DANIEL SHAW

2023

CM3035 MIDTERM

SHIP PASSENGER ANALYSIS

For my Mid-Term coursework I've decided to create a RESTAPI that allows a user to view and interpret various interesting data points from the Titanic Passenger Dataset from Kaggle - Titanic - Will Cukierski. (2012). Machine Learning from Disaster, https://www.kaggle.com/competitions/titanic/data. This dataset consisted of two primary .csv files a test.csv and a train.csv. For the purpose of this project I will only be using the train dataset as it contains all of the passengers and is already cleaned as it was meant to be used to train AI to predict wether or not a random person would survive. Hence the test data set containing randomly generated people who weren't actually on the titanic is not used.

This data caught my attention as it includes data on the ultimate fate of the passengers aboard the ship after it crashed. This piqued my curiosity and made me want to find out more about the people aboard the titanic that day.

The columns of the dataset are as follows:

survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	

parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

Notes :

OS: Ventura 13.0.1 M1 MAX

Python: 3.10

Admin:

Username: dansh

Password: test

DEVELOPMENT

In this section I will be discussing various points in relation to the development of this program such as the various endpoints and queries used. I will talk about what makes the endpoints work and why I believe that they're interesting.

I've created two sides to the API, one being a web-facing html version that returns webpages with css styling to present data in a nice and readable way, I then created a JSON version of the API where a client is able to access all of the same features as the HTML version programmatically.

APPLICATION STRUCTURE

The application is called TitanicDataRest with an application module called Main inside it. The Main module does all of the work as it contains all of the models and views inside, whilst the TitanicDataRest contains the administration interface.

SEEDING THE DATABASE

Seeding the database is done through the DataImport.py script this program loads a .csv file (titanic.csv), creates a table for the data to be inserted in, then it reads the .csv file and inserts all of the records into the DB. It then prints a message to let the user know that it's finished successfully.

THE MODEL

My only model is the TitanicPassenger model. Each passenger is defined through this model. I migrated the models by running the following command: manage.py migrate Main

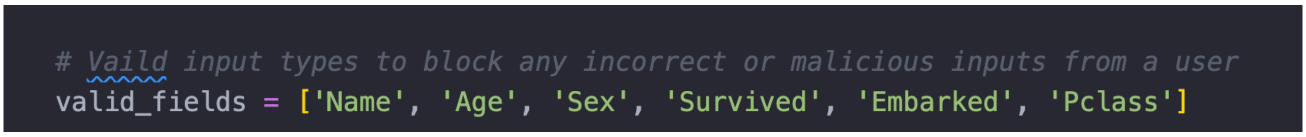
HTML VERSION OF API (MVC)

PASSENGERS ENDPOINT (GET)

I started development with the most simple endpoint to develop, this being the passengers endpoint as all it does is gather all of the records and display them in a table for a user to view. This endpoint calls upon the TitanicPassenger object to retrieve all of the data. It calls the objects.all() method which returns all objects in the table. This is then displayed through the passenger_list.html template. This endpoint caters to GET requests. The RestAPI version of this endpoint serialises the data and returns it as a JSON response.

SEARCH-PASSENGERS ENDPOINT (POST)

This endpoint allows a user to filter passengers by various fields. Name, Age, Sex, Survived, Boarded Class. The interface consists of a dropdown field selection, a text input and a search button. The development behind this endpoint was considered carefully as we are taking data input from a user which has to be assumed to be unsafe so proper filtering and data validation was necessary. Since Django already handles database interfacing not much validation was needed, I made an array of accepted filter types to block any incorrect or malicious requests.

