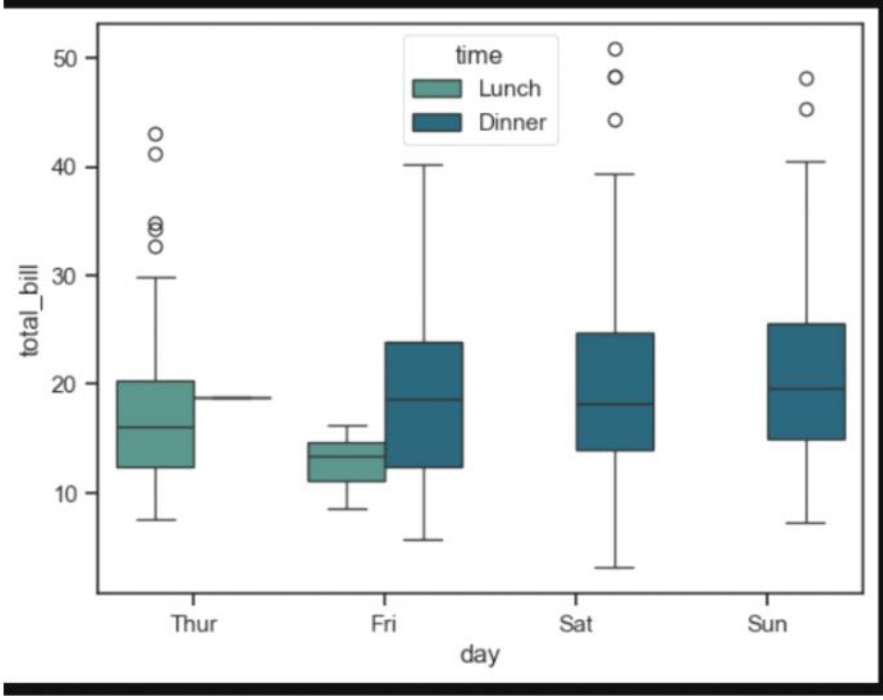
```
sns.boxplot(data=tips, x="day", y="total_bill", hue="time")
```

```
<Axes: xlabel='day', ylabel='total_bill'>
```



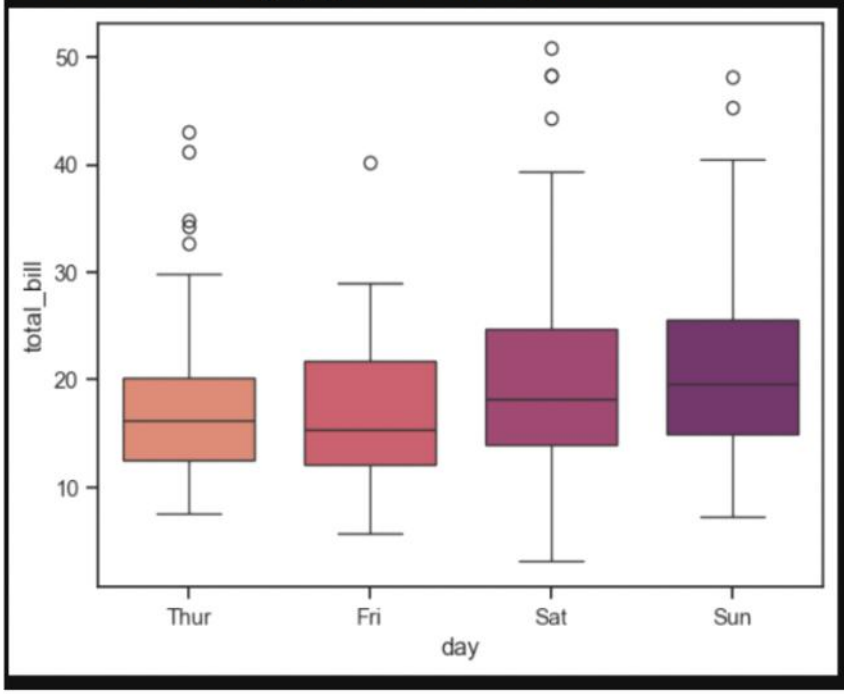

```
sns.boxplot(data=tips, x="day", y="total_bill", hue="time" , palette = 'crest')
```

```
<Axes: xlabel='day', ylabel='total_bill'>
```



```
sns.boxplot(data=tips, x="day", y="total_bill", hue="day",fill=True, gap=.1,palette = 'flare')
```

```
<Axes: xlabel='day', ylabel='total_bill'>
```




```
# Correlation: HeatMap [Multivariate Analysis] [Tips -> 3 Continuous columns]
numeric_tips = tips.select_dtypes(include = 'number')
numeric_tips
```

