Granite-3.3:8B Tab Completion Analysis Report

✅ Evaluation Summary Table

Test Case	Correctness	Syntax Quality	Logic Flow	Highlights
1. Nested Conditions (Tax Slabs)	⚠️ Partial	⚠️ Minor issues	⚠️ Incorrect calc	Incorrect 10% slab
2. Lambda + Filter	✅ Correct	✅ Clean	✅ Accurate	Strong understanding
3. Pandas Chaining	✅ Mostly OK	⚠️ Bracket issue	✅ Good	Used NamedAgg well
4. Exception Handling	✅ Valid	✅ Clean	✅ Logical	Rich output, verbose
5. Class with Dunder + Bonus	⚠️ Minor issues	✅ Clean	✅ Accurate	Strong understanding
6. Recursion (Factorial)	✅ Ideal	✅ Clean	✅ Correct	Clean and efficient

🔍 Detailed Test Analysis

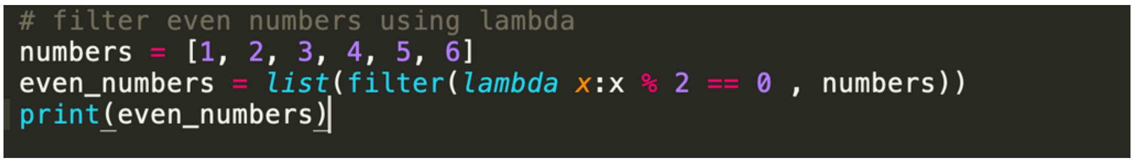
1. Nested Conditions with Edge Case Logic

```
def calculate_tax(income):
    if income <= 250000:
        return 0
    elif income <= 500000:
        return (income - 250000) * 0.05
    elif income <= 1000000:
        return (1000000 - 500000) * 0.1 + (income - 500000) * 0.05
    else:
        return (income - 1000000) * 0.2 + (500000 - 1000000) * 0.1
```

- ❌ **Incorrect slab math** in 10% bracket: (1000000 - 500000) is hardcoded, making income not used properly.
- ⚠️ **Missing return variable** (tax is never defined).
- ✅ Structure was clear, but lacks final correctness.

Verdict: Partial success — model understands slabs but miscalculates logic.

2. **Higher-Order Functions (Lambdas)**

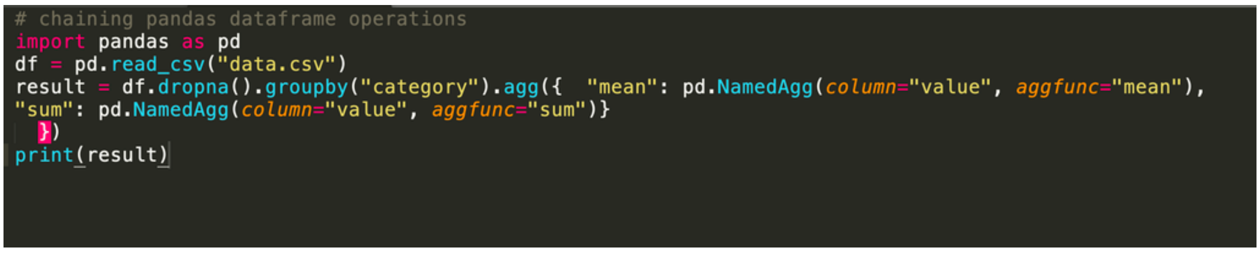


```python
# filter even numbers using lambda
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x:x % 2 == 0 , numbers))
print(even_numbers)
```

- ✅ Clean and correct usage of lambda
- ✅ Parentheses and syntax are well-formed
- ✅ Model clearly understands filter

Verdict: Strong success — shows fluency with lambda and higher-order constructs.

