```
"""
Training command: import the race header details (and horses) from a single
racecard HTML file and write them to the DB.

Run:
  python manage.py import_racecard path/to/file.html
  python manage.py import_racecard path/to/file.html --update

This prints detailed debug info at each step so you can see how parsing flows.
"""

import os
import re
from datetime import datetime, date, time

from bs4 import BeautifulSoup
from django.core.management.base import BaseCommand
from racecard.models import Race, Horse, Run, Ranking
import json
from django.db.models import Q

# Import the new ClassAnalysisService
from racecard.services.class_analysis import ClassAnalysisService

# -------------------------
# Helpers
# -------------------------
def ensure_date(val):
    if isinstance(val, date) and not isinstance(val, datetime):
        return val
    if isinstance(val, datetime):
        return val.date()
    if isinstance(val, str):
        # Clean the string first
        clean_val = val.strip()
        # Handle cases like "(5) 24.10.05" or "(20)25.01.11"
        if '(' in clean_val and ')' in clean_val:
            clean_val = clean_val.split(')')[-1].strip()

        # Try different date formats
        for fmt in ("%y.%m.%d", "%d.%m.%y", "%y%m%d", "%d/%m/%Y", "%d/%m/%y"):
            try:
                return datetime.strptime(clean_val, fmt).date()
            except ValueError:
                continue
    raise ValueError(f"Cannot parse date from value: {val!r}")


def ensure_time(val):
    if isinstance(val, time):
        return val
    if isinstance(val, str):
        s = val.strip().replace(".", ":")
        if ":" in s:
            h, m = map(int, s.split(":")[:2])
        else:
            h, m = divmod(int(s), 100)
        return time(h, m)
    if isinstance(val, int):
```

```
        h, m = divmod(val, 100)
        return time(h, m)
    raise ValueError(f"Cannot parse time from value: {val!r}")


def _text(node, default=""):
    return node.get_text(strip=True) if node else default


class Command(BaseCommand):
    help = (
        "Import race header and horses from a racecard HTML file. "
        "Use --update to update existing DB rows."
    )

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Initialize the class analysis service
        self.class_analyzer = ClassAnalysisService()

    # -------------------------
    # CLI arguments
    # -------------------------
    def add_arguments(self, parser):
        parser.add_argument("html_file", type=str, help="Path to a single racecard
HTML file")
        parser.add_argument(
            "--update",
            action="store_true",
            dest="update",
            help="If set, update the existing Race (and Horses) with parsed
values.",
        )

    # -------------------------
    # Jockey-Trainer Analysis Functions
    # -------------------------
    def analyze_jockey_trainer_combination(self, html_row):
        """
        Analyze jockey-trainer combination from HTML row and calculate a score
        based on placing percentage and other performance metrics.
        """
        # Extract data from table cells
        cells = html_row.find_all('td')

        if len(cells) < 9:  # We need at least 9 cells for the basic data
            return None

        # Extract key information
        horse_number = cells[0].get_text(strip=True)
        jockey = cells[1].get_text(strip=True)
        trainer = cells[2].get_text(strip=True)

        # Performance metrics
        try:
            percentage_placings = int(cells[3].get_text(strip=True))
            starts = int(cells[4].get_text(strip=True))
            first_places = int(cells[5].get_text(strip=True))
            second_places = int(cells[6].get_text(strip=True))
```

```
            third_places = int(cells[7].get_text(strip=True))
        except (ValueError, IndexError):
            return None

        # Calculate additional metrics
        total_races = starts
        total_wins = first_places
        total_places = first_places + second_places + third_places

        win_percentage = (total_wins / total_races * 100) if total_races > 0 else 0
        place_percentage = (total_places / total_races * 100) if total_races > 0
else 0

        # Calculate combination score (weighted formula)