	total_bill	tip	size
0	16.99	1.01	2
1	10.34	1.66	3
2	21.01	3.50	3
3	23.68	3.31	2
4	24.59	3.61	4
...	...	...	...
239	29.03	5.92	3
240	27.18	2.00	2
241	22.67	2.00	2
242	17.82	1.75	2
243	18.78	3.00	2

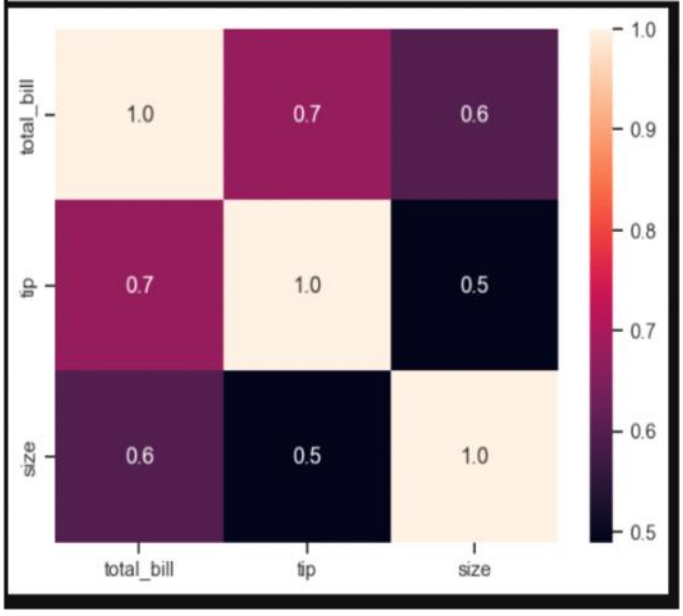
244 rows × 3 columns

```
numeric_tips = tips[['total_bill','tip','size']]
numeric_tips
```

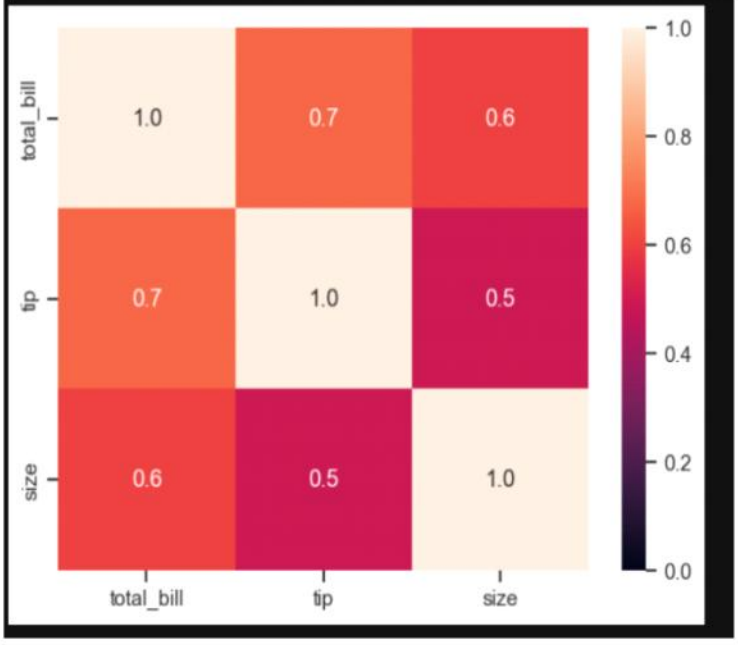
	total_bill	tip	size
0	16.99	1.01	2
1	10.34	1.66	3
2	21.01	3.50	3
3	23.68	3.31	2
4	24.59	3.61	4
...	...	...	...
239	29.03	5.92	3
240	27.18	2.00	2
241	22.67	2.00	2
242	17.82	1.75	2
243	18.78	3.00	2

