**Code:**

**Inventory\_management.py:**

"""

This module deals with the inventory management.

It also deals with saving and loading of all the states of

the batches and tanks.

:attribute BEER\_PROCESS:

A Process object that holds the list of Batch objects

by the stage of brewing it is in.

:attribute TANKS: List of all the Tank objects.

:attribute FERMENTERS:

List of Tank objects with fermentation capabilities.

:attribute CONDITIONERS:

List of Tank objects with conditioning capabilities.

"""

import os

import time

from datetime import time as t\_time

from typing import Tuple, List

import logging

import \_pickle

ABS\_PATH = os.path.abspath(\_\_file\_\_)

D\_NAME = os.path.dirname(ABS\_PATH)

os.chdir(D\_NAME)

LOGGER = logging.getLogger("inv\_management")

LOGGER.setLevel(logging.DEBUG)

F\_HANDLER = logging.FileHandler('log\_file.log')

F\_FORMAT = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

F\_HANDLER.setFormatter(F\_FORMAT)

LOGGER.addHandler(F\_HANDLER)

class Process:

"""

This class stores lists of Batch objects.

Each batch object is stored in a list depending on which stage

of the production it is in.

"""

# pylint: disable=too-many-instance-attributes

# pylint: disable=too-few-public-methods

def \_\_init\_\_(self):

"""

Initialises each list/dictionary.

The dictionary steps allows each list to be accessed by its name.

The list step\_names allows the lists to be put in order.

The dictionary finished stores each finished beer and its quantity.

"""

LOGGER.debug("Initialising Process object")

self.waiting = []

self.brewing = []

self.fermenting = []

self.conditioning = []

self.bottling = []

self.finished = {}

self.orders = []

self.steps = {"waiting": self.waiting, "brewing": self.brewing,

"fermenting": self.fermenting, "conditioning": self.conditioning,

"bottling": self.bottling, "finished": self.finished}

self.step\_names = ["waiting", "brewing", "fermenting",

"conditioning", "bottling", "finished"]

class Tank:

"""This class is the class for each tank."""

# pylint: disable=too-few-public-methods

def \_\_init\_\_(self, name: str, volume: int, function: str):

"""

Initialising the tank by setting its current batch to None.

:param name: Name of the tank.

:param volume: The maximum volume of tank.

:param function:

The capability of tank. 'fermenter', 'conditioner' or 'both'.

"""

LOGGER.debug("Initialising Tank object")

self.name = name

self.volume = volume

self.function = function

self.current\_batch = None

class Batch:

"""This class is the class for each batch."""

# pylint: disable=too-few-public-methods

def \_\_init\_\_(self, beer: str, volume: int):

"""

Initialising the batch.

:param beer: Name of the beer for the batch.

:param volume: The volume for the batch.

"""

LOGGER.debug("Initialising Batch object")

self.beer = beer

self.current\_step = 0

self.current\_start\_time = time.time()

self.next\_step = 1

self.volume = volume

self.current\_tank = None

def go\_next\_step(self, process\_obj: Process, next\_tank: str = None) -> int:

"""

Handles all functions necessary for going to the next stage for the batch.

:param process\_obj: The Process object the batches and tanks are in.

:param next\_tank: The Tank to handle the next stage if any.

:return: The current step of the Batch object.

"""

LOGGER.info("Going to the next step")

# Don't go to brewing stage if brewing equipment is occupied.

if self.next\_step == 1 and process\_obj.brewing:

return self.current\_step

# Removing self from previous stage

if self.next\_step <= 5:

current\_step\_name = process\_obj.step\_names[self.current\_step]

process\_obj.steps[current\_step\_name].remove(self)

LOGGER.debug("Removed self from previous step")

# Adding to the finished dictionary when finished.

if self.next\_step == 5:

if self.beer in process\_obj.finished.keys():

process\_obj.finished[self.beer] += self.volume

else:

process\_obj.finished[self.beer] = self.volume

LOGGER.debug("Added self to next step")

# If it was in a tank, set tank as empty.

if self.current\_step in [2, 3]:

self.current\_tank.current\_batch = None

# If the next step requires a tank.

if self.next\_step in [2, 3]:

if next\_tank is not None:

next\_tank = find\_tank\_from\_name(next\_tank.split(" ")[0])

if next\_tank is not None:

next\_tank.current\_batch = self

self.current\_tank = next\_tank

self.current\_step = self.next\_step

self.next\_step += 1

else:

# If there are no available tanks set state to waiting.

LOGGER.warning("No tanks available")

self.current\_step = 0

else:

self.current\_step = self.next\_step

self.next\_step += 1

if self.current\_step <= 4:

next\_step\_name = process\_obj.step\_names[self.current\_step]

process\_obj.steps[next\_step\_name].append(self)

self.current\_start\_time = time.time()

LOGGER.debug("Gone to next step")

return self.current\_step

def load\_objects() -> Tuple[Process, List[Tank]]:

"""

This module loads saved data.

:return: The process object and the list of tanks.

"""

LOGGER.info("Loading data")

try:

with open('process\_object.dictionary', 'rb') as file:

result = \_pickle.load(file)

process\_obj = result[0]

tanks = result[1]

return process\_obj, tanks

except FileNotFoundError:

LOGGER.error("File for objects not found")

print("error: file not found")

return False, False

except EOFError:

LOGGER.warning("Empty objects file. Creating new.")

return False, False

def get\_tank\_types(tanks: List[Tank]) -> Tuple[List[Tank], List[Tank]]:

"""Making the list of fermenters and conditioners from the list of tanks"""

LOGGER.debug("Sorting tanks by funciton")

fermenters, conditioners = [], []

for tank in tanks:

if tank.function in ["both", "fermenter"]:

fermenters.append(tank)

if tank.function in ["both", "conditioner"]:

conditioners.append(tank)

return fermenters, conditioners

BEER\_PROCESS, TANKS = load\_objects()

# If file could not be loaded

if not BEER\_PROCESS:

LOGGER.warning("Couldn't load previous objects")

BEER\_PROCESS = Process()

# If tanks failed to be loaded from file.

if not TANKS:

LOGGER.warning("Couldn't load previous tanks")

TANKS = [Tank("Albert", 1000, "both"),

Tank("Brigadier", 800, "both"),

Tank("Camilla", 1000, "both"),

Tank("Dylon", 800, "both"),

Tank("Emily", 1000, "both"),

Tank("Florence", 800, "both"),

Tank("Gertrude", 680, "conditioner"),

Tank("Harry", 680, "conditioner"),

Tank("R2D2", 800, "fermenter")]

FERMENTERS, CONDITIONERS = get\_tank\_types(TANKS)

def show\_beer\_steps(process\_obj=BEER\_PROCESS) -> Tuple[List[Batch], List[str]]:

"""

Returns list of batch of objects and strings describing each batch and its current stage.

Looks through each step in the process object and returns

list of all the batch objects found and a list of strings

describing each batch.

:param process\_obj: The Process object containing each stage.

:return: List of processing batches and a list describing each batch.

