



BAD DATA SCIENCE CODING PRACTICES

Vi Ly

7 Jun 2023

AGENDA

- Why Are Data Scientists Bad Coders?
- Technical Debt
- Bad Coding Practices
- Resources
- Self-Documenting Code
- Python Tricks

Data Scientist: Person who is better at statistics than any software engineering and ~~better at software engineering than any statistician.~~



- All examples were egregiously committed by a Lead Data Scientist in a single project
- If it can happen to a Lead DS, it can happen to you!



WHY ARE DATA SCIENTISTS BAD CODERS?

- Clean code not “sexy”
- Most DS curriculum do not focus on clean code
 - Too much emphasis on what is the newest, most advanced algorithm
 - Too much emphasis on pandas, numpy, sklearn, etc.
 - Not enough emphasis on basic Python data structures (list, tuple, dict, set)
 - Do not cover Space / Time Complexity (Big O Notation)
 - No concept of Technical Debt
 - No coverage of PEP8
 - Too much emphasis on Notebooks
 - Not enough emphasis on using Integrated Development Environment (IDE)
- What changes between school DS vs. real world DS?
 - Individual sport → Team sport

TECHNICAL DEBT

- Tech Debt: implied cost of future reworking required when choosing an easy but limited solution instead of a better approach that could take more time (https://en.wikipedia.org/wiki/Technical_debt)
- Bad code incurs higher tech debt and higher cognitive load
- When do you pay Tech Debt?
 - Someone else on your team takes over your code
 - More time (higher chance for error) is required to understand bad code than good code
 - Someone on another team has to productionize your model
 - You look back at your own code X months later and forgot what it does
 - Data migration
 - Package changes
 - Lots more examples



BAD CODING PRACTICES

- DRY Violations
- Version Control by Naming
- Confounding / Similar names
- Inconsistent Naming Convention
- Commenting Out Unused / Dead Code
- Useless Comments
- Using Magic Numbers
- Verbose Conditionals
- Long Functions
- Crowded Code
- Wildcard Imports
- Mid Code Violations
- Out-of-Order Notebook Cells

DRY VIOLATIONS

- Don't Repeat Yourself (DRY)
- How does it happen?
 - Copy & Paste code blocks
 - Hard-coding values
- Why is it bad?
 - When change is required
 - Spend time finding all the different instances
 - Update code in multiple places – higher chance of missing
- Fix
 - Use functions / classes (Vi's Rule of Thumb - 2 Probably, 3 Definitely)
 - Use loops
 - Separate files for functions / classes
 - Use better naming / variables

DRY VIOLATION (BAD)

```
array1 = [1, 2, 3, 4, 5, 6]
numeric_array = array1
n = len(numeric_array)
mean = sum(numeric_array) / n
stdev_numerator = []
```

```
for x in numeric_array:
    stdev_numerator.append(
        # Note: This is purposely wrong for demo purposes
        (x + mean) ** 2
    )
```

```
stdev = math.sqrt(
    sum(stdev_numerator) / (n - 1)
)
```

```
z_scores = []
for x in numeric_array:
    z_scores.append(
        (x - mean) / stdev
    )
array1_zscores = z_scores
```

```
array2 = [7, 8, 9, 10, 11, 12]
numeric_array = array2
n = len(numeric_array)
mean = sum(numeric_array) / n
stdev_numerator = []
```

```
for x in numeric_array:
    stdev_numerator.append(
        # Note: This is purposely wrong for demo purposes
        (x + mean) ** 2
    )
```

```
stdev = math.sqrt(
    sum(stdev_numerator) / (n - 1)
)
```

```
z_scores = []
for x in numeric_array:
    z_scores.append(
        (x - mean) / stdev
    )
array2_zscores = z_scores
```

```
array3 = [13, 14, 15, 16, 17, 18]
numeric_array = array2
n = len(numeric_array)
mean = sum(numeric_array) / n
stdev_numerator = []
```

```
for x in numeric_array:
    stdev_numerator.append(
        # Note: This is purposely wrong for demo purposes
        (x + mean) ** 2
    )
```

```
stdev = math.sqrt(
    sum(stdev_numerator) / (n - 1)
)
```

```
z_scores = []
for x in numeric_array:
    z_scores.append(
        (x - mean) / stdev
    )
array3_zscores = z_scores
```