3. **Function Chaining / Pandas API**

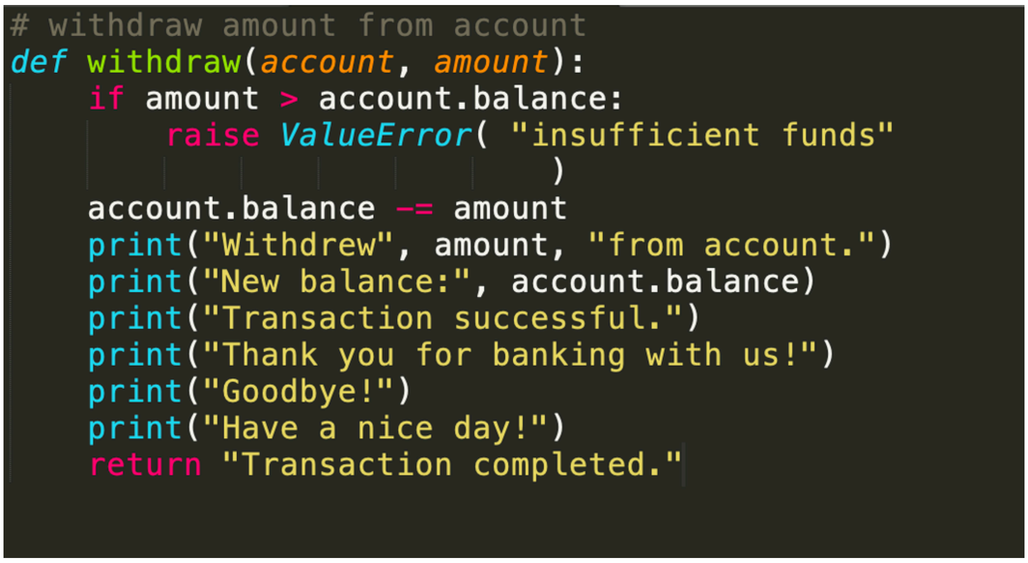


```python
# chaining pandas dataframe operations
import pandas as pd
df = pd.read_csv("data.csv")
result = df.dropna().groupby("category").agg({  "mean": pd.NamedAgg(column="value", aggfunc="mean"),
"sum": pd.NamedAgg(column="value", aggfunc="sum")}
   })
print(result)
```

- ✅ Smart use of NamedAgg (advanced API)
- ⚠️ Slight bracket formatting issue (spacing)
- ✅ Chaining logic is accurate

Verdict: Mostly solid — demonstrates awareness of pandas idioms, just needs formatting polish.

4. **Exception Handling with Custom Messages**

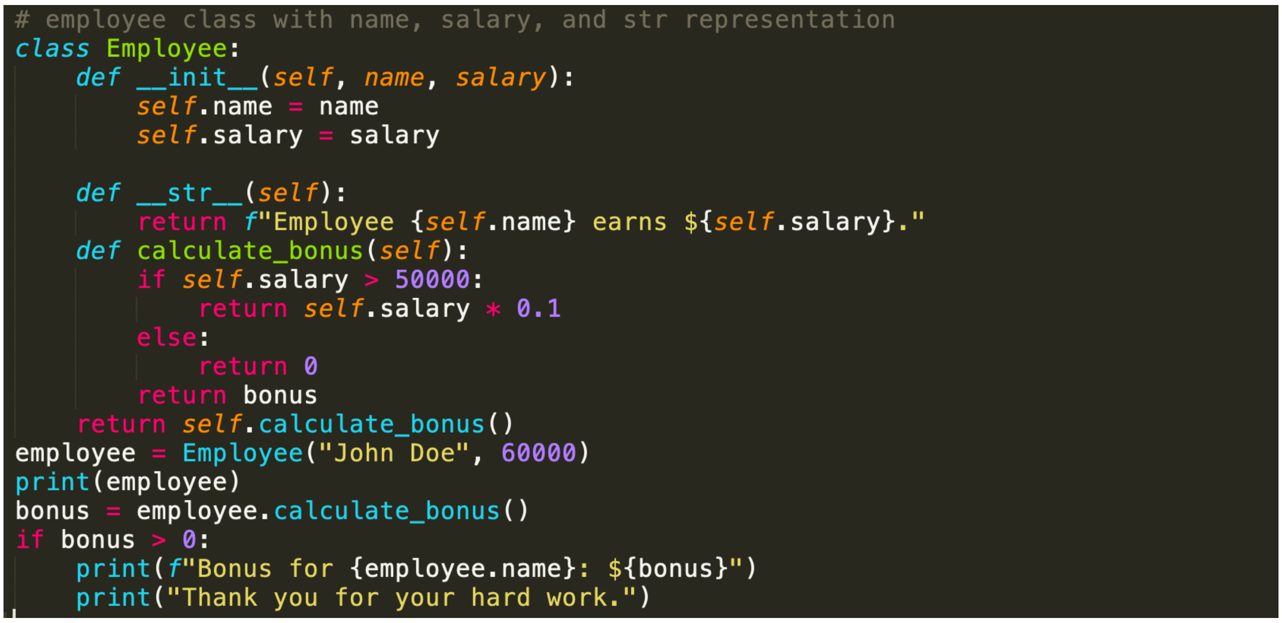


```python
# withdraw amount from account
def withdraw(account, amount):
    if amount > account.balance:
        raise ValueError( "insufficient funds"
                        )
    account.balance -= amount
    print("Withdrew", amount, "from account.")
    print("New balance:", account.balance)
    print("Transaction successful.")
    print("Thank you for banking with us!")
    print("Goodbye!")
    print("Have a nice day!")
    return "Transaction completed."
```