244 rows × 3 columns

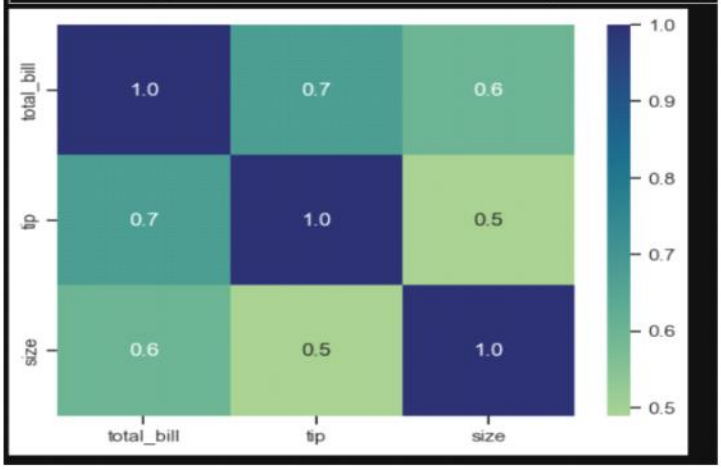
```
corr = numeric_tips.corr()
sns.heatmap(corr,annot=True,fmt=".1f");
```



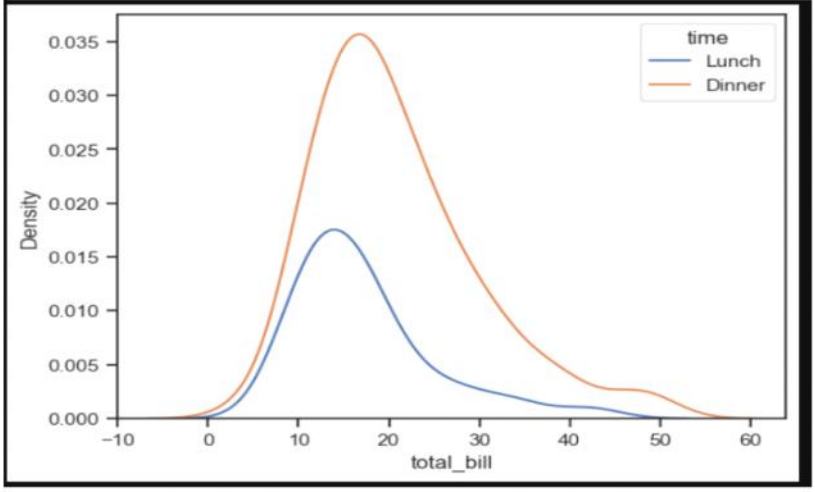

```
sns.heatmap(corr,annot=True,fmt=".1f",vmin=0, vmax=1);
```



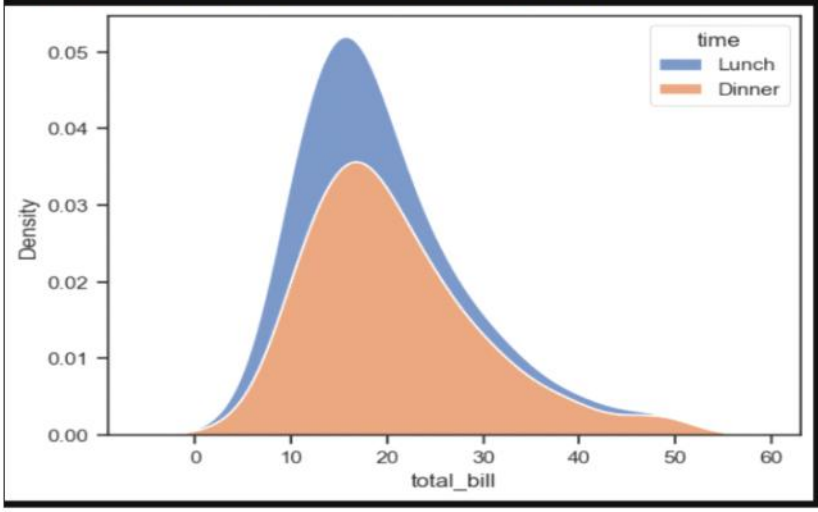
```
sns.heatmap(corr,annot=True,fmt=".1f",cmap = 'crest');
```




```
# Skewness [Normal Distribution Curve] ~ Bell Curve
# KDE Plot
sns.kdeplot(data=tips, x="total_bill",hue="time");
```