DRY VIOLATION (GOOD)

```
import math

def calculate_zscore(numeric_array):
    n = len(numeric_array)
    mean = sum(numeric_array) / n
    # stdev_numerator = [(x - mean) ** 2 for x in numeric_array]
    stdev_numerator = [(x - mean) ** 2 for x in numeric_array]

    stdev = math.sqrt(
        sum(stdev_numerator) / (n - 1)
    )
    return [(x - mean) / stdev for x in numeric_array]

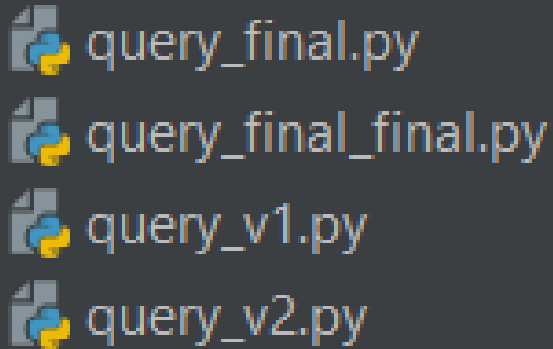
# Good, but still some duplication in that we are calling calculate_zscore 3X)
array1 = [1, 2, 3, 4, 5, 6]
array1_zscores = calculate_zscore(array1)

array2 = [7, 8, 9, 10, 11, 12]
array2_zscores = calculate_zscore(array2)

array3 = [13, 14, 15, 16, 17, 18]
array3_zscores = calculate_zscore(array3)

# Even better - call calculate_zscore once using a loop
array1_zscores, array2_zscores, array3_zscores = [calculate_zscore(x) for x in (array1, array2, array3)]
```

VERSION CONTROL BY NAMING



query_final.py
query_final_final.py
query_v1.py
query_v2.py

- Why is it bad?
 - Increased cognitive load
 - For someone reading / inheriting your code, they will always think:
 - “Is this the latest version?”
 - “Am I using the right version?”
 - Spend more time searching through all files to ensure there are no other versions

```
def train_test_split_v1(*args, **kwargs):  
    ...  
  
def train_test_split_v2(*args, **kwargs):  
    ...  
  
def train_test_split_v3(*args, **kwargs):  
    ...
```

- Fix
 - Use actual version control like Git to track changes
 - If no access to version control:
 - Raise Error / Warning (<https://docs.python.org/3/library/exceptions.html>)
 - NotImplementedError
 - DeprecationWarning
 - As a last resort: add comment / doc string

CONFOUNDING SIMILAR NAMES

```
def split_train_test(data, test_size):  
    ...  
  
def train_test_splitting(data_indices, test_size, random_state):  
    ...  
  
def train_test(df_input, month_train, month_test, features):  
    ...
```

- Actual Example
 - All 3 functions in the same file
 - Similar names
 - Similar functionality – performs some type of train / test split
- Why is it bad?
 - Reader has more to remember
 - Duplication – violates DRY principle
 - Harder to refactor
- Fix
 - Generalize function to handle multiple cases
 - Use better naming convention

INCONSISTENT NAMING CONVENTION

```
def calculate_total_amount(amt: List[float], discount: float, tax: float) → float:
    total_amnt = sum(amt)
    if discount is not None:
        discounted_amt = (total_amnt * discount)
        total_amnt -= discounted_amt
    return total_amnt * (1 + tax)
```

- How many versions of amount are used?
 - 4 (amount, amts, amt, amnt)
- Other Examples
 - Using different verb tenses: calculate vs calculating
- Why is it bad?
 - Users expect consistent naming convention
 - More cognitive load having to remember multiple conventions
 - Ctrl + F (Find) & Ctrl + R (Replace) become ineffective
 - Must manually find and replace
 - Higher chance something gets missed
- Fix
 - Use consistent naming convention (variables, function / method names, parameters)
 - When working with others, establish convention early on