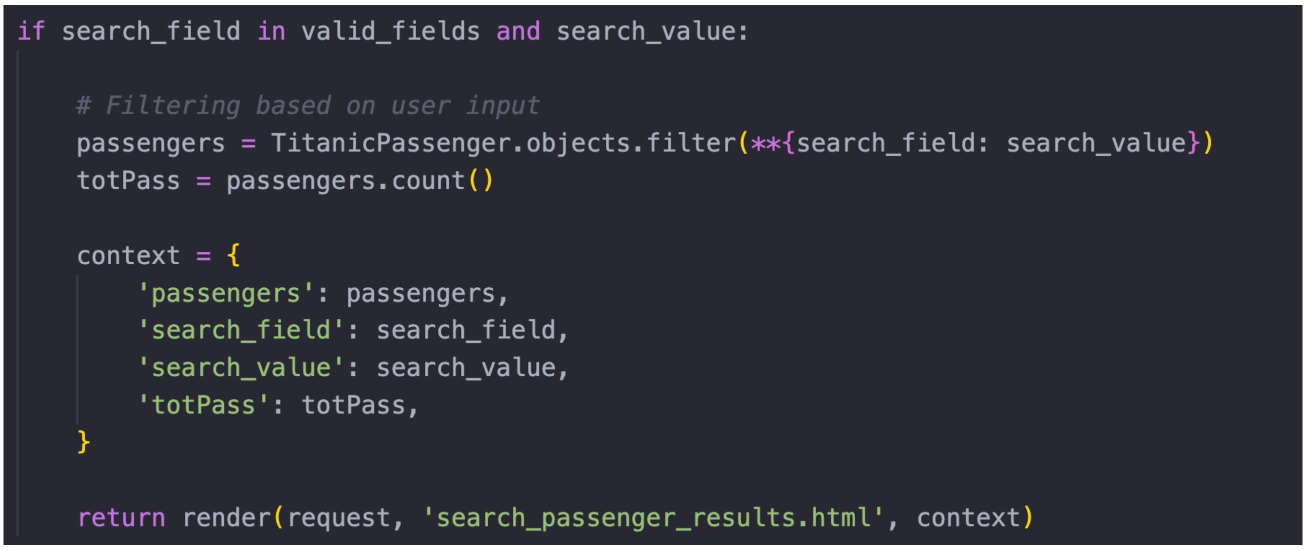


```python
# Vaild input types to block any incorrect or malicious inputs from a user
valid_fields = ['Name', 'Age', 'Sex', 'Survived', 'Embarked', 'Pclass']
```

views.py line 82.

This validation was achieved by the pictured above array of accepted terms and a simple IF statement.

We then define a passengers var and query the database using Django's ORM



```python
if search_field in valid_fields and search_value:

    # Filtering based on user input
    passengers = TitanicPassenger.objects.filter(**{search_field: search_value})
    totPass = passengers.count()

    context = {
        'passengers': passengers,
        'search_field': search_field,
        'search_value': search_value,
        'totPass': totPass,
    }

    return render(request, 'search_passenger_results.html', context)
```

