BỘ THÔNG TIN VÀ TRUYỀN THÔNG

HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG


# BÀI BÁO CÁO

## MÔN: LẬP TRÌNH PYTHON

**Giảng viên:** Kim Ngọc Bách

**Tên sinh viên:** Nguyễn Minh Tuấn Kiệt

**Mã sinh viên:** B23DCCE060

**Lớp:** D23CQCE06 - B

**Môn:** Lập trình Python

**Hà Nội, Tháng 6/2025**

# Bài 1:

# Mã Nguồn Đầy Đủ

```python
import time
import pandas as pd
from collections import Counter
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
from bs4 import BeautifulSoup, Comment
import sys
import traceback
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium.common.exceptions import TimeoutException

# Safe text retrieval function, returns 'N/a' on error
def safe_get_text(element, default='N/a'):
    if not element: return default
    text = element.get_text(strip=True)
    return text or default

# Function to process nationality (for 3-letter codes)
def get_nationality(td_element):
    if td_element is None: return 'N/a'
    try:
        strings = list(td_element.stripped_strings)
        full_text = ' '.join(strings)
        if not full_text: return 'N/a'
        parts = full_text.split()
        if not parts: return 'N/a'
        # Prioritize standalone 3-letter uppercase codes
        for i in range(len(parts) - 1, -1, -1):
             part = parts[i]
             if len(part) == 3 and part.isupper() and part.isalpha():
                  return part
        # If not found, check text within link (if any)
        link = td_element.find('a')
        if link:
            link_text = safe_get_text(link)
            if link_text != 'N/a' and len(link_text) >= 2 and len(link_text) <=
                4 and link_text.isupper() and link_text.isalpha():
                return link_text
        # If still not found, take the last element if it's a country code (can
            be 2-4 letters)
        last_part = parts[-1]
        if len(last_part) <= 4 and last_part.isupper() and last_part.isalpha():
```

```
            return last_part
        # Last case, if only one element and it's a country code
        if len(parts) == 1 and len(parts[0]) <= 4 and parts[0].isupper() and
            parts[0].isalpha():
            return parts[0]
        return 'N/a'
    except Exception as e:
        # print(f"Error processing nationality: {e} for element: {td_element}")
        return 'N/a'

# Function to calculate age from birth year string or age string
def calculate_age(age_or_birth_str, current_year=None):
    if current_year is None:
        try: current_year = pd.Timestamp.now().year
        except: current_year = 2025 # Fallback year

    if not isinstance(age_or_birth_str, str) or age_or_birth_str == 'N/a':
        return 'N/a'
    age_or_birth_str = age_or_birth_str.strip()

    # Case 1: String already contains age
    try:
        if '-' in age_or_birth_str:
            age_part = age_or_birth_str.split('-')[0]
            if age_part.isdigit() and 14 < int(age_part) < 50: return age_part
        elif age_or_birth_str.isdigit() and 14 < int(age_or_birth_str) < 50:
            return age_or_birth_str
    except (ValueError, TypeError): pass

    # Case 2: String contains birth year
    try:
        if '-' in age_or_birth_str and len(age_or_birth_str.split('-')) == 3: #
            YYYY-MM-DD format
            parts = age_or_birth_str.split('-')
            if len(parts[0]) == 4 and parts[0].isdigit():
                birth_year = int(parts[0])
                if 1900 < birth_year <= current_year: return str(current_year -
                    birth_year)

        year_part = ''.join(filter(str.isdigit, age_or_birth_str)) # Find any
            4-digit year
        if len(year_part) >= 4:
            potential_years = [year_part[i:i+4] for i in range(len(year_part) -
                3)]
            for year_str in potential_years:
                try:
                    birth_year = int(year_str)
                    if 1900 < birth_year <= current_year: return
                        str(current_year - birth_year)
                except ValueError: continue

