**Instructions for Implementation Claudes**

**Your Role**

You are an Implementation Claude tasked with developing a specific section of the strategy\_calculators.py file for the Heage Betting App. This file will contain 500 bet type calculation methods, and you are responsible for implementing a specific range of these methods.

**Important Rules**

1. **DO NOT rewrite the entire code** when making changes. Only modify the specific sections you're working on.
2. **Follow the exact template structure** provided for consistency across all methods.
3. **Use docstrings for all methods** with clear descriptions, parameters, return values, and examples.
4. **Implement robust error handling** for all edge cases (negative values, division by zero, etc.).
5. **Ensure detailed logging** at appropriate levels (INFO for results, WARNING for potential issues, ERROR for failures).

**Technical Requirements**

**Code Structure**

* Implement all methods as @staticmethod under the StrategyCalculators class
* Use consistent naming: calculate\_bet\_type\_name
* Follow the binary or multi-outcome template based on the bet type
* Each method must validate inputs, calculate profits, and return standardized results

**Performance Considerations**

* Use @lru\_cache for helper methods that may be called frequently
* Optimize multi-outcome calculations to avoid redundant computations
* Ensure efficiency in large dictionaries by pre-computing shared values

**Error Handling**

* Validate all inputs (odds >= 1.0, stake > 0, etc.)
* Handle division by zero in lay stake calculations
* Check for missing keys in dictionaries
* Return meaningful error messages

**Logging**

* Log the start of each calculation with input values
* Log the result of each calculation with profit values
* Use appropriate log levels based on severity
* Include context in log messages (method name, parameters, outcome)

**Your Specific Assignment**

Based on your Claude number (which I will tell you separately), you will implement:

* **Claude 1**: Preamble + Bet Types 1-100 (Match Outcome & Goals Part 1)
* **Claude 2**: Bet Types 101-200 (Goals Part 2 & Score)
* **Claude 3**: Bet Types 201-300 (Player-Specific)
* **Claude 4**: Bet Types 301-400 (Team Event)
* **Claude 5**: Bet Types 401-500 (Game Incident & Specialty/Combo)

**Templates to Follow**

**Binary Outcome Bet Template**

@staticmethod

def calculate\_binary\_bet\_type(

back\_odds: Union[int, float, Decimal],

lay\_odds: Union[int, float, Decimal],

stake: Union[int, float, Decimal],

commission: Union[int, float, Decimal] = Decimal('0.02'),

back\_commission: Union[int, float, Decimal] = Decimal('0.00')

) -> Dict[str, Union[str, bool, Decimal, Dict]]:

"""

Calculate profit for Binary Bet Type.

Args:

back\_odds: Decimal odds for the back bet (e.g., 2.0)

lay\_odds: Decimal odds for the lay bet (e.g., 2.1)

stake: Amount for the back bet

commission: Commission rate for lay bet (default: 0.02 or 2%)

back\_commission: Commission rate for back bet (default: 0.00 or 0%)

Returns:

Dictionary with profit scenarios, lay stakes, and profitability status

Example:

>>> calculate\_binary\_bet\_type(2.0, 2.1, 100.0)

{"type": "Binary Bet Type", "profit\_scenarios": {"win": -8.33}, "min\_profit": -8.33,

"is\_profitable": False, "lay\_stakes": {"win": 100.0}, "stake": 100.0,

"commission": 0.02, "back\_commission": 0.00}

"""

logger.info(f"Calculating Binary Bet Type: back\_odds={back\_odds}, lay\_odds={lay\_odds}, stake={stake}, commission={commission}, back\_commission={back\_commission}")

inputs = {

'back\_odds': back\_odds,

'lay\_odds': lay\_odds,

'stake': stake,

'commission': commission,

'back\_commission': back\_commission

}

StrategyCalculators.validate\_inputs(

inputs,

['back\_odds', 'lay\_odds', 'stake'],

{k: (int, float, Decimal) for k in inputs.keys()}

)

stake\_d = inputs['stake']

lay\_stake = StrategyCalculators.calculate\_lay\_stake(stake\_d, inputs['back\_odds'], inputs['lay\_odds'], inputs['commission'])

if lay\_stake is None:

logger.error("Failed to calculate lay stake")

raise DivisionByZeroError("Invalid odds or commission leading to division by zero in lay stake calculation")

profit = (stake\_d \* (inputs['back\_odds'] - 1) \* (1 - inputs['back\_commission'])) - (

lay\_stake \* (inputs['lay\_odds'] - 1) \* (1 - inputs['commission'])

)

return StrategyCalculators.create\_standard\_result(

bet\_type="Binary Bet Type",

profit\_scenarios={"win": profit},

lay\_stakes={"win": lay\_stake},

stake=stake\_d,

commission=inputs['commission'],

back\_commission=inputs['back\_commission']