        score = (
            (percentage_placings * 0.4) +           # 40% weight to placing
percentage
            (win_percentage * 0.3) +                # 30% weight to win percentage
            (place_percentage * 0.2) +              # 20% weight to place
percentage
            (min(total_races, 50) * 0.1)            # 10% weight to experience
(capped at 50 races)
        )

        return {
            'horse_number': horse_number,
            'jockey': jockey,
            'trainer': trainer,
            'combination': f"{jockey} - {trainer}",
            'percentage_placings': percentage_placings,
            'starts': total_races,
            'wins': total_wins,
            'places': total_places,
            'win_percentage': round(win_percentage, 1),
            'place_percentage': round(place_percentage, 1),
            'score': round(score, 2),
            'rating': self.get_jt_rating(score)
        }

    def get_jt_rating(self, score):
        """Convert numerical score to qualitative rating"""
        if score >= 80:
            return "Excellent"
        elif score >= 60:
            return "Very Good"
        elif score >= 40:
            return "Good"
        elif score >= 20:
            return "Average"
        else:
            return "Poor"

    # -------------------------
    # Header parsing
    # -------------------------
    @staticmethod
    def _parse_header_td(soup):
        td = soup.find("td", align="center")
        if not td:
```

```
        return {}

    lines = list(td.stripped_strings)
    result = {
        "lines": lines,
        "course": lines[0] if lines else None,
        "date_text": None,
        "race_date": None,
        "race_no": None,
        "race_time_text": None,
        "race_time_hhmm": None,
    }

    # Date detection (accept 25/07/2025 or 25/07/25)
    for text in lines[1:4]:
        clean = text.strip()
        if re.fullmatch(r"\d{2}/\d{2}/\d{4}", clean):
            parsed_date = datetime.strptime(clean, "%d/%m/%Y").date()
            result["date_text"] = clean
            result["race_date"] = parsed_date
            break
        if re.fullmatch(r"\d{2}/\d{2}/\d{2}", clean):
            parsed_date = datetime.strptime(clean, "%d/%m/%y").date()
            result["date_text"] = clean
            result["race_date"] = parsed_date
            break

    # Race number in <div class="rev4">
    rev4 = td.find("div", class_="rev4")
    if rev4:
        m = re.search(r"\d+", rev4.get_text(strip=True))
        if m:
            result["race_no"] = int(m.group())

    # Race time in <div class="b1">
    b1 = td.find("div", class_="b1")
    if b1:
        raw = b1.get_text(strip=True)
        result["race_time_text"] = raw
        normalized = raw.replace(".", ":")
        try:
            t = datetime.strptime(normalized, "%H:%M").time()
            result["race_time_hhmm"] = t.hour * 100 + t.minute
        except Exception:
            m2 = re.search(r"\b(\d{3,4})\b", raw)
            if m2:
                try:
                    result["race_time_hhmm"] = int(m2.group(1))
                except Exception:
                    result["race_time_hhmm"] = None

    return result

@staticmethod
def _parse_right_td_for_details(right_td):
    result = {
        "race_name": None,
        "race_distance": None,
        "race_class": None,
```

```
            "race_merit": None,
        }
        if not right_td:
            return result

        b2 = right_td.find("div", class_="b2")
        if b2:
            b2_lines = list(b2.stripped_strings)
            if b2_lines:
                result["race_name"] = b2_lines[0]
            if len(b2_lines) > 1:
                m = re.search(r"(\d+)\s*Metres", b2_lines[1], flags=re.I)
                if m:
                    result["race_distance"] = m.group(1)

        for text in (t.strip() for t in right_td.stripped_strings if t.strip()):
            low = text.lower()
            if any(k in low for k in (
                "class", "maiden", "merit rated", "benchmark", "handicap",
                "stakes", "conditions", "plate", "allowance", "apprentice",
                "novice", "graduation", "restricted"
            )):
                result["race_class"] = text
