**Kellerman Kinase Database Documentation**

Contents

[Overview 2](#_Toc1037037)

[Data collection and Database Structure 3](#_Toc1037038)

[Kinases 3](#_Toc1037039)

[Phophosites 3](#_Toc1037040)

[Inhibitors 3](#_Toc1037041)

[Database schema 4](#_Toc1037042)

[Software Structure 4](#_Toc1037043)

[Software Strucutre 5](#_Toc1037044)

[Run.py 5](#_Toc1037045)

[App.py 5](#_Toc1037046)

[Config.py 6](#_Toc1037047)

[Procfile 6](#_Toc1037048)

[Requirements.txt 6](#_Toc1037049)

[.gitignore 6](#_Toc1037050)

[Common Folder 7](#_Toc1037051)

[Models Folder 7](#_Toc1037052)

[Templates Folder 10](#_Toc1037053)

[Static Folder 10](#_Toc1037054)

[Upload and Download Folders 11](#_Toc1037055)

[The Analysis Code 11](#_Toc1037056)

[Limitations and Next Steps 11](#_Toc1037057)

[Technologies 13](#_Toc1037058)

# Overview

The Kellerman Kinase application is an interactive searchable database. It provides the user with background information about protein kinases such as their gene name, the families they belong to, the cell location and their target phosphorylation sites. As well as target phosphorylation sites it also provides the user with the genomic location and neighbouring sequences of their respective phosphosites. Inhibitor information for each kinase is retrieved and this database allows you to search known inhibitors on the inhibitor page, displaying their name, chemical structure and respective kinase they are known to inhibit. Furthermore, this database contains an analysis section allowing the user to upload their own phosphoproteomics data. The analysis section calculates and graphically presents the relative active of each kinase in the sample data.

The protein kinase and phosphosite data was collected through UniProt API and inhibitor data was collected from the MRC Interational Center of Kinase Profiling. The collected data was populated into a PostgreSQL database in PgAdmin4. The application itself was built through Flask in Python 2.7 and was deployed using Heroku which is a cloud based development platform as service (PaaS). This platform provides cloud-based recovery and back-up options. It allows a consistent and continuous GitHub integration, while the infrastructure is handled by Heroku itself. This means that deployment of a new version is achieved by a GitHub push. There is no need to change infrastructure settings on Heroku. Heroku allows horizontal, vertical or automatic scaling which allows the user to use as much resources as needed for the application. Database configuration, and add-on management is generally a single line command or a ‘click’ on the dashboard, without having to configure anything else. Heroku is set up to be able to deploy rapidly and efficiently. Our goal was to develop the app continuously, therefore the ease of deployment was our driving reason when choosing a deployment platform. Heroku’s official database recommendation is PostgreSQL. Heroku Postgres database is fully supported, managed and developed to be fully integrated into the Heroku platform. To set up a PostgreSQL database on Heroku is extremely easy and yet highly maintained and secure. All Heroku supported languages allow easy access to the database (Ruby, Java, PHP, Python, Node, Go, Scala and Clojure). Heroku allows forking on databases and let users to be read only followers of a database. The database is constantly monitored for protection and health checks. Using this system allows full recovery of the database in an event of hardware failure.

# Data collection and Database Structure

## Kinases

Collected through HumanKinome – downloaded as a CSV file. To circumvent the problem of alias’ preventing successful database searches, we downloaded a list of their known alias’ from NCBI. This was then allocated alongside the appropriate kinase.