"""

LOGGER.info("Making string for each batch")

return\_list = []

object\_list = []

for key, step in process\_obj.steps.items():

if key != "finished":

object\_list += step

for batch in step:

ongoing\_time = time.time()-batch.current\_start\_time

minutes, seconds = divmod(ongoing\_time, 60)

hours, minutes = divmod(minutes, 60)

days, hours = divmod(hours, 24)

weeks, days = divmod(days, 7)

# Creating the string for each batch

return\_list.append(str(batch.volume) + " Litres of " +

batch.beer + " has been " + key

+ " for " + f"{weeks} weeks, {days} days and "

f"{hours}:{minutes}:{int(seconds)}")

return object\_list, return\_list

def show\_tanks(tanks: List[Tank] = TANKS) -> List[str]:

"""

Returns a list of string describing each occupied tank.

:param tanks: List of all the tanks.

:return: List containing description for each occupied tank.

"""

LOGGER.info("Creating string for each tank")

return\_list = []

for tank in tanks:

if tank.current\_batch is not None:

batch = tank.current\_batch

return\_list.append(tank.name + " is currently processing " +

str(batch.volume) + "L of " + batch.beer)

return return\_list

def add\_batch(process\_obj: Process = BEER\_PROCESS, beer: str = "Organic Pilsner",

volume: int = 1000) -> Batch:

"""

Function for starting a new batch and creating a new batch object.

If there is no brewing, the new batch starts brewing. If there is already a batch brewing,

the new batch is put into the list of waiting beers.

:param process\_obj: The Process object containing the stages of production.

:param beer: Name of beer for the batch.

:param volume: Volume for the batch.

:return: The batch object created.

"""

LOGGER.info("Creating a new batch")

batch = Batch(beer, volume)

if not process\_obj.brewing:

batch.current\_step = 1

batch.next\_step = 2

process\_obj.brewing.append(batch)

else:

LOGGER.warning("Batch added to waiting list")

process\_obj.waiting.append(batch)

return batch

def available\_tanks(volume: int, step: int) -> List[Tank]:

"""

Gives all the available tanks for the given step and volume of batch.

:param volume: Volume of batch.

:param step: The step in the stage of production.

:return: List of available tank objects.

"""

LOGGER.info("Finding all available tanks")

if step == 2:

tank\_list = FERMENTERS

elif step == 3:

tank\_list = CONDITIONERS

return\_list = [tank for tank in tank\_list

if volume <= tank.volume and tank.current\_batch is None]

return return\_list

def find\_tank\_from\_name(name: str) -> Tank:

"""Finds the tank object given its name"""

for tank in TANKS:

if tank.name == name:

return tank

return None

def process\_done(batch\_list: List[Batch], time\_limit: t\_time) -> List[Batch]:

"""Returns all batches that have been processing longer than the given time limit"""

return\_list = []

for batch in batch\_list:

if time.time() - batch.current\_start\_time > time\_limit:

return\_list.append(batch)

return return\_list

def finished\_processes() -> List[Batch]:

"""

Finds all batches that are finished done with its stage.

All batches in all stages are looked at and compared with the

required time for that stage. If the batch has been processing

longer than that, it is returned.

:return: List of batches finished with the stage.

"""

LOGGER.info("Finding all the processes that are finished")

done\_waiting = []

if not BEER\_PROCESS.brewing:

waits = [batch for batch in BEER\_PROCESS.waiting if batch.next\_step == 1]

if waits:

done\_waiting.append(waits[0])

done\_brewing = process\_done(BEER\_PROCESS.brewing, 180\*3600)

done\_fermenting = process\_done(BEER\_PROCESS.fermenting, 4\*7\*24\*3600)

done\_conditioning = process\_done(BEER\_PROCESS.conditioning, 2\*7\*24\*3600)

done\_bottling = [batch for batch in BEER\_PROCESS.bottling

if time.time()-batch.current\_start\_time >= batch.volume\*3600]

return done\_waiting + done\_brewing + done\_fermenting + done\_conditioning + done\_bottling

def save\_objects(process\_obj: Process, tanks\_list: List[Tank]) -> str:

"""Saves the process object and list of tanks."""

LOGGER.info("Saving objects")

try:

with open('process\_object.dictionary', 'wb') as file:

\_pickle.dump([process\_obj, tanks\_list], file)

return "success"

except FileNotFoundError:

LOGGER.error("Objects file not found")

return "File not found"

if \_\_name\_\_ == "\_\_main\_\_":

save\_objects(BEER\_PROCESS, TANKS)

**read\_file.py:**

"""This module deals with reading the csv file and adding to it"""

import os

import csv

from datetime import timedelta, datetime

from typing import Dict, List, Union

import logging

from pandas import read\_csv, concat

from pandas.errors import ParserError

from dateutil.parser import parse

ABS\_PATH = os.path.abspath(\_\_file\_\_)

D\_NAME = os.path.dirname(ABS\_PATH)

os.chdir(D\_NAME)

LOGGER = logging.getLogger("inv\_management")

LOGGER.setLevel(logging.DEBUG)

F\_HANDLER = logging.FileHandler('log\_file.log')

F\_FORMAT = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

F\_HANDLER.setFormatter(F\_FORMAT)

LOGGER.addHandler(F\_HANDLER)

def parse\_data() -> Dict[str, Dict[str, List[Union[datetime, int]]]]:

"""

Parses the data in the csv to a dictionary in the following structure.

{'x': {'Beer1': [date1, date2, date3...],

'Beer2': [date1, date2, date3...],

'Beer3': [date1, date2, date3...]}

'y': {'Beer1': [value1, value2, value3...],

'Beer2': [value1, value2, value3...],

'Beer3': [value1, value2, value3...]}}

:return: Parsed data.

"""

LOGGER.info("Reading the csv file")

try:

with open('Barnabys\_sales\_fabriacted\_data.csv') as file:

# Adding missing data to the beginning

data\_dict = {'x': {'Organic Red Helles': [datetime(2018, 11, 1)],

'Organic Dunkel': [datetime(2018, 11, 1)]},

'y': {'Organic Red Helles': [0], 'Organic Dunkel': [0]}}

# Iterating through each row, ignoring the header.

for row in list(csv.reader(file, delimiter=','))[1:]:

# row[3] is the beer name

# If the beer has not been added to the dictionary yet

if row[3] not in data\_dict['x'].keys():

data\_dict['x'][row[3]] = []

data\_dict['y'][row[3]] = []

date\_obj = parse(row[2])

quantity = int(row[5])

# If date to be added exists, add sales to the existing data.

# If it doesn't, add the date and sale to the dictionary.

if date\_obj not in data\_dict['x'][row[3]]:

data\_dict['x'][row[3]].append(date\_obj)

data\_dict['y'][row[3]].append(quantity)

else:

# Adding sale to existing sale value

data\_dict['y'][row[3]][data\_dict['x'][row[3]].index(date\_obj)] += quantity

LOGGER.debug("Initial reading done")