```

```
        if ',' in age_or_birth_str: # "Month Day, YYYY" format
            year_str = age_or_birth_str.split(',')[-1].strip()
            if len(year_str) == 4 and year_str.isdigit():
                birth_year = int(year_str)
                if 1900 < birth_year <= current_year: return str(current_year - birth_year)
    except (ValueError, TypeError, IndexError): pass

    # Case 3: Only a 4-digit birth year
    try:
        if len(age_or_birth_str) == 4 and age_or_birth_str.isdigit():
            birth_year = int(age_or_birth_str)
            if 1900 < birth_year <= current_year: return str(current_year - birth_year)
    except (ValueError, TypeError): pass
    return 'N/a'

# Function to scrape a table from a given URL
def scrape_fbref_table(driver, url, table_id=None, required_stats=None, min_minutes=90):
    print(f"Attempting to scrape data from: {url}")
    try:
        driver.get(url)
        print(" Page requested. Waiting for table...")

        wait_time = 25
        locator = (By.ID, table_id) if table_id else (By.CSS_SELECTOR, "table.stats_table")
        print(f" Waiting for table with {'ID: ' + table_id if table_id else 'class stats_table'}")

        try:
            WebDriverWait(driver, wait_time).until(EC.visibility_of_element_located(locator))
            print(f" Table {locator} visible.")
        except TimeoutException:
            print(f" Warning: Table {locator} not visible within {wait_time}s on {url}. Checking HTML comments...")

        time.sleep(1) # Allow JS to fully render
        html = driver.page_source
        soup = BeautifulSoup(html, 'html.parser')
        print(" Parsed page source.")

        data_table = soup.find('table', {'id': table_id}) if table_id else None
        if not data_table: data_table = soup.find('table', {'class': lambda x: x and 'stats_table' in x.split()})

        if not data_table: # Check HTML comments
            comments = soup.find_all(string=lambda text: isinstance(text, Comment))
```

```
            for comment in comments:
                comment_soup = BeautifulSoup(comment, 'html.parser')
                potential_table = comment_soup.find('table', {'id': table_id})
                    if table_id else None
                if not potential_table: potential_table =
                    comment_soup.find('table', {'class': lambda x: x and
                    'stats_table' in x.split()})
                if potential_table:
                    print(f" Found table {'with ID '+table_id if table_id else
                        ''} in HTML comment.")
                    data_table = potential_table
                    break
        if not data_table:
            print(f"Error: Table not found on {url}.")
            return pd.DataFrame()

        tbody = data_table.find('tbody')
        rows = tbody.find_all('tr') if tbody else [r for r in
            data_table.find_all('tr') if r.find(['th', 'td'], {'data-stat':
            True}) and not r.find('th', {'scope':'col'})]
        if not tbody and not rows:
            print(f"Error: No data rows found in table on {url}")
            return pd.DataFrame()
        elif not tbody:
            print(f" Found {len(rows)} potential data rows directly in table.")

        print(f" Found {len(rows)} rows for {url}. Processing...")

        base_stats_needed = {'player', 'team', 'nationality', 'position',
            'age', 'birth_year', 'minutes', 'minutes_90s'}
        stats_to_extract = set(base_stats_needed)
        if required_stats:
             stats_to_extract.update(required_stats)
        else: # If no required_stats, get from header
             print(" Warning: No specific list of required stats, will fetch
                from table header.")
             thead = data_table.find('thead')
             if thead and (header_rows := thead.find_all('tr')):
                 last_header_row = header_rows[-1]
                 header_stats = {th.get('data-stat', '').strip() for th in
                     last_header_row.find_all('th')}
                 stats_to_extract.update(stat for stat in header_stats if stat
                     and stat not in ['ranker', 'matches', 'match_report'])
                 print(f" Dynamically fetching stats from header:
                     {sorted(list(stats_to_extract - base_stats_needed))}")

        players_data = []
        collected_count, skipped_header, skipped_minutes, skipped_no_player =
            0, 0, 0, 0

        for i, row in enumerate(rows):
```

```
            if row.has_attr('class') and any(c in row['class'] for c in
                ['thead', 'partial_table', 'spacer']):
                skipped_header += 1; continue
            if not row.find(['th','td'], {'data-stat' : True}): continue

            player_cell = row.find(['th', 'td'], {'data-stat': 'player'})
            player_name = safe_get_text(player_cell)
            if player_name == 'N/a' or player_name == '' or player_name ==
                'Player':
                skipped_no_player += 1; continue

            minutes_played_num = -1
            minutes_td = row.find('td', {'data-stat': 'minutes'})
            minutes_90s_td = row.find('td', {'data-stat': 'minutes_90s'})
            minutes_str = safe_get_text(minutes_td, '').replace(',', '')
            minutes_90s_str = safe_get_text(minutes_90s_td, '').replace(',', '')

            try:
                if minutes_str.isdigit():
                    minutes_played_num = int(minutes_str)
                elif minutes_90s_str:
                    try: minutes_played_num = float(minutes_90s_str) * 90
                    except ValueError:
                        if minutes_90s_str.isdigit(): minutes_played_num =
                            int(minutes_90s_str) * 90
                        else: minutes_played_num = -1
                if not (minutes_played_num < 0 and minutes_td is None and
                    minutes_90s_td is None) and minutes_played_num <
                    min_minutes: # Allow players with no minutes data if fields
                    are missing, otherwise filter
                    skipped_minutes += 1; continue
            except (ValueError, TypeError, AttributeError):
                 skipped_minutes += 1; continue

            player_stats = {}
            all_cells = row.find_all(['th', 'td'])
            processed_stats_in_row = set()

            for cell in all_cells:
                stat = cell.get('data-stat', '').strip()
                if stat and stat in stats_to_extract and stat not in
                    processed_stats_in_row:
                    processed_stats_in_row.add(stat)
                    if stat == 'nationality': player_stats['nationality'] =
                        get_nationality(cell)
                    elif stat == 'birth_year' or stat == 'age': # 'age' column
                        often contains birth year or age
                         age_birth_text = safe_get_text(cell)
                         if 'original_age_value' not in player_stats:
                             player_stats['original_age_value'] = age_birth_text
                         calculated_age = calculate_age(age_birth_text)
```

```
                    if calculated_age != 'N/a': player_stats['Age'] =
                        calculated_age
                    elif player_stats.get('Age', 'N/a') == 'N/a':
                        player_stats['Age'] = age_birth_text if
                        age_birth_text != 'N/a' else 'N/a'
                elif stat == 'player': player_stats['Player'] = player_name
                elif stat == 'team':
                    team_name = safe_get_text(cell.find('a'),
                        default=safe_get_text(cell))
                    player_stats['Team'] = team_name
                elif stat == 'position':
                    position_text = safe_get_text(cell)
                    player_stats['Position'] =
                        position_text.split(',')[0].strip() if ',' in
                        position_text and
                        position_text.split(',')[0].strip() else
                        position_text
                elif stat == 'minutes': player_stats['minutes'] =
                    minutes_str or '0'
                elif stat == 'minutes_90s': player_stats['minutes_90s'] =
                    minutes_90s_str or '0.0'
                else: player_stats[stat] = safe_get_text(cell)

            # Fallbacks for essential columns if not picked up by general loop
            player_stats.setdefault('Player', player_name)
            if 'Team' not in player_stats:
                team_td_fallback = row.find('td', {'data-stat': 'team'})
                player_stats['Team'] =
                    safe_get_text(team_td_fallback.find('a'),
                    default=safe_get_text(team_td_fallback)) if team_td_fallback
                    else 'N/a'
            if 'Position' not in player_stats:
                pos_td_fallback = row.find('td', {'data-stat': 'position'})
                pos_text_fb = safe_get_text(pos_td_fallback)
                player_stats['Position'] = (pos_text_fb.split(',')[0].strip()
                    if ',' in pos_text_fb and pos_text_fb.split(',')[0].strip()
                    else pos_text_fb) if pos_td_fallback else 'N/a'
            if player_stats.get('Age', 'N/a') == 'N/a':
                age_td_fallback = row.find('td', {'data-stat': 'age'})
                age_text_fallback = safe_get_text(age_td_fallback)
                player_stats['Age'] = calculate_age(age_text_fallback)
                if 'original_age_value' not in player_stats:
                    player_stats['original_age_value'] = age_text_fallback
            if 'nationality' not in player_stats:
                nat_td_fallback = row.find('td', {'data-stat': 'nationality'})
                player_stats['nationality'] = get_nationality(nat_td_fallback)
            player_stats.setdefault('minutes', minutes_str or '0')
            player_stats.setdefault('minutes_90s', minutes_90s_str or '0.0')

            players_data.append(player_stats)
            collected_count += 1
```

```
245
246         print(f" Finished processing rows for {url}. Summary - Found:
                {len(rows)}, Skipped header: {skipped_header}, No player name:
                {skipped_no_player}, Low minutes ({min_minutes}):
                {skipped_minutes}, Collected: {collected_count}")
247         if not players_data:
248             print(f"Warning: No player data met criteria from {url}.")
249             return pd.DataFrame()
250
251         df = pd.DataFrame(players_data)
252         if 'Player' in df.columns and 'Team' in df.columns: # Deduplication
253             if 'minutes' in df.columns:
254                 df['minutes_numeric'] =
                        pd.to_numeric(df['minutes'].astype(str).str.replace(',',
                        ''), errors='coerce').fillna(0)
255                 df = df.sort_values(by=['Player', 'Team', 'minutes_numeric'],
                        ascending=[True, True, False])
256                 df = df.drop_duplicates(subset=['Player', 'Team'],
                        keep='first').drop(columns=['minutes_numeric'])
257             else: df = df.drop_duplicates(subset=['Player', 'Team'],
                    keep='first')
258
259             try: # Set index
260                 df['Player'] = df['Player'].astype(str)
261                 df['Team'] = df['Team'].astype(str)
262                 if 'position' in df.columns and 'Position' not in df.columns:
                        df.rename(columns={'position': 'Position'}, inplace=True)
263                 elif 'position' in df.columns and 'Position' in df.columns:
                        df.drop(columns=['position'], inplace=True)
264                 df = df.set_index(['Player', 'Team'])
265                 print(f" Created and indexed DataFrame for {url}. Shape:
                        {df.shape}")
266             except KeyError as e:
267                 print(f"Error setting index for {url}: {e}. Columns:
                        {df.columns.tolist()}")
268                 return df if not df.empty else pd.DataFrame()
269         else:
270             print(f"Error: Missing 'Player' or 'Team' in {url}. Columns:
                    {df.columns.tolist()}")
271             return df if not df.empty else pd.DataFrame()
272         time.sleep(1.5) # Anti-blocking delay
273         return df
274     except TimeoutException as e:
275         print(f"Scraping error for {url}: Page element timed out. {e}")
276     except Exception as e:
277         print(f"Unknown error scraping {url}: {e}\nTraceback:
                {traceback.format_exc()}")
278     return pd.DataFrame()
279
280 # User-requested stats and FBRef mapping (Category, Sub-Category, Statistic
    Name) -> FBRef Key
```

```
USER_REQUESTED_STAT_MAPPING = {
    ('', '', 'Nation'): 'nationality', ('', '', 'Position'): 'Position', ('', '', 'Age'): 'Age',
    ('Playing Time', '', 'MP'): 'games', ('Playing Time', '', 'Starts'): 'games_starts', ('Playing Time', '', 'Min'): 'minutes',
    ('Performance', '', 'Gls'): 'goals', ('Performance', '', 'Ast'): 'assists', ('Performance', '', 'CrdY'): 'cards_yellow', ('Performance', '', 'CrdR'): 'cards_red',
    ('Expected', '', 'xG'): 'xg', ('Expected', '', 'xAG'): 'xg_assist', # FBRef uses xAG
    ('Progression', '', 'PrgC'): 'progressive_carries', ('Progression', '', 'PrgP'): 'progressive_passes', ('Progression', '', 'PrgR'): 'progressive_passes_received',
    ('Per 90 Minutes', '', 'Gls'): 'goals_per90', ('Per 90 Minutes', '', 'Ast'): 'assists_per90', ('Per 90 Minutes', '', 'xG'): 'xg_per90', ('Per 90 Minutes', '', 'xGA'): 'xg_assist_per90',
    ('Goalkeeping', 'Performance', 'GA90'): 'gk_goals_against_per90', ('Goalkeeping', 'Performance', 'Save%'): 'gk_save_pct', ('Goalkeeping', 'Performance', 'CS%'): 'gk_clean_sheets_pct', ('Goalkeeping', 'Penalty Kicks', 'Save%'): 'gk_pens_save_pct',
    ('Shooting', 'Standard', 'SoT%'): 'shots_on_target_pct', ('Shooting', 'Standard', 'SoT/90'): 'shots_on_target_per90', ('Shooting', 'Standard', 'G/Sh'): 'goals_per_shot', ('Shooting', 'Standard', 'Dist'): 'average_shot_distance',
    ('Passing', 'Total', 'Cmp'): 'passes_completed', ('Passing', 'Total', 'Cmp%'): 'passes_pct', ('Passing', 'Total', 'TotDist'): 'passes_total_distance',
    ('Passing', 'Short', 'Cmp%'): 'passes_pct_short', ('Passing', 'Medium', 'Cmp%'): 'passes_pct_medium', ('Passing', 'Long', 'Cmp%'): 'passes_pct_long',
    ('Passing', 'Expected', 'KP'): 'assisted_shots', ('Passing', 'Expected', '1/3'): 'passes_into_final_third', ('Passing', 'Expected', 'PPA'): 'passes_into_penalty_area', ('Passing', 'Expected', 'CrsPA'): 'crosses_into_penalty_area', ('Passing', 'Expected', 'PrgP'): 'progressive_passes',
    ('Goal and Shot Creation', 'SCA', 'SCA'): 'sca', ('Goal and Shot Creation', 'SCA', 'SCA90'): 'sca_per90', ('Goal and Shot Creation', 'GCA', 'GCA'): 'gca', ('Goal and Shot Creation', 'GCA', 'GCA90'): 'gca_per90',
    ('Defensive Actions', 'Tackles', 'Tkl'): 'tackles', ('Defensive Actions', 'Tackles', 'TklW'): 'tackles_won', ('Defensive Actions', 'Challenges', 'Att'): 'challenges', ('Defensive Actions', 'Challenges', 'Lost'): 'challenges_lost',
    ('Defensive Actions', 'Blocks', 'Blocks'): 'blocks', ('Defensive Actions', 'Blocks', 'Sh'): 'blocked_shots', ('Defensive Actions', 'Blocks', 'Pass'): 'blocked_passes', ('Defensive Actions', 'Blocks', 'Int'): 'interceptions',
    ('Possession', 'Touches', 'Touches'): 'touches', ('Possession', 'Touches', 'Def Pen'): 'touches_def_pen_area', ('Possession', 'Touches', 'Def 3rd'): 'touches_def_3rd', ('Possession', 'Touches', 'Mid 3rd'): 'touches_mid_3rd', ('Possession', 'Touches', 'Att 3rd'):
```

```
        'touches_att_3rd', ('Possession', 'Touches', 'Att Pen'):
        'touches_att_pen_area',
297     ('Possession', 'Take-Ons', 'Att'): 'take_ons', ('Possession', 'Take-Ons',
        'Succ%'): 'take_ons_won_pct', ('Possession', 'Take-Ons', 'Tkld%'):
        'take_ons_tackled_pct',
298     ('Possession', 'Carries', 'Carries'): 'carries', ('Possession', 'Carries',
        'PrgDist'): 'carries_progressive_distance', ('Possession', 'Carries',
        'ProgC'): 'progressive_carries', ('Possession', 'Carries', '1/3'):
        'carries_into_final_third', ('Possession', 'Carries', 'CPA'):
        'carries_into_penalty_area', ('Possession', 'Carries', 'Mis'):
        'miscontrols', ('Possession', 'Carries', 'Dis'): 'dispossessed',
299     ('Possession', 'Receiving', 'Rec'): 'passes_received', ('Possession',
        'Receiving', 'PrgR'): 'progressive_passes_received',
300     ('Miscellaneous Stats', 'Performance', 'Fls'): 'fouls', ('Miscellaneous
        Stats', 'Performance', 'Fld'): 'fouled', ('Miscellaneous Stats',
        'Performance', 'Off'): 'offsides', ('Miscellaneous Stats',
        'Performance', 'Crs'): 'crosses', ('Miscellaneous Stats',
        'Performance', 'Recov'): 'ball_recoveries',
301     ('Miscellaneous Stats', 'Aerial Duels', 'Won'): 'aerials_won',
        ('Miscellaneous Stats', 'Aerial Duels', 'Lost'): 'aerials_lost',
        ('Miscellaneous Stats', 'Aerial Duels', 'Won%'): 'aerials_won_pct',
302 }
303 required_fbref_keys = set(USER_REQUESTED_STAT_MAPPING.values()) | {'player',
    'team', 'birth_year', 'minutes_90s'} # Add basic keys
304 print(f"\nTargeting {len(required_fbref_keys)} FBRef keys for scraping
    (user-requested + basic).")
305
306 urls = {
307     'standard': 'https://fbref.com/en/comps/9/stats/Premier-League-Stats',
        'shooting':
        'https://fbref.com/en/comps/9/shooting/Premier-League-Stats',
308     'passing': 'https://fbref.com/en/comps/9/passing/Premier-League-Stats',
        'gca': 'https://fbref.com/en/comps/9/gca/Premier-League-Stats',
309     'defense': 'https://fbref.com/en/comps/9/defense/Premier-League-Stats',
        'possession':
        'https://fbref.com/en/comps/9/possession/Premier-League-Stats',
310     'misc': 'https://fbref.com/en/comps/9/misc/Premier-League-Stats',
        'keepers': 'https://fbref.com/en/comps/9/keepers/Premier-League-Stats',
311 }
312 table_ids = {
313     'standard': 'stats_standard', 'shooting': 'stats_shooting', 'passing':
        'stats_passing', 'gca': 'stats_gca',
314     'defense': 'stats_defense', 'possession': 'stats_possession',
        'playingtime': 'stats_playing_time', # Keep ID if URL re-enabled
315     'misc': 'stats_misc', 'keepers': 'stats_keeper',
316 }
317
318 print("\nSetting up Selenium WebDriver...")
319 try:
320     options = Options()
321     # options.add_argument("--headless")
```

```
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    options.add_argument("--log-level=3") # Reduce browser logs
    options.add_argument("user-agent=Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko) Chrome/124.0.0.0 Safari/537.36")
    options.add_argument("--disable-blink-features=AutomationControlled")
    options.add_experimental_option("excludeSwitches", ["enable-automation"])
    options.add_experimental_option('useAutomationExtension', False)
    try:
        service = Service(ChromeDriverManager().install())
        driver = webdriver.Chrome(service=service, options=options)
        print("WebDriver using ChromeDriverManager.")
    except Exception as driver_manager_err:
        print(f"Warning: ChromeDriverManager failed ({driver_manager_err}).
            Trying default ChromeDriver from PATH...")
        driver = webdriver.Chrome(options=options) # Fallback to system PATH
        print("WebDriver using ChromeDriver from system PATH.")
    driver.execute_script("Object.defineProperty(navigator, 'webdriver', {get:
        () => undefined})") # Hide webdriver property
    print("WebDriver setup complete.")
except Exception as e:
    print(f"Critical error during WebDriver setup: {e}")
    sys.exit(1)

all_dfs = {}
MIN_MINUTES_PLAYED = 90
scraping_successful = False

print("\n--- Starting to scrape data from URLs ---")
for category, url in urls.items():
    table_id = table_ids.get(category)
    df_cat = scrape_fbref_table(driver, url, table_id=table_id,
        min_minutes=MIN_MINUTES_PLAYED, required_stats=required_fbref_keys)
    if df_cat is not None and not df_cat.empty:
        cols_to_keep = [col for col in df_cat.columns if col in
            required_fbref_keys]
        if cols_to_keep:
            all_dfs[category] = df_cat[cols_to_keep]
            print(f"--> Success: Fetched data for {category}
                ({all_dfs[category].shape[0]} players, {len(cols_to_keep)}
                stats)")
            scraping_successful = True
        else: print(f"--> Warning: {category} contained no required stats.")
    else: print(f"--> Warning: Fetching failed or no data for {category} from
        {url}")
    print("-" * 30)
driver.quit()

if not scraping_successful or not all_dfs:
    print("ERROR: No data successfully fetched. Cannot continue.")
    sys.exit(1)
```

```
365
366 print("\n--- Merging scraped DataFrames ---")
367 merged_df = None
368 df_keys_priority = ['standard', 'keepers'] + [k for k in all_dfs.keys() if k
    not in ['standard', 'keepers']]
369 for category in df_keys_priority:
370     if category not in all_dfs or all_dfs[category].empty: continue
371     df_cat = all_dfs[category]
372     if merged_df is None: merged_df = df_cat
373     else:
374         try:
375             merged_df = merged_df.merge(df_cat, left_index=True,
                right_index=True, how='outer', suffixes=(None, f'__{category}'))
376         except Exception as merge_error:
377             print(f"CRITICAL ERROR merging '{category}': {merge_error}") #
                Potentially log more details or stop
378     print(f" {'Started with' if merged_df is df_cat else 'Merged'}
        '{category}'. Current shape: {merged_df.shape if merged_df is not None
        else 'N/A'}")
379 if merged_df is None:
380      print("ERROR: No DataFrames merged. Cannot create result file.")
381      sys.exit(1)
382 print(f"\nInitial merge complete. Total unique Player/Team pairs:
    {len(merged_df)}")
383 merged_df = merged_df.reset_index().fillna('N/a')
384
385 print("\n--- Building final DataFrame based on user request ---")
386 final_df = pd.DataFrame({'Player': merged_df['Player'], 'Team':
    merged_df['Team']})
387 final_columns_structure = []
388 missing_stats_log = []
389 processed_fbref_keys_final = {'Player', 'Team'}
390 merged_df_columns_list = merged_df.columns.tolist()
391
392 def find_column_match(df_columns, base_key, suffix_marker='__'):
393     if base_key in df_columns: return base_key
394     suffixed_cols = [c for c in df_columns if isinstance(c, str) and
        c.startswith(base_key + suffix_marker)]
395     return suffixed_cols[0] if suffixed_cols else None
396
397 print(f"Processing {len(USER_REQUESTED_STAT_MAPPING)} requested stats...")
398 for col_tuple, base_key in USER_REQUESTED_STAT_MAPPING.items():
399     final_columns_structure.append(col_tuple)
400     matched_col = find_column_match(merged_df_columns_list, base_key)
401     if matched_col:
402         final_df[col_tuple] = merged_df[matched_col]
403         processed_fbref_keys_final.add(matched_col)
404         if matched_col != base_key: missing_stats_log.append(f"Used suffixed
            '{matched_col}' for {col_tuple} (orig: {base_key})")
405     else:
406          final_df[col_tuple] = 'N/a'
```

```
        missing_stats_log.append(f"Missing {col_tuple} (orig: {base_key}).")

print("\n--- Checks and Reports ---")
unused_original_columns = [col for col in merged_df_columns_list if col not in processed_fbref_keys_final]
if unused_original_columns: print(f"Info: {len(unused_original_columns)} unrequested columns dropped. (e.g., {', '.join(sorted(unused_original_columns)[:5])}{'...' if len(unused_original_columns) > 5 else ''})")
if missing_stats_log:
    print("Warning - Mapping issues:")
    for warning in sorted(list(set(missing_stats_log))): print(f" - {warning}")
else: print("All requested stats mapped successfully.")
print("------------------------------------------------------\n")

print("Creating column index for final DataFrame...")
try:
    multiindex_tuples = [('','','Player'), ('','','Team')] + final_columns_structure
    if len(multiindex_tuples) == final_df.shape[1]:
        final_df.columns = pd.MultiIndex.from_tuples(multiindex_tuples, names=['Category', 'Sub-Category', 'Statistic'])
        print("MultiIndex created.")
    else: raise ValueError(f"Column count mismatch for MultiIndex: DF has {final_df.shape[1]}, tuples {len(multiindex_tuples)}.")
except Exception as multiindex_error:
    print(f"Error creating MultiIndex: {multiindex_error}. Using flat column names as fallback.")
    flat_fallback_cols = ['Player', 'Team'] + ['_'.join(filter(None, map(str, tpl))) for tpl in final_columns_structure]
    final_df.columns = [f"{col}_{i}" if flat_fallback_cols.count(col) > 1 else col for i, col in enumerate(flat_fallback_cols)]

is_multiindex = isinstance(final_df.columns, pd.MultiIndex)
player_col_id = ('', '', 'Player') if is_multiindex else 'Player'
if player_col_id in final_df.columns:
    try:
        final_df = final_df.sort_values(by=player_col_id, ascending=True, key=lambda col: col.astype(str).str.lower(), na_position='last')
        print("Sorted DataFrame by Player name.")
    except Exception as e: print(f"Warning: Could not sort by Player ('{player_col_id}'): {e}.")
else: print(f"Warning: Player column '{player_col_id}' not found for sorting.")

print("\nReordering final columns...")
PRIORITY_COLS_TUPLE = [('', '', 'Player'), ('', '', 'Team'), ('', '', 'Nation'), ('', '', 'Position'), ('', '', 'Age')]
PRIORITY_COLS_FLAT = ['Player', 'Team', 'Nation', 'Position', 'Age']
priority_cols_definition = PRIORITY_COLS_TUPLE if is_multiindex else PRIORITY_COLS_FLAT
```

```
all_current_cols = final_df.columns.tolist()
priority_cols_present = [col for col in priority_cols_definition if col in all_current_cols]
other_cols = sorted([col for col in all_current_cols if col not in priority_cols_present])
final_column_order = priority_cols_present + other_cols
try:
    final_df = final_df[final_column_order]
    print("Column reordering successful.")
except Exception as e: print(f"Error reordering columns: {e}.")

print("\nPreparing to export final CSV file...")
final_df_export = final_df.copy()
if isinstance(final_df_export.columns, pd.MultiIndex):
    print("Flattening MultiIndex columns for CSV...")
    flat_columns = []
    processed_flat_names = set()
    for col_tuple in final_df_export.columns:
        parts = [str(c).strip().replace(' ', '_').replace('/', '_').replace('%', 'Pct').replace('+/-','_Net').replace('#','Num').replace('(','').replace(')','').replace('_') for c in col_tuple if str(c).strip()]
        base_flat_col = '_'.join(parts) if parts else f"col_{len(flat_columns)}"
        original_base = base_flat_col
        current_count = 1
        while base_flat_col in processed_flat_names:
            base_flat_col = f"{original_base}_{current_count}"; current_count += 1
        flat_columns.append(base_flat_col)
        processed_flat_names.add(base_flat_col)
    if len(flat_columns) == final_df_export.shape[1]: final_df_export.columns = flat_columns
    else:
        print(f"CRITICAL ERROR: Column count mismatch after flattening ({len(flat_columns)} vs {final_df_export.shape[1]}). Aborting save.")
        sys.exit(1)

print("Reordering flattened columns for export...")
PRIORITY_COLS_FLAT_FINAL = ['Player', 'Team', 'Nation', 'Position', 'Age']
id_cols_flat_final = [c for c in PRIORITY_COLS_FLAT_FINAL if c in final_df_export.columns]
other_cols_flat_final = sorted([c for c in final_df_export.columns if c not in id_cols_flat_final])
final_export_order_flat = id_cols_flat_final + other_cols_flat_final
try:
    final_df_export = final_df_export[final_export_order_flat]
    print("Flat column reordering for export successful.")
except Exception as e: print(f"Error reordering flat columns for export: {e}")

output_filename = 'results.csv'
```

```
print(f"\nSaving final results to {output_filename}...")
try:
    if final_df_export.empty or final_df_export.shape[1] == 0: print("Warning: Final DataFrame is empty or has no columns. Saving empty CSV.")
    missing_protected = [col for col in id_cols_flat_final if col not in final_df_export.columns]
    if missing_protected: print(f"CRITICAL WARNING: Basic ID columns lost before saving: {missing_protected}.")
    final_df_export.to_csv(output_filename, index=False, encoding='utf-8-sig')
    print(f"Successfully saved results to {output_filename}. Shape: {final_df_export.shape}")
    print(f"Final columns (first 25): {final_df_export.columns.tolist()[:25]}{'...' if len(final_df_export.columns) > 25 else ''}")
except Exception as e:
    print(f"ERROR saving CSV '{output_filename}': {e}\nTraceback: {traceback.format_exc()}")
print("\n--- Script complete ---")
```