                m = re.search(r"Merit\s*Rated\s*(\d{1,3})", text, flags=re.I)
                if not m:
                    m = re.search(r"Benchmark\s*(\d{1,3})", text, flags=re.I)
                if not m:
                    m = re.search(r"\b(\d{2,3})\b", text)
                if m:
                    try:
                        result["race_merit"] = int(m.group(1))
                    except Exception:
                        result["race_merit"] = 0
                else:
                    result["race_merit"] = 0
                break

        return result

    def _parse_horse_runs(self, horse_table, horse_obj):
        """Extract and save the last 4 runs for a horse"""
        runs = []

        # Find all run rows - looking for class="small" in your HTML
        run_rows = horse_table.find_all('tr', class_='small') or []

        for run_row in run_rows[:4]:  # Only take last 4 runs
            try:
                cols = [td.get_text(strip=True) for td in run_row.find_all('td')]
                if len(cols) < 15:  # Make sure we have enough columns
                    continue

                # Extract and parse date
                date_text = cols[0]
                run_date = ensure_date(date_text)

                # Extract position (from what appears to be column 13 in debug
output)
                position = cols[12] if len(cols) > 12 else ""
```

```
                # Extract margin (from column 13 in debug output)
                margin = cols[13] if len(cols) > 13 else ""

                # Extract distance (from column 7 in debug output)
                distance = cols[6] if len(cols) > 6 else ""

                # Extract race class (from column 5 in debug output)
                race_class = cols[4] if len(cols) > 4 else ""

                # Create the run record
                Run.objects.create(
                    horse=horse_obj,
                    run_date=run_date,
                    position=position,
                    margin=margin,
                    distance=distance,
                    race_class=race_class
                )
                runs.append({
                    'date': run_date,
                    'position': position,
                    'margin': margin,
                    'distance': distance,
                    'class': race_class
                })

            except Exception as e:
                self.stdout.write(self.style.WARNING(f"⚠️ Could not parse run row:
{e}"))
                continue

        return runs

   # -------------------------
    # Horse parsing
    # -------------------------
    def _parse_horses(self, soup, race, update_existing: bool):
        """
        Parse horse blocks. We only consider tables that:
        - have border="border"
        - contain a <div class="b4"> with a numeric horse number
        """
        self.stdout.write("\n🔍 Extracting Horses...")
        created_or_updated = 0
        horse_tables = soup.select('table[border="border"]')

        # FIRST: Find and parse the jockey-trainer stats table
        jt_analysis_data = self._parse_jockey_trainer_table(soup)

        for idx, table in enumerate(horse_tables, start=1):
            try:
                first_tr = table.find("tr")
                if not first_tr:
                    continue
                main_tds = first_tr.find_all("td", recursive=False)
                if len(main_tds) < 2:
                    continue
```

```
# --- TD 0: number/odds/rating ---
td0 = main_tds[0]
num_div = td0.find("div", class_="b4")
if not num_div:
    # Not a horse row
    continue
try:
    horse_no = int(_text(num_div))
except Exception:
    continue

odds_el = td0.find("div", class_="b1")
odds = _text(odds_el)

merit_el = td0.find("span", class_="b1")
horse_merit = None
if merit_el:
    m = re.search(r"\d+", merit_el.get_text())
    if m:
        horse_merit = int(m.group())

# --- TD 1: name + age/blinkers ---
td1 = main_tds[1] if len(main_tds) > 1 else None
horse_name = ""
blinkers = False
age = ""

if td1:
    name_cell = td1.find("td", class_="b1")
    horse_name = _text(name_cell) or _text(td1)
    # Blinkers if "(B)" appears anywhere in the name block
    block_text_upper = td1.get_text(" ", strip=True).upper()
    blinkers = "(B" in block_text_upper