)

**Multi-Outcome Bet Template**

@staticmethod

def calculate\_multi\_outcome\_bet\_type(

back\_odds\_dict: Dict[str, Union[int, float, Decimal]],

lay\_odds\_dict: Dict[str, Union[int, float, Decimal]],

stake: Union[int, float, Decimal],

commission: Union[int, float, Decimal] = Decimal('0.02'),

back\_commission: Union[int, float, Decimal] = Decimal('0.00')

) -> Dict[str, Union[str, bool, Decimal, Dict]]:

"""

Calculate profit for Multi-Outcome Bet Type.

Args:

back\_odds\_dict: Dictionary mapping outcomes to back odds

lay\_odds\_dict: Dictionary mapping outcomes to lay odds

stake: Amount for each back bet

commission: Commission rate for lay bet (default: 0.02 or 2%)

back\_commission: Commission rate for back bet (default: 0.00 or 0%)

Returns:

Dictionary with profit scenarios, lay stakes, and profitability status

Example:

>>> calculate\_multi\_outcome\_bet\_type(

... {"home": 2.0, "draw": 3.0, "away": 4.0},

... {"home": 2.1, "draw": 3.1, "away": 4.1},

... 100.0

... )

{"type": "Multi-Outcome Bet Type", "profit\_scenarios": {"home": -15.0, "draw": -20.0, "away": -25.0},

"min\_profit": -25.0, "is\_profitable": False, "lay\_stakes": {"home": 95.24, "draw": 96.77, "away": 97.56},

"stake": 100.0, "commission": 0.02, "back\_commission": 0.00}

"""

logger.info(f"Calculating Multi-Outcome Bet Type: back\_odds\_dict={back\_odds\_dict}, lay\_odds\_dict={lay\_odds\_dict}, stake={stake}, commission={commission}, back\_commission={back\_commission}")

inputs = {

'back\_odds\_dict': back\_odds\_dict,

'lay\_odds\_dict': lay\_odds\_dict,

'stake': stake,

'commission': commission,

'back\_commission': back\_commission

}

StrategyCalculators.validate\_inputs(

inputs,

['back\_odds\_dict', 'lay\_odds\_dict', 'stake'],

{'stake': (int, float, Decimal), 'back\_odds\_dict': (dict,), 'lay\_odds\_dict': (dict,),

'commission': (int, float, Decimal), 'back\_commission': (int, float, Decimal)}

)

# Verify dictionaries have matching keys

if set(back\_odds\_dict.keys()) != set(lay\_odds\_dict.keys()):

logger.error(f"Mismatched keys in back\_odds\_dict and lay\_odds\_dict: {back\_odds\_dict.keys()} vs {lay\_odds\_dict.keys()}")

raise InvalidInputError("back\_odds\_dict and lay\_odds\_dict must have the same keys")

stake\_d = inputs['stake']

# Calculate lay stakes for each outcome

lay\_stakes = {}

for outcome in back\_odds\_dict:

lay\_stakes[outcome] = StrategyCalculators.calculate\_lay\_stake(

stake\_d,

back\_odds\_dict[outcome],

lay\_odds\_dict[outcome],

inputs['commission']

)

if lay\_stakes[outcome] is None:

logger.error(f"Failed to calculate lay stake for outcome {outcome}")

raise DivisionByZeroError(f"Invalid odds or commission for outcome {outcome}")

# Calculate profit scenarios

profit\_scenarios = StrategyCalculators.calculate\_profit\_scenarios(

stake\_d,

back\_odds\_dict,

lay\_odds\_dict,

lay\_stakes,

inputs['commission'],

inputs['back\_commission']

)

return StrategyCalculators.create\_standard\_result(

bet\_type="Multi-Outcome Bet Type",

profit\_scenarios=profit\_scenarios,

lay\_stakes=lay\_stakes,

stake=stake\_d,

commission=inputs['commission'],

back\_commission=inputs['back\_commission']

)

**Preamble Code (For Claude 1 Only)**

import logging

import json

from decimal import Decimal, ROUND\_HALF\_UP

from typing import Dict, Union, Optional, List, Any, Tuple

from functools import lru\_cache

# Configure logging

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'

)

logger = logging.getLogger(\_\_name\_\_)

# Custom exceptions

class BettingCalculationError(Exception):

"""Base exception for all betting calculation errors."""

pass

class InvalidInputError(BettingCalculationError):

"""Exception raised when input validation fails."""

pass

class DivisionByZeroError(BettingCalculationError):

"""Exception raised when division by zero occurs in calculations."""

pass

class UnsupportedMarketError(BettingCalculationError):

"""Exception raised when an unsupported market type is requested."""

pass

class StrategyCalculators:

"""

Class providing methods to calculate profits for various betting strategies.