# Giải thích đoạn code

## Thư viện cần cài đặt

Các thư viện Python cần thiết cho đoạn code bao gồm:

- `time`: Thư viện chuẩn để làm việc với thời gian.
- `pandas`: Thư viện để xử lý và phân tích dữ liệu.
- `collections`: Thư viện chuẩn, sử dụng `Counter`.
- `selenium`: Thư viện để tự động hóa trình duyệt web.
- `webdriver_manager`: Thư viện để quản lý driver cho Selenium.
- `bs4` (BeautifulSoup): Thư viện để phân tích cú pháp HTML và XML.
- `sys`: Thư viện chuẩn để làm việc với hệ thống.
- `traceback`: Thư viện chuẩn để in ra thông tin lỗi.

Để cài đặt các thư viện không chuẩn, sử dụng lệnh:

```
pip install pandas selenium webdriver-manager beautifulsoup4
```

## Cách code vận hành chung

Đoạn code được thiết kế để thu thập dữ liệu bóng đá từ trang web `FBRef.com`, cụ thể là từ các bảng thống kê của Premier League. Nó sử dụng `Selenium` để tự động hóa trình duyệt web, `BeautifulSoup` để phân tích HTML, và `pandas` để xử lý dữ liệu.

Quy trình chung của code như sau:

1. **Thiệt lập Selenium WebDriver**: Khởi tạo trình duyệt Chrome với các tùy chọn để tránh bị phát hiện là bot.
2. **Thu thập dữ liệu từ các URL**: Sử dụng hàm `scrape_fbref_table` để lấy dữ liệu từ các bảng thống kê khác nhau.
3. **Xử lý và làm sạch dữ liệu**: Sử dụng các hàm như `safe_get_text`, `get_nationality`, và `calculate_age`.
4. **Hợp nhất dữ liệu**: Hợp nhất các DataFrame từ các bảng thành một DataFrame duy nhất.
5. **Tạo DataFrame cuối cùng**: Tạo DataFrame theo yêu cầu người dùng dựa trên `USER_REQUESTED_STAT_MAPPING`.
6. **Xuất dữ liệu ra file CSV**: Lưu DataFrame vào file `results.csv`.

## Giải thích từng hàm chính

`safe_get_text(element, default='N/a')`

- **Mục đích**: Trích xuất văn bản từ phần tử HTML một cách an toàn.
- **Cách hoạt động**:
  - Nếu `element` là `None`, trả về `default`.
  - Lấy văn bản từ `element` và loại bỏ khoảng trắng thừa.
  - Nếu văn bản rỗng, trả về `default`.

`get_nationality(td_element)`

- **Mục đích**: Trích xuất mã quốc tịch từ ô HTML.
- **Cách hoạt động**:
  - Nếu `td_element` là `None`, trả về 'N/a'.
  - Lấy tất cả chuỗi văn bản từ `td_element`.
  - Tìm mã quốc tịch 3 chữ cái in hoa.
  - Nếu không tìm thấy, kiểm tra văn bản trong liên kết (nếu có).
  - Nếu vẫn không tìm thấy, lấy phần cuối nếu là mã quốc tịch (2-4 chữ cái).

`calculate_age(age_or_birth_str, current_year=None)`

- **Mục đích**: Tính tuổi từ chuỗi chứa tuổi hoặc năm sinh.
- **Cách hoạt động**:
  - Nếu chuỗi là số từ 14 đến 50, trả về tuổi đó.
  - Nếu chuỗi chứa năm sinh (ví dụ: "1990-01-01"), tính tuổi bằng cách trừ năm sinh từ năm hiện tại.
  - Xử lý các định dạng như "YYYY-MM-DD" hoặc "Month Day, YYYY".
  - Nếu không xử lý được, trả về 'N/a'.

`scrape_fbref_table(driver, url, table_id=None, required_stats=None, min_minutes=90)`

- **Mục đích**: Thu thập dữ liệu từ bảng trên `FBRef.com`.
- **Cách hoạt động**:
  - Mở URL bằng Selenium.
  - Đợi bảng xuất hiện với `table_id` hoặc class `stats_table`.
  - Phân tích HTML bằng BeautifulSoup.
  - Trích xuất dữ liệu từ các hàng, lọc người chơi có số phút dưới `min_minutes`.
  - Tạo DataFrame từ dữ liệu thu thập.

`find_column_match(df_columns, base_key, suffix_marker='__')`

- **Mục đích**: Tìm cột trong DataFrame khớp với `base_key`.
- **Cách hoạt động**:
  - Nếu `base_key` tồn tại trong danh sách cột, trả về nó.
  - Nếu không, tìm cột bắt đầu bằng `base_key` + `suffix_marker`.

## Kết quả đầu ra của file CSV

Kết quả của đoạn code là một file CSV tên `results.csv`, chứa dữ liệu thống kê bóng đá từ các bảng trên trang `FBRef.com` (Premier League). Dưới đây là mô tả chi tiết về cấu trúc và nội dung của file:

- **Cấu trúc cột**:
  - File chứa các cột được định nghĩa trong `USER_REQUESTED_STAT_MAPPING`, bao gồm:
    - *Thông tin cơ bản*: `Player` (tên cầu thủ), `Team` (đội bóng), `Nation` (quốc tịch, mã 2-4 chữ cái in hoa), `Position` (vị trí thi đấu), `Age` (tuổi của cầu thủ).
    - *Thống kê hiệu suất*: Số trận (`MP`), số lần ra sân từ đầu (`Starts`), số phút thi đấu (`Min`), số bàn thắng (`Gls`), số kiến tạo (`Ast`), xG (`xG`), xAG (`xAG`), số thẻ vàng (`CrdY`), số thẻ đỏ (`CrdR`), v.v.
    - *Thống kê chi tiết*: Tỷ lệ sút trúng đích (`SoT%`), số đường chuyền hoàn thành (`Cmp`), tỷ lệ chuyền bóng chính xác (`Cmp%`), số lần tắc bóng (`Tkl`), số lần đánh chặn (`Int`), số lần chạm bóng (`Touches`), số lần rê bóng thành công (`Succ%`), v.v.
    - *Thống kê thủ môn* (nếu có): Tỷ lệ cứu thua (`Save%`), tỷ lệ giữ sạch lưới (`CS%`), tỷ lệ cứu penalty (`Save%`), v.v.
  - Các cột được sắp xếp với thứ tự ưu tiên: `Player`, `Team`, `Nation`, `Position`, `Age`, sau đó là các cột thống kê khác theo thứ tự alphabet.
- **Định dạng dữ liệu**:

* Mỗi hàng đại diện cho một cầu thủ duy nhất trong một đội bóng, với các giá trị thống kê tương ứng.
* Giá trị thiếu được điền bằng 'N/a'.
* Dữ liệu được sắp xếp theo cột `Player` (theo thứ tự bảng chữ cái, không phân biệt hoa thường).
* File sử dụng mã hóa `utf-8-sig` để hỗ trợ các ký tự đặc biệt (như tên cầu thủ có dấu).

– **Ví dụ nội dung** (minh họa, dữ liệu thực tế phụ thuộc vào `FBRef.com`):

```
Player,Team,Nation,Position,Age,MP,Starts,Min,Gls,Ast,xG,xAG,...
Mohamed Salah,Liverpool,EGY,FW,32,30,28,2500,15,10,12.5,8.0,...
Erling Haaland,Manchester City,NOR,FW,24,32,30,2700,25,5,20.0,3.5,...
```

– **Đặc điểm bổ sung**:

* Các cột được đặt tên phẳng (flat) để phù hợp với định dạng CSV, với các ký tự đặc biệt (như `%`, `/`, `-`) được thay thế bằng các chuỗi hợp lệ (ví dụ: `%` thành `Pct`, `-` thành `_`).
* File không chứa cột chỉ mục (`index=False`), đảm bảo đầu ra là bảng dữ liệu thuần túy.
* Nếu không thu thập được dữ liệu (do lỗi mạng, trang web thay đổi cấu trúc, v.v.), file có thể rỗng hoặc chỉ chứa tiêu đề cột.