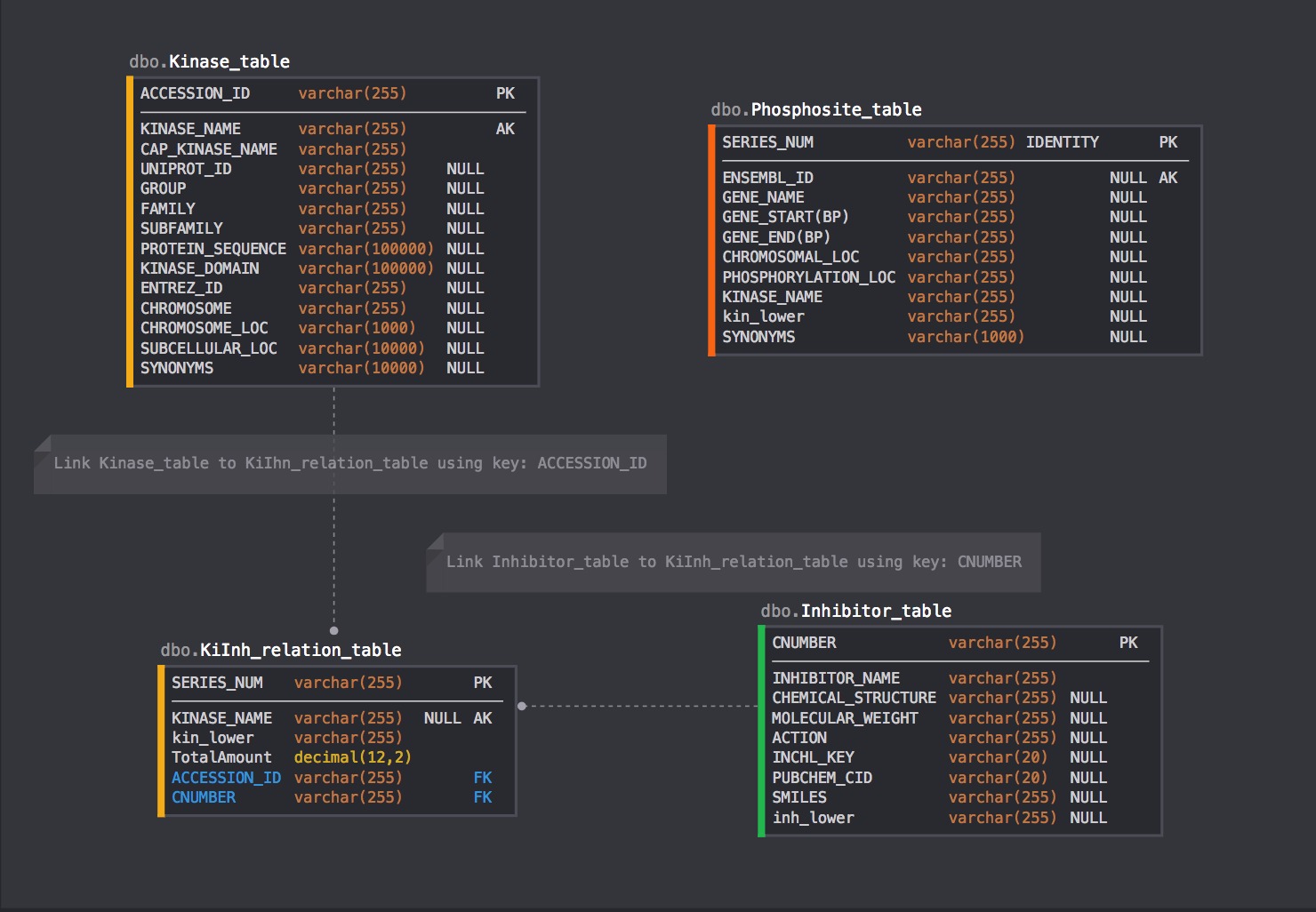
## Phophosites

Firstly, a list of the human proteome including accession IDs recorded on Uniprot was downloaded; this was then run through a loop that accessed Uniprot’s API service which downloaded information that is specific to the post-translational modifications of the given protein, this information was saved into an intermediate CSV file. Secondly, the CSV file was loaded into Rstudio and the Bioconductor: biomaRt package was installed. Resultantly, we accessed BiomaRts API and downloaded our selected attributes from the website. In this case, we downloaded genomic information on each protein. The first result file from the Uniprot access and the second from the BiomaRt access were looped together via python, a final CSV file was output.

## Inhibitors

Inhibitor information was collected from MRC International Center of Kinase Profiling. 239 inhibitors are listed in total with references annotated for each inhibitor that reviewed by the MRC Protein Phosphorylation Unit at the University of Dundee.