# Sorting each list in the dictionaries.

for key, x\_beer in data\_dict['x'].items():

data\_dict['y'][key] = [x for \_, x in sorted(zip(x\_beer, data\_dict['y'][key]))]

data\_dict['x'][key] = sorted(x\_beer)

# If the latest dates don't match, add dates so the latest date matches.

last\_date\_list = [item[-1] for item in data\_dict['x'].values()]

if not (last\_date\_list[0] == last\_date\_list[1]

and last\_date\_list[1] == last\_date\_list[2]):

for key, value in data\_dict['x'].items():

if max(last\_date\_list) not in value:

data\_dict['x'][key].append(max(last\_date\_list))

data\_dict['y'][key].append(0)

LOGGER.debug("Latest date matched")

for key, x\_beer in data\_dict['x'].items():

# Filling in the gaps with sales of 0.

for counter, (date\_obj, quantity) in \

enumerate(zip(x\_beer[:-1], data\_dict['y'][key][:-1])):

next\_date\_obj = x\_beer[counter+1]

gap = (next\_date\_obj - date\_obj).days - 1

for i in range(gap):

get\_date\_obj = date\_obj + timedelta(days=1+i)

new\_quantity = 0

x\_beer.append(get\_date\_obj)

data\_dict['y'][key].append(new\_quantity)

# Sorting the data in case it got mixed while filling in gaps.

data\_dict['y'][key] = [x for \_, x in sorted(zip(x\_beer, data\_dict['y'][key]))]

data\_dict['x'][key] = sorted(x\_beer)

LOGGER.debug("Filled gap and sorted %s", key)

except FileNotFoundError:

LOGGER.critical("CSV File Not Found")

return data\_dict

def write\_data(file\_dir: str) -> str:

"""

This function adds data from a csv file in the given the directory.

:param file\_dir: The directory of the csv file to be added.

:return: Error message.

"""

LOGGER.info("Writing csv data#")

try:

open(file\_dir)

except FileNotFoundError:

return "File not found"

try:

file\_a = read\_csv('Barnabys\_sales\_fabriacted\_data.csv', index\_col=0)

file\_b = read\_csv(file\_dir, index\_col=0)

concat([file\_a, file\_b]).to\_csv('Barnabys\_sales\_fabriacted\_data.csv')

return "success"

except FileNotFoundError:

LOGGER.error("csv file was not found")

return "file not found"

except ParserError:

LOGGER.error("Invalid data in new csv file")

return "Valid data not found in file"

DATA\_DICT = parse\_data()

if \_\_name\_\_ == "\_\_main\_\_":

print(parse\_data())

**Sales\_predictions.py:**

"""

This module deals with the prediction of sales for a year into the future.

This is done by finding the growth rates for each day in the csv file and multiplying

that to the latest data point for each beer over and over until all the growth rates have

been multiplied.

In addition, the module can plot past data and the prediction using

matplotlib and find the total predicted sale of a given time period.

"""

import os

from datetime import datetime, timedelta

from typing import List, Tuple, Dict

import logging

import matplotlib.pyplot as plt

from matplotlib.lines import Line2D

from read\_file import parse\_data

ABS\_PATH = os.path.abspath(\_\_file\_\_)

D\_NAME = os.path.dirname(ABS\_PATH)

os.chdir(D\_NAME)

LOGGER = logging.getLogger("sales\_predictions")

LOGGER.setLevel(logging.DEBUG)

F\_HANDLER = logging.FileHandler('log\_file.log')

F\_FORMAT = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

F\_HANDLER.setFormatter(F\_FORMAT)

LOGGER.addHandler(F\_HANDLER)

def calculate\_growth(start\_date\_obj: datetime, end\_date\_obj: datetime,

x\_list: List[datetime], y\_list: List[int], days: int = 1) -> int:

"""

This function calculates the growth % given two dates and the sales for those dates.

This function can also calculate the growth rate between two periods.

:param start\_date\_obj: First date.

:param end\_date\_obj: Second date.

:param x\_list: List of dates.

:param y\_list: List of sales.

:param days: Length of time period.

:return: Growth % as a rounded integer.

"""

x\_index = x\_list.index(start\_date\_obj)

start\_data = sum(y\_list[x\_index:x\_index+days]) # Summing the sales values for the given period.

if start\_data == 0:

# Setting a default replacement value for 0 so prediction division by 0 doesn't occur.

LOGGER.debug("Start data changed to 1.1")

start\_data = 1.1

x\_index = x\_list.index(end\_date\_obj)

if x\_index+days <= len(y\_list):

# Summing the sales values in the next given period.

end\_data = sum(y\_list[x\_index:x\_index+days])

else:

end\_data = sum(y\_list[x\_index:]) # If the end of the list is reached.

growth\_percent = round((end\_data - start\_data) / start\_data, 5) \* 100

return growth\_percent

def multiply\_rate(value: int, percent\_list: List[int],

index: int = 0, new\_list: List = None) -> List[int]:

"""

This function multiplies all the growth rates recursively.

:param value: Value for the growth % to be applied to.

:param percent\_list: List of growth %.

:param index: Index of the growth % to be applied.

:param new\_list: List of the new values after growth % was applied.

:return: After recursion is finished, a list of the new values is returned.

"""

if new\_list is None:

new\_list = []

if index >= len(percent\_list):

return new\_list

value += 1 # To avoid the prediction flat-lining.

value \*= ((percent\_list[index]) / 100) + 1

new\_list.append(value)

index += 1

return multiply\_rate(value, percent\_list, index, new\_list)

def plot\_past\_data(key\_name: str = None) -> List[Line2D]:

"""

This function plots the past data using matplotlib.

The optional argument allows you to see the graph for only one beer type.

:param key\_name: A specific type of beer to be shown.

:return: The plot.

"""

LOGGER.info("Plotting past data")

data\_dict = parse\_data()

for key in data\_dict['x']:

if key\_name is None or key == key\_name:

x\_data = data\_dict['x'][key]

y\_data = data\_dict['y'][key]

line\_2d = plt.plot(x\_data, y\_data, label=key, marker=".", linewidth=0.5, markersize=1)

plt.legend()

return line\_2d

def plot\_growth\_percent(days: int = 1, key\_name: str = None, plot: bool = True,

start\_date: datetime = None) -> Tuple[List[datetime], Dict[str, List[int]]]:

"""

This function returns the growth rates for each beer.

The growth can be calculated for a total of a given period or day by day. It can

also be calculated for all the beers or just one specific beer.

This module also has the capability to plot the grow rates using matplotlib.

:param days: The period to sum the data.

:param key\_name: The name of the specific beer.

:param plot: Whether to plot the graph or not.

:param start\_date: The date to start the growth rate list from.

:return: A dictionary of the growth rates and the corresponding dates.