# Bài 2:

# Mã Nguồn Đầy Đủ

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import os
import sys
import traceback
import re

# --- Configuration ---
INPUT_CSV = 'results.csv'
OUTPUT_TOP_BOTTOM = 'top_3.txt'
OUTPUT_STATS_SUMMARY = 'results2.csv'
OUTPUT_HISTOGRAM_DIR = 'histograms'
HIST_SUBDIR_ALL = 'all_players'
HIST_SUBDIR_TEAMS = 'by_team'
OUTPUT_HIGHEST_SCORING_TEAMS = 'highest_scoring_teams.txt' # For highest
    scoring teams output

ID_COLS = ['Player', 'Team', 'Nation', 'Position', 'Age']

# Patterns for selecting specific stats for histograms
HISTOGRAM_OFFENSIVE_PATTERNS = [
```

```
    'gls', 'goal', 'sh', 'shot', 'sot', 'xg', 'npxg', 'xa', 'assist', 'keypass', 'kp',
    'sca', 'gca', 'att_pen', 'crspa', 'succ_dribbles', 'prog_passes_rec', 'touches_att_pen',
    'progcarry', 'progpass' # Added for progressive carries/passes if named as such
]
HISTOGRAM_DEFENSIVE_PATTERNS = [
    'tkl', 'tackle', 'tklw', 'int', 'interception', 'block', 'clr', 'clearance',
    'sav', 'save', 'cs', 'clean_sheet', 'ga', 'goals_against', 'err', # 'ga' and 'goals_against' for goals against
    'crdy', 'crdr', 'card', 'foul', 'aerialswon', 'pkcon', 'pressure', 'recover'
]


# --- Helper Functions ---
def clean_numeric_column(series):
    series_str = series.astype(str)
    series_cleaned = series_str.str.replace('%', '', regex=False)
    series_cleaned = series_cleaned.str.replace(',', '', regex=False)
    series_numeric = pd.to_numeric(series_cleaned, errors='coerce')
    return series_numeric

def get_numeric_columns(df, exclude_cols):
    numeric_cols = []
    potential_cols = [col for col in df.columns if col not in exclude_cols]
    print(f" Potentially analyzing {len(potential_cols)} columns (excluding: {', '.join(exclude_cols)})")
    original_dtypes = df[potential_cols].dtypes

    for col in potential_cols:
        if pd.api.types.is_numeric_dtype(original_dtypes[col]):
            numeric_cols.append(col)
            continue
        try:
            # Attempt to coerce to numeric, then check ratio of valid numbers
            coerced = pd.to_numeric(df[col].astype(str).str.replace(',', ''), errors='coerce')
            valid_ratio = coerced.notna().sum() / len(coerced) if len(coerced) > 0 else 0
            if valid_ratio > 0.1: # Consider a column numeric if more than 10% can be converted
                numeric_cols.append(col)
        except Exception as e:
            print(f" Skipping column '{col}' due to error during numeric check: {e}", file=sys.stderr)
    return sorted(list(set(numeric_cols)))

def format_player_list(series):
```

```
    return [f"{player} ({score})" for player, score in series.items()]

# --- Main Analysis Logic ---
if __name__ == "__main__":
    print(f"Loading data from {INPUT_CSV}...")
    try:
        df = pd.read_csv(INPUT_CSV)
        print(f"Data loaded successfully. Shape: {df.shape}")
        if df.empty:
            print(f"Error: {INPUT_CSV} is empty. Cannot proceed.",
                file=sys.stderr)
            sys.exit(1)
        print(f"Original columns ({len(df.columns)}): {',
            '.join(df.columns[:min(10, len(df.columns))])}...") # Show first 10
        if 'Player' in df.columns: df['Player'] = df['Player'].astype(str)
        if 'Team' in df.columns: df['Team'] = df['Team'].astype(str)
    except FileNotFoundError:
        print(f"Error: {INPUT_CSV} not found. Please ensure the file exists.",
            file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"Error loading {INPUT_CSV}: {e}", file=sys.stderr)
        print(traceback.format_exc(), file=sys.stderr)
        sys.exit(1)

    print("\nApplying cleaning to potential numeric columns...")
    df_cleaned = df.copy()
    potential_numeric_cols_for_cleaning = [col for col in df.columns if col
        not in ID_COLS]
    cleaned_count = 0
    for col in potential_numeric_cols_for_cleaning:
        if col in df_cleaned.columns and not
            pd.api.types.is_numeric_dtype(df_cleaned[col]):
            df_cleaned[col] = clean_numeric_column(df_cleaned[col])
            cleaned_count += 1
    print(f"Attempted cleaning on {cleaned_count} non-numeric columns
        (excluding ID cols).")

    print("\nIdentifying numeric columns for analysis after cleaning...")
    NON_STAT_COLS = list(ID_COLS)
    # Add specific playing time columns from your CSV structure to
        NON_STAT_COLS
    # Common playing time columns that might appear from Problem1.py
    # (e.g., Playing_Time_Min, Playing_Time_MP, Playing_Time_Starts)
    # Adjust these based on the actual output of Problem1.py if needed
    generated_playing_time_cols = ['Playing_Time_Min', 'Playing_Time_MP',
        'Playing_Time_Starts', 'Min', 'MP', 'Starts']
    for pt_col in generated_playing_time_cols:
        if pt_col in df_cleaned.columns:
            NON_STAT_COLS.append(pt_col)
    NON_STAT_COLS = sorted(list(set(NON_STAT_COLS))) # Ensure unique
```

```
105
106     stat_cols = get_numeric_columns(df_cleaned, NON_STAT_COLS)
107
108     if not stat_cols:
109         print("\nError: No numeric statistic columns identified after
                cleaning.", file=sys.stderr)
110         print("Please check the input CSV structure and the
                cleaning/identification logic.")
111         print(f" Columns excluded as non-stats: {NON_STAT_COLS}")
112         sys.exit(1)
113
114     print(f"\nIdentified {len(stat_cols)} numeric statistic columns for
            analysis.")
115     # Example: print first 5 stat_cols
116     print(f" Sample stats: {', '.join(stat_cols[:min(5, len(stat_cols))])}...")
117     # GK stats identification (remains useful)
118     potential_gk_cols_in_stats = [c for c in stat_cols if 'gk' in c.lower() or
            'goal' in c.lower() or 'sav' in c.lower() or 'pk' in c.lower() or 'ga'
            in c.lower() or 'cs' in c.lower()]
119     if potential_gk_cols_in_stats:
120         print(f" Potential GK stats identified: {',
                '.join(sorted(potential_gk_cols_in_stats))}")
121     else:
122         print(" No columns matching typical Goalkeeping patterns found in
                identified stats.")
123     df_numeric = df_cleaned
124
125     print(f"\nCalculating Top/Bottom 3 players per statistic ->
            {OUTPUT_TOP_BOTTOM}")
126     try:
127         with open(OUTPUT_TOP_BOTTOM, 'w', encoding='utf-8') as f:
128             f.write("Top and Bottom 3 Players per Statistic\n")
129             f.write("====================================\n\n")
130             for col in stat_cols:
131                 if 'Player' not in df_numeric.columns:
132                     f.write(f"--- {col} ---\n")
133                     f.write("Error: 'Player' column not found. Cannot determine
                            top/bottom players.\n\n")
134                     continue
135
136                 # Ensure the column actually exists in df_numeric before
                        proceeding
137                 if col not in df_numeric.columns:
138                     f.write(f"--- {col} ---\n")
139                     f.write(f"Error: Column '{col}' not found in DataFrame for
                            Top/Bottom analysis.\n\n")
140                     continue
141
142                 stat_df = df_numeric[['Player', col]].copy()
143                 stat_df[col] = pd.to_numeric(stat_df[col], errors='coerce') #
                        Coerce to numeric
```

```
144                 stat_df.dropna(subset=[col], inplace=True)
145
146                 if stat_df.empty:
147                     f.write(f"--- {col} ---\n")
148                     f.write(f"No valid numeric data for this statistic
                            ('{col}').\n\n")
149                     continue
150                 try:
151                     highest = stat_df.sort_values(by=col,
                            ascending=False).set_index('Player')[col]
152                     lowest = stat_df.sort_values(by=col,
                            ascending=True).set_index('Player')[col]
153                     top_3 = highest.head(3)
154                     bottom_3 = lowest.head(3)
155                     f.write(f"--- {col} ---\n")
156                     f.write("Top 3:\n")
157                     if not top_3.empty:
158                         for player, score in top_3.items():
159                             score_str = f"{score:.2f}" if pd.notna(score) else
                                    "N/A"
160                             f.write(f" - {player}: {score_str}\n")
161                     else:
162                         f.write(" (No players found for Top 3)\n")
163                     f.write("\nBottom 3:\n")
164                     if not bottom_3.empty:
165                         for player, score in bottom_3.items():
166                             score_str = f"{score:.2f}" if pd.notna(score) else
                                    "N/A"
167                             f.write(f" - {player}: {score_str}\n")
168                     else:
169                         f.write(" (No players found for Bottom 3)\n")
170                     f.write("\n----------------------------------------\n\n")
171                 except Exception as sort_err:
172                     f.write(f"--- {col} ---\n")
173                     f.write(f"Error sorting data for statistic '{col}':
                            {sort_err}\n\n")
174                     f.write("----------------------------------------\n\n")
175         print("Top/Bottom 3 players saved.")
176     except Exception as e:
177         print(f"Error during Task 1 (Top/Bottom 3): {e}", file=sys.stderr)
178         print(traceback.format_exc(), file=sys.stderr)
179
180     print(f"\nCalculating Median, Mean, Std Dev per statistic ->
            {OUTPUT_STATS_SUMMARY}")
181     results_data = []
182     try:
183         if not stat_cols:
184             print("Warning: No numeric stats columns identified for Task 2.",
                    file=sys.stderr)
185         else:
186             # Ensure only existing stat_cols are used for aggregation
```

```
            valid_stat_cols_for_agg = [sc for sc in stat_cols if sc in
                df_numeric.columns]
            if not valid_stat_cols_for_agg:
                print("Warning: None of the identified stat_cols exist in the
                    DataFrame for aggregation.", file=sys.stderr)
            else:
                global_agg = df_numeric[valid_stat_cols_for_agg].agg(['median',
                    'mean', 'std'])
                for stat in valid_stat_cols_for_agg:
                    if stat in global_agg.columns:
                        results_data.append({
                            'Team': 'all',
                            'Statistic': stat,
                            'Median': global_agg.loc['median', stat],
                            'Mean': global_agg.loc['mean', stat],
                            'Std': global_agg.loc['std', stat]
                        })
                    else:
                        print(f"Warning: Statistic '{stat}' not found in global
                            aggregation results.", file=sys.stderr)

        if 'Team' in df_numeric.columns:
            valid_teams_df =
                df_numeric[df_numeric['Team'].astype(str).str.lower() != 'all']
            if not valid_teams_df.empty:
                teams_for_grouping = valid_teams_df['Team'].unique()
                if len(teams_for_grouping) > 0 and valid_stat_cols_for_agg: #
                    Also check if there are stats to group by
                    grouped =
                        valid_teams_df.groupby('Team')[valid_stat_cols_for_agg]
                    if not grouped.groups:
                        print("Warning: Grouping by 'Team' resulted in empty
                            groups.", file=sys.stderr)
                    else:
                        try:
                            team_agg = grouped.agg(['median', 'mean', 'std'])
                            if team_agg.empty:
                                print("Warning: Aggregation per team produced
                                    empty results.", file=sys.stderr)
                            else:
                                for team_name_idx in team_agg.index:
                                    for stat_col_name_agg in
                                        valid_stat_cols_for_agg:
                                        median_key = (stat_col_name_agg, 'median')
                                        mean_key = (stat_col_name_agg, 'mean')
                                        std_key = (stat_col_name_agg, 'std')
                                        if median_key in team_agg.columns and
                                            mean_key in team_agg.columns and
                                            std_key in team_agg.columns:
                                            results_data.append({
                                                'Team': team_name_idx,
```

```
                                        'Statistic': stat_col_name_agg,
                                        'Median':
                                            team_agg.loc[team_name_idx,
                                            median_key],
                                        'Mean': team_agg.loc[team_name_idx,
                                            mean_key],
                                        'Std': team_agg.loc[team_name_idx,
                                            std_key]
                                    })
                    except Exception as group_agg_e:
                        print(f"Error during per-team aggregation:
                            {group_agg_e}", file=sys.stderr)
                        print(traceback.format_exc(), file=sys.stderr)
            elif not valid_stat_cols_for_agg:
                print("Warning: No valid stat columns to perform per-team
                    aggregation.", file=sys.stderr)
            else:
                print("Warning: No unique teams found for per-team
                    statistics (excluding 'all').", file=sys.stderr)
        else:
            print("Warning: DataFrame became empty after filtering out
                'all' team. No per-team stats.", file=sys.stderr)
    else:
        print("Warning: 'Team' column not found. Cannot calculate per-team
            statistics.", file=sys.stderr)

    if not results_data:
        print("Error: No statistics could be calculated for Task 2.",
            file=sys.stderr)
    else:
        summary_long_df = pd.DataFrame(results_data)
        summary_long_df.fillna(value=np.nan, inplace=True)
        if summary_long_df.empty or not {'Team', 'Statistic', 'Median',
            'Mean', 'Std'}.issubset(summary_long_df.columns):
            print("Error: Cannot create pivot table due to missing columns
                or empty data frame after aggregation.", file=sys.stderr)
        else:
            try:
                summary_pivot = summary_long_df.pivot_table(
                    index='Team', columns='Statistic', values=['Median',
                        'Mean', 'Std']
                )
                if isinstance(summary_pivot.columns, pd.MultiIndex):
                    summary_pivot.columns =
                        summary_pivot.columns.swaplevel(0, 1)
                    metric_order = pd.CategoricalDtype(['Median', 'Mean',
                        'Std'], ordered=True)
                    summary_pivot.sort_index(axis=1, level=0, inplace=True)
                    summary_pivot.sort_index(axis=1, level=1, key=lambda x:
                        x.astype(metric_order), inplace=True)
```