    # Age e.g. "6 y. o. b g."
    age_text = ""
    for s in td1.stripped_strings:
        if re.search(r"\by\.?\s*o\.?", s, flags=re.I):
            age_text = s
            break
    m_age = re.search(r"\b(\d{1,2})\b", age_text)
    age = m_age.group(1) if m_age else ""

# --- Jockey / Trainer (nested table) ---
# Look inside the current horse's table for any "div.itbld"
itbld_divs = table.select("div.itbld")
jockey, trainer = "", ""
if len(itbld_divs) >= 1:
    jockey = " ".join(itbld_divs[0].stripped_strings)
if len(itbld_divs) >= 2:
    trainer = " ".join(itbld_divs[1].stripped_strings)

# --- Jockey-Trainer Analysis ---
jt_score = 50  # Default neutral score
jt_rating = "Average"

# Use the pre-parsed jockey-trainer data if available
if horse_no in jt_analysis_data:
    jt_data = jt_analysis_data[horse_no]
```

```
                    jt_score = jt_data['score']
                    jt_rating = jt_data['rating']
                    # Use the jockey/trainer from analysis if available (more
accurate)
                    jockey = jt_data.get('jockey', jockey)
                    trainer = jt_data.get('trainer', trainer)

                # --- Debug prints ---
                print(f"[DEBUG] Horse {horse_no}: name={horse_name}")
                print(f"[DEBUG]  -> Odds={odds}, Merit={horse_merit},
Blinkers={blinkers}, Age={age}")
                print(f"[DEBUG]  -> Jockey={jockey}, Trainer={trainer}")
                print(f"[DEBUG]  -> Jockey-Trainer Score={jt_score},
Rating={jt_rating}")

                # Ensure safe field lengths
                age = (age or "")[:10]
                odds = (odds or "")[:20]

                # Upsert
                defaults = dict(
                    horse_name=horse_name,
                    blinkers=bool(blinkers),
                    age=age,
                    dob="",  # not present in provided markup
                    odds=odds,
                    horse_merit=horse_merit if horse_merit is not None else 0,
                    race_class=race.race_class or "",
                    trainer=trainer,
                    jockey=jockey,
                )
                obj, created = Horse.objects.update_or_create(
                    race=race, horse_no=horse_no, defaults=defaults
                )
                created_or_updated += 1
                self.stdout.write(
                    f"🐎 Horse {horse_no}: {horse_name} | "
                    f"Blinkers={blinkers} | Odds={odds or '-'} | "
                    f"Merit={defaults['horse_merit']} | "
                    f"Jockey={jockey or '-'} | Trainer={trainer or '-'} | "
                    f"J-T Score={jt_score} | J-T Rating={jt_rating}"
                )

                # Add runs extraction
                runs = self._parse_horse_runs(table, obj)
                if runs:
                    self.stdout.write(f"    📜 Added {len(runs)} past runs:")
                    for run in runs:
                        self.stdout.write(f"      - {run['date']}: Pos
{run['position']} ({run['margin']}) {run['distance']}m {run['class']}")

            except Exception as e:
                self.stdout.write(self.style.WARNING(f"⚠️ Skipping one table (idx
{idx}) due to error: {e}"))

        # Store jockey-trainer analysis in race for later use
        race.jt_analysis_data = jt_analysis_data

        self.stdout.write(self.style.SUCCESS(f"✅ Horses saved:
```

```
{created_or_updated}"))
        return created_or_updated

    def _parse_jockey_trainer_table(self, soup):
        """Find and parse the jockey-trainer statistics table"""
        jt_analysis_data = {}

        # Look for tables that contain jockey-trainer stats
        for table in soup.find_all('table'):
            # Check if this table contains jockey-trainer headers
            headers = table.find_all(['th', 'td'])
            header_texts = [header.get_text(strip=True) for header in headers]

            # Look for characteristic headers
            has_jockey = any('jockey' in text.lower() for text in header_texts)
            has_trainer = any('trainer' in text.lower() for text in header_texts)
            has_rns = any('rns' in text.lower() for text in header_texts)

            if has_jockey and has_trainer and has_rns:
                self.stdout.write("📊 Found jockey-trainer statistics table")
                self.stdout.write(f"DEBUG: Table headers: {header_texts}")

                # Parse all data rows
                for row in table.find_all('tr'):
                    # Skip header rows
                    if any('jockey' in cell.get_text().lower() or
                        'trainer' in cell.get_text().lower() for cell in
row.find_all('td')):
                        continue

                    jt_analysis = self.analyze_jockey_trainer_combination(row)
                    if jt_analysis:
                        try:
                            horse_no = int(jt_analysis['horse_number'])
                            jt_analysis_data[horse_no] = {
                                'score': jt_analysis['score'],
                                'rating': jt_analysis['rating'],
                                'jockey': jt_analysis['jockey'],
                                'trainer': jt_analysis['trainer']
                            }
                            self.stdout.write(f"  ➕ Horse {horse_no}: J-T
Score={jt_analysis['score']}")
                        except (ValueError, KeyError) as e:
                            self.stdout.write(self.style.WARNING(f"⚠️ Could not
parse J-T row: {e}"))
                            continue
                break

        if not jt_analysis_data:
            self.stdout.write("⚠️ No jockey-trainer table found or parsed")

        return jt_analysis_data
    # -------------------------
    # Jockey-Trainer Analysis Functions (updated)
    # -------------------------
    def analyze_jockey_trainer_combination(self, html_row):
        """
        Analyze jockey-trainer combination from HTML row and calculate a score
        based on placing percentage and other performance metrics.