"""

LOGGER.info("Calculating growth rates")

data\_dict = parse\_data()

growth\_dict = {}

for key in data\_dict['x']: # Iterating through each beer.

if key\_name is None or key\_name == key:

x\_data, y\_data = data\_dict['x'][key], data\_dict['y'][key]

growth\_rates = []

dates = []

# Iterating through the dates in required steps.

for index in range(0, len(x\_data)-days, days):

date = x\_data[index]

if start\_date is None or start\_date <= date:

dates.append(date)

growth = calculate\_growth(date, date + timedelta(days), x\_data, y\_data, days)

growth\_rates.append(growth)

LOGGER.debug("Data and date added")

growth\_dict[key] = growth\_rates

if plot:

plt.plot(dates, growth\_rates,

label=key + " growth % "+str(days),

linestyle='None', marker="D", markersize=2)

if plot:

plt.legend()

return dates, growth\_dict

def plot\_next\_year(days: int = 1, key\_name: str = None, next\_year: bool = True,

start\_date: datetime = None, date\_range: int = None)\

-> Tuple[List[datetime], Dict[str, List[int]]]:

"""

This module calculates the predictions for 1 year from the latest data in the csv file.

The module has several options including calculating the data for just the given period.

A prediction is made by calculating the growth rates per day in the csv file, and multiplying

that to the latest sales made in the csv file.

:param days:

The period in which to calculate growth.

Set this to values other than one to predict using time periods.

:param key\_name: Can be set to find the prediction for just one beer.

:param next\_year: Whether to adjust the dates for prediction.

:param start\_date: The date to start the prediction from.

:param date\_range: The number of days in the future the prediction should go on for.

:return: A dictionary of predicted data and the corresponding dates.

"""

LOGGER.info("Calculating next year")

data\_dict = parse\_data()

start\_past\_date = data\_dict['x']['Organic Red Helles'][-1] - timedelta(days=364)

dates, percent\_dict = plot\_growth\_percent(days=days, key\_name=key\_name, plot=False,

start\_date=start\_past\_date)

if dates is None:

return None, None

dates = [date + timedelta(days=days-1) for date in dates]

prediction\_dict = {}

if next\_year:

dates = [date + timedelta(days=365) for date in dates]

# Start date + date range is larger than date list

if start\_date is not None and start\_date + timedelta(date\_range) not in dates:

return False, False

for key, percent\_list in percent\_dict.items():

start\_sale = data\_dict['y'][key][-1]

LOGGER.debug("Multiplying rate")

prediction\_dict[key] = multiply\_rate(start\_sale, percent\_list)

if start\_date is not None:

index = dates.index(start\_date)

prediction\_dict[key] = prediction\_dict[key][index:]

if date\_range is not None:

prediction\_dict[key] = prediction\_dict[key][:date\_range]

LOGGER.debug("All rates multiplied")

if start\_date is not None:

dates = dates[index:]

if date\_range is not None:

dates = dates[:date\_range]

return dates, prediction\_dict

def get\_total(start\_date: datetime, date\_range: datetime,

data\_tuple: Tuple[List[datetime], Dict[str, List[int]]]) -> Dict[str, int]:

"""

This module gets the total sales in a given time period.

:param start\_date: The date to start counting the sales.

:param date\_range: The period in which to calculate the total.

:param data\_tuple: The list of dates and the dictionary which holds the sales data.

:return:

A dictionary containing the total for each beer.

"""

LOGGER.info("Getting total from %s", str(start\_date))

dates, data = data\_tuple

return\_dict = {}

for key, batch in data.items():

if start\_date not in dates:

LOGGER.warning("start date not in dates")

return None

index = dates.index(start\_date)

if index + date\_range >= len(batch):

LOGGER.warning("Date range out of range of batch list")

return None

return\_dict[key] = sum(batch[index:index+date\_range])

return return\_dict

if \_\_name\_\_ == "\_\_main\_\_":

plot\_past\_data(key\_name="Organic Pilsner")

# plot\_next\_year(days=1, key\_name="Organic Pilsner", next\_year=False)

# plot\_growth\_percent(days=7, key\_name="Organic Pilsner")

# plot\_average\_growths(days=3, key\_name="Organic Pilsner")

# dates, data = plot\_next\_year(plot=False)

# print(get\_total(datetime(2019, 12, 2), 7, dates, data))

plt.show()

**User\_interface.py:**

"""

This module is the main module. It deals with all of the user interface.

The layout of the interface was created using PyQt5 designer.

Form implementation generated from reading ui file 'interface.ui'.

Created by: PyQt5 UI code generator 5.13.0.

"""

import sys

from os import path as os\_path, chdir

from math import ceil

from threading import Thread

from datetime import datetime, timedelta

from typing import Tuple, List, Dict, Callable, Union

from time import sleep as time\_sleep

import logging

from PyQt5 import QtCore, QtGui, QtWidgets

import pyqtgraph as pg

from sales\_predictions import plot\_next\_year, get\_total

from inventory\_management import Tank, Batch, \

BEER\_PROCESS, TANKS, show\_beer\_steps, show\_tanks, add\_batch, \

available\_tanks, finished\_processes, save\_objects

from read\_file import write\_data

LOGGER = logging.getLogger("user\_interface")

LOGGER.setLevel(logging.DEBUG)

F\_HANDLER = logging.FileHandler('log\_file.log')

F\_FORMAT = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

F\_HANDLER.setFormatter(F\_FORMAT)

LOGGER.addHandler(F\_HANDLER)

ABS\_PATH = os\_path.abspath(\_\_file\_\_)

D\_NAME = os\_path.dirname(ABS\_PATH)

chdir(D\_NAME)

# pylint: disable=c-extension-no-member

def current\_datetime() -> datetime:

"""Returning today's date as a datetime object."""

return datetime.combine(datetime.today().date(), datetime.min.time())

def beer\_suggestion() -> Tuple[str, int]:

"""

Making a recommendation on the next batch to brew.

The recommendation algorithm is as follows:

1. Look at the 6 week period 10 weeks from now.

2. For all the predicted demands, subtract the volumes

currently waiting, brewing and fermenting.

3. Get the beer with the highest demand after step 2.

4. Look to see if any batches are brewing or waiting.

5. If the result from step 4 is None, recommend the beer from step 3.

:return: The name and suggested volume for the next beer to be brewed.

"""

LOGGER.info("Calculating next batch suggestion")

if plot\_next\_year()[0]:

totals\_dict = get\_total(current\_datetime() + timedelta(weeks=10), 7 \* 6, plot\_next\_year())

for batch in BEER\_PROCESS.waiting + BEER\_PROCESS.brewing + BEER\_PROCESS.fermenting:

LOGGER.info("There are batches waiting, brewing or fermenting")

totals\_dict[batch.beer] -= batch.volume \* 2

maximum = max(totals\_dict.values())

key = list(totals\_dict.keys())[list(totals\_dict.values()).index(maximum)]

LOGGER.debug("Beer with max demand fetched")

volume = int(ceil((maximum \* 0.5) / 10.0)) \* 10

tanks = available\_tanks(800, 2)

if tanks and not BEER\_PROCESS.brewing and \

not [batch for batch in BEER\_PROCESS.waiting if batch.next\_step == 1]:

max\_possible = max([tank.volume for tank in tanks])

volume = min(max\_possible, volume)

LOGGER.info("There is a beer suggestion")

return key, volume

LOGGER.info("No beer suggestion")

return None, None

def get\_next\_tanks(batch\_object: Batch) -> List[Tank]:

"""

Returns the list of tanks that may be required for the next stage for a batch.