## Database schema

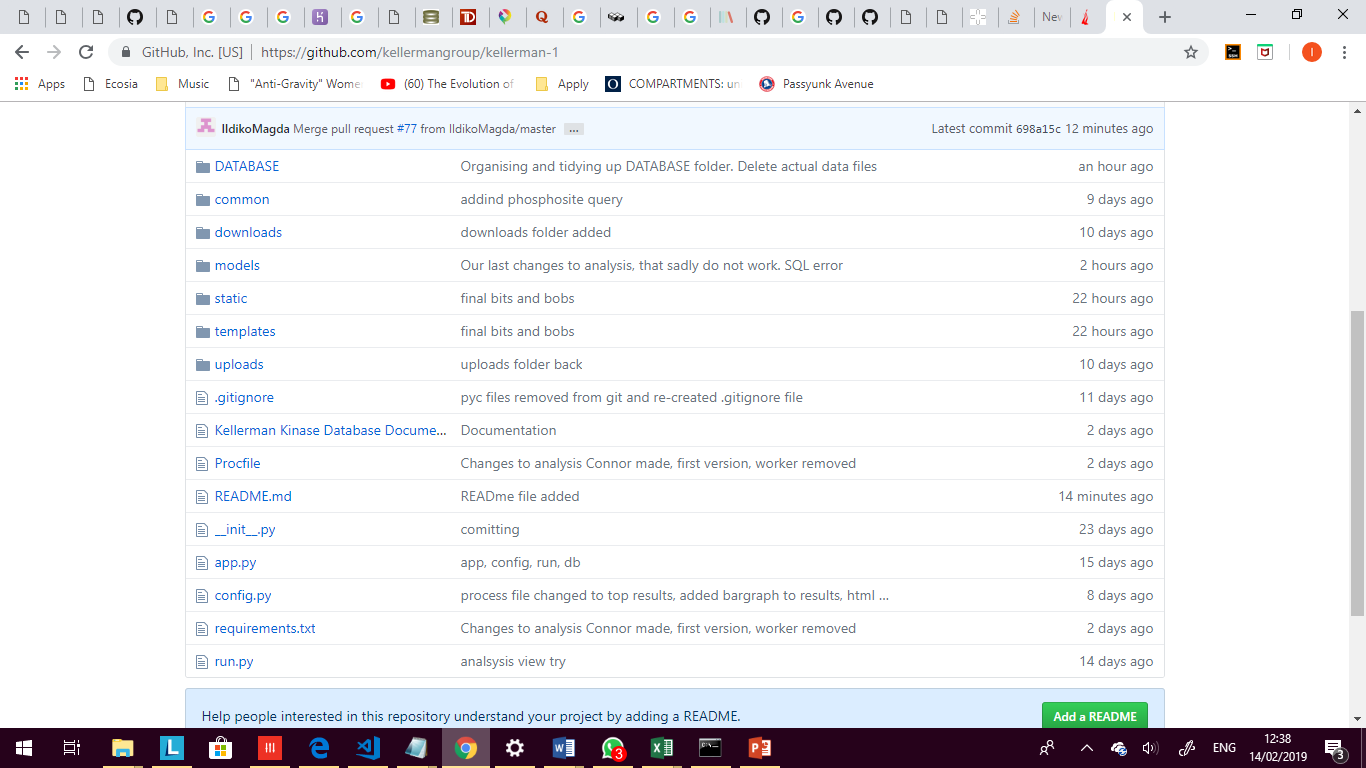
The database was curated and administered in PgAdmin4. The SQL tree to show the tables is found below. 

The four tables are built to store the whole dataset and three separate tables are shown as: kinase information, kinase phosphorylation information and kinase inhibitors information. One relation table called KiInh\_relation\_table was set up as a bridge to link the information in Kinase\_table and Inhibitor\_table. This table uses ACCESSION\_ID as foreign key to match with the same ACCESSION\_ID from Kinase\_table and CNUMBER(aka the inhibitor id) as foreign key to match with the same CNUMBER from the Inhibitor\_table. The ACCESSION\_ID and CNUMBER are set as primary keys in Kinase\_table and Inhibitor\_table respectively. This table records the ACCESSION\_ID and CNUMBER in pairs to link each kinase with several inhibitors and each inhibitor to their target kinases.

For simplicity, we use a series number as the primary key for Phosphosite\_table due to a primary key uniqueness constraints.

# Software Structure

The software was structured through organised folders and files stored in a GitHub repository. Below shows a screenshot of the software directory taken from GitHub. In each folder, there is a respective \_\_init\_\_.py file to mark directories as python packages when the app is running and should remain empty.



## Software Design

### Run.py

The file runs the app either locally or deployed. It imports the operating system (os) to set the environment to run through a port and it also imports the app to apply debug mode.