COMMENTING OUT DEAD / USELESS CODE

- What is it?
 - You are experimenting and making changes to your code.
 - You comment out old code (so that it may be retrieved in case something goes wrong) during the updates, but never remove the commented code.
- Why is it bad?
 - Comment Hoarder - No one ever removes commented code. Comments just stay in perpetuity, or worse, keeps increasing.
 - New people will wonder if they ever need that code
 - More lines of code for people to read
- Fix
 - Delete unused / dead code instead of commenting out
 - Remember: This is why you have version control

USELESS COMMENTS

- What are useless comments?
 - Commenting every line of code with what the code does
- Why is it bad?
 - Waste time / no added value
 - Cry wolf – someone reviewing your code will skip critical comments because they think all comments are useless
 - DRY Violation – what happens if your code changes?
 - 2 changes instead of 1 (code and in the comment – higher chance for error)
- Fix
 - Delete useless comments
 - Use self-documenting code
 - Comments should describe why (and sometimes how) but not what.

USING MAGIC NUMBERS

- What is a Magic Number?
 - A unique value with unexplained meaning or multiple occurrences which could (preferably) be replaced with a named constant
 - [https://en.wikipedia.org/wiki/Magic_number_\(programming\)](https://en.wikipedia.org/wiki/Magic_number_(programming))
- Why is it bad?
 - Does not provide context on significance of the number
 - Potential DRY Violation
 - What happens if you have that number hard-coded in multiple places and it changes?
- Fix
 - Use self-documenting code / variable (aka better naming convention)
 - Use comments

MAGIC NUMBER EXAMPLE

- In this example, 1944 is the Magic Number

```
def was_olympics_held_bad(year: int) → bool:  
    return ((year % 4) == 0) and (year ≠ 1944)
```

```
def was_olympic_held_good(year: int) → bool:  
    year_olympics_cancelled_worldwar2 = 1944  
    return ((year % 4) == 0) and (year ≠ year_olympics_cancelled_worldwar2)
```

VERBOSE CONDITIONALS

```
def was_olympics_held_verbose(year: int) → bool:
    if ((year % 4) == 0) and (year != 1944):
        return True
    else:
        return False

def was_olympics_held_notverbose(year: int) → bool:
    return ((year % 4) == 0) and (year != 1944)
```

```
ViLy
def is_list_empty_verbose(some_list: List) → bool:
    if len(some_list) == 0:
        return True
    else:
        return False

ViLy
def is_list_empty_better(some_list: List) → bool:
    return len(some_list) == 0

ViLy
def is_list_empty_best(some_list: List) → bool:
    return bool(some_list)
```

- Why is it bad?
 - More lines of code for reader
- Fix
 - Just use the condition
 - For native Python data structures, use their natural 'truthiness' value

```
def get_email_domain_bad1(email: str, case: str) → str:
    domain = email.split('@')[-1]
    return domain.upper() if case.lower() == 'upper' else domain
```

```
get_email_domain_bad1('bdsc@gmail.com', 'upper')
'GMAIL.COM'

get_email_domain_bad1('bdsc@gmail.com', 'up')
'gmail.com'
```

```
def get_email_domain_bad2(email: str, upper_case: str) → str:
    domain = email.split('@')[-1]

    if upper_case.upper()[0] == 'Y':
        return domain.upper()
    else:
        return domain
```

```
get_email_domain_bad2('bdsc@gmail.com', 'yes')
'GMAIL.COM'

get_email_domain_bad2('bdsc@gmail.com', 'y')
'GMAIL.COM'

get_email_domain_bad2('bdsc@gmail.com', 'no')
'gmail.com'
```

```
def get_email_domain_better(email: str, upper_case: bool) → str:
    domain = email.split('@')[-1]
    return domain.upper() if upper_case else domain
```

```
get_email_domain_better('bdsc@gmail.com', True)
'GMAIL.COM'

get_email_domain_better('bdsc@gmail.com', False)
'gmail.com'
```

VERBOSE CONDITIONALS (CONT'D)

- Don't use str for binary arguments
- Why is it bad?
 - More code to “standardize” argument (i.e. lower, strip)
 - Makes both reader and user think there are more than 2 possible values
- Fix
 - Use boolean argument