```
                    summary_pivot.columns = [f"{metric} of {stat}" for stat,
                        metric in summary_pivot.columns]
                summary_pivot = summary_pivot.reset_index()
                if 'all' in summary_pivot['Team'].values:
                    all_row = summary_pivot[summary_pivot['Team'] == 'all']
                    other_rows = summary_pivot[summary_pivot['Team'] !=
                        'all'].sort_values(by='Team')
                    summary_pivot = pd.concat([all_row, other_rows],
                        ignore_index=True)
                summary_pivot.to_csv(OUTPUT_STATS_SUMMARY, index=False,
                    encoding='utf-8-sig', float_format='%.3f')
                print(f"Median/Mean/Std Dev summary saved to
                    {OUTPUT_STATS_SUMMARY}")
            except Exception as pivot_e:
                print(f"Error during pivoting or formatting results for Task
                    2: {pivot_e}", file=sys.stderr)
                print(traceback.format_exc(), file=sys.stderr)
except Exception as e:
    print(f"Error during Task 2 (Median/Mean/Std Dev): {e}",
        file=sys.stderr)
    print(traceback.format_exc(), file=sys.stderr)

# --- Task: Identify teams with the highest average score per statistic
    (Requirement 1) ---
print(f"\nIdentifying teams with the highest average score per statistic
    -> {OUTPUT_HIGHEST_SCORING_TEAMS}")
highest_scoring_teams_dict = {} # Renamed to avoid conflict
if 'Team' in df_numeric.columns and stat_cols: # Check if stat_cols is not
    empty
    try:
        valid_teams_df_for_means =
            df_numeric[df_numeric['Team'].astype(str).str.lower() !=
            'all'].copy()
        # Ensure only existing stat_cols are used
        existing_stat_cols_for_means = [sc for sc in stat_cols if sc in
            valid_teams_df_for_means.columns]

        if not valid_teams_df_for_means.empty and
            existing_stat_cols_for_means:
            for col in existing_stat_cols_for_means: # Ensure numeric type
                for mean calculation
                valid_teams_df_for_means[col] =
                    pd.to_numeric(valid_teams_df_for_means[col],
                    errors='coerce')

            team_means =
                valid_teams_df_for_means.groupby('Team')[existing_stat_cols_for_means].m

            if not team_means.empty:
                with open(OUTPUT_HIGHEST_SCORING_TEAMS, 'w',
                    encoding='utf-8') as f_highest:
```

```
                    f_highest.write("Team with Highest Average Score per Statistic\n")
                    f_highest.write("==========================================\n\n")
                    for col in existing_stat_cols_for_means:
                        if col in team_means.columns and team_means[col].notna().any():
                            try:
                                best_team_idx = team_means[col].idxmax()
                                highest_score_val = team_means[col].max()
                                highest_scoring_teams_dict[col] = (best_team_idx, highest_score_val)
                                f_highest.write(f"- Highest Avg {col}: {best_team_idx} ({highest_score_val:.2f})\n")
                                # print(f"- Highest Avg {col}: {best_team_idx} ({highest_score_val:.2f})") # Optional console print
                            except ValueError:
                                f_highest.write(f"- Highest Avg {col}: N/A (all values NaN or empty after grouping)\n")
                            except Exception as idxmax_e:
                                f_highest.write(f"- Highest Avg {col}: Error ({idxmax_e})\n")
                        else:
                            f_highest.write(f"- Highest Avg {col}: N/A (column data insufficient or all NaN)\n")
                print(f"Highest scoring team data saved to {OUTPUT_HIGHEST_SCORING_TEAMS}")
            else:
                print("Warning: Calculating team means resulted in an empty DataFrame.", file=sys.stderr)
        elif not existing_stat_cols_for_means:
             print("Warning: No valid statistic columns found in DataFrame to calculate team means.", file=sys.stderr)
        else: # valid_teams_df_for_means is empty
            print("Warning: No valid team data (excluding 'all') to calculate highest scores.", file=sys.stderr)
    except Exception as e:
        print(f"Error during Highest Team Scores task: {e}", file=sys.stderr)
        print(traceback.format_exc(), file=sys.stderr)
elif not 'Team' in df_numeric.columns:
    print("Warning: 'Team' column not found. Cannot perform Highest Team Scores task.", file=sys.stderr)
else: # stat_cols is empty
    print("Warning: No numeric statistics identified. Cannot perform Highest Team Scores task.", file=sys.stderr)


```

```
    # --- Filter stats for histograms (Requirement 2) ---
    print("\nSelecting Offensive and Defensive statistics for histogram generation...")
    stats_for_histograms = []
    if stat_cols: # Ensure stat_cols is not empty
        all_hist_patterns = HISTOGRAM_OFFENSIVE_PATTERNS + HISTOGRAM_DEFENSIVE_PATTERNS
        for col in stat_cols:
            if col not in df_numeric.columns: continue # Skip if col somehow isn't in df
            col_lower = col.lower()
            if any(pattern in col_lower for pattern in all_hist_patterns):
                stats_for_histograms.append(col)

    if not stats_for_histograms:
        print("Warning: No offensive or defensive statistics identified for histogram plotting based on current patterns.", file=sys.stderr)
    else:
        print(f"Identified {len(stats_for_histograms)} offensive/defensive stats for histograms: {', '.join(stats_for_histograms[:min(5, len(stats_for_histograms))])}...")


    print(f"\nGenerating histograms for Offensive/Defensive Stats -> {OUTPUT_HISTOGRAM_DIR}/")
    hist_path_all = os.path.join(OUTPUT_HISTOGRAM_DIR, HIST_SUBDIR_ALL)
    hist_path_teams = os.path.join(OUTPUT_HISTOGRAM_DIR, HIST_SUBDIR_TEAMS)
    try:
        os.makedirs(hist_path_all, exist_ok=True)
        os.makedirs(hist_path_teams, exist_ok=True)
    except OSError as e:
        print(f"Error creating histogram directories: {e}", file=sys.stderr)
        # Decide if to exit or continue: sys.exit(1) or pass

    plot_errors_all = 0
    plots_generated_all = 0
    plot_errors_teams = 0
    plots_generated_teams = 0

    for col in stats_for_histograms: # Use the filtered list
        try:
            data_to_plot_all = df_numeric[col].dropna() # Ensure col exists
            if data_to_plot_all.empty or not pd.api.types.is_numeric_dtype(data_to_plot_all):
                pass
            else:
                plt.figure(figsize=(10, 6))
                plt.hist(data_to_plot_all, bins=20, edgecolor='black', color='skyblue')
                plt.title(f'Distribution of {col} (All Players)')
                plt.xlabel(col)
```

```
            plt.ylabel('Frequency (Number of Players)')
            plt.grid(axis='y', alpha=0.75)
            safe_col_name = "".join(c if c.isalnum() else "_" for c in col)
            plot_filename_all = os.path.join(hist_path_all,
                f'hist_all_{safe_col_name}.png')
            plt.savefig(plot_filename_all)
            plt.close()
            plots_generated_all += 1
        except Exception as e:
            plot_errors_all += 1
            print(f"Error generating histogram for {col} (All Players): {e}",
                file=sys.stderr)
            plt.close()

        if 'Team' in df_numeric.columns:
            # Ensure team names are strings for filtering and file naming
            teams_list = df_numeric[df_numeric['Team'].astype(str).str.lower()
                != 'all']['Team'].astype(str).dropna().unique()
            if len(teams_list) == 0:
                continue
            for team_name_str in teams_list:
                try:
                    team_data = df_numeric[df_numeric['Team'] ==
                        team_name_str][col].dropna()
                    if team_data.empty or not
                        pd.api.types.is_numeric_dtype(team_data):
                        continue
                    plt.figure(figsize=(8, 5))
                    plt.hist(team_data, bins=15, edgecolor='black',
                        color='lightcoral')
                    plt.title(f'Distribution of {col} for {team_name_str}',
                        fontsize=10)
                    plt.xlabel(col, fontsize=9)
                    plt.ylabel('Frequency', fontsize=9)
                    plt.xticks(fontsize=8); plt.yticks(fontsize=8)
                    plt.grid(axis='y', alpha=0.6)
                    safe_col_name = "".join(c if c.isalnum() else "_" for c in
                        col)
                    safe_team_name = "".join(c if c.isalnum() else "_" for c in
                        team_name_str) # team_name_str is already string
                    plot_filename_team = os.path.join(hist_path_teams,
                        f'hist_{safe_team_name}_{safe_col_name}.png')
                    plt.savefig(plot_filename_team)
                    plt.close()
                    plots_generated_teams += 1
                except Exception as e:
                    plot_errors_teams += 1
                    print(f"Error generating histogram for {col} -
                        {team_name_str}: {e}", file=sys.stderr)
                    plt.close()

```

```
406     print(f"\nHistograms generation summary (Offensive/Defensive Stats):")
407     print(f" - All Players: {plots_generated_all} successful,
            {plot_errors_all} errors.")
408     if 'Team' in df_numeric.columns:
409         print(f" - Per Team: {plots_generated_teams} successful,
                {plot_errors_teams} errors.")
410
411     # --- Best Performing Team Analysis (using highest_scoring_teams_dict) ---
412     print("\n--- Best Performing Team Analysis (Based on Average Stats) ---")
413     analysis_text = "Based on the average statistics per team:\n"
414     team_mentions_high = {}
415     team_mentions_low = {}
416
417     # Define patterns for interpreting stats (can be reused or adjusted)
418     # These are used for the textual analysis part.
419     LOWER_IS_BETTER_PATTERNS = ['ga', 'goals_against', 'offside', 'fls',
            'foul', 'lost', 'crd', 'card', 'pkcon', 'err_leading_to_shot'] # Added
            more specific
420     KEY_OFFENSIVE_PATTERNS_TEXT = ['gls', 'goal', 'xg', 'sot', 'sca', 'gca',
            'att_pen', 'shot', 'assist', 'key_pass', 'prog_pass_rec']
421     KEY_DEFENSIVE_PATTERNS_TEXT = ['tklw', 'tackles_won', 'int',
            'interception', 'block', 'clr', 'clearance', 'sav', 'save', 'cs',
            'clean_sheet', 'aerial_won']
422     KEY_POSSESSION_PATTERNS_TEXT = ['cmp_pct', 'pass_accuracy', 'prgp',
            'progressive_pass', 'prgc', 'progressive_carr', 'touch', 'progression',
            'prog']
423
424
425     if highest_scoring_teams_dict:
426         current_team_means_for_analysis = pd.DataFrame() # Initialize
427         if 'Team' in df_numeric.columns and stat_cols:
428             temp_df = df_numeric[df_numeric['Team'].astype(str).str.lower() !=
                    'all'].copy()
429             valid_cols = [c for c in stat_cols if c in temp_df.columns]
430             if valid_cols:
431                 for c in valid_cols: temp_df[c] = pd.to_numeric(temp_df[c],
                        errors='coerce')
432                 current_team_means_for_analysis =
                        temp_df.groupby('Team')[valid_cols].mean()
433
434         for stat, (team, score) in highest_scoring_teams_dict.items():
435             stat_lower = stat.lower()
436             is_lower_better_stat = any(pattern in stat_lower for pattern in
                    LOWER_IS_BETTER_PATTERNS)
437
438             if is_lower_better_stat:
439                 if not current_team_means_for_analysis.empty and stat in
                        current_team_means_for_analysis.columns and
                        current_team_means_for_analysis[stat].notna().any():
440                     try:
```

```
                    min_team_idx =
                        current_team_means_for_analysis[stat].idxmin()
                    team_mentions_low[min_team_idx] =
                        team_mentions_low.get(min_team_idx, 0) + 1
                except ValueError: pass # All NaN for this stat
                except Exception as min_err: print(f" (Error determining min
                    for {stat}: {min_err})")
            else:
                team_mentions_high[team] = team_mentions_high.get(team, 0) + 1

        most_mentioned_high = sorted(team_mentions_high.items(), key=lambda
            item: item[1], reverse=True)
        most_mentioned_low = sorted(team_mentions_low.items(), key=lambda item:
            item[1], reverse=True)

        if most_mentioned_high:
            analysis_text += f"- '{most_mentioned_high[0][0]}' leads
                {most_mentioned_high[0][1]} 'higher-is-better' stats.\n"
        if most_mentioned_low:
            analysis_text += f"- '{most_mentioned_low[0][0]}' leads
                {most_mentioned_low[0][1]} 'lower-is-better' stats.\n"

        def get_leaders_text_analysis(patterns, means_df, lower_better_def):
            leaders = {} # Store as {team: count_of_leading_stats}
            if means_df.empty: return {"N/A"}
            for stat_col in means_df.columns:
                stat_col_lower = stat_col.lower()
                if any(p in stat_col_lower for p in patterns):
                    if means_df[stat_col].notna().any():
                        try:
                            best_team = means_df[stat_col].idxmin() if any(lb in
                                stat_col_lower for lb in lower_better_def) else
                                means_df[stat_col].idxmax()
                            leaders[best_team] = leaders.get(best_team, 0) + 1
                        except ValueError: pass # All NaN
            # Return teams sorted by how many relevant stats they lead
            sorted_leaders = sorted(leaders.items(), key=lambda x:x[1],
                reverse=True)
            return {team for team, count in sorted_leaders[:3]} if
                sorted_leaders else {"N/A"} # Top 3 or N/A


        if not current_team_means_for_analysis.empty:
            off_leaders =
                get_leaders_text_analysis(KEY_OFFENSIVE_PATTERNS_TEXT,
                current_team_means_for_analysis, LOWER_IS_BETTER_PATTERNS)
            def_leaders = get_leaders_text_analysis(KEY_DEFENSIVE_PATTERNS_TEXT
                + LOWER_IS_BETTER_PATTERNS, current_team_means_for_analysis,
                LOWER_IS_BETTER_PATTERNS) # include lower_is_better in def
                patterns
```