```

```
        """
        # Extract data from table cells
        cells = html_row.find_all('td')

        # Debug: print what we're working with
        cell_texts = [cell.get_text(strip=True) for cell in cells]
        self.stdout.write(f"DEBUG: J-T Row cells: {cell_texts}")

        if len(cells) < 9:  # We need at least 9 cells for the basic data
            self.stdout.write(f"DEBUG: Not enough cells ({len(cells)})")
            return None

        try:
            # Extract key information - adjust indices based on actual structure
            horse_number = cells[0].get_text(strip=True)
            trainer = cells[1].get_text(strip=True)  # Based on your HTML: <td>
Trainer</td>
            jockey = cells[2].get_text(strip=True)   # Based on your HTML: <td>
Jockey</td>

            # Performance metrics - adjust indices based on actual positions
            starts = int(cells[3].get_text(strip=True))        # Rns
            first_places = int(cells[4].get_text(strip=True))  # 1st
            second_places = int(cells[5].get_text(strip=True)) # 2nd
            third_places = int(cells[6].get_text(strip=True))  # 3rd
            win_percentage = int(cells[7].get_text(strip=True)) # Win%
            place_percentage = int(cells[8].get_text(strip=True)) # PLC%

            self.stdout.write(f"DEBUG: Parsed - Horse:{horse_number}, Jockey:
{jockey}, Trainer:{trainer}")

        except (ValueError, IndexError) as e:
            self.stdout.write(f"DEBUG: Error parsing J-T row: {e}")
            return None

        # Calculate combination score (weighted formula)
        score = (
            (place_percentage * 0.4) +           # 40% weight to placing percentage
            (win_percentage * 0.3) +             # 30% weight to win percentage
            (min(starts, 50) * 0.1) +            # 10% weight to experience (capped
at 50 races)
            (25 if starts > 10 else 0)           # Bonus for experience
        )

        # Ensure score is within 0-100 range
        score = max(0, min(100, score))

        return {
            'horse_number': horse_number,
            'jockey': jockey,
            'trainer': trainer,
            'combination': f"{jockey} - {trainer}",
            'percentage_placings': place_percentage,
            'starts': starts,
            'wins': first_places,
            'places': first_places + second_places + third_places,
            'win_percentage': win_percentage,
            'place_percentage': place_percentage,
            'score': round(score, 2),
```

```
            'rating': self.get_jt_rating(score)
        }

    def analyze_horse_runs(self, horse):
        """Analyze a horse's past runs and return performance metrics"""
        # Use the new class analysis service
        class_history = self.class_analyzer.analyze_horse_class_history(horse)

        runs = Run.objects.filter(horse=horse).order_by('-run_date')[:4]

        if not runs:
            return {
                'average_position': None,
                'recent_class': None,
                'recent_distance': None,
                'form_rating': 0,
                'consistency': 0,
                'class_history': class_history
            }

        positions = []
        distances = []

        for run in runs:
            try:
                pos = float(run.position) if run.position and
run.position.isdigit() else None
                if pos:
                    positions.append(pos)
                if run.distance:
                    distances.append(run.distance)
            except:
                continue

        avg_position = sum(positions)/len(positions) if positions else None
        most_common_distance = max(set(distances), key=distances.count) if
distances else None

        # Form rating (lower is better)
        form_rating = 0
        if positions:
            form_rating = sum(p * (0.8 ** i) for i, p in enumerate(positions)) /
sum(0.8 ** i for i in range(len(positions)))

        # Consistency (percentage of runs within 2 positions of average)
        if avg_position and len(positions) > 1:
            consistency = sum(1 for p in positions if abs(p - avg_position) <= 2) /
len(positions)
        else:
            consistency = 0

        return {
            'average_position': avg_position,
            'recent_class': class_history.get('average_class_weight'),