This class contains:

1. Helper methods for input validation and stake calculation

2. 500 bet type calculation methods for different football markets

3. Utility methods for standardized result creation

All calculation methods return a standardized dictionary containing:

- The bet type

- Profit scenarios

- Minimum profit

- Profitability status

- Lay stakes

- Original stake

- Commission rates

"""

@staticmethod

def validate\_inputs(inputs: Dict[str, Any], required\_keys: List[str],

type\_checks: Optional[Dict[str, tuple]] = None) -> None:

"""

Validate input parameters for bet calculations.

Args:

inputs: Dictionary of input parameters

required\_keys: List of keys that must be present

type\_checks: Dictionary mapping keys to expected types

Raises:

InvalidInputError: If validation fails

"""

# Check required keys

for key in required\_keys:

if key not in inputs:

logger.warning(f"Missing required parameter: {key}")

raise InvalidInputError(f"Missing required parameter: {key}")

# Check types if specified

if type\_checks:

for key, expected\_types in type\_checks.items():

if key in inputs and not isinstance(inputs[key], expected\_types):

logger.warning(f"Invalid type for {key}. Expected {expected\_types}, got {type(inputs[key])}")

raise InvalidInputError(f"Invalid type for {key}. Expected {expected\_types}, got {type(inputs[key])}")

# Convert numeric values to Decimal and validate odds/commission

for key, value in inputs.items():

if isinstance(value, (int, float)):

inputs[key] = Decimal(str(value)).quantize(Decimal('0.01'), rounding=ROUND\_HALF\_UP)

if key in ['commission', 'back\_commission'] and inputs[key] >= 1:

logger.warning(f"Commission {inputs[key]} >= 1 is invalid")

raise InvalidInputError("Commission must be between 0 and 1")

@staticmethod

@lru\_cache(maxsize=1000)

def calculate\_lay\_stake(stake: Union[int, float, Decimal],

back\_odds: Union[int, float, Decimal],

lay\_odds: Union[int, float, Decimal],

commission: Union[int, float, Decimal] = Decimal('0.02')) -> Decimal:

"""

Calculate the appropriate lay stake to use when laying off a back bet.

Args:

stake: Amount for the back bet

back\_odds: Decimal odds for the back bet

lay\_odds: Decimal odds for the lay bet

commission: Commission rate for lay bet (default: 0.02 or 2%)

Returns:

Calculated lay stake

Raises:

InvalidInputError: If inputs are invalid

DivisionByZeroError: If division by zero occurs

"""

inputs = {'stake': stake, 'back\_odds': back\_odds, 'lay\_odds': lay\_odds, 'commission': commission}

StrategyCalculators.validate\_inputs(

inputs,

['stake', 'back\_odds', 'lay\_odds', 'commission'],

{k: (int, float, Decimal) for k in inputs.keys()}

)

stake\_d, back\_odds\_d, lay\_odds\_d, commission\_d = (

inputs['stake'],

inputs['back\_odds'],

inputs['lay\_odds'],

inputs['commission']

)

# Validate positivity

if stake\_d <= 0 or back\_odds\_d <= 0 or lay\_odds\_d <= 0:

logger.warning(f"Invalid inputs: stake={stake\_d}, back\_odds={back\_odds\_d}, lay\_odds={lay\_odds\_d}")

raise InvalidInputError("Stake and odds must be positive")

# Validate odds >= 1.0

if back\_odds\_d < 1 or lay\_odds\_d < 1:

logger.warning(f"Odds below 1: back\_odds={back\_odds\_d}, lay\_odds={lay\_odds\_d}")

raise InvalidInputError("Odds must be >= 1.0")

# Calculate denominator and check for division by zero

denominator = lay\_odds\_d \* (1 - commission\_d) - 1

if denominator <= 0:

logger.warning(f"Division by zero: lay\_odds={lay\_odds\_d}, commission={commission\_d}")

raise DivisionByZeroError("Invalid lay odds or commission leading to division by zero")

# Calculate and return lay stake

lay\_stake = (stake\_d \* back\_odds\_d / denominator).quantize(Decimal('0.01'), rounding=ROUND\_HALF\_UP)

logger.debug(f"Calculated lay\_stake: {lay\_stake}")

return lay\_stake

@staticmethod

def calculate\_profit\_scenarios(stake: Union[int, float, Decimal],

back\_odds\_dict: Dict[str, Union[int, float, Decimal]],

lay\_odds\_dict: Dict[str, Union[int, float, Decimal]],

lay\_stakes: Dict[str, Decimal],

commission: Union[int, float, Decimal],

back\_commission: Union[int, float, Decimal]) -> Dict[str, Decimal]:

"""

Calculate profit scenarios for multi-outcome bets.