""" This File is used to run the app deployed or locally"""

import os

from app import APP

# run deployed

port = int(os.environ.get('PORT', 33506))

APP.run(debug=APP.config['DEBUG'], use\_reloader=APP.config['RELOADER'], port=port, host='0.0.0.0')

### App.py

The app.py file imports the Flask module and render template. The application is created by applying a variable to the Flask module and contains the application home route index in which renders the home index template. Moreover, it imports Flask blueprints which connects all the route pages through the models folder to the respective views.py files. Flask blueprints were favoured over having each route on the app page as it was easier to manage each route as the inhibitor and kinase pages varied very differently to analysis.

# import Flask and render template

from flask import Flask, render\_template

# import blueprints from models folder

from models.kinase.views import kinase\_blueprint

from models.inhibitor.views import inhibitor\_blueprint

from models.analysis.views import analysis\_blueprint

# create a flask application

APP = Flask(\_\_name\_\_)

APP.config.from\_object('config')

# define action of top level routes

@APP.route("/")

def index():

""" Description: Home Route that renders the index template """

return render\_template('index.html')

# register other route blueprints

APP.register\_blueprint(kinase\_blueprint, url\_prefix="/kinase")

APP.register\_blueprint(inhibitor\_blueprint, url\_prefix="/inhibitor")

APP.register\_blueprint(analysis\_blueprint, url\_prefix="/analysis")

### Config.py

The config file sets the configurations needed for the app.

""" Define All essential configurations we will need for the app itself"""

import os

DEBUG = True # Turns on debugging features in Flask

RELOADER = True

UPLOAD\_FOLDER = os.path.dirname(os.path.abspath(\_\_file\_\_)) + '/uploads'

DOWNLOAD\_FOLDER = os.path.dirname(os.path.abspath(\_\_file\_\_)) + '/downloads'

STATIC\_FOLDER = os.path.dirname(os.path.abspath(\_\_file\_\_)) + '/static'

### Procfile

The Procfile is linked to Heroku and is required to specify the commands to execute on the app start up. It tells it to use python and execute the run.py file.

### Requirements.txt

Requirements necessary for the app to run. These include Flask, psycopg2, FlaskSQLAlchemy.

### .gitignore

.gitignore file is required to tell git which commits to ignore and is not necessarily required for the application itself.

### Common Folder

In the common folder is the database.py file that connects to the Postgresql database server through pyscopg2. There is a Query function in this folder that runs the queries for retrieving kinase, inhibitor, and phosphosite data with the argument ‘query’ defined in the kinase and inhibitor routes.

import psycopg2

# Example of query function

def Query(query):

# Define our connection string

conn\_string = ("host='ec2-54-75-245-94.eu-west-1.compute.amazonaws.com' dbname='d71uh4v1fd2hq' user='tdsneouerzmxkj' password='92a500cb091fe70168b32c66fa6a3d6c376d467d57fb9b663eb5d13446ecb2e6'")

try: # get a connection, if a connect cannot be made an exception will be raised here

conn = psycopg2.connect(conn\_string)

# conn.cursor will return a cursor object, you can use this cursor to perform queries

cur = conn.cursor()

cur.execute(query)

query = cur.fetchall() # fetch query

conn.close() # close connection

return query

except:

return 'An error occured while executing SQL query'

### Models Folder

The models folder contains the views.py files for each route. It consists of 3 further folders for each route; analysis, kinase, and inhibitors. The blueprints for each route are in the respective views.py folder. The kinase and inhibitor index page contain a search request form for the user to input their search and renders their respective index HTML templates. The input is stored as the nameFilter variable which is parsed into the results route and into the database queries. The route is redirected to the results page when the search has been submitted in which the results HTML page loops through the database variable data to display to the user.