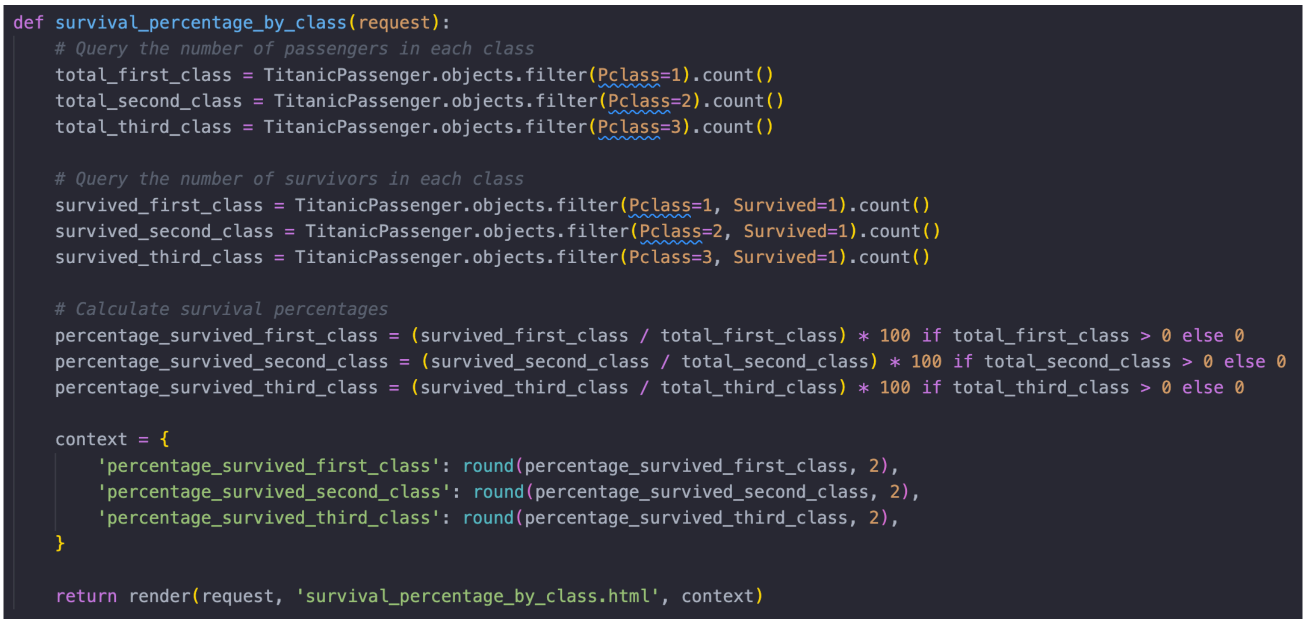
views.py line 84-97

We then count the total returned passengers to return to the user as a datapoint which can be used to quickly extract data such as quantity of males on board, quantity of passengers aged 10 etc…

We then form a data object called context to pass to the interface passing the information of all of the passengers that fit the criteria, the search field used and the search value used also including the total people returned from the query.

SURVIVAL-PERCENTAGE-BY-CLASS (GET)

This endpoint provides insightful information on the survival of passengers of different class, this can tell us a lot about how the ship was organised in the event of an emergency which as you can see results in a big disparity between classes of ticket.



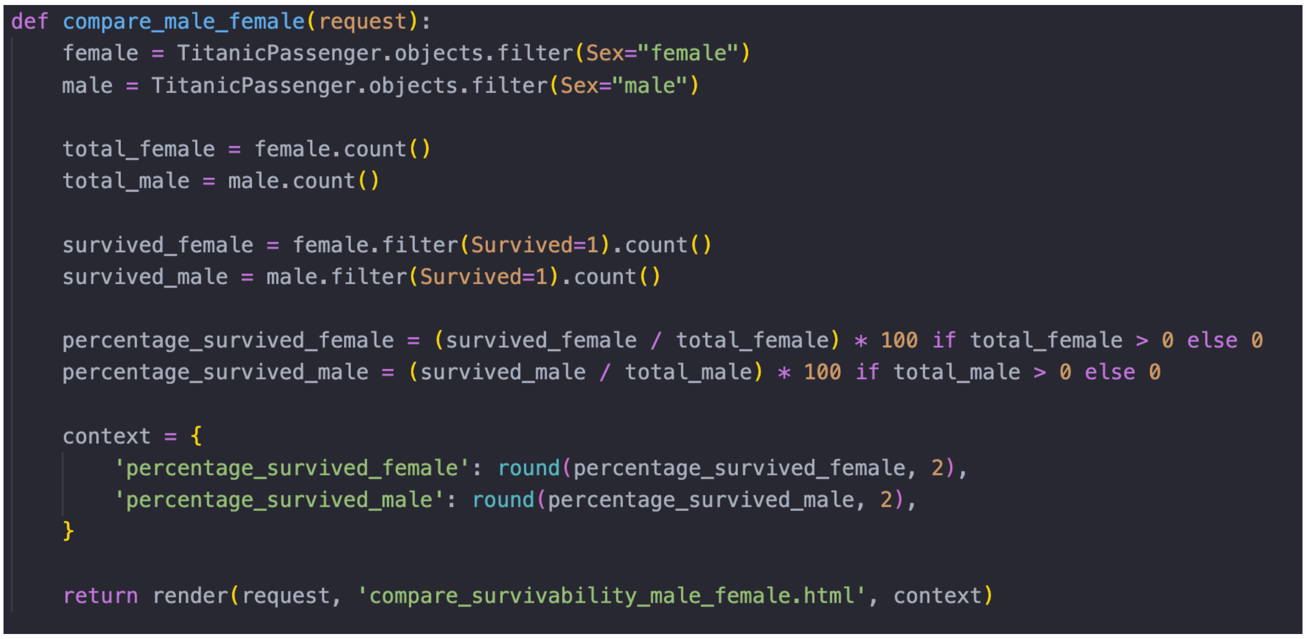
```python
def survival_percentage_by_class(request):
    # Query the number of passengers in each class
    total_first_class = TitanicPassenger.objects.filter(Pclass=1).count()
    total_second_class = TitanicPassenger.objects.filter(Pclass=2).count()
    total_third_class = TitanicPassenger.objects.filter(Pclass=3).count()

    # Query the number of survivors in each class
    survived_first_class = TitanicPassenger.objects.filter(Pclass=1, Survived=1).count()
    survived_second_class = TitanicPassenger.objects.filter(Pclass=2, Survived=1).count()
    survived_third_class = TitanicPassenger.objects.filter(Pclass=3, Survived=1).count()

    # Calculate survival percentages
    percentage_survived_first_class = (survived_first_class / total_first_class) * 100 if total_first_class > 0 else 0
    percentage_survived_second_class = (survived_second_class / total_second_class) * 100 if total_second_class > 0 else 0
    percentage_survived_third_class = (survived_third_class / total_third_class) * 100 if total_third_class > 0 else 0

    context = {
        'percentage_survived_first_class': round(percentage_survived_first_class, 2),
        'percentage_survived_second_class': round(percentage_survived_second_class, 2),
        'percentage_survived_third_class': round(percentage_survived_third_class, 2),
    }

    return render(request, 'survival_percentage_by_class.html', context)
```