LONG FUNCTIONS

- “Brevity is the soul of wit” – William Shakespeare (Hamlet)
- “Simple is better than complex” – Zen of Python
- Why is it bad?
 - Harder to debug
 - Harder to test
 - May need portions of code in other functions (DRY Violation)
 - Much higher cognitive load
 - Red flag: Do you have to put comments to remind you of what the code is doing?
- Fix
 - Break into smaller functions
 - Define large function from smaller functions

CROWDED CODE

```
bad = '''
    SELECT acct_nbr, transaction_date, SUM(amount) AS total_amount FROM
    transaction_table WHERE transaction_date ≥ '2022-08-01' AND
    transaction_date ≤ '2022-08-31' AND transaction_type = 'Credit Card' GROUP BY
    transaction_date ORDER BY acct_nbr, transaction_date
'''

good = '''
    SELECT
        acct_nbr,
        transaction_date,
        SUM(amount) AS total_amount
    FROM
        transaction_table
    WHERE
        transaction_date ≥ '2022-08-01'
        AND transaction_date ≤ '2022-08-31'
        AND transaction_type = 'Credit Card'
    GROUP BY
        transaction_date
    ORDER BY
        acct_nbr, transaction_date
'''
```

- Why is it bad?
 - Harder to read
 - Higher cognitive load
- Fix
 - Use line breaks and indentations liberally (code permitting)

WILDCARD IMPORTS

```
from sklearn import *
```

- Why is it bad?
 - Violates PEP8
 - Naming collision
 - Spend more time tracing object / variable origin
 - Becomes bigger problem when version changes break compatibility
- Fix
 - Follow PEP8
 - Use namespaces

MID-CODE VIOLATIONS

- Mid-Code Violations
 - Imports & Function / Class Definitions placed in the middle of code instead of at the beginning
 - Violates PEP 8
- Why is it bad?
 - Tech debt - More work to productionize code / notebook
 - Forces others to go through all of code looking for imports / definitions
 - Higher chance something gets missed
- Fix
 - Follow PEP8

OUT OF ORDER NOTEBOOK CELLS

```
In [1]: import pandas as pd
```

```
In [4]: df['FLOSS_NAME'] = df['FLOSS_NAME'].str.upper().str.replace(' ', '_')
```

```
In [5]: df.head()
```

Out[5]:

	DMC_CODE	FLOSS_NAME	R	G	B	RGB_CODE
0	3713	SALMON_VERY_LIGHT	255	226	226	#FFE2E2
1	761	SALMON_LIGHT	255	201	201	#FFC9C9
2	760	SALMON	245	173	173	#F5ADAD
3	3712	SALMON_MEDIUM	241	135	135	#F18787
4	3328	SALMON_DARK	227	109	109	#E36D6D

```
In [2]: df = pd.read_csv('c:/users/viqua/desktop/DMC Color Chart.csv')
```

```
In [3]: df.head()
```

Out[3]:

	DMC_CODE	FLOSS_NAME	R	G	B	RGB_CODE
0	3713	Salmon Very Light	255	226	226	#FFE2E2
1	761	Salmon Light	255	201	201	#FFC9C9
2	760	Salmon	245	173	173	#F5ADAD
3	3712	Salmon Medium	241	135	135	#F18787
4	3328	Salmon Dark	227	109	109	#E36D6D

- Why is it bad?
 - Not reproducible
 - Creates errors when run
 - Code expected to run top to bottom
 - More time to debug
 - More time to reorder
- Fix
 - Use IDE
 - Re-run notebook after completion
 - Should run error-free

RESOURCES

- Python PEP 8
 - <https://peps.python.org/pep-0008/>
- Zen of Python
 - <https://peps.python.org/pep-0020/>
- Ottinger's Rules for Variable and Class Naming
 - <https://exelearning.org/wiki/OttingersNaming/>
- The Mental Game of Python – Raymond Hettinger
 - <https://www.youtube.com/watch?v=UANN2Eu6ZnM>
- Refactoring by Martin Fowler
 - <https://martinfowler.com/books/refactoring.html>
- Clean Code by Robert C. Martin
 - <https://learning.oreilly.com/library/view/clean-code-a/9780136083238/>
- Fluent Python by Luciano Ramalho
 - <https://learning.oreilly.com/library/view/fluent-python-2nd/9781492056348/>