# import requirements for the model to run

from flask import Flask, Blueprint, render\_template, request, url\_for, redirect

# import database from common database

import common.database as db

# name kinase blueprint

kinase\_blueprint = Blueprint('kinase', \_\_name\_\_)

# make search box using request form for the index page

@kinase\_blueprint.route('/', methods=['GET', 'POST'])

def index():

if request.method == 'POST':

if request.form['name']:

# make user input a variable named nameFilter

nameFilter = request.form['name']

# turn kinase name into uppercase

nameFilter = nameFilter.upper()

# redirect to results page

return redirect(url\_for('kinase.results',

return render\_template('kinase/index.html')

@kinase\_blueprint.route('/results/<nameFilter>')

def results(nameFilter):

query = 'SELECT \* FROM public."Kinase\_table" WHERE "CAP\_KINASE\_NAME" LIKE \'' "%" + nameFilter + "%" '\' OR "SYNONYMS" LIKE \'' "%" + nameFilter + "%" '\''

# gather data from database

data = db.Query(query)

return render\_template('kinase/results.html', data=data,

The analysis models folder contains the route index views file and a process file. The index route contains an upload form that accepts a tsv file. The user is allowed to upload multiple .tsv files and temporarily save it to the uploads folder. The uploaded file is analysed via functions executed in the proccess\_file module. The uploaded file is analysed via functions executed in the proccess\_file module. The results of the analysis is sent to the uploaded route, together with a bar graph created to show the graphical representation of the results. Under the results the user can find a download button that initiates the download of a zip file containing the analysis files and bar graph. While the analysis is running through an entire function, there are multiple supporting functions to the app and the uploaded file. The process\_file function can have any file file open and ready to process. The aim of the delete\_folder function is to be able to delete folder contents uploaded by the user. This function can later be generalised to delete all saved files from our server related to the users’ uploaded data.

ALLOWED\_EXTENSIONS = {'tsv'} #specify the file extension allowed

def allowed\_file(filename):

""" allowed extensions to upload""" # somehow works in google chrome not in firefox....

return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED\_EXTENSIONS

@analysis\_blueprint.route('upload/', methods=['POST', 'GET'])

def upload():

"""This function uploads the file to uploads folder """

if request.method == 'POST':

uploadedfile = None

file = request.files['file']

# if file sent save it to upload\_folder and redirect to analysis:

for file in request.files.getlist('file'):

uploadedfile = secure\_filename(file.filename)

file.save(os.path.join(UPLOAD\_FOLDER, uploadedfile))

return redirect(url\_for('analysis.uploaded', uploadedfile=uploadedfile))

return render\_template("analysis/index.html")

@analysis\_blueprint.route('uploaded/', methods=['GET', 'POST'])

def uploaded():

"""This function maybe runs the analysis """

result\_object =process\_file.actual\_analysis()

ourprecious = process\_file.create\_fancybargraph(result\_object)

return render\_template("analysis/results.html",

tables=[result\_object.to\_html(classes='data', header="true")],

ourprecious=ourprecious)

@analysis\_blueprint.route('/dowload\_all', methods=['GET', 'POST'])

def download\_all():

"""This is the function which dowloads the results for the user"""

zipf = zipfile.ZipFile('Results.zip','w', zipfile.ZIP\_DEFLATED)

for root, dirs, files in os.walk('downloads/'):

for file in files:

zipf.write('downloads/'+file)

zipf.close()

return send\_file('Results.zip',

mimetype = 'zip',

attachment\_filename= 'Results.zip',

as\_attachment = True)

Process file:

def process\_file():

""" this function will find and open the file"""

pathjoin = os.path.join

for file in os.listdir(UPLOAD\_FOLDER):

mydearfile = open(pathjoin(UPLOAD\_FOLDER, file))

return mydearfile

def delete\_foldercontent():

"""This function might be able to delete data from our upload folder"""

folder = UPLOAD\_FOLDER

for the\_file in os.listdir(folder):

file\_path = os.path.join(folder, the\_file)

try:

if os.path.isfile(file\_path):

os.unlink(file\_path)

except Exception as e\_ception:

print(e\_ception)

### Templates Folder

The templates folder holds all the html code for the app. The menu.html file is the initial page that sets the html paraments for the entire app. It is styled with Bootstrap basic template enabling clarity for the app across different platforms. It also calls into the static folder to enable custom CSS style sheet. The menu page sets the settings for the top navigation bar to direct to the different routes. Each html file is an extension of the menu page. The index file is the homepage for the home route, and there is a respective index page for the route indexes.

### Static Folder

This folder contains the style of the application in CSS. The style.css file is an addition to the Bootstraps file and applies various settings for each component in the HTML files.

### Upload and Download Folders

Stores the uploaded files. The downloads folder is the files that are available for downloading. This contains 2 csv files and one png files.