Views.py line 52-74

We first started by querying and counting the total amounts of people from each class. We then proceeded to query the total amount of passengers that survived from each class, the percentages were then calculated and before being sent to the user the percentages were rounded to 2DP for ease of interpretation.

COMPARE-MALE-FEMALE (GET)

I've made this endpoint to explore the differences between the survival of men and women in the disaster of the titanic. The results interested me as the gap in-between the males and females is really surprising. Where 74.2% of women survived only 18.89% of males survived. Whilst women and children would usually be loaded onto lifeboats first the large gap in-between the two genders really surprised me.



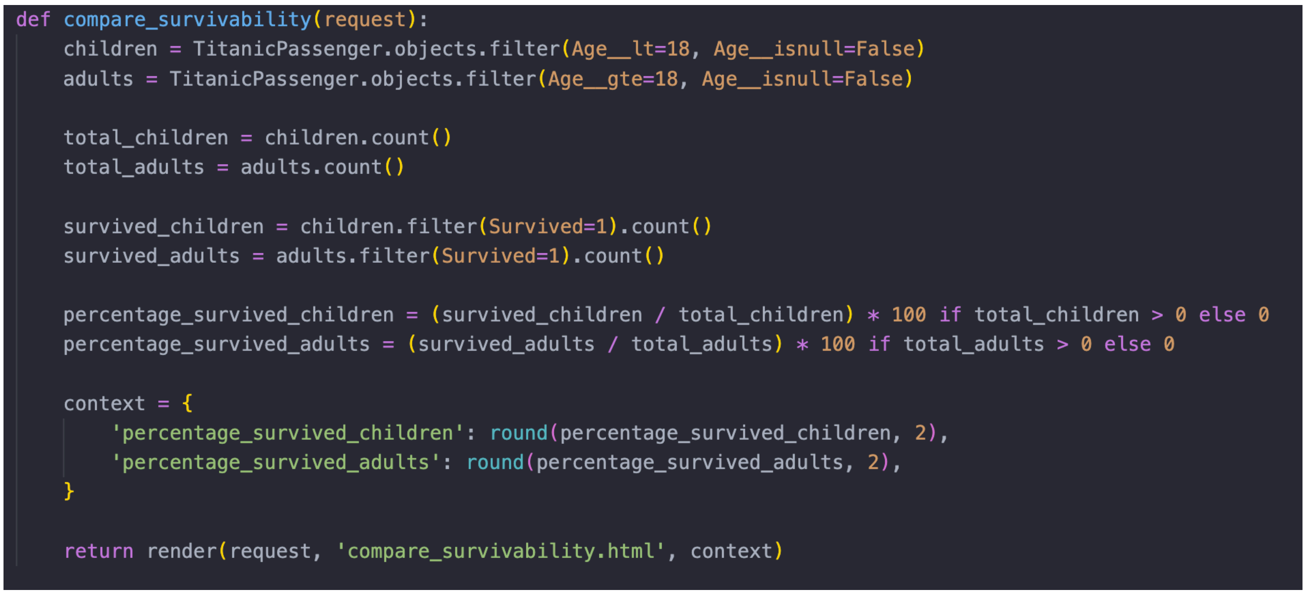
```python
def compare_male_female(request):
    female = TitanicPassenger.objects.filter(Sex="female")
    male = TitanicPassenger.objects.filter(Sex="male")

    total_female = female.count()
    total_male = male.count()

    survived_female = female.filter(Survived=1).count()
    survived_male = male.filter(Survived=1).count()

    percentage_survived_female = (survived_female / total_female) * 100 if total_female > 0 else 0
    percentage_survived_male = (survived_male / total_male) * 100 if total_male > 0 else 0

    context = {
        'percentage_survived_female': round(percentage_survived_female, 2),
        'percentage_survived_male': round(percentage_survived_male, 2),
    }

    return render(request, 'compare_survivability_male_female.html', context)
```

views.py lines 32-50

We first count the total amount of female and male passengers aboard. Then we count the total amount of females and males that survived respectively. A percentage is then calculated and is then rounded to 2DP for ease of interpretation.