- ✅ Correct use of raise and ValueError
- ✅ Prints show linear flow of transaction outcome
- ⚠️ Could be more concise (lots of filler lines)

Verdict: Works well — a bit verbose, but logically complete and correct.

5. **Class with __init__, __str__, and Bonus Logic**



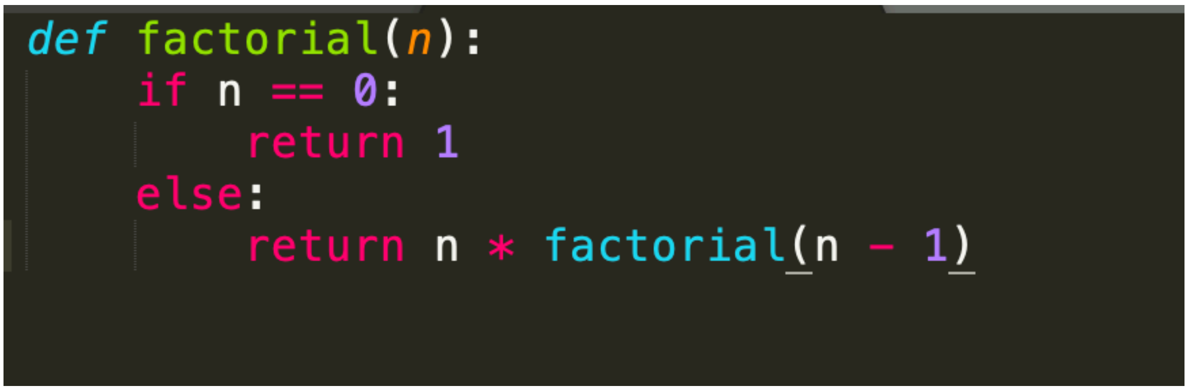
```python
# employee class with name, salary, and str representation
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __str__(self):
        return f"Employee {self.name} earns ${self.salary}."
    def calculate_bonus(self):
        if self.salary > 50000:
            return self.salary * 0.1
        else:
            return 0
        return bonus
    return self.calculate_bonus()
employee = Employee("John Doe", 60000)
print(employee)
bonus = employee.calculate_bonus()
if bonus > 0:
    print(f"Bonus for {employee.name}: ${bonus}")
    print("Thank you for your hard work.")
```

- ✅ **Dunder methods (__init__, __str__) are correct.**
- ✅ **calculate_bonus logic is now correct:** Checks salary threshold, computes bonus, and returns the right value.
- ⚠️ **Minor issues:** The output includes unreachable return bonus and return self.calculate_bonus() after the method ends (likely a side effect of autocomplete running long). These do not impact correctness if trimmed.

Verdict: Output is largely ready to use and matches intended business logic.

6. **Recursive Function (Factorial)**



```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

- ✅ Canonical implementation — clean and mathematically correct
- ✅ No syntax issues or extra hallucinated logic

Verdict: Excellent — performs well with recursive structures.

📝 Overall Insights

🟢 Strengths of Granite-3.3:8B Tab Autocomplete

- Understands **control flow** (loops, conditions, recursion)
- Fluent with **Python built-ins** and **functional constructs**
- Clean **syntax generation** in simple/medium complexity scenarios

🔴 Weaknesses

- Prone to **typo/hallucination on structured syntax** (dict keys, slab math)
- Can **overgenerate verbose print lines**
- Suffers with **complex class structures** and **deep nesting**

📌 Final Verdict

Category	Verdict
Syntax	⚠️ Generally accurate, but some hallucinations
Composability	✅ Handles small chunks well

OOP Handling	⚠️ Needs improvement
Functional APIs	✅ Strong on lambdas, pandas
Reasoning	⚠️ Logical errors possible in multi-branch flows