:param batch\_object: The Batch object to look tanks for.

:return: List of tanks required if any.

"""

LOGGER.info("Getting the list of tanks for next step")

if batch\_object.next\_step in [2, 3]:

LOGGER.info("Tanks are required for next step of %s", batch\_object.beer)

tanks = available\_tanks(batch\_object.volume, batch\_object.next\_step)

if batch\_object.current\_tank is not None and\

batch\_object.current\_tank.function == "conditioner":

LOGGER.debug("Adding current tank to list.")

tanks = [batch\_object.current\_tank] + tanks

if tanks:

LOGGER.info("Tanks returned")

return tanks

LOGGER.info("No tank required")

return False

def save\_continuously():

"""Procedure to save the state of batches and tanks every 2 minutes"""

while True:

save\_objects(BEER\_PROCESS, TANKS)

LOGGER.warning("Current state auto-saved")

time\_sleep(120)

def pop\_up(text: str):

"""Shows a pop up box for the given text"""

message = QtWidgets.QMessageBox()

message.setText(text)

message.exec\_()

class UiMainWindow:

"""This class is for creating the user interface"""

# pylint: disable=too-many-instance-attributes

def get\_graph(self, dates: List[datetime] = None,

data: Dict[str, List[int]] = None, symbol: str = None):

"""

This function plots the graph.

A part of the graph can be plotted if the dates and data for

it is given. The symbol for the data points can be changed.

:param dates: List of dates when plotting part of graph.

:param data: List of sales for each beer when plotting part of graph.

:param symbol: Symbol for data points.

"""

LOGGER.info("Getting graph")

self.widget.clear()

if self.widget.plotItem.legend:

LOGGER.debug("Legend cleared")

self.widget.plotItem.legend.scene().removeItem(self.widget.plotItem.legend)

self.widget.setTitle("Prediction")

if not dates or not data or len(dates) != len(data['Organic Pilsner']):

start\_date = current\_datetime()

dates, predict\_dict = plot\_next\_year(start\_date=start\_date, date\_range=180)

else:

predict\_dict = data

# Date in strings to show on graph

string\_dates = [date.strftime("%d/%m/%y") for date in dates]

# Changing dates to integer so graph can be plotted.

dates = [round((date.timestamp()-dates[0].timestamp())/(3600\*24)) for date in dates]

colours = ['r', 'g', 'b']

self.widget.addLegend(size=(70, 50), offset=(10, 1))

plots = []

LOGGER.debug("Starting to plot")

for counter, (key, beer) in enumerate(predict\_dict.items()):

colour = colours[counter]

# Showing the total for the region the graph shows.

plots.append(self.widget.plotItem.plot(dates, beer,

pen=pg.mkPen(colour, width=1),

name=key + ", " + str(int(sum(beer))) + " btls",

symbol=symbol))

LOGGER.info("Graph plotted")

# Adjusting ticks

x\_axis = self.widget.getAxis('bottom')

x\_axis.setTicks([list(zip(dates[::18], string\_dates[::18]))])

x\_axis.setTickSpacing([(20, 0), (1, 0), (0.25, 0)])

def show\_tanks(self):

"""Showing tank states for each tank that is processing a batch."""

LOGGER.info("Showing each tank state")

self.tanks\_list.clear()

tank\_strings = show\_tanks()

self.tanks\_list.addItems(tank\_strings)

def make\_step\_function(self, batch: Batch, combo\_box: QtWidgets.QComboBox = None) -> Callable:

"""Making the function to be linked for the next step button."""

def go\_next\_step():

"""The function to be linked to the 'next step' button."""

# If next step requires a tank.

LOGGER.info("next step button clicked")

if combo\_box is None:

batch.go\_next\_step(BEER\_PROCESS)

else:

batch.go\_next\_step(BEER\_PROCESS, combo\_box.currentText())

self.refresh\_page()

return go\_next\_step

def make\_add\_function(self, name: str, volume: str) -> Callable:

"""Making the function to connect to the button to add a batch."""

LOGGER.info("Making function to connect to add batch button")

def add\_batch2():

"""The function to add a batch"""

LOGGER.info("Add batch button clicked")

add\_batch(beer=name, volume=volume)

self.refresh\_page()

return add\_batch2

def show\_beers(self):

"""

Showing all batches that is currently being processed.

Each batch that is being processed and its description is found

using show\_beer\_steps and a button to go to the next step and a

combo box if necessary is created.

"""

LOGGER.info("Showing each processing batch to interface")

self.batches\_list.clear()

# Getting each Batch object and the description.

batch\_objects, beers = show\_beer\_steps()

for counter, beer in enumerate(beers):

batch\_object = batch\_objects[counter]

add\_item = QtGui.QListWidgetItem(beer)

add\_item.setFlags(add\_item.flags() ^ QtCore.Qt.ItemIsSelectable)

self.batches\_list.addItem(add\_item)

# Layout for button and combo box.

widget = QtWidgets.QWidget(self.batches\_list)

layout = QtWidgets.QHBoxLayout(widget)

layout.setContentsMargins(230, 20, 0, 0)

LOGGER.debug("Getting next function")

next\_function = self.make\_step\_function(batch\_object)

get\_next = get\_next\_tanks(batch\_object)

# If the next step requires a tank, create combo box.

if get\_next:

tanks = get\_next

combo\_box = QtWidgets.QComboBox(widget)

layout.addWidget(combo\_box)

for tank in tanks:

combo\_box.addItem(tank.name + " " + str(tank.volume) + "L")

next\_function = self.make\_step\_function(batch\_object, combo\_box)

LOGGER.info("Next function retrieved")

# Making the button to go to the next step.

button = QtWidgets.QToolButton(widget)

button.setText("Next Step")

button.clicked.connect(next\_function)

layout.addStretch()

layout.addWidget(button)

self.batches\_list.setItemWidget(add\_item, widget)

LOGGER.debug("Beers shown")

def refresh\_page(self):

"""Function to update all descriptions in the interface"""

LOGGER.info("Refreshing page")

self.show\_beers()

self.show\_tanks()

self.show\_bottled()

self.show\_orders()

self.get\_recommendation()

save\_objects(BEER\_PROCESS, TANKS)

def add\_beers(self):

"""

This is the function to add batches.

The volume and beer name is obtained from the interface

and passed to add\_batch.