COMPARE-SURVIVABILITY(GET)

This function was created to determine the survivability between passengers over the age of 18 and passengers under the age of 18. I thought that this would allow us to gauge the difference that age and experience had on survival rates. Whilst 53.98% of children aboard survived only 38.1% of adults survived this is due to children being loaded onto lifeboats first with the females. However these numbers also put into



```python
def compare_survivability(request):
    children = TitanicPassenger.objects.filter(Age__lt=18, Age__isnull=False)
    adults = TitanicPassenger.objects.filter(Age__gte=18, Age__isnull=False)

    total_children = children.count()
    total_adults = adults.count()

    survived_children = children.filter(Survived=1).count()
    survived_adults = adults.filter(Survived=1).count()

    percentage_survived_children = (survived_children / total_children) * 100 if total_children > 0 else 0
    percentage_survived_adults = (survived_adults / total_adults) * 100 if total_adults > 0 else 0

    context = {
        'percentage_survived_children': round(percentage_survived_children, 2),
        'percentage_survived_adults': round(percentage_survived_adults, 2),
    }

    return render(request, 'compare_survivability.html', context)
```

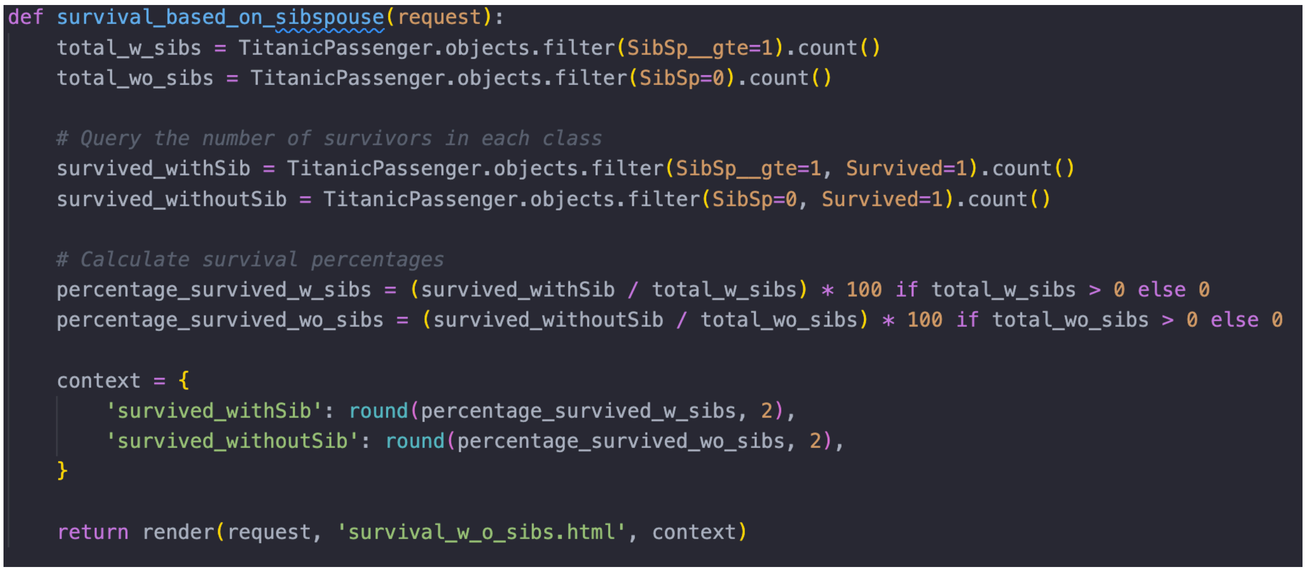
perspective the danger and harsh conditions that the passengers faced on that fateful day.

views.py lines 12-30

The function first collects all of the children and adults, the query works using Django's operators (lt - less than, gte - greater than or equal to). The totals are then gathered and it then filters out the survivors of children and adults and counts them, after that a percentage is calculated and is rounded to 2DP for ease of interpretation.