SELF-DOCUMENTING CODE



SELF-DOCUMENTING CODE

BAD

```
def do_something(l, c, y, h):  
  
    z = y / l  
  
    if h or (c ≥ 780) or (z ≥ 10):  
        return True  
  
    if c ≥ 650:  
        return z ≥ 5  
    else:  
        return z ≥ 10
```

SELF-DOCUMENTING CODE

BAD

```
def do_something(l, c, y, h):  
  
    z = y / l  
  
    if h or (c ≥ 780) or (z ≥ 10):  
        return True  
  
    if c ≥ 650:  
        return z ≥ 5  
    else:  
        return z ≥ 10
```

BETTER

```
def approve_loan(loan_amount, credit_score, yearly_income, has_cosigner):  
    excellent_credit_threshold = 780  
    good_credit_threshold = 650  
    excellent_income_amount_ratio_threshold = 10  
    good_income_amount_ratio_threshold = 5  
  
    income_amount_ratio = yearly_income / loan_amount  
  
    if any([  
        has_cosigner,  
        credit_score ≥ excellent_credit_threshold,  
        income_amount_ratio ≥ excellent_income_amount_ratio_threshold  
    ]):  
        return True  
    else:  
        if credit_score ≥ good_credit_threshold:  
            return income_amount_ratio ≥ good_income_amount_ratio_threshold  
        else:  
            return income_amount_ratio ≥ excellent_credit_threshold
```

SELF-DOCUMENTING CODE

BETTER

```
def approve_loan(loan_amount, credit_score, yearly_income, has_cosigner):
    excellent_credit_threshold = 780
    good_credit_threshold = 650
    excellent_income_amount_ratio_threshold = 10
    good_income_amount_ratio_threshold = 5

    income_amount_ratio = yearly_income / loan_amount

    if any([
        has_cosigner,
        credit_score ≥ excellent_credit_threshold,
        income_amount_ratio ≥ excellent_income_amount_ratio_threshold
    ]):
        return True
    else:
        if credit_score ≥ good_credit_threshold:
            return income_amount_ratio ≥ good_income_amount_ratio_threshold
        else:
            return income_amount_ratio ≥ excellent_credit_threshold
```

EVEN BETTER (Even better with Doc Strings)

```
excellent_credit_threshold = 780
good_credit_threshold = 650
excellent_income_amount_ratio_threshold = 10
good_income_amount_ratio_threshold = 5

def approve_immediately(credit_score: int, income_amount_ratio: float, has_cosigner: bool) → bool:
    return any([
        has_cosigner,
        credit_score ≥ excellent_credit_threshold,
        income_amount_ratio ≥ excellent_income_amount_ratio_threshold
    ])

def approve_based_on_ratio(credit_score: int, income_amount_ratio: float) → bool:
    if credit_score ≥ good_credit_threshold:
        return income_amount_ratio ≥ good_income_amount_ratio_threshold
    else:
        return income_amount_ratio ≥ excellent_credit_threshold

def approve_loan(loan_amount: int, credit_score: int, yearly_income: int, has_cosigner: bool) → bool:
    income_amount_ratio = yearly_income / loan_amount
    return (
        approve_immediately(credit_score, income_amount_ratio, has_cosigner)
        or approve_based_on_ratio(credit_score, income_amount_ratio)
    )
```

VI

Refactor this code:

```
def do_something(l, c, y, h):  
  
    z = y / l  
  
    if h or (c >= 780) or (z >= 10):  
        return True  
  
    if c >= 650:  
        return z >= 5  
    else:  
        return z >= 10
```

Bad Code Example



Sure, here's a refactored version of the code:



python

Copy code

```
def should_do_something(l, c, y, h):  
    z = y / l  
    if should_perform_action(h, c, z):  
        return True  
    return should_perform_alternative_action(c, z)  
  
def should_perform_action(h, c, z):  
    return h or c >= 780 or z >= 10  
  
def should_perform_alternative_action(c, z):  
    if c >= 650:  
        return z >= 5  
    return z >= 10
```