            'recent_distance': most_common_distance,
            'form_rating': form_rating,
            'consistency': consistency * 100,  # as percentage
            'class_history': class_history
        }
```

```
    def calculate_horse_score(self, horse):
        """Calculate a comprehensive score for a horse including jockey-trainer"""
        run_analysis = self.analyze_horse_runs(horse)

        # Base score from merit rating
        merit_score = horse.horse_merit or 0

        # Class suitability from the new service
        class_suitability = self.class_analyzer.calculate_class_suitability(horse,
horse.race)

        # Run performance factors
        form_score = 100 - (run_analysis['form_rating'] * 5) if
run_analysis['form_rating'] else 50
        consistency_score = run_analysis['consistency'] or 50

        # Distance suitability (simple version - could be enhanced)
        distance_score = 70  # base
        if run_analysis['recent_distance'] and horse.race.race_distance:
            if run_analysis['recent_distance'] == horse.race.race_distance:
                distance_score = 90

        # Jockey-Trainer score - get from stored analysis data
        jt_score = 50  # Default neutral score
        jt_rating = "Average"
        jockey = horse.jockey or ""
        trainer = horse.trainer or ""

        if hasattr(horse.race, 'jt_analysis_data') and horse.horse_no in
horse.race.jt_analysis_data:
            jt_data = horse.race.jt_analysis_data[horse.horse_no]
            jt_score = jt_data['score']
            jt_rating = jt_data['rating']
            # Use the jockey/trainer from analysis if available (more accurate)
            jockey = jt_data.get('jockey', jockey)
            trainer = jt_data.get('trainer', trainer)

        # Calculate final score with jockey-trainer included
        score = (
            (merit_score * 0.3) +           # Reduced from 0.4 to make room for JT
            (class_suitability * 0.2) +     # New: 20% weight to class suitability
            (form_score * 0.2) +            # Reduced from 0.3
            (consistency_score * 0.15) +    # Reduced from 0.2
            (distance_score * 0.1) +        # Reduced from 0.1
            (jt_score * 0.25)               # New: 25% weight to jockey-trainer
        )

        # Get class trend from the new service
        class_trend = self.class_analyzer.get_class_trend(horse)

        return {
            'horse': horse,
            'score': round(score, 2),
            'merit_score': merit_score,
            'class_score': class_suitability,
            'form_score': round(form_score, 2),
            'consistency_score': round(consistency_score, 2),
            'distance_score': distance_score,
```

```
            'jt_score': jt_score,
            'jt_rating': jt_rating,
            'jockey': jockey,
            'trainer': trainer,
            'run_analysis': run_analysis,
            'class_trend': class_trend
        }

    def rank_horses(self, race):
        """Rank all horses in a race by their calculated scores"""
        horses = Horse.objects.filter(race=race)
        ranked_horses = []

        for horse in horses:
            ranked_horses.append(self.calculate_horse_score(horse))

        # Sort by score descending
        ranked_horses.sort(key=lambda x: x['score'], reverse=True)

        # Add ranking position
        for i, horse in enumerate(ranked_horses, 1):
            horse['rank'] = i

        return ranked_horses

    def print_horse_rankings(self, ranked_horses):
        """Print the horse rankings in a readable format"""
        self.stdout.write("\n🏇 Horse Rankings:")
        self.stdout.write("{:<5} {:<5} {:<20} {:<8} {:<8} {:<8} {:<8} {:<8}
{:<10}".format(
            "Rank", "No", "Name", "Score", "Merit", "Class", "Form", "J-T", "Trend"
        ))
        for horse in ranked_horses:
            self.stdout.write("{:<5} {:<5} {:<20} {:<8.1f} {:<8} {:<8.1f} {:<8.1f}
{:<8.1f} {:<10}".format(
                horse['rank'],
                horse['horse'].horse_no,
                horse['horse'].horse_name[:18],
                horse['score'],
                horse['merit_score'],
                horse['class_score'],
                horse['form_score'],
                horse['jt_score'],
                horse['class_trend']
            ))
    def _save_rankings_to_db(self, race, ranked_horses):