"""

LOGGER.info("Adding a new batch")

volume = self.volume\_edit.toPlainText()

try:

volume = int(volume)

except ValueError:

LOGGER.error("Integer not entered for volume")

pop\_up("Please enter an integer")

else:

if volume != "" and 0 < volume <= 1000:

beer = self.combo\_box.currentText()

add\_batch(beer=beer, volume=volume)

self.refresh\_page()

else:

LOGGER.warning("Value between 0 and 1000 not entered for volume")

pop\_up("Please enter a value between 0 and 1000")

self.volume\_edit.setText("")

def search\_graph(self):

"""

This is the function to search the graph

Date and width is grabbed from the user interface

which is then passed to plot\_next\_year to obtain the

data. Next, get\_graph is used to plot the graph.

"""

LOGGER.info("Grabbing graph")

# Grabbing the date and region from the interface

date = self.date\_edit.date().toPyDate()

width = self.range\_choice.currentText()

# Changing date object to a datetime object

date\_time = datetime.combine(date, datetime.min.time())

if width == "Week":

d\_range = 7

else:

d\_range = 30

# Getting the dates and data for specified region.

dates, data = plot\_next\_year(start\_date=date\_time, date\_range=d\_range)

if dates:

self.get\_graph(dates, data, "d")

else:

LOGGER.error("Date not found")

pop\_up("Failed to find date")

def make\_start\_suggestion(self):

"""

Make suggestions to start a new batch.

If starting a new brew is suggested, the recommendation

is shown and a button next to it to actually start the brew.

"""

LOGGER.info("Showing the suggestions for batches to start")

self.suggestions\_list.clear()

# Getting the suggestion

key, volume = beer\_suggestion()

widget = QtWidgets.QWidget(self.suggestions\_list)

# If it is suggested to start a batch.

if key is not None:

LOGGER.debug("There is a suggestions")

if volume > 10:

# Creating the string to display.

add\_item = QtGui.QListWidgetItem("Start brewing " + str(volume) + "L of " + key)

add\_item.setFlags(add\_item.flags() ^ QtCore.Qt.ItemIsSelectable)

self.suggestions\_list.addItem(add\_item)

# Creating the button

button = QtWidgets.QToolButton(widget)

button.setText("Execute")

next\_function = self.make\_add\_function(key, volume)

button.clicked.connect(next\_function)

layout = QtWidgets.QHBoxLayout(widget)

layout.setContentsMargins(0, 0, 0, 0)

layout.addStretch()

layout.addWidget(button)

self.suggestions\_list.setItemWidget(add\_item, widget)

LOGGER.debug("Suggestion shown")

def make\_next\_suggestion(self):

"""

This is the function to make a suggestion for batches that should be moved to the next step.

It gets the batches that have finished in their process from finished\_processes

and displays it. If the next step doesn't require a tank, a button is displayed

to actually move it to the next step.

"""

LOGGER.info("Showing suggestion for batches to advance")

step\_finished = finished\_processes()

for batch in step\_finished:

add\_item = QtGui.QListWidgetItem("Process next step for " + batch.beer + " that is "

+ BEER\_PROCESS.step\_names[batch.current\_step] + "\n")

add\_item.setFlags(add\_item.flags() ^ QtCore.Qt.ItemIsSelectable)

self.suggestions\_list.addItem(add\_item)

widget = QtWidgets.QWidget(self.suggestions\_list)

layout = QtWidgets.QHBoxLayout(widget)

layout.setContentsMargins(0, 20, 0, 0)

layout.addStretch()

get\_next = get\_next\_tanks(batch)

# If a tank is not required in the next step

if not get\_next:

LOGGER.debug("Next step does not requires a tank.")

# Getting the function to link to the button.

next\_function = self.make\_step\_function(batch)

# Making the button.

button = QtWidgets.QToolButton(widget)

button.setText("Execute")

button.clicked.connect(next\_function)

layout.addWidget(button)

self.suggestions\_list.setItemWidget(add\_item, widget)

LOGGER.debug("Suggestions shown")

def get\_recommendation(self):

"""This shows the recommendations on the interface"""

LOGGER.info("Retrieving recommendations")

self.make\_start\_suggestion()

self.make\_next\_suggestion()

def show\_bottled(self):

"""This shows all the bottled beers."""

LOGGER.info("Showing bottled beers")

self.bottled\_list.clear()

for beer, volume in BEER\_PROCESS.finished.items():

self.bottled\_list.addItem(str(int(volume/0.5)) + " bottles of " + beer)

def add\_order(self):

"""

Adding orders.

The name, quantity and due date is retrieved from the interface

and added to the orders list in the Process object.

"""

LOGGER.info("Adding order")

beer = self.combo\_box\_3.currentText()

bottle\_quantity = self.spin\_box.value()

due\_date = self.date\_edit\_2.date().toPyDate()

if bottle\_quantity > 0:

BEER\_PROCESS.orders.append([beer, bottle\_quantity, due\_date])

LOGGER.debug("Order added")

else:

pop\_up("Please enter a value larger than 0")

self.refresh\_page()

def make\_deliver\_button(self, order: List[Union[str, int]]) -> Callable:

"""

Makes the function for the 'deliver' button for orders to be connected to.

:param order: The list of attributes of an order.

:return: The function for the 'deliver' button to be linked to.

"""

LOGGER.info("Making function to link to deliver button")

def deliver\_order():

"""

The function that is executed when the 'deliver' button is pressed.

The order is removed from BEER\_PROCESS, the number of bottles is

updated if there is enough in the inventory.

"""

LOGGER.info("Deliver button clicked")

if order[0] in BEER\_PROCESS.finished and \

BEER\_PROCESS.finished[order[0]] >= order[1]\*0.5:

BEER\_PROCESS.orders.remove(order)

BEER\_PROCESS.finished[order[0]] -= order[1]\*0.5

self.refresh\_page()

LOGGER.info("Order removed successfully")

else:

LOGGER.warning("Not enough inventory")

pop\_up("Not enough inventory")

return deliver\_order

def show\_orders(self):

"""

Shows all the orders.

For each order, the order displayed and a button to deliver the

order is created.

"""

LOGGER.info("Showing all orders")

self.orders\_list.clear()

self.spin\_box.setValue(1)

# Retrieving the orders

orders = BEER\_PROCESS.orders

for order in orders:

# Making the string to show.

add\_item = QtGui.QListWidgetItem(str(order[1]) + " bottles of " + order[0]

+ " due " + order[2].strftime("%d/%m/%Y"))

add\_item.setFlags(add\_item.flags() ^ QtCore.Qt.ItemIsSelectable)

self.orders\_list.addItem(add\_item)

next\_function = self.make\_deliver\_button(order)

LOGGER.debug("Next function retrieved")

widget = QtWidgets.QWidget(self.orders\_list)

# Creating the button.

button = QtWidgets.QToolButton(widget)

button.setText("Deliver")

button.clicked.connect(next\_function)

layout = QtWidgets.QHBoxLayout(widget)

layout.setContentsMargins(0, 0, 0, 0)

layout.addStretch()

layout.addWidget(button)

self.orders\_list.setItemWidget(add\_item, widget)

LOGGER.debug("All orders shown")

def add\_file(self):

"""

Adding a csv file to the existing file.

File adding attempt is made and a message is shown in a pop up box.