Regenerate response

My Refactored Code (Before Checking Out ChatGPT)

```
excellent_credit_threshold = 780  
good_credit_threshold = 650  
excellent_income_amount_ratio_threshold = 10  
good_income_amount_ratio_threshold = 5
```

```
def approve_immediately(credit_score: int, income_amount_ratio: float, has_cosigner: bool) → bool:  
    return any([  
        has_cosigner,  
        credit_score ≥ excellent_credit_threshold,  
        income_amount_ratio ≥ excellent_income_amount_ratio_threshold  
    ])
```

```
def approve_based_on_ratio(credit_score: int, income_amount_ratio: float) → bool:  
    if credit_score ≥ excellent_credit_threshold:  
        raise ValueError(f'credit_score should be < {excellent_credit_threshold}')  
    if credit_score ≥ good_credit_threshold:  
        return income_amount_ratio ≥ good_income_amount_ratio_threshold  
    else:  
        return income_amount_ratio ≥ excellent_credit_threshold
```

```
def approve_loan(loan_amount: int, credit_score: int, yearly_income: int, has_cosigner: bool) → bool:  
    income_amount_ratio = yearly_income / loan_amount  
    return (  
        approve_immediately(credit_score, income_amount_ratio, has_cosigner)  
        or approve_based_on_ratio(credit_score, income_amount_ratio)  
    )
```

VI

Refactor the code which is delimited by triple quotes.

Replace the original variables with new variable names using this mapping:

l: loan_amount

c: credit_score

y: yearly_income

h: has_cosigner

"""

def do_something(l, c, y, h):

z = y / l

if h or (c >= 780) or (z >= 10):

return True

if c >= 650:

return z >= 5

else:

return z >= 10

"""



Here's the refactored code using the new variable names:

arduino

Copy code

```
def check_loan_eligibility(loan_amount, credit_score, yearly_income, has_cosigner):
    income_loan_ratio = yearly_income / loan_amount

    if has_cosigner or (credit_score >= 780) or (income_loan_ratio >= 10):
        return True

    if credit_score >= 650:
        return income_loan_ratio >= 5
    else:
        return income_loan_ratio >= 10
```

Regenerate response

Send a message.

- ChatGPT inferred 2 new variable names based on the context:
 - check_loan_eligibility
 - income_loan_ratio

vi Write the docstrings for each of the Python functions. Functions are delimited by triple backticks.

```
...
def should_do_something(l, c, y, h):
    z = y / l
    if should_perform_action(h, c, z):
        return True
    return should_perform_alternative_action(c, z)
...
def should_perform_action(h, c, z):
    return h or c >= 780 or z >= 10
...
def should_perform_alternative_action(c, z):
    if c >= 650:
        return z >= 5
    return z >= 10
```

```
python Copy code

def should_perform_action(h, c, z):
    """Determine whether a certain action should be performed.

    Args:
        - h (bool): the value of h.
        - c (int): the value of c.
        - z (float): the value of z.

    Returns:
        - bool: True if the certain action should be performed, False otherwise.
    """
    return h or c >= 780 or z >= 10
```

```
python Copy code

def should_perform_alternative_action(c, z):
    """Determine whether an alternative action should be performed.

    Args:
        - c (int): the value of c.
        - z (float): the value of z.

    Returns:
        - bool: True if the alternative action should be performed, False otherwise.
    """
    if c >= 650:
        return z >= 5
    return z >= 10
```

```
python Copy code

def should_do_something(l, c, y, h):
    """Determine whether a certain action should be performed or an alternative action should be performed.

    Args:
        - l (int): the value of l.
        - c (int): the value of c.
        - y (int): the value of y.
        - h (bool): the value of h.

    Returns:
        - bool: True if the certain action should be performed, False otherwise.
    """
    z = y / l
    if should_perform_action(h, c, z):
        return True
    return should_perform_alternative_action(c, z)
```

```
python Copy code

def approve_immediately(credit_score: int, income_amount_ratio: float, has_cosigner: bool) -> bool:
    """Determine if a loan application can be approved immediately or not.

    Args:
        - credit_score (int): the credit score of the loan applicant.
        - income_amount_ratio (float): the ratio of yearly income to the loan amount.
        - has_cosigner (bool): True if the loan applicant has a cosigner, False otherwise.

    Returns:
        - bool: True if the loan application can be approved immediately, False otherwise.
    """
    return any([
        has_cosigner,
        credit_score >= excellent_credit_threshold,
        income_amount_ratio >= excellent_income_amount_ratio_threshold
    ])
```

```
python Copy code

def approve_based_on_ratio(credit_score: int, income_amount_ratio: float) -> bool:
    """Determine if a loan application can be approved based on the income amount ratio.

    Args:
        - credit_score (int): the credit score of the loan applicant.
        - income_amount_ratio (float): the ratio of yearly income to the loan amount.

    Returns:
        - bool: True if the loan application can be approved based on the income amount ratio, False otherwise.

    Raises:
        - ValueError: If credit_score is greater than or equal to excellent_credit_threshold.
    """
    if credit_score >= excellent_credit_threshold:
        raise ValueError(f'credit_score should be < {excellent_credit_threshold}')
    if credit_score >= good_credit_threshold:
        return income_amount_ratio >= good_income_amount_ratio_threshold
    else:
        return income_amount_ratio >= excellent_credit_threshold
```

```
python Copy code

def approve_loan(loan_amount: int, credit_score: int, yearly_income: int, has_cosigner: bool) -> bool:
    """Determine if a loan application can be approved or not.

    Args:
        - loan_amount (int): the amount of the loan requested.
        - credit_score (int): the credit score of the loan applicant.
        - yearly_income (int): the yearly income of the loan applicant.
        - has_cosigner (bool): True if the loan applicant has a cosigner, False otherwise.

    Returns:
        - bool: True if the loan application can be approved, False otherwise.
    """
    income_amount_ratio = yearly_income / loan_amount
    return (
        approve_immediately(credit_score, income_amount_ratio, has_cosigner)
        or approve_based_on_ratio(credit_score, income_amount_ratio)
    )
```