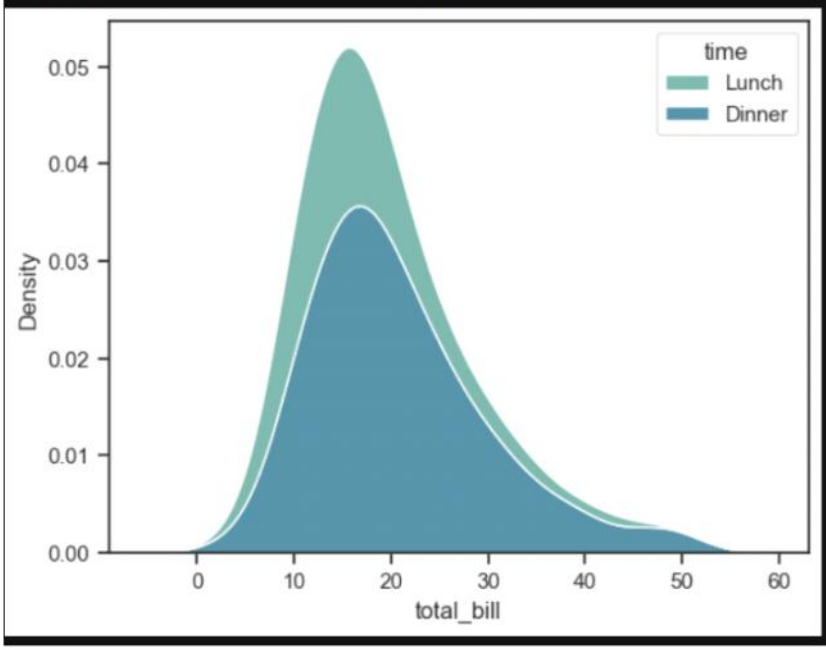



```
sns.kdeplot(data=tips, x="total_bill", hue="time", multiple="stack")
<Axes: xlabel='total_bill', ylabel='Density'>
```



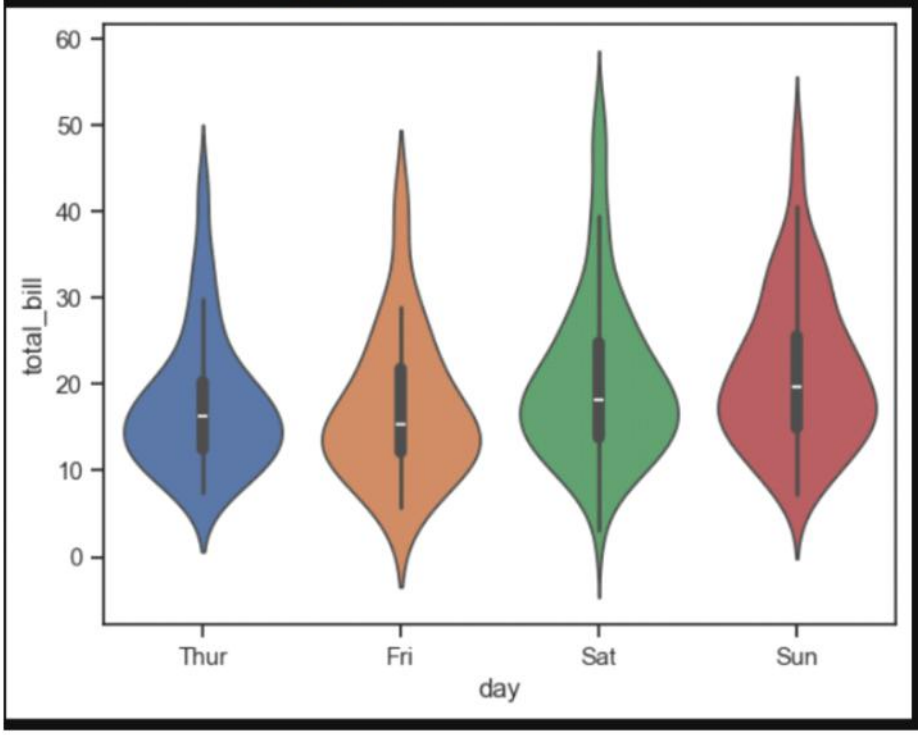
```
sns.kdeplot(data=tips, x="total_bill", hue="time", multiple="stack", palette='crest')
```

```
<Axes: xlabel='total_bill', ylabel='Density'>
```




```
# Violin Plot
sns.violinplot(data=tips, x="day", y="total_bill", hue="day")
```

```
<Axes: xlabel='day', ylabel='total_bill'>
```