```
            poss_leaders =
                get_leaders_text_analysis(KEY_POSSESSION_PATTERNS_TEXT,
                current_team_means_for_analysis, LOWER_IS_BETTER_PATTERNS)
            analysis_text += f"- Offensive Leaders (top teams by # of led
                stats): {', '.join(sorted(list(off_leaders)))}\n"
            analysis_text += f"- Defensive Leaders: {',
                '.join(sorted(list(def_leaders)))}\n"
            analysis_text += f"- Possession Leaders: {',
                '.join(sorted(list(poss_leaders)))}\n"
        else:
             analysis_text += "Could not generate detailed categorical leaders
                 as team means were not available for analysis.\n"

    else:
        analysis_text += "Could not determine highest scoring teams, so further
            detailed analysis is limited.\n"

    analysis_text += "\nDisclaimer: This analysis is based solely on average
        player statistics per team derived from the input data..."
    print(analysis_text)
    try:
        with open("team_performance_analysis_summary.txt", "w",
            encoding="utf-8") as f_analysis:
            f_analysis.write(analysis_text)
        print("\nTeam performance analysis summary saved to
            team_performance_analysis_summary.txt")
    except Exception as e_write_analysis:
        print(f"Error writing team performance analysis summary:
            {e_write_analysis}")

    print("\n--- Analysis Finished ---")
```

# Giải thích chi tiết về mã code

## Tổng quan về mã code

Mã code này được viết bằng Python và sử dụng các thư viện như `pandas`, `numpy`, và `matplotlib` để phân tích dữ liệu bóng đá từ file `results.csv`. Nó thực hiện các tác vụ sau:

- **Đọc dữ liệu**: Tải dữ liệu từ file CSV.
- **Làm sạch dữ liệu**: Chuyển đổi các cột không phải số thành dạng số.
- **Phân tích thống kê**: Tính toán top/bottom 3 cầu thủ, median, mean, và độ lệch chuẩn (std) cho từng chỉ số và từng đội.
- **Xác định đội xuất sắc**: Tìm đội có điểm trung bình cao nhất cho mỗi chỉ số.
- **Tạo biểu đồ**: Vẽ histogram cho các chỉ số tấn công và phòng thủ.
- **Tóm tắt hiệu suất**: Phân tích và ghi kết quả hiệu suất đội vào file.

## Các hàm chính và cách hoạt động

**Hàm** `clean_numeric_column(series)`

- **Mục đích**: Chuyển đổi một cột dữ liệu (series) chứa ký tự không phải số (như `%`, `,`) thành dạng số.
- **Cách hoạt động**:
  1. Chuyển series thành chuỗi bằng `astype(str)`.
  2. Loại bỏ ký tự `%` và `,` bằng `str.replace`.
  3. Chuyển đổi thành số bằng `pd.to_numeric`, nếu lỗi thì trả về `NaN`.
- **Ví dụ**: `"15%"` → 15.0, `"1,234"` → 1234.0.

**Hàm** `get_numeric_columns(df, exclude_cols)`

- **Mục đích**: Xác định các cột trong DataFrame có thể là số, ngoại trừ các cột trong `exclude_cols`.
- **Cách hoạt động**:
  1. Lọc các cột không nằm trong `exclude_cols`.
  2. Kiểm tra kiểu dữ liệu gốc; nếu là số thì thêm vào danh sách.
  3. Nếu không phải số, thử chuyển đổi và tính tỷ lệ giá trị hợp lệ (>10% thì coi là cột số).
- **Ví dụ**: Với DataFrame có cột `['Player', 'Goals', 'Assists']` và `exclude_cols=['Player` hàm trả về `['Goals', 'Assists']`.

**Hàm** `format_player_list(series)`

- **Mục đích**: Định dạng danh sách cầu thủ và điểm số thành chuỗi để ghi file.
- **Cách hoạt động**: Tạo danh sách chuỗi `"player (score)"` từ series.
- **Ví dụ**: Series `{"Salah": 28, "Haaland": 21}` → `["Salah (28)", "Haaland (21)"]`.

**Phần chính** (`if __name__ == "__main__":`)

- **Đọc dữ liệu**: Tải `results.csv` bằng `pd.read_csv`. Nếu lỗi (file không tồn tại, trống), chương trình dừng.
- **Làm sạch dữ liệu**: Áp dụng `clean_numeric_column` cho các cột không phải số (ngoại trừ `ID_COLS`).
- **Xác định cột số**: Gọi `get_numeric_columns` để lấy danh sách `stat_cols`.
- **Top/Bottom 3**:
  - Sắp xếp từng cột số, lấy 3 giá trị cao nhất/thấp nhất cùng tên cầu thủ.
  - Ghi vào `top_3.txt`.

- **Median, Mean, Std**:
  - Tính toán cho toàn bộ dữ liệu và từng đội bằng `agg(['median', 'mean', 'std'])`.
  - Ghi vào `results2.csv`.
- **Đội cao nhất**:
  - Tính trung bình từng chỉ số theo đội, tìm đội có giá trị cao nhất.
  - Ghi vào `highest_scoring_teams.txt`.
- **Histogram**:
  - Lọc chỉ số tấn công/phòng thủ dựa trên `HISTOGRAM_OFFENSIVE_PATTERNS` và `HISTOGRAM_DEFENSIVE_PATTERNS`.
  - Vẽ histogram cho tất cả cầu thủ (`all_players`) và từng đội (`by_team`), lưu vào `histograms`.
- **Phân tích đội**:
  - Đếm số chỉ số mà mỗi đội dẫn đầu (cao hơn tốt hoặc thấp hơn tốt).
  - Xác định đội dẫn đầu về tấn công, phòng thủ, kiểm soát bóng.
  - Ghi vào `team_performance_analysis_summary.txt`.

## Kết quả đầu ra

**File** `top_3.txt`

- **Nội dung**: Top 3 và bottom 3 cầu thủ cho mỗi chỉ số.
- **Ví dụ**:

```
--- Defensive_Actions_Blocks_Blocks ---
Top 3:
  - Nathan Collins: 69.00
  - Murillo: 66.00
  - Tyrick Mitchell: 62.00
Bottom 3:
  - Aaron Ramsdale: 0.00
  - Ali Al Hamadi: 0.00
  - Alisson: 0.00
```

**File** `results2.csv`

- **Nội dung**: Median, mean, std của từng chỉ số, cho tất cả cầu thủ (`all`) và từng đội.
- **Ví dụ**:

```
Team,Median of Defensive_Actions_Blocks_Blocks,Mean of Defensive_Actions_Blocks
all,11.000,15.062,13.699,...
Arsenal,10.000,13.682,11.167,...
```

**File** `highest_scoring_teams.txt`

- **Nội dung**: Đội có trung bình cao nhất cho mỗi chỉ số.
- **Ví dụ**:

```
- Highest Avg Defensive_Actions_Blocks_Blocks: Brentford (20.38)
- Highest Avg Defensive_Actions_Blocks_Int: Bournemouth (13.83)
```

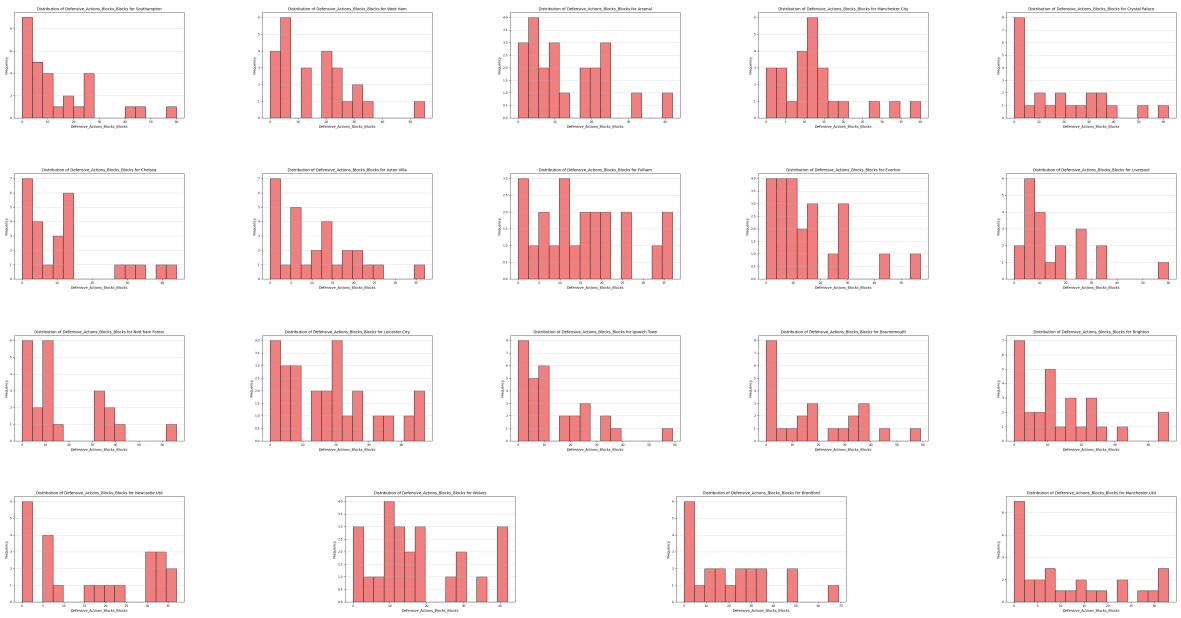
**Thư mục** `histograms`

- **Nội dung**: Biểu đồ histogram cho các chỉ số tấn công/phòng thủ.
- **Cấu trúc**:
  - `all_players`: Histogram cho tất cả cầu thủ (ví dụ: `hist_all_Defensive_Actions_Blocks`
  - `by_team`: Histogram cho từng đội (ví dụ: `hist_Arsenal_Defensive_Actions_Blocks_Blo`

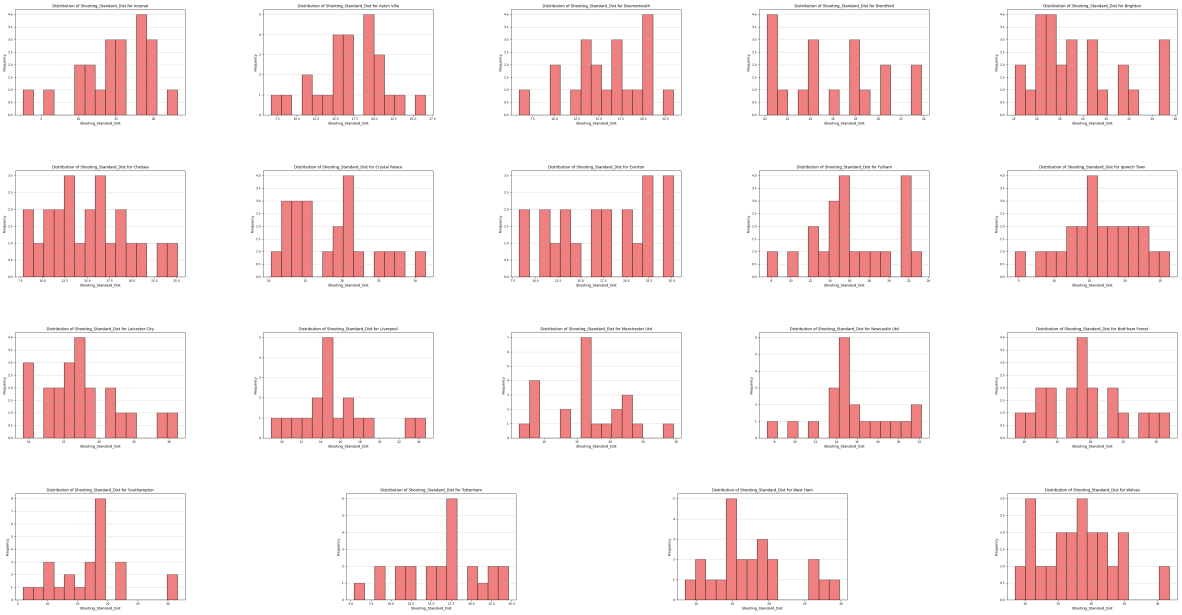
**File** `team_performance_analysis_summary.txt`

- **Nội dung**: Tóm tắt hiệu suất đội dựa trên số chỉ số dẫn đầu.
- **Ví dụ**:

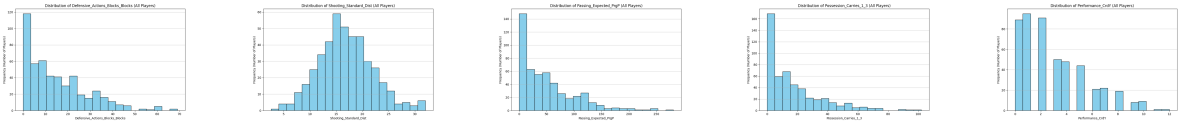
```
Based on the average statistics per team:
- 'Liverpool' leads 23 'higher-is-better' stats.
- 'Manchester City' leads 2 'lower-is-better' stats.
- Offensive Leaders: Brentford, Everton, Liverpool
- Defensive Leaders: Bournemouth, Brentford, Manchester City
- Possession Leaders: Brentford, Liverpool, Manchester City
```



Hình 1: Chỉ số phòng thủ của các đội



Hình 2: Chỉ số tấn công của các đội



Hình 3: Chỉ số của tất cả các cầu thủ

# Bài 3:

## Mã nguồn đầy đủ

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import seaborn as sns

# Load the dataset
try:
    df = pd.read_csv('result.csv')
    print(f"Successfully loaded result.csv. Dataset size: {df.shape}")
except FileNotFoundError:
    print("Error: File 'result.csv' not found.")
    print("Please ensure you have run the BTL-BAI1.py script first and the CSV
        file is created in the same directory.")
    exit()
except Exception as e:
    print(f"Error while reading CSV file: {e}")
    exit()

# Extract player information
player_info = df[['Player', 'Team', 'Position', 'Age']].copy()

# Identify numeric and categorical features
potential_numeric_cols = df.select_dtypes(include=np.number).columns.tolist()
cols_to_exclude = ['Age']
numeric_features = [col for col in potential_numeric_cols if col not in
    cols_to_exclude]
categorical_features = ['Position']

# Create features dataframe
features_df = df[numeric_features + categorical_features].copy()

# Define preprocessing pipelines
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])

categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])
```

```
46
47 # Combine preprocessors
48 preprocessor = ColumnTransformer(
49     transformers=[
50         ('num', numeric_transformer, numeric_features),
51         ('cat', categorical_transformer, categorical_features)
52     ],
53     remainder='drop'
54 )
55
56 # Preprocess the data
57 try:
58     X_processed = preprocessor.fit_transform(features_df)
59     print(f"Data preprocessing completed. Feature matrix size:
        {X_processed.shape}")
60     try:
61         feature_names_out = preprocessor.get_feature_names_out()
62     except AttributeError:
63         feature_names_out = numeric_features + \
64                             list(preprocessor.transformers_[1][1].named_steps['onehot']
                                    \
65                                  .get_feature_names_out(categorical_features))
66 except Exception as e:
67     print(f"Error during data preprocessing: {e}")
68     print("Selected numeric columns:", numeric_features)
69     print("Selected categorical columns:", categorical_features)
70     print("Data types of numeric columns:")
71     print(df[numeric_features].dtypes)
72     print("Data types of categorical columns:")
73     print(df[categorical_features].dtypes)
74     print("Number of NA values in numeric columns:")
75     print(df[numeric_features].isna().sum())
76     print("Number of NA values in categorical columns:")
77     print(df[categorical_features].isna().sum())
78     exit()
79
80 # Calculate inertia for different k values (Elbow Method)
81 inertia = []
82 possible_k = range(2, 11)
83
84 print("\nCalculating Inertia for different k values (Elbow Method)...")
85 for k in possible_k:
86     kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42, n_init=10)
87     kmeans.fit(X_processed)
88     inertia.append(kmeans.inertia_)
89
90 # Plot the Elbow curve
91 plt.figure(figsize=(10, 6))
92 plt.plot(possible_k, inertia, marker='o')
93 plt.title('Elbow Method for Determining Optimal Number of Clusters (k)')
94 plt.xlabel('Number of Clusters (k)')
```

```
plt.ylabel('Inertia (Within-cluster Sum of Squares)')
plt.xticks(possible_k)
plt.grid(True)
plt.show()

# Select optimal number of clusters
optimal_k = 4
print(f"\n=> Based on the Elbow plot, selected k = {optimal_k}")

# Perform final clustering
kmeans_final = KMeans(n_clusters=optimal_k, init='k-means++', random_state=42,
    n_init=10)
clusters = kmeans_final.fit_predict(X_processed)

# Add cluster labels to dataframes
player_info['Cluster'] = clusters
df['Cluster'] = clusters

print(f"\nAssigned {len(df)} players to {optimal_k} clusters.")
print("Number of players in each cluster:")
print(player_info['Cluster'].value_counts().sort_index())

# Perform PCA for dimensionality reduction
print("\nPerforming PCA to reduce data to 2 dimensions...")
pca = PCA(n_components=2, random_state=42)
X_pca = pca.fit_transform(X_processed)

# Create PCA dataframe
pca_df = pd.DataFrame(data=X_pca, columns=['Principal Component 1', 'Principal
    Component 2'])
pca_df['Cluster'] = clusters
pca_df['Player'] = player_info['Player'].values
pca_df['Position'] = player_info['Position'].values

# Plot 2D cluster visualization
print("Plotting 2D cluster visualization...")
plt.figure(figsize=(12, 8))
sns.scatterplot(
    x="Principal Component 1", y="Principal Component 2",
    hue="Cluster",
    palette=sns.color_palette("hsv", optimal_k),
    data=pca_df,
    legend="full",
    alpha=0.8
)

plt.title(f'Player Clustering ({optimal_k} Clusters) After PCA Reduction')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.grid(True)
plt.show()
```

```python

# Analyze cluster characteristics
print(f"\nAnalyzing basic characteristics of {optimal_k} clusters:")
cluster_summary = player_info.groupby('Cluster').agg(
    count=('Player', 'size'),
    common_position=('Position', lambda x: x.mode()[0] if not x.mode().empty
        else 'N/A'),
    avg_age=('Age', lambda x: pd.to_numeric(x, errors='coerce').mean())
).reset_index()

print("\nOverview of cluster characteristics (Count, Most Common Position,
    Average Age):")
print(cluster_summary)

# Calculate mean statistics for each cluster
print("\nMean values of original statistics for each cluster:")
numeric_original_df = df[numeric_features + ['Cluster']].copy()
for col in numeric_features:
    numeric_original_df[col] = pd.to_numeric(numeric_original_df[col],
        errors='coerce')

cluster_means = numeric_original_df.groupby('Cluster').mean()
print(cluster_means.round(2))

# PCA information
print("\nPCA Information:")
explained_variance = pca.explained_variance_ratio_
print(f"Variance explained by PC1: {explained_variance[0]:.2%}")
print(f"Variance explained by PC2: {explained_variance[1]:.2%}")
print(f"Total variance explained by 2 PCs: {explained_variance.sum():.2%}")

print("\n--- End ---")
```

