Granite-3.3:8B Tab Completion Analysis Report — FIM Template

✅ Evaluation Summary Table

Test Case	Correctness	Syntax Quality	Logic Flow	Highlights
1. Nested Conditions (Tax Slabs)	❌ Broken	❌ Syntax error	❌ Dead code	Misses slab logic
2. Lambda + Filter	✅ Correct	✅ Clean	✅ Accurate	Fully correct
3. Pandas Chaining	❌ Vague	⚠️ Acceptable	⚠️ Underspecified	Only "all columns" given
4. Exception Handling	✅ Correct	✅ Clean	✅ Accurate	Ideal logic & message
5. Class/Dunder/Bonus	❌ Broken	❌ Incomplete	❌ Fails logic	No return, faulty bonus
6. Recursion (Factorial)	✅ Correct	✅ Clean	✅ Accurate	Ideal recursive impl.

FIM Prompting Setup

For each scenario, the **prompt was split into:**

- **Prefix:** Code before the intended completion
- **Suffix:** Code after the intended completion
- **Cursor:** Placed at the "hole" (i.e., the gap for the FIM model to fill)

Model's FIM completion for Calculator code:

Prompt (omitted the *divide* function to test FIM autocomplete):

```python
# Basic Calculator code using Python
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

# <<< cursor here for FIM >>>

def calculate(operation, x, y):
    if operation == 'add':
        return add(x, y)
    elif operation == 'subtract':
        return subtract(x, y)
    elif operation == 'multiply':
        return multiply(x, y)
    elif operation == 'divide':
        return divide(x, y)
    else:
        return "Invalid operation. Please choose from 'add', 'subtract', 'multiply', or 'divide'."
```

Model's FIM completion:

```python
# Basic Calculator code using Python
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        return "Error! Division by zero."
    else:
        return x / y

def calculate(operation, x, y):
    if operation == 'add':
        return add(x, y)
    elif operation == 'subtract':
        return subtract(x, y)
    elif operation == 'multiply':
        return multiply(x, y)
    elif operation == 'divide':
        return divide(x, y)
    else:
        return "Invalid operation. Please choose from 'add', 'subtract', 'multiply', or 'divide'."
```

Analysis

Evaluation Aspect	FIM Output	Comments

Function header	✅ Included (def divide(x, y):)	Matches expected function style
Logic correctness	✅ Correct logic for division and zero-division guard	Safe and Pythonic
Formatting/indentation	✅ Properly formatted	No manual fixes needed
Context awareness	✅ Integrated well with surrounding code	Used the right argument names and pattern
Completeness	✅ Fully functional, drop-in ready	No additional edits required

🔍 Detailed Test Analysis

1️⃣ Nested Conditions with Edge Case Logic

Prompt Details:

- **Prefix:**

```
def calculate_tax(income):
    if income <= 250000:
        return 0
    elif income <= 500000:
        return (income - 250000) * 0.05
```

- **Suffix:**

```
    else:
        return (income - 1000000) * 0.2 + (500000 - 1000000) * 0.1
```

- **Cursor:** Placed at the next logical branch, after the second elif, before the else.

Model Completion:

```
    return (income - 500000) * 0.2
```

Final Output:

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

Analysis:

- ❌ **Logic/Syntax Error:** Inserts a return outside the if/elif/else, causing dead code and syntax issues.
- ❌ **Did not generate elif income <= 1000000: slab logic.**
- ❌ **Slab math is not preserved.**

2️⃣ Higher-Order Functions (Lambdas + Filter)

Prompt Details:

- **Prefix:**

```python
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(
```

- **Suffix:**

```python
, numbers))
print(even_numbers)
```

- **Cursor:** Between the open parenthesis after *filter(* and the comma.

Model Completion:

```python
lambda x: x % 2 == 0
```

Final Output:

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

Analysis:

- ✅ **Logic and syntax correct**
- ✅ **Exactly matches expected code**

3️⃣ Function Chaining / Pandas API

Prompt Details:

- **Prefix:**

```
import pandas as pd
df = pd.read_csv("data.csv")
result = df.dropna().groupby("category").agg({
```

- **Suffix:**

```
})
print(result)
```

- **Cursor:** Between the opening curly brace in *agg({* and the closing brace in *})*.

Model Completion:

```
    all columns
```

Final Output:

```
import pandas as pd
df = pd.read_csv("data.csv")
result = df.dropna().groupby("category").agg({
    all columns
})
print(result)
```

Analysis:

- ❌ **Non-Python output ("all columns") is not valid code**
- ❌ **No use of pd.NamedAgg or correct aggregations**
- ⚠️ **Vague, would not run**

4️⃣ Exception Handling with Custom Messages

Prompt Details:

- **Prefix:**

```
def withdraw(account, amount):
    if amount > account.balance:
```

- **Suffix:**

```
    account.balance -= amount
    print("Withdrew", amount, "from account.")
    print("New balance:", account.balance)
    print("Transaction successful.")
    return "Transaction completed."
```

- **Cursor:** Indented after the *if amount > account.balance:* line.

Model Completion:

```
        raise ValueError("Insufficient funds.")
```

Final Output:

```
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.")
    return "Transaction completed."
```

Analysis:

- ✅ **Ideal, robust exception logic**
- ✅ **Correct Python syntax**

5️⃣ Class Definitions with init, str, and Method Logic

Prompt Details:

- **Prefix:**

```
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):
```

- **Suffix:**

```
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.")
```

- **Cursor:** Indented within the *calculate_bonus* method.

Model Completion:

```
        self.salary * 0.10
```

Final Output:

```
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):
        self.salary * 0.10
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.")
```

Analysis:

- ❌ **Missing return statement**; method always returns None.

- ❌ No check for salary threshold (should be if salary > 50000).
- ❌ Not robust for real business logic.

6️⃣ Recursive Logic

Prompt Details:

- **Prefix:**

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

- **Suffix:** *(No suffix needed; this is the end of the function)*

- **Cursor:** After the *return* statement inside the *else*.

Model Completion:

```
n * factorial(n - 1)
```

Final Output:

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

Analysis:

- ✅ Ideal recursion logic
- ✅ Matches canonical Python pattern

📝 Overall Insights

🟢 Strengths of Granite-3.3:8B (FIM)

- Handles basic logic blocks, lambdas, simple recursion, and exception raising very well.
- Great at "slotting in" missing logic when pattern is obvious and context is small.

🔴 Weaknesses

- Fails for more complex, multi-branch business logic or multi-line completions (e.g., slab tax, pandas aggregation, OOP bonus).
- Sometimes produces code that is non-Pythonic or vague (e.g., "all columns").
- Incomplete logic in class/method definitions.

⚠️ Usability Notes

- Sometimes required manual cursor placement at line start for FIM to work.
- Good completions for isolated logic; unreliable for structural or multi-line/multi-branch completions.

📌 Final Verdict

Category	Verdict (FIM)
Syntax	⚠️ Accurate for simple cases, buggy for complex
Composability	✅ Good for isolated blocks
OOP Handling	❌ Needs improvement
Functional APIs	❌ Vague or incomplete
Reasoning	⚠️ Simple logic okay, multi-branch fails

✅ Recommendations

- Use FIM for small, focused logic holes (lambdas, exceptions, recursion).
- Avoid FIM for complex, multi-branch, or highly contextual completions.