Procedural Programming:

Procedural style = "writing instructions step by step as procedures (functions)".

- ✔ Data is stored separately (variables, lists, dicts).
- ✔ Functions operate on that data.
- ✔ No natural grouping of "data + behaviour".

Object-Oriented Programming (OOP):

OOP style = "bundle data + behavior together into objects (instances of classes)".

- ✔ A class defines the template (blueprint).
- ✔ Each object has its own data (state).
- ✔ Methods operate on that object's data via self.
- ✔ Promotes encapsulation, reusability, and scalability.

Class -> Blueprint

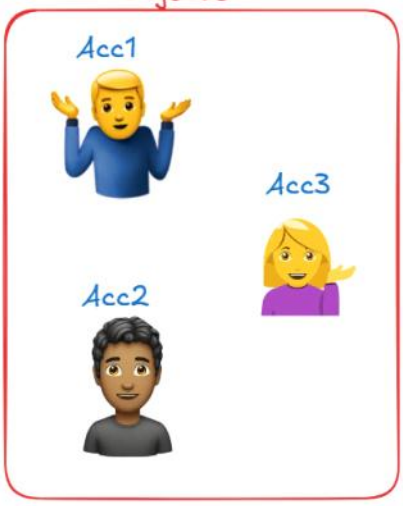
Acc Info. ->
1. Account Number
2. PIN [Private]
3. Balance