        """Save rankings to the database - clear existing first"""
        # Clear existing rankings for this race
        deleted_count, _ = Ranking.objects.filter(race=race).delete()
        self.stdout.write(f"  🗑️ Cleared {deleted_count} existing rankings")

        # Create new rankings
        for horse_data in ranked_horses:
            ranking_data = {
                'race': race,
                'horse': horse_data['horse'],
                'score': horse_data['score'],
                'class_score': horse_data['class_score'],
                'rank': horse_data['rank'],
```

```
                'jt_score': horse_data.get('jt_score'),
                'jt_rating': horse_data.get('jt_rating'),
                'jockey': horse_data.get('jockey'),
                'trainer': horse_data.get('trainer'),
                'class_trend': horse_data.get('class_trend'),
            }
            Ranking.objects.create(**ranking_data)

        self.stdout.write(f"  ✅ Created {len(ranked_horses)} new rankings")

    # In your Command class - Update the _calculate_advanced_scores method
    def _calculate_advanced_scores(self, race):
        """Calculate and store advanced HorseScore records"""
        from racecard.services.scoring_service import HorseScoringService

        self.stdout.write("\n[STEP 9] Calculating advanced horse scores...")

        horses = Horse.objects.filter(race=race)
        scores_created = 0
        scores_updated = 0

        for horse in horses:
            try:
                scoring_service = HorseScoringService(horse, race)
                score_record, created = scoring_service.create_score_record()

                action = "Created" if created else "Updated"
                if created:
                    scores_created += 1
                else:
                    scores_updated += 1

                self.stdout.write(
                    f"  📊 {horse.horse_name}: {action} score -
Overall={score_record.overall_score:.3f} "
                    f"(M:{score_record.merit_score:.3f}, F:
{score_record.form_score:.3f}, "
                    f"D:{score_record.distance_score:.3f}, C:
{score_record.consistency_score:.3f})"
                )

            except Exception as e:
                self.stdout.write(self.style.WARNING(f"⚠️ Could not calculate score
for {horse.horse_name}: {e}"))

        self.stdout.write(self.style.SUCCESS(
            f"✅ Advanced scores processed: {scores_created} created,
{scores_updated} updated"
        ))
        return scores_created + scores_updated
    # -------------------------
    # Main handler
    # -------------------------
    def handle(self, *args, **options):
        html_file = options["html_file"]
        update_existing = options["update"]

        # Step 1: file existence
        self.stdout.write(f"\n[STEP 1] Checking file: {html_file}")
```

```
        if not os.path.exists(html_file):
            self.stdout.write(self.style.ERROR("❌ File not found. Aborting."))
            return
        self.stdout.write(self.style.SUCCESS("✅ File exists."))

        # Step 2: load HTML
        self.stdout.write("\n[STEP 2] Loading and parsing HTML...")
        with open(html_file, "r", encoding="utf-8") as fh:
            soup = BeautifulSoup(fh, "html.parser")
        self.stdout.write(self.style.SUCCESS("✅ HTML loaded into BeautifulSoup."))

        # Step 3: parse header (left) td
        self.stdout.write("\n[STEP 3] Extracting header block
(course/date/no/time)...")
        header = self._parse_header_td(soup)

        self.stdout.write(f"  • Raw header lines: {header.get('lines')}")
        self.stdout.write(f"  • Course/Field: {header.get('course')!r}")
        self.stdout.write(f"  • Date text: {header.get('date_text')!r} -> Parsed:
{header.get('race_date')!r}")
        self.stdout.write(f"  • Race No (rev4): {header.get('race_no')!r}")
        self.stdout.write(f"  • Race time raw: {header.get('race_time_text')!r} ->
HHMM: {header.get('race_time_hhmm')!r}")

        essential_ok = all([
            bool(header.get("course")),
            bool(header.get("race_date")),
            header.get("race_no") is not None,
        ])
        if not essential_ok:
            self.stdout.write(self.style.ERROR("❌ Missing essential header info
(course/date/race_no). Aborting."))
            return
        self.stdout.write(self.style.SUCCESS("✅ Header looks good."))