"""

LOGGER.info("Adding file")

file\_dir = self.file\_dir\_edit.text()

self.file\_dir\_edit.setText("Enter file directory for new csv file")

result = write\_data(file\_dir)

if result == "success":

message\_text = "Successfully added csv file"

else:

LOGGER.error("Failed to add file")

message\_text = str(result)

pop\_up(message\_text)

# pylint: disable=too-many-statements

def \_\_init\_\_(self, main\_window: QtWidgets.QMainWindow):

"""

This is the layout of each object in the user interface.

:param main\_window: QtMainWindow object.

"""

main\_window.setObjectName("main\_window")

main\_window.resize(1751, 869)

self.central\_widget = QtWidgets.QWidget(main\_window)

font = QtGui.QFont()

font.setPointSize(9)

font.setUnderline(True)

font2 = QtGui.QFont()

font2.setPointSize(9)

font2.setUnderline(False)

self.date\_edit = QtWidgets.QDateEdit(self.central\_widget)

self.date\_edit.setGeometry(QtCore.QRect(160, 10, 110, 22))

self.date\_edit.setDateTime(QtCore.QDateTime.currentDateTime())

self.search\_label = QtWidgets.QLabel(self.central\_widget)

self.search\_label.setGeometry(QtCore.QRect(105, 10, 55, 16))

self.search\_label.setText("Search:")

self.search\_label.setFont(font)

self.range\_choice = QtWidgets.QComboBox(self.central\_widget)

self.range\_choice.setGeometry(QtCore.QRect(275, 10, 73, 22))

self.range\_choice.addItem("Week")

self.range\_choice.addItem("Month")

self.search\_button = QtWidgets.QPushButton(self.central\_widget)

self.search\_button.setGeometry(QtCore.QRect(352, 7, 91, 26))

self.search\_button.setText("Search")

self.search\_button.clicked.connect(self.search\_graph)

self.graph\_button = QtWidgets.QPushButton(self.central\_widget)

self.graph\_button.setGeometry(QtCore.QRect(0, 5, 93, 28))

self.graph\_button.setText("Full Graph")

self.graph\_button.clicked.connect(self.get\_graph)

self.beer\_label = QtWidgets.QLabel(self.central\_widget)

self.beer\_label.setGeometry(QtCore.QRect(280, 760, 55, 16))

self.beer\_label.setObjectName("beer\_label")

self.beer\_label.setText("Beer:")

self.beer\_label.setFont(font2)

self.volume\_label = QtWidgets.QLabel(self.central\_widget)

self.volume\_label.setGeometry(QtCore.QRect(450, 760, 55, 16))

self.volume\_label.setObjectName("volume\_label")

self.volume\_label.setText("Volume:")

self.volume\_label.setFont(font2)

self.volume\_edit = QtWidgets.QTextEdit(self.central\_widget)

self.volume\_edit.setGeometry(QtCore.QRect(450, 780, 161, 31))

self.volume\_edit.setObjectName("volume\_edit")

self.combo\_box = QtWidgets.QComboBox(self.central\_widget)

self.combo\_box.setGeometry(QtCore.QRect(280, 780, 170, 31))

self.combo\_box.setObjectName("combo\_box")

self.combo\_box.addItem("Organic Red Helles")

self.combo\_box.addItem("Organic Pilsner")

self.combo\_box.addItem("Organic Dunkel")

self.batch\_button = QtWidgets.QPushButton(self.central\_widget)

self.batch\_button.setGeometry(QtCore.QRect(360, 810, 90, 28))

self.batch\_button.setObjectName("batch\_button")

self.batch\_button.setText("Add Batch")

self.batch\_button.clicked.connect(self.add\_beers)

self.refresh\_button = QtWidgets.QPushButton(self.central\_widget)

self.refresh\_button.setGeometry(QtCore.QRect(450, 810, 90, 28))

self.refresh\_button.setObjectName("refresh\_button")

self.refresh\_button.setText("Refresh")

self.refresh\_button.clicked.connect(self.refresh\_page)

self.widget = pg.PlotWidget(self.central\_widget)

self.widget.setGeometry(QtCore.QRect(0, 40, 901, 421))

self.widget.setObjectName("widget")

self.get\_graph()

self.batches\_list = QtWidgets.QListWidget(self.central\_widget)

self.batches\_list.setGeometry(QtCore.QRect(0, 485, 450, 276))

self.batches\_list.setObjectName("batches\_list")

self.batches\_list.setWordWrap(True)

self.tanks\_list = QtWidgets.QListWidget(self.central\_widget)

self.tanks\_list.setGeometry(QtCore.QRect(450, 485, 450, 276))

self.tanks\_list.setObjectName("tanks\_list")

self.tanks\_list.setWordWrap(True)

self.batch\_label = QtWidgets.QLabel(self.central\_widget)

self.batch\_label.setGeometry(QtCore.QRect(5, 465, 86, 16))

self.batch\_label.setText("Batch Status:")

self.batch\_label.setFont(font)

self.tank\_label = QtWidgets.QLabel(self.central\_widget)

self.tank\_label.setGeometry(QtCore.QRect(460, 465, 90, 16))

self.tank\_label.setText("Tank Status:")

self.tank\_label.setFont(font)

self.suggestions\_list = QtWidgets.QListWidget(self.central\_widget)

self.suggestions\_list.setGeometry(QtCore.QRect(900, 40, 426, 721))

self.suggestions\_list.setWordWrap(True)

self.suggest\_label = QtWidgets.QLabel(self.central\_widget)

self.suggest\_label.setGeometry(QtCore.QRect(904, 10, 120, 21))

self.suggest\_label.setText("Suggested To Do:")

self.suggest\_label.setFont(font)

self.get\_recommendation()

self.order\_button = QtWidgets.QPushButton(self.central\_widget)

self.order\_button.setGeometry(QtCore.QRect(1493, 80, 93, 28))

self.order\_button.setText("Add Order")

self.order\_button.clicked.connect(self.add\_order)

self.date\_edit\_2 = QtWidgets.QDateEdit(self.central\_widget)

self.date\_edit\_2.setGeometry(QtCore.QRect(1613, 40, 111, 31))

self.date\_edit\_2.setDateTime(QtCore.QDateTime.currentDateTime())

self.spin\_box = QtWidgets.QSpinBox(self.central\_widget)

self.spin\_box.setGeometry(QtCore.QRect(1513, 40, 46, 31))

self.spin\_box.setRange(1, 1000)

self.combo\_box\_3 = QtWidgets.QComboBox(self.central\_widget)

self.combo\_box\_3.setGeometry(QtCore.QRect(1358, 40, 146, 31))

self.combo\_box\_3.addItem("Organic Red Helles")

self.combo\_box\_3.addItem("Organic Pilsner")

self.combo\_box\_3.addItem("Organic Dunkel")

self.beer\_type\_label = QtWidgets.QLabel(self.central\_widget)

self.beer\_type\_label.setGeometry(QtCore.QRect(1358, 18, 76, 21))

self.beer\_type\_label.setFont(font2)