Data

1. Deposit()
2. Withdraw()
3. Check_Balance()

Behaviour

Objects




```
Acc1.deposit(10000)
Acc1.withdraw(1000, 'PIN')
Acc1.Check_balance('PIN')
```

```python
# Procedural Programming ❌❌
# Data [Stored Seperately]
accounts = {
    '12345' : {'balance' : 1000, 'pin' : 1111},
    '67890' : {'balance' : 2000, 'pin' : 2222}
}

# Functions that operates on that data
def deposit(account_no , amount):
    if account_no in accounts:
        accounts[account_no]['balance'] += amount
    else:
        print("Account is not Found!")

def withdraw(account_no , amount , pin):
    if pin == accounts[account_no]['pin']:
        if accounts[account_no]['balance'] >= amount:
            accounts[account_no]['balance'] -= amount
            print("Withdraw Successful ✅")
        else:
            print("Insufficients Funds")
    else:
        print("Invalid PIN")

def check_balance(account_no, pin):
    if pin == accounts[account_no]['pin']:
        return accounts[account_no]['balance']
    else:
        print("Invalid PIN")
```

```python
# Sensitive info is not protected
print(accounts['12345']['pin'])
print(accounts['67890']['balance'])
```

```
1111
2000
```

```python
deposit('12345',500)
print(check_balance('12345',1111))
```

```
1500
```

```python
withdraw('12345',2500,1111)
print(check_balance('12345',1111))
```

```
Insufficients Funds
1500
```

```python
withdraw('12345',2500,1234)
```

```
Invalid PIN
```

```python
withdraw('12345',1000,1111)
print(check_balance('12345',1111))
```

```
Withdraw Successful ✅
500
```

```python
# Object Oriented Programming
# Class - bundle the [Data + Behaviour]

class Account:
    def __init__(self, account_number, balance = 0):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self ,amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficients Funds")

    def check_balance(self):
        return f"Account {self.account_number} having a balance : {self.balance}"
```

```python
# Making an Object [Instance of a Class]
acc1 = Account('12345', 1000) # Creating a new Account
acc1.deposit(500)
acc1.check_balance()
```

```
'Account 12345 having a balance : 1500'
```

```python
acc2 = Account('67890', 2000) # Creating a new Account
acc2.withdraw(1000)
acc2.check_balance()
```

```
'Account 67890 having a balance : 1000'
```

```python
acc3.deposit(1000) # NameError: name 'acc3' is not defined
```

```python
acc1.withdraw(2000)
acc1.check_balance()
```

```
Insufficients Funds
'Account 12345 having a balance : 1500'
```

```python
# Encapsulation [Hide the Sensitive Data] -> Control Access
class Account:
    def __init__(self, account_number, pin, balance = 0):
        self.account_number = account_number # Public
        self.__pin = pin # Private
        self._balance = balance # Protected

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
        else:
            print("Deposit Must be Positive")

    def withdraw(self ,amount , pin):
        if pin == self.__pin:
            if amount <= self._balance:
                self._balance -= amount
            else:
                print("Insufficients Funds")
        else:
            print("Invalid PIN")

    def check_balance(self,pin):
        if pin == self.__pin:
            return f"Account {self.account_number} having a balance : {self._balance}"
        else:
            print("Invalid PIN")
```

```python
acc1 = Account('12345',1111,1000)
print(acc1._balance)
```

```
1000
```

```python
acc1.check_balance(1111)
```

```
'Account 12345 having a balance : 1000'
```

```python
acc1.deposit(2000)
acc1.check_balance(1111)
```

```
'Account 12345 having a balance : 3000'
```

```python
acc1.withdraw(1000,1111)
acc1.check_balance(1111)
```

```
'Account 12345 having a balance : 2000'
```

```python
# Inheritance 🚀 -> Create a new class without writing fresh
# Subclass ['Saving Account'] -> Superclass (Account)

class SavingsAccount(Account):
    def __init__(self, account_number, pin, balance = 0, withdrawl_limit = 5000):
        super().__init__(account_number, pin, balance)
        self.withdrawl_limit = withdrawl_limit

    # Override the withdrawl method
    def withdraw(self ,amount , pin):
        if amount > self.withdrawl_limit:
            print(f"Cannot Withdraw more than {self.withdrawl_limit} at once.")
        else:
            super().withdraw(amount , pin)
```

```python
# Parent Class
acc_p = Account(101,1234,10000)
acc_p.withdraw(5500,1234)
acc_p.check_balance(1234)
```

```
'Account 101 having a balance : 4500'
```

```python
# Child Class
acc_s = SavingsAccount(102,5678,5000,withdrawl_limit = 3000)
acc_s.withdraw(3500,5678)
acc_s.check_balance(5678)
```

```
Cannot Withdraw more than 3000 at once.
'Account 102 having a balance : 5000'
```

```python
acc_s.deposit(2000)
acc_s.check_balance(5678)
```

```
'Account 102 having a balance : 7000'
```

```python
acc_s.withdraw(2000,5678)
acc_s.check_balance(5678)
```

```
'Account 102 having a balance : 5000'
```

```python
acc_s.withdraw(2500,5678)
acc_s.check_balance(5678)
```

```
'Account 102 having a balance : 2500'
```

```python
acc_s.withdraw(2700,5678)
acc_s.check_balance(5678)
```

```
Insufficients Funds
'Account 102 having a balance : 2500'
```

```python
acc_s.withdraw(2700,1234)
acc_s.check_balance(5678)
```

```
Invalid PIN
'Account 102 having a balance : 2500'
```