SURVIVAL-BASED-ON-SIBSPOUSE (GET)

I then decided to determine wether families containing siblings and spouses had a higher chance of surviving than ones that don't. Whilst the differences are minimal the results are interesting as families definitely survived more coming in at 46.64% survival rates. Whilst passengers that were travelling solo had a survival rate of 34.54%



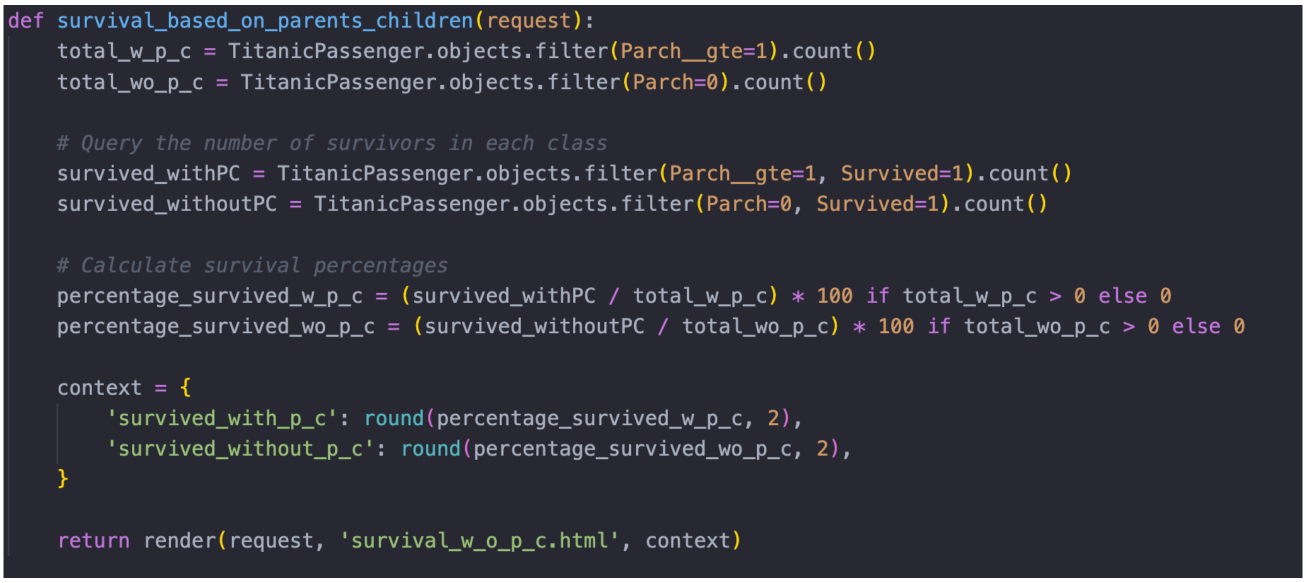
```python
def survival_based_on_sibspouse(request):
    total_w_sibs = TitanicPassenger.objects.filter(SibSp__gte=1).count()
    total_wo_sibs = TitanicPassenger.objects.filter(SibSp=0).count()

    # Query the number of survivors in each class
    survived_withSib = TitanicPassenger.objects.filter(SibSp__gte=1, Survived=1).count()
    survived_withoutSib = TitanicPassenger.objects.filter(SibSp=0, Survived=1).count()

    # Calculate survival percentages
    percentage_survived_w_sibs = (survived_withSib / total_w_sibs) * 100 if total_w_sibs > 0 else 0
    percentage_survived_wo_sibs = (survived_withoutSib / total_wo_sibs) * 100 if total_wo_sibs > 0 else 0

    context = {
        'survived_withSib': round(percentage_survived_w_sibs, 2),
        'survived_withoutSib': round(percentage_survived_wo_sibs, 2),
    }

    return render(request, 'survival_w_o_sibs.html', context)
```

views.py lines 103-120

We first filter out people who have siblings/spouses and people who don't. I used the gte operator to account for people who have more than 1 spouse. The survivors are then counted for each respectively and then the percentages are calculated and rounded accordingly before being returned by the API.

SURVIVAL-BASED-ON-PARENTS-CHILDREN (GET)

This endpoint gives us insight onto wether people travelling with parents or children had a higher percentage of survival as to people who weren't. The results of this were quite interesting as people with parents or children aboard survived 51.17% of the time whilst people without parents or children aboard survived only 34.37%. This is a notable difference and most likely stems from the fact that people with children are usually women.