# The Analysis Code

The analysis takes the users raw data file set as a variable and loads it into python. Using psycog2, the script connects to our phosphosite table in our database; using pandas.io.sql this is implemented into a pandas dataframe. A z-score calculation was performed with the following equation: The calculation is then performed as:

A loop then finds the kinases responsible for phosphorylating at these sites with our own database as reference. The data is cleaned and formatted correctly to flow correctly through the code. the maths module was imported to use to calculate the following: *s*, is the mean log2(fold change(fc)) for each kinase known substrate is calculated and stored in a dictionary, *p*, is the mean log2(fc) is calculated for all substrates., *m,* is the square-root of the number of substrates with known kinases is calculated, *d* is the standard deviation of the log2(fc) of all substrates in the sample. The script then makes two tables – one that contains the top ten most downregulated kinases and another with all results. The top ten results are then displayed as a bar graph for easy visualisation.

# Limitations and Next Steps

Unfortunately, the first deployment of the analysis contained a script that was too time intensive; this is because of the number of iterations required in a singular loop. But, the analysis did function as it was meant to. A new script has been produced that runs in less than ten seconds in comparison to the previous that took more than five minutes. The main differences between the scripts are displayed below with figure 3 containing the new script and 1 and 2 containing snippets of the obsolete version:

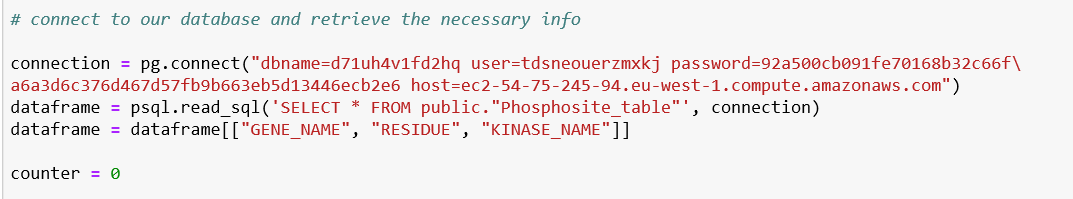


Figure 1 - downloads our phosphosite database into a panda’s data frame.

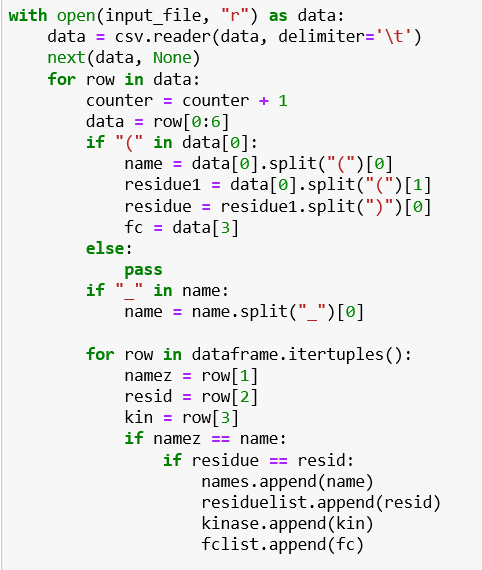


Figure 2 – iterates through the raw data file and database data frame, two conditions are implemented to make sure the gene name and the residue are matched and appends all required information into lists. Finally, these are then converted into a new data frame.



Figure 3 - contains the new version of the analysis script. The script searches for matches between the substrate’s gene name and the residue of phosphorylation for matches. If a match is found, the data appended into a new panda’s data frame. The "pd.merge" function then merges the corresponding fold change value from the raw data.

As seen above, the first version of the code was too time intensive due to the sheer number of iterations required, this is omitted in the second script. Unfortunately, the second script had a couple of bugs which prevented its full deployment in the websites. Therefore, in the next steps, this analysis must be debugged and fully integrated.

Furthermore, the kinase and inhibitors models require an error exception when the user input does not match our database. There is an error HTML page in the templates folder that redirects them back to the search page.

Using the hobby development package from Heroku as a deployment platform for the database has its limitations. Unfortunately, we have a limited number of rows, number of dynos, and size in Mb. A solution to overcome this to be able to an extensive database without these issues is to pay for the other various subscriptions. The ideal package would be the standard 2x as it has the optimal features for our application.

# Technologies

The overall technologies required for the app are:

* Python 2.7
  + Panadas, Flask, Matplotlib, Maths
* PostgreSQ:
  + PgAdmin4
* Heroku
  + Hobby package