        # Step 4: parse right td for name/distance/class/merit
        self.stdout.write("\n[STEP 4] Extracting race details
(name/distance/class/merit)...")
        left_td = soup.find("td", align="center")
        right_td = left_td.find_next_sibling("td") if left_td else None
        details = self._parse_right_td_for_details(right_td)

        self.stdout.write(f"  • race_name: {details.get('race_name')!r}")
        self.stdout.write(f"  • race_distance: {details.get('race_distance')!r}")
        self.stdout.write(f"  • race_class: {details.get('race_class')!r}")
        self.stdout.write(f"  • race_merit: {details.get('race_merit')!r}")
        self.stdout.write(self.style.SUCCESS("✅ Details extracted."))

        # Step 5: prepare DB fields (match your Race model fields)
        race_date = ensure_date(header["race_date"])
        race_no = int(header["race_no"])
        race_time_obj = ensure_time(header["race_time_hhmm"])  # -> datetime.time
        race_field = header["course"].strip()
        race_name = details.get("race_name") or ""
        race_distance = details.get("race_distance") or ""
        race_class = details.get("race_class") or ""
        race_merit = details.get("race_merit") or 0

        self.stdout.write("\n[STEP 5] Prepared DB values:")
```

```
        self.stdout.write(f"  • race_date={race_date}, race_no={race_no},
race_time={race_time_obj}")
        self.stdout.write(f"  • race_field={race_field!r}, race_name={race_name!
r}")
        self.stdout.write(f"  • race_distance={race_distance!r},
race_class={race_class!r}, race_merit={race_merit!r}")

        # Step 6: write Race to DB
        self.stdout.write("\n[STEP 6] Writing Race to database...")
        try:
            race, created = Race.objects.get_or_create(
                race_date=race_date,
                race_no=race_no,
                race_field=race_field,
                defaults={
                    "race_time": race_time_obj,
                    "race_name": race_name,
                    "race_distance": race_distance,
                    "race_class": race_class,
                    "race_merit": race_merit,
                },
            )

            if not created and update_existing:
                race.race_time = race_time_obj
                race.race_name = race_name
                race.race_distance = race_distance
                race.race_class = race_class
                race.race_merit = race_merit
                race.save()
                self.stdout.write(self.style.SUCCESS("🔄 Updated existing Race
(because --update was used)."))
            elif created:
                self.stdout.write(self.style.SUCCESS("✅ Created new Race row."))
            else:
                self.stdout.write("ℹ️ Race already exists (same date/no/field).")

            self.stdout.write(
                f"  id={race.id} | date={race.race_date} | no={race.race_no} | "
                f"field={race.race_field} | time={race.race_time} | "
                f"name={race.race_name!r} | distance={race.race_distance!r} | "
                f"class={race.race_class!r} | merit={race.race_merit!r}"
            )
        except Exception as e:
            self.stdout.write(self.style.ERROR(f"❌ DB write failed (Race): {e}"))
            return

        # Step 7: write Horses to DB
        self._parse_horses(soup, race, update_existing)

        # Step 8: Calculate and store rankings
        self.stdout.write("\n[STEP 8] Calculating horse rankings...")
        try:
            ranked_horses = self.rank_horses(race)
            self.print_horse_rankings(ranked_horses)
            self._save_rankings_to_db(race, ranked_horses)
            self.stdout.write(self.style.SUCCESS("✅ Rankings calculated and
stored."))
        except Exception as e:
```

```
            self.stdout.write(self.style.ERROR(f"❌ Error calculating rankings:
{e}"))
            import traceback
            self.stdout.write(traceback.format_exc())

        self.stdout.write(self.style.SUCCESS("\n✅ Done. Racecard import
finished."))

        # Step 9: Calculate advanced scores for AI
        self._calculate_advanced_scores(race)
```