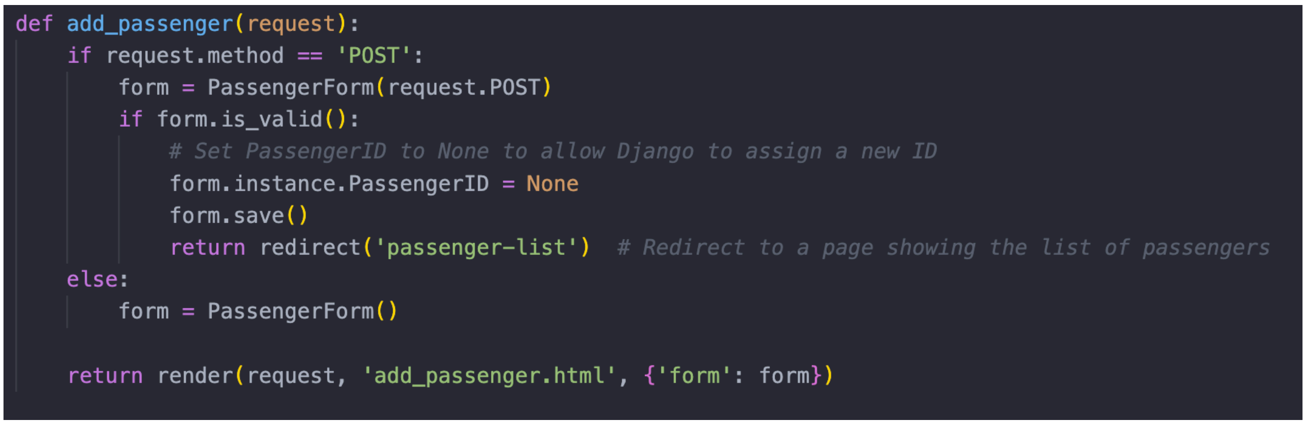
```python
def survival_based_on_parents_children(request):
    total_w_p_c = TitanicPassenger.objects.filter(Parch__gte=1).count()
    total_wo_p_c = TitanicPassenger.objects.filter(Parch=0).count()

    # Query the number of survivors in each class
    survived_withPC = TitanicPassenger.objects.filter(Parch__gte=1, Survived=1).count()
    survived_withoutPC = TitanicPassenger.objects.filter(Parch=0, Survived=1).count()

    # Calculate survival percentages
    percentage_survived_w_p_c = (survived_withPC / total_w_p_c) * 100 if total_w_p_c > 0 else 0
    percentage_survived_wo_p_c = (survived_withoutPC / total_wo_p_c) * 100 if total_wo_p_c > 0 else 0

    context = {
        'survived_with_p_c': round(percentage_survived_w_p_c, 2),
        'survived_without_p_c': round(percentage_survived_wo_p_c, 2),
    }

    return render(request, 'survival_w_o_p_c.html', context)
```

views.py lines 122-139

First we gather the people that have parents and children aboard and ones that don't, they are counted and then the survivors are counted respectively. Percentages are calculated and rounded accordingly.

ADD-PASSENGER (POST)

I've added this endpoint in the case that someone would like to use this tool to possibly update the passenger list of the titanic or in the case that a different dataset is loaded into the program where a user might want to add more entries to complete it.



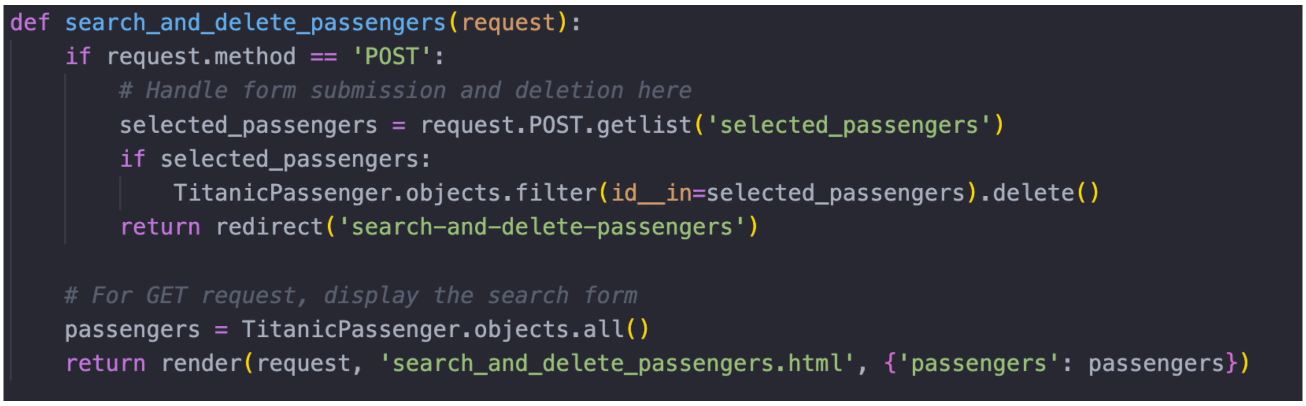
```python
def add_passenger(request):
    if request.method == 'POST':
        form = PassengerForm(request.POST)
        if form.is_valid():
            # Set PassengerID to None to allow Django to assign a new ID
            form.instance.PassengerID = None
            form.save()
            return redirect('passenger-list')  # Redirect to a page showing the list of passengers
    else:
        form = PassengerForm()

    return render(request, 'add_passenger.html', {'form': form})
```