# Giải thích chi tiết từng bước của code

## Bước 1: Tải và kiểm tra dữ liệu

Code bắt đầu bằng việc tải file CSV `result.csv` sử dụng hàm `pd.read_csv` từ thư viện pandas. Nếu file không tồn tại hoặc có lỗi, chương trình sẽ in thông báo lỗi và dừng thực thi.

## Bước 2: Chuẩn bị dữ liệu

- Tách thông tin cầu thủ: Các cột `Player`, `Team`, `Position`, `Age` được sao chép vào DataFrame `player_info`.
- Xác định đặc trưng:
  - Đặc trưng số: Các cột chứa giá trị số, ngoại trừ `Age`.
  - Đặc trưng phân loại: Cột `Position`.

- Tạo DataFrame `features_df` chứa các đặc trưng đã chọn.

### Bước 3: Tiền xử lý dữ liệu

Sử dụng `Pipeline` và `ColumnTransformer` để tiền xử lý:

- Đối với đặc trưng số:
  - Điền giá trị thiếu bằng giá trị trung bình (`SimpleImputer`).
  - Chuẩn hóa dữ liệu bằng `StandardScaler`.
- Đối với đặc trưng phân loại:
  - Điền giá trị thiếu bằng giá trị phổ biến nhất (`SimpleImputer`).
  - Mã hóa one-hot bằng `OneHotEncoder`.

Kết quả là ma trận `X_processed` đã được tiền xử lý.

### Bước 4: Xác định số cụm tối ưu

Sử dụng phương pháp Elbow:

- Tính inertia cho $k$ từ 2 đến 10.
- Vẽ biểu đồ inertia theo $k$ để tìm điểm "elbow".
- Chọn $k = 4$ làm số cụm tối ưu.

### Bước 5: Phân cụm

- Áp dụng thuật toán KMeans với $k = 4$ trên `X_processed`.
- Gán nhãn cụm (0, 1, 2, 3) cho từng cầu thủ, lưu vào `player_info` và `df`.

### Bước 6: Giảm chiều dữ liệu

Sử dụng PCA để giảm chiều dữ liệu xuống 2 chiều:

- Tạo ma trận `X_pca` từ `X_processed`.
- Tạo DataFrame `pca_df` chứa hai thành phần chính và nhãn cụm.

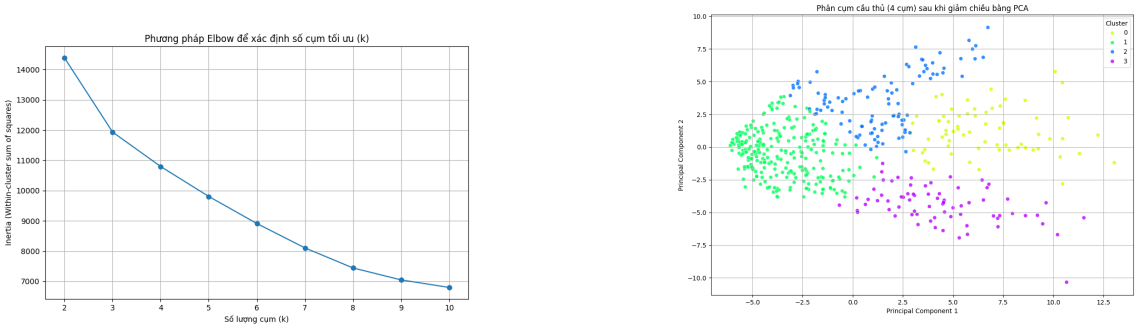
### Bước 7: Trực quan hóa

Sử dụng `sns.scatterplot` từ seaborn để vẽ biểu đồ phân tán 2D:

- Trục x: Thành phần chính 1 (PC1).
- Trục y: Thành phần chính 2 (PC2).
- Màu sắc: Phân biệt các cụm bằng nhãn cụm.

## Bước 8: Phân tích cụm

- Tính toán:
  - Số lượng cầu thủ trong mỗi cụm.
  - Vị trí thi đấu phổ biến nhất và tuổi trung bình của mỗi cụm.
  - Giá trị trung bình của các chỉ số thống kê gốc cho mỗi cụm.
- Hiển thị tỷ lệ phương sai được giải thích bởi PCA.




(a) Biểu đồ Elbow

(b) Phân cụm PCA

Hình 4: Elbow and PCA

# Bài 4:

## Mã nguồn lấy data cho chuyển nhượng cầu thủ

```python
# --- Import necessary libraries ---
import time
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager
from selenium.common.exceptions import WebDriverException

# --- Function to set up Selenium WebDriver ---
def setup_driver():
    """Initialize and return an instance of Chrome WebDriver."""
    try:
        service = ChromeService(ChromeDriverManager().install())
        driver = webdriver.Chrome(service=service)
        print("WebDriver initialized successfully.")
        return driver
    except WebDriverException as e:
        print(f"Error initializing WebDriver: {e}")
        print("Ensure Google Chrome is installed.")
        print("Or try updating webdriver-manager: pip install --upgrade
            webdriver-manager")
        return None
    except Exception as e:
        print(f"Unknown error initializing driver: {e}")
        return None

# --- Function to scrape data from a specific URL ---
def scrape_page(driver, url):
    """Scrape player data from a URL using Selenium driver."""
    if driver is None:
        print("Error: Invalid driver.")
        return []
    try:
        driver.get(url)
        print(f"Accessing: {url}")
        time.sleep(3)
        soup = BeautifulSoup(driver.page_source, 'html.parser')

        table = soup.find('table', class_='table table-hover no-cursor
            table-striped leaguetable mvp-table similar-players-table mb-0')
        if not table:
            print(f"Warning: No data table found on page {url}")
            return []

        tbody = table.find('tbody')
        if not tbody:
```

```
            print(f"Warning: No tbody tag found in table on page {url}")
            return []

        data = []
        rows = tbody.find_all('tr')
        print(f"Found {len(rows)} rows on page {url}")

        for row in rows:
            try:
                skill_div = row.find('div', class_='table-skill__skill')
                pot_div = row.find('div', class_='table-skill__pot')
                skill_text = skill_div.text.strip() if skill_div else None
                pot_text = pot_div.text.strip() if pot_div else None
                skill = float(skill_text) if skill_text else None
                pot = float(pot_text) if pot_text else None
                skill_pot = f"{skill}/{pot}" if skill is not None and pot is
                    not None else None

                player_link = row.select_one('td.td-player div.text a')
                player_name = player_link.text.strip() if player_link else None

                team_span = row.find('span', class_='td-team__teamname')
                team = team_span.text.strip() if team_span else None

                etv_span = row.find('span', class_='player-tag')
                etv = etv_span.text.strip() if etv_span else None

                if player_name and team and etv and skill_pot:
                    data.append({
                        'player_name': player_name,
                        'team': team,
                        'price': etv,
                        'skill/pot': skill_pot
                    })

            except Exception as e:
                print(f"Error processing a row: {e}. Skipping this row.")
                continue

        return data

    except WebDriverException as e:
        print(f"WebDriver error accessing {url}: {e}")
        return []
    except Exception as e:
        print(f"Unknown error scraping page {url}: {e}")
        return []

# --- Main section to perform scraping ---
base_url = "https://www.footballtransfers.com/en/players/uk-premier-league"
total_pages = 22
```

```
all_data = []

print("Initializing WebDriver...")
driver = setup_driver()

if driver:
    try:
        print(f"Starting to scrape data from {total_pages} pages...")
        for page in range(1, total_pages + 1):
            print(f"\n--- Processing page {page}/{total_pages} ---")
            if page == 1:
                url = base_url
            else:
                url = f"{base_url}/{page}"

            page_data = scrape_page(driver, url)

            if page_data:
                all_data.extend(page_data)
                print(f"Added {len(page_data)} records from page {page}.")
            else:
                print(f"No valid data returned from page {page}.")

    except Exception as e:
        print(f"An error occurred during scraping: {e}")
    finally:
        print("\nClosing WebDriver...")
        driver.quit()

    if all_data:
        print(f"\nTotal of {len(all_data)} records scraped.")
        df_final = pd.DataFrame(all_data)
        try:
            df_final.to_csv('football_transfers_players.csv', index=False,
                encoding='utf-8-sig')
            print("Data successfully saved to 'football_transfers_players.csv'")
            print("\nPreview of the first 5 rows of data:")
            print(df_final.head())
        except Exception as e:
            print(f"Error saving CSV file: {e}")
    else:
        print("\nNo data collected. CSV file will not be created.")
else:
    print("Failed to initialize WebDriver. Unable to proceed with scraping.")

print("\nCompleted.")
```