vi Write the docstrings for each of the Python functions. Functions are delimited by triple backticks.

```
...
def approve_immediately(credit_score: int, income_amount_ratio: float, has_cosigner: bool) -> bool:
    """Determine if a loan application can be approved immediately or not.

    Args:
        - credit_score (int): the credit score of the loan applicant.
        - income_amount_ratio (float): the ratio of yearly income to the loan amount.
        - has_cosigner (bool): True if the loan applicant has a cosigner, False otherwise.

    Returns:
        - bool: True if the loan application can be approved immediately, False otherwise.
    """
    return any([
        has_cosigner,
        credit_score >= excellent_credit_threshold,
        income_amount_ratio >= excellent_income_amount_ratio_threshold
    ])
...
def approve_based_on_ratio(credit_score: int, income_amount_ratio: float) -> bool:
    """Determine if a loan application can be approved based on the income amount ratio.

    Args:
        - credit_score (int): the credit score of the loan applicant.
        - income_amount_ratio (float): the ratio of yearly income to the loan amount.

    Returns:
        - bool: True if the loan application can be approved based on the income amount ratio, False otherwise.

    Raises:
        - ValueError: If credit_score is greater than or equal to excellent_credit_threshold.
    """
    if credit_score >= excellent_credit_threshold:
        raise ValueError(f'credit_score should be < {excellent_credit_threshold}')
    if credit_score >= good_credit_threshold:
        return income_amount_ratio >= good_income_amount_ratio_threshold
    else:
        return income_amount_ratio >= excellent_credit_threshold
...
def approve_loan(loan_amount: int, credit_score: int, yearly_income: int, has_cosigner: bool) -> bool:
    """Determine if a loan application can be approved or not.

    Args:
        - loan_amount (int): the amount of the loan requested.
        - credit_score (int): the credit score of the loan applicant.
        - yearly_income (int): the yearly income of the loan applicant.
        - has_cosigner (bool): True if the loan applicant has a cosigner, False otherwise.

    Returns:
        - bool: True if the loan application can be approved, False otherwise.
    """
    income_amount_ratio = yearly_income / loan_amount
    return (
        approve_immediately(credit_score, income_amount_ratio, has_cosigner)
        or approve_based_on_ratio(credit_score, income_amount_ratio)
    )
```


PYTHON TRICKS





PYTHON TRICKS

- Number Underscore
- Tuple Unpacking
- Collections
- Set
- Mapping
- Itertools
- Functools
- Type Hints
- Jinja2
- Black