Views.py lines 143-153

For this endpoint we start by checking if the form is indeed valid - we then save the form and redirect the user to the passenger-list page where they are able to view the new addition to the DB.

SEARCH-AND-DELETE-PASSENGERS (POST/GET)

This endpoint was also added to allow a potential user to remove any entries that are incorrect or extra.



```python
def search_and_delete_passengers(request):
    if request.method == 'POST':
        # Handle form submission and deletion here
        selected_passengers = request.POST.getlist('selected_passengers')
        if selected_passengers:
            TitanicPassenger.objects.filter(id__in=selected_passengers).delete()
        return redirect('search-and-delete-passengers')

    # For GET request, display the search form
    passengers = TitanicPassenger.objects.all()
    return render(request, 'search_and_delete_passengers.html', {'passengers': passengers})
```

Views.py lines 155-165

Upon a POST request being received it gets a list of selected passengers and deletes them. Upon GET request being received it returns all passengers in a list for selection. This allows a user to first select passengers that they wish to delete and upon completion of the form and submit being hit the POST request deletes selected passengers.

DESCRIPTION OF THE REST API (JSON)

Since I wanted the API to be accessible by a user with just a web-browser I created a HTML user-friendly set of endpoints, however I also wanted to allow for someone to access the API programmatically to enable creation of custom UI. All of the endpoints are the same in their principal function however passing values differs between the HTML API and the JSON one as instead of taking input through HTML I had to take it as Query Params. Serialisation had to be used and as such I implemented a serialiser called TitanicPassengerSerializer. I use it extensively whenever I need to serialise any passenger objects into JSON. I also use it to deserialise upon addition of a passenger.

UNIT-TESTING

Any web project needs a good way of testing the functionality to confirm its implementation. Whilst I'm not completely familiar with Django's testing framework I have implemented quite a few tests that check for the basic functionalities operation. However - during my development of this API I've used postman extensively as that is what I'm most familiar with. Through postman I've confirmed the functionality of all of the endpoints that I've created.

REQUIREMENTS AND HOW'VE THEY BEEN MET

During the development of this coursework I've paid attention to details such as input validation and I've maximised the usefulness of the data through my application. I feel that it gives an insightful look into the sinking of the titanic and the reality of the fate of those on board. I found the research behind the project fun and rewarding. The research behind the disaster infatuated me and That is why I ultimately chose this dataset for development and evaluation. All kinds of endpoints are covered and I've followed and applied what we've learnt up until now in my program. The code is clean and non-verbose. Serialisers are used and unit testing and factory boy is used. The API is both an HTML one and a JSON one this allows for maximum potential modularity as any REST client would be able to consume my API with ease.

THE STATE OF THE ART

Whilst Django is widely used in the Web-Development space it is considered an MVC type of framework. Whilst this isn't a bad thing when it comes to the state of the art in web development we use a new form of development called "single-page-applications" which are client sided. Django is server sided in its rendering meaning that the web server returns HTML pages. Django works perfectly as a backend only framework and can be used in conjunction with react. React runs completely on JavaScript (or TypeScript) and uses something called JSX (or TSX) to integrate JavaScript with HTML code for fully dynamic and functional applications. This makes for a more separated approach to development which encourages higher levels of modularity and fully separates the front end and back end. All of my experience is in the SPA space and this is my first time using a MVC framework. Whilst I found the ORM to be very convenient and the passing of data around to be easier. I do prefer development using a SPA framework such as React and the many others that are available.

Thank you for taking the time to consider my coursework!