## Mã nguồn chính

```
import pandas as pd

def combine_and_filter_player_data():
    """
    Combine data from football_transfers_players.csv and results.csv,
    then filter players with playing time > 900 minutes and display that time.
    """
    try:
        df_transfers = pd.read_csv('football_transfers_players.csv')
        df_fbref = pd.read_csv('results.csv')
        print("Successfully read 'football_transfers_players.csv' and
            'results.csv'.")
    except FileNotFoundError as e:
        print(f"Error: One of the required CSV files not found: {e}")
        print("Ensure 'football_transfers_players.csv' and 'results.csv' are
            created and in the same directory.")
        return
    except pd.errors.EmptyDataError as e:
        print(f"Error: One of the CSV files is empty: {e}")
        return
    except Exception as e:
        print(f"Unknown error reading CSV files: {e}")
        return

    if 'player_name' in df_transfers.columns:
        df_transfers.rename(columns={'player_name': 'Player'}, inplace=True)
    elif 'Player' not in df_transfers.columns:
        print("Error: No player name column ('player_name' or 'Player') found
            in 'football_transfers_players.csv'.")
        return

    if 'Player' not in df_fbref.columns:
        print("Error: No 'Player' column found in 'results.csv'.")
        return

    minutes_col_fbref = None
    candidate_minute_cols = ['Playing_Time_Min', 'Min', 'minutes']

    for col_name in candidate_minute_cols:
        if col_name in df_fbref.columns:
            minutes_col_fbref = col_name
            break

    if minutes_col_fbref is None:
        possible_min_cols = [col for col in df_fbref.columns if 'min' in
            col.lower() and ('time' in col.lower() or 'play' in col.lower() or
            col.lower() == 'min')]
        if possible_min_cols:
            minutes_col_fbref = possible_min_cols[0]
```

```
        print(f"Warning: No standard minutes column found. Using heuristic column: '{minutes_col_fbref}'")
    else:
        print(f"Error: Could not identify minutes played column in 'results.csv'. Available columns: {df_fbref.columns.tolist()}")
        return

print(f"Using column '{minutes_col_fbref}' from 'results.csv' for filtering minutes played.")

df_fbref[minutes_col_fbref] = pd.to_numeric(df_fbref[minutes_col_fbref], errors='coerce')
df_fbref.dropna(subset=[minutes_col_fbref], inplace=True)

df_fbref_filtered = df_fbref[df_fbref[minutes_col_fbref] > 900][['Player', minutes_col_fbref]].copy()

if df_fbref_filtered.empty:
    print(f"\nNo players in 'results.csv' with playing time ({minutes_col_fbref}) > 900 minutes.")
    return

df_fbref_filtered.rename(columns={minutes_col_fbref: 'Total_Minutes_Played'}, inplace=True)

players_with_high_minutes_count = df_fbref_filtered['Player'].nunique()
print(f"\nFound {players_with_high_minutes_count} players in 'results.csv' with > 900 minutes played.")

players_to_keep = df_fbref_filtered['Player'].unique()
df_transfers_filtered_by_name = df_transfers[df_transfers['Player'].isin(players_to_keep)].copy()

df_fbref_for_merge = df_fbref_filtered.drop_duplicates(subset=['Player'], keep='first')
df_final_output = pd.merge(df_transfers_filtered_by_name, df_fbref_for_merge, on='Player', how='left')

print(f"\nInitial number of players in 'football_transfers_players.csv': {len(df_transfers)}")
print(f"Final number of players (matching > 900 minutes criteria and in transfers): {len(df_final_output)}")

if not df_final_output.empty:
    print("\n--- Preview of first 5 rows of filtered player data (including total minutes played): ---")
    print(df_final_output.head())

    try:
        output_filename = 'filtered_football_transfers_players_gt900min_with_total_time.csv'
```

```
            cols = ['Player'] + [col for col in df_final_output.columns if col
                != 'Player' and col != 'Total_Minutes_Played'] +
                ['Total_Minutes_Played']
            cols_exist = [col for col in cols if col in df_final_output.columns]
            df_final_output_ordered = df_final_output[cols_exist]
            df_final_output_ordered.to_csv(output_filename, index=False,
                encoding='utf-8-sig')
            print(f"\nSaved filtered player data to '{output_filename}'")
        except Exception as e:
            print(f"Error saving output CSV file: {e}")
    else:
        print("\nNo players from 'football_transfers_players.csv' match the >
            900 minutes criteria or could not be merged.")

if __name__ == '__main__':
    combine_and_filter_player_data()
```

# Phân tích code trong BTL-BAI4.py

File này chịu trách nhiệm thu thập dữ liệu cầu thủ từ trang web `https://www.footballtransfers.com/en/players/uk-premier-league` và lưu vào file CSV có tên `football_transfers_players.csv`

## Import các thư viện cần thiết

```
import time
import pandas as pd
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.chrome.service import Service as ChromeService
from webdriver_manager.chrome import ChromeDriverManager
from selenium.common.exceptions import WebDriverException
```

Danh sách các thư viện:

- **time**: Dùng để tạm dừng chương trình nhằm tránh tải quá nhanh.
- **pandas**: Thư viện xử lý và phân tích dữ liệu.
- **BeautifulSoup**: Phân tích cú pháp HTML từ trang web.
- **selenium**: Tự động hóa trình duyệt để tải trang động.
- **ChromeDriverManager**: Quản lý và cài đặt ChromeDriver tự động.
- **WebDriverException**: Xử lý các ngoại lệ từ Selenium.

### Hàm `setup_driver()`

```
def setup_driver():
    try:
        service = ChromeService(ChromeDriverManager().install())
        driver = webdriver.Chrome(service=service)
        print("WebDriver initialized successfully.")
        return driver
    except WebDriverException as e:
        print(f"Error initializing WebDriver: {e}")
        print("Ensure Google Chrome is installed.")
        print("Or try updating webdriver-manager: pip install --upgrade webdriver-m
        return None
    except Exception as e:
        print(f"Unknown error initializing driver: {e}")
        return None
```

Chức năng:

- Khởi tạo trình duyệt Chrome thông qua Selenium WebDriver.
- Sử dụng `ChromeDriverManager().install()` để cài đặt ChromeDriver tự động.
- Trả về đối tượng `driver` nếu thành công, hoặc `None` nếu có lỗi.

### Hàm `scrape_page(driver, url)`

```
def scrape_page(driver, url):
    if driver is None:
        print("Error: Invalid driver.")
        return []
    try:
        driver.get(url)
        print(f"Accessing: {url}")
        time.sleep(3)
        soup = BeautifulSoup(driver.page_source, 'html.parser')
```

Chức năng:

- Thu thập dữ liệu từ một URL cụ thể.
- Kiểm tra `driver` hợp lệ, tải trang web, chờ 3 giây, sau đó phân tích HTML bằng BeautifulSoup.

```
        table = soup.find('table', class_='table table-hover no-cursor table-stripe
        if not table:
            print(f"Warning: No data table found on page {url}")
            return []

        tbody = table.find('tbody')
        if not tbody:
            print(f"Warning: No tbody tag found in table on page {url}")
            return []
```

- Tìm bảng dữ liệu với class cụ thể trên trang web.
- Nếu không tìm thấy bảng hoặc thẻ `<tbody>`, trả về danh sách rỗng.

```
data = []
rows = tbody.find_all('tr')
print(f"Found {len(rows)} rows on page {url}")

for row in rows:
    try:
        skill_div = row.find('div', class_='table-skill__skill')
        pot_div = row.find('div', class_='table-skill__pot')
        skill_text = skill_div.text.strip() if skill_div else None
        pot_text = pot_div.text.strip() if pot_div else None
        skill = float(skill_text) if skill_text else None
        pot = float(pot_text) if pot_text else None
        skill_pot = f"{skill}/{pot}" if skill and pot else None
```

- Xử lý từng hàng trong bảng dữ liệu.
- Trích xuất và chuyển đổi thông tin kỹ năng (`skill`) và tiềm năng (`pot`).

```
        player_link = row.select_one('td.td-player div.text a')
        player_name = player_link.text.strip() if player_link else None

        team_span = row.find('span', class_='td-team__teamname')
        team = team_span.text.strip() if team_span else None

        etv_span = row.find('span', class_='player-tag')
        etv = etv_span.text.strip() if etv_span else None
```

- Trích xuất tên cầu thủ, tên đội bóng, và giá trị chuyển nhượng (ETV).

```
        if player_name and team and etv and skill_pot:
            data.append({
                'player_name': player_name,
                'team': team,
                'price': etv,
                'skill/pot': skill_pot
            })
```

- Nếu tất cả thông tin đầy đủ, thêm dữ liệu vào danh sách `data`.

```
except WebDriverException as e:
    print(f"WebDriver error accessing {url}: {e}")
    return []
except Exception as e:
    print(f"Unknown error scraping page {url}: {e}")
    return []
```

- Xử lý lỗi từ WebDriver hoặc lỗi khác, trả về danh sách rỗng nếu có vấn đề.

### Phần chính (Main Section)

```
base_url = "https://www.footballtransfers.com/en/players/uk-premier-league"
total_pages = 22
all_data = []

driver = setup_driver()

if driver:
    try:
        for page in range(1, total_pages + 1):
            if page == 1:
                url = base_url
            else:
                url = f"{base_url}/{page}"
            page_data = scrape_page(driver, url)
            if page_data:
                all_data.extend(page_data)
```

- Thiết lập URL cơ bản và tổng số trang (22).
- Khởi tạo danh sách `all_data` để lưu trữ toàn bộ dữ liệu.
- Dùng vòng lặp để thu thập dữ liệu từ 22 trang.

```
    finally:
        driver.quit()

    if all_data:
        df_final = pd.DataFrame(all_data)
        df_final.to_csv('football_transfers_players.csv', index=False, encoding='ut
        print("\nPreview of the first 5 rows of data:")
        print(df_final.head())
```

- Đóng trình duyệt sau khi hoàn tất.
- Nếu có dữ liệu, chuyển thành DataFrame và lưu vào file CSV, sau đó in 5 hàng đầu tiên để kiểm tra.

## Phân tích code trong "FINAL RESULT.py"

File này xử lý dữ liệu từ `football_transfers_players.csv`, kết hợp với `results.csv`, lọc các cầu thủ chơi hơn 900 phút và lưu kết quả.

### Import thư viện

```
import pandas as pd
```

- Sử dụng **pandas** để đọc và xử lý dữ liệu từ các file CSV.

## Hàm `combine_and_filter_player_data()`

```
def combine_and_filter_player_data():
    try:
        df_transfers = pd.read_csv('football_transfers_players.csv')
        df_fbref = pd.read_csv('results.csv')
        print("Successfully read 'football_transfers_players.csv' and 'results.csv'
    except FileNotFoundError as e:
        print(f"Error: One of the required CSV files not found: {e}")
        return
```

- Đọc dữ liệu từ hai file CSV, xử lý lỗi nếu file không tồn tại.

```
    if 'player_name' in df_transfers.columns:
        df_transfers.rename(columns={'player_name': 'Player'}, inplace=True)
    elif 'Player' not in df_transfers.columns:
        print("Error: No player name column ('player_name' or 'Player') found in 'f
        return

    if 'Player' not in df_fbref.columns:
        print("Error: No 'Player' column found in 'results.csv'.")
        return
```

- Chuẩn hóa tên cột thành `Player` để đồng nhất giữa hai file.

```
    minutes_col_fbref = None
    candidate_minute_cols = ['Playing_Time_Min', 'Min', 'minutes']
    for col_name in candidate_minute_cols:
        if col_name in df_fbref.columns:
            minutes_col_fbref = col_name
            break

    if minutes_col_fbref is None:
        possible_min_cols = [col for col in df_fbref.columns if 'min' in col.lower(
        if possible_min_cols:
            minutes_col_fbref = possible_min_cols[0]
```

- Tìm cột chứa thông tin phút thi đấu trong `results.csv`.

```
    df_fbref[minutes_col_fbref] = pd.to_numeric(df_fbref[minutes_col_fbref], errors
    df_fbref.dropna(subset=[minutes_col_fbref], inplace=True)

    df_fbref_filtered = df_fbref[df_fbref[minutes_col_fbref] > 900][['Player', minu
```

- Chuyển đổi dữ liệu phút thi đấu sang số, loại bỏ giá trị `NaN`, và lọc các cầu thủ chơi hơn 900 phút.

```
    df_fbref_filtered.rename(columns={minutes_col_fbref: 'Total_Minutes_Played'}, i
    players_to_keep = df_fbref_filtered['Player'].unique()
    df_transfers_filtered_by_name = df_transfers[df_transfers['Player'].isin(player
```

- Đổi tên cột phút thi đấu thành `Total_Minutes_Played`.
- Lọc dữ liệu chuyển nhượng dựa trên danh sách cầu thủ đã chọn.

```
df_fbref_for_merge = df_fbref_filtered.drop_duplicates(subset=['Player'], keep=
df_final_output = pd.merge(df_transfers_filtered_by_name, df_fbref_for_merge, o
```

- Xóa các hàng trùng lặp trong dữ liệu phút thi đấu.
- Gộp dữ liệu chuyển nhượng và phút thi đấu dựa trên cột `Player`.

```
if not df_final_output.empty:
    cols = ['Player'] + [col for col in df_final_output.columns if col != 'Play
    df_final_output_ordered = df_final_output[cols]
    df_final_output_ordered.to_csv('filtered_football_transfers_players_gt900mi
```

- Nếu có dữ liệu, sắp xếp lại thứ tự cột và lưu vào file CSV mới.

## Phần chính

```
if __name__ == '__main__':
    combine_and_filter_player_data()
```

- Chạy hàm chính khi file được thực thi.