Args:

stake: Amount for each back bet

back\_odds\_dict: Dictionary mapping outcomes to back odds

lay\_odds\_dict: Dictionary mapping outcomes to lay odds

lay\_stakes: Dictionary mapping outcomes to lay stakes

commission: Commission rate for lay bet

back\_commission: Commission rate for back bet

Returns:

Dictionary mapping outcomes to profit amounts

Raises:

InvalidInputError: If inputs are invalid

"""

inputs = {

'stake': stake,

'back\_odds\_dict': back\_odds\_dict,

'lay\_odds\_dict': lay\_odds\_dict,

'lay\_stakes': lay\_stakes,

'commission': commission,

'back\_commission': back\_commission

}

StrategyCalculators.validate\_inputs(

inputs,

['stake', 'back\_odds\_dict', 'lay\_odds\_dict', 'lay\_stakes', 'commission', 'back\_commission'],

{

'stake': (int, float, Decimal),

'back\_odds\_dict': (dict,),

'lay\_odds\_dict': (dict,),

'lay\_stakes': (dict,),

'commission': (int, float, Decimal),

'back\_commission': (int, float, Decimal)

}

)

stake\_d = inputs['stake']

# Convert odds to Decimal

back\_odds = {k: Decimal(str(v)) for k, v in back\_odds\_dict.items()}

lay\_odds = {k: Decimal(str(v)) for k, v in lay\_odds\_dict.items()}

commission\_d = inputs['commission']

back\_commission\_d = inputs['back\_commission']

# Verify all dictionaries have matching keys

if set(back\_odds.keys()) != set(lay\_odds.keys()) or set(back\_odds.keys()) != set(lay\_stakes.keys()):

logger.warning("Mismatched keys in back\_odds\_dict, lay\_odds\_dict, and lay\_stakes")

raise InvalidInputError("All dictionaries must have the same keys")

# Calculate profit for each scenario

profit\_scenarios = {}

# Optimize by pre-computing total liability

total\_lay\_liability = sum(

lay\_stakes[outcome] \* (lay\_odds[outcome] - 1) \* (1 - commission\_d)

for outcome in lay\_stakes

)

for outcome in back\_odds:

# For each outcome, we win on the back bet but lose on all other lay bets

target\_liability = lay\_stakes[outcome] \* (lay\_odds[outcome] - 1) \* (1 - commission\_d)

# Profit = back winnings - (total lay liability - current outcome liability)

profit = (stake\_d \* (back\_odds[outcome] - 1) \* (1 - back\_commission\_d)) - (total\_lay\_liability - target\_liability)

# Round to 2 decimal places and store

profit\_scenarios[outcome] = profit.quantize(Decimal('0.01'), rounding=ROUND\_HALF\_UP)

return profit\_scenarios

@staticmethod

def create\_standard\_result(bet\_type: str,

profit\_scenarios: Dict[str, Decimal],

lay\_stakes: Dict[str, Decimal],

stake: Decimal,

commission: Decimal,

back\_commission: Decimal,

extra\_context: Optional[Dict[str, Any]] = None) -> Dict[str, Union[str, bool, Decimal, Dict]]:

"""

Create a standardized result dictionary for bet calculations.

Args:

bet\_type: Name of the bet type

profit\_scenarios: Dictionary mapping outcomes to profit amounts

lay\_stakes: Dictionary mapping outcomes to lay stakes

stake: Original back bet amount

commission: Commission rate for lay bet

back\_commission: Commission rate for back bet

extra\_context: Optional additional context to include

Returns:

Standardized result dictionary

"""

# Find minimum profit across all scenarios

min\_profit = min(profit\_scenarios.values())

# Create standard result structure

result = {

"type": bet\_type,

"profit\_scenarios": profit\_scenarios,

"min\_profit": min\_profit,

"is\_profitable": min\_profit > 0,

"lay\_stakes": lay\_stakes,

"stake": stake,

"commission": commission,

"back\_commission": back\_commission

}

# Add any extra context if provided

if extra\_context:

result.update(extra\_context)

# Log result summary

logger.info(f"{bet\_type} calculation: min\_profit={min\_profit:.2f}, is\_profitable={min\_profit > 0}")

return result

**Output Format**

Your implementation should be ready to be directly added to the strategy\_calculators.py file without any modification. Make sure your code is well-formatted, follows PEP 8 guidelines, and includes detailed docstrings for all methods.

**Submission**

Submit your completed code chunk as an artifact. The Manager Claude will review your work and may request specific modifications.