self.beer\_type\_label.setText("Beer Type:")

self.due\_label = QtWidgets.QLabel(self.central\_widget)

self.due\_label.setGeometry(QtCore.QRect(1613, 20, 55, 16))

self.due\_label.setFont(font2)

self.due\_label.setText("Due:")

self.bottles\_label = QtWidgets.QLabel(self.central\_widget)

self.bottles\_label.setGeometry(QtCore.QRect(1561, 50, 46, 16))

self.bottles\_label.setFont(font2)

self.bottles\_label.setText("bottles")

self.orders\_list = QtWidgets.QListWidget(self.central\_widget)

self.orders\_list.setGeometry(QtCore.QRect(1325, 140, 426, 291))

self.orders\_list.setWordWrap(True)

self.orders\_label = QtWidgets.QLabel(self.central\_widget)

self.orders\_label.setGeometry(QtCore.QRect(1330, 115, 91, 16))

self.orders\_label.setText("Orders:")

self.orders\_label.setFont(font)

self.bottled\_list = QtWidgets.QListWidget(self.central\_widget)

self.bottled\_list.setGeometry(QtCore.QRect(1325, 465, 431, 296))

self.bottled\_list.setWordWrap(True)

self.bottled\_label = QtWidgets.QLabel(self.central\_widget)

self.bottled\_label.setGeometry(QtCore.QRect(1330, 440, 200, 16))

self.bottled\_label.setText("Bottled and ready:")

self.bottled\_label.setFont(font)

self.file\_dir\_edit = QtWidgets.QLineEdit(self.central\_widget)

self.file\_dir\_edit.setGeometry(QtCore.QRect(467, 10, 341, 22))

self.file\_dir\_edit.setText("Enter file directory for new csv file")

self.add\_file\_button = QtWidgets.QPushButton(self.central\_widget)

self.add\_file\_button.setGeometry(QtCore.QRect(810, 7, 81, 26))

self.add\_file\_button.setText("Add File")

self.add\_file\_button.clicked.connect(self.add\_file)

self.explain\_label = QtWidgets.QLabel(self.central\_widget)

self.explain\_label.setGeometry(QtCore.QRect(900, 765, 846, 71))

self.explain\_label.setText("Prediction - Calculated by multiplying "

"latest sales data with daily growth rates.\n\n"

"Suggestions - Obtained by looking 6 weeks "

"into the future and calculating beer "

"with the most demand, along with the available equipment")

main\_window.setCentralWidget(self.central\_widget)

QtCore.QMetaObject.connectSlotsByName(main\_window)

LOGGER.info("Finished creating user interface")

# Refresh the page.

self.refresh\_page()

# Constantly save the batches and tanks.

process = Thread(target=save\_continuously, args=(), daemon=True)

process.start()

LOGGER.debug("New thread started")

if \_\_name\_\_ == "\_\_main\_\_":

LOGGER.info("Starting Program")

if hasattr(QtCore.Qt, 'AA\_EnableHighDpiScaling'):

LOGGER.info("High Dpi scaling enabled")

QtWidgets.QApplication.setAttribute(QtCore.Qt.AA\_EnableHighDpiScaling, True)

if hasattr(QtCore.Qt, 'AA\_UseHighDpiPixmaps'):

LOGGER.info("High Dpi Pixmaps enabled")

QtWidgets.QApplication.setAttribute(QtCore.Qt.AA\_UseHighDpiPixmaps, True)

APP = QtWidgets.QApplication(sys.argv)

WINDOW = QtWidgets.QMainWindow()

UI = UiMainWindow(WINDOW)

WINDOW.show()

sys.exit(APP.exec\_())

**Log\_file.log**

Example:

2019-12-06 02:17:45,415 - sales\_predictions - DEBUG - Start data changed to 1.1

2019-12-06 02:17:45,415 - sales\_predictions - DEBUG - Start data changed to 1.1

2019-12-06 02:17:45,415 - sales\_predictions - DEBUG - Start data changed to 1.1

2019-12-06 02:17:45,415 - sales\_predictions - DEBUG - Data and date added

2019-12-06 02:17:45,416 - sales\_predictions - DEBUG - Multiplying rate

2019-12-06 02:17:45,417 - sales\_predictions - DEBUG - Multiplying rate

2019-12-06 02:17:45,417 - sales\_predictions - DEBUG - Multiplying rate

2019-12-06 02:17:45,418 - sales\_predictions - DEBUG - All rates multiplied

2019-12-06 02:17:45,418 - sales\_predictions - INFO - Getting total from 2020-02-14 00:00:00

2019-12-06 02:17:45,418 - user\_interface - DEBUG - Beer with max demand fetched

2019-12-06 02:17:45,418 - inv\_management - INFO - Finding all available tanks

2019-12-06 02:17:45,418 - inv\_management - INFO - Finding all available tanks

2019-12-06 02:17:45,418 - user\_interface - INFO - There is a beer suggestion

2019-12-06 02:17:45,418 - user\_interface - DEBUG - There is a suggestions

2019-12-06 02:17:45,418 - user\_interface - INFO - Making function to connect to add batch button

2019-12-06 02:17:45,419 - user\_interface - DEBUG - Suggestion shown

2019-12-06 02:17:45,419 - user\_interface - INFO - Showing suggestion for batches to advance

2019-12-06 02:17:45,419 - inv\_management - INFO - Finding all the processes that are finished

2019-12-06 02:17:45,419 - inv\_management - INFO - Finding all the processes that are finished

2019-12-06 02:17:45,419 - user\_interface - DEBUG - Suggestions shown

2019-12-06 02:17:45,419 - inv\_management - INFO - Saving objects

2019-12-06 02:17:45,419 - inv\_management - INFO - Saving objects

2019-12-06 02:17:45,426 - user\_interface - INFO - Order removed successfully

2019-12-06 02:17:48,537 - user\_interface - INFO - Deliver button clicked

2019-12-06 02:17:48,537 - user\_interface - WARNING - Not enough inventory

**README.md:**

# Beer Management

Beer Management is a Python script for providing a user interface for

Barnaby's Brewhouse to manage the sale and organisation of beers.

## Framework

User interface built using [PyQt5](https://github.com/PyQt5)

## Features

\* Graph of prediction up to a year in the future

\* Ability to search through prediction

\* Interface to manage batches

\* Suggestions on what needs to be done

\* Interface to manage orders and inventory

## Installation

Python needs to be installed on the system. This can be done through the

official python website by downloading the installer and running it.

Check python is correctly installed by running the following command.

```bash

python --help

```

## Usage

To open the user interface, locate to the folder user\_interface.py is located in the commandline.

Then execute the following:

```bash

python user\_interface.py

```

Graph:

![Graph Image](Images/screenshot1.JPG)

\* The graph is a prediction obtained by multiplying the last sales data provided

for each beer provided with the daily growth rates for every day.

\* To search for a specific week or a month, enter the starting date for

that period, choose week or month from the drop down menu.

\* The full graph can be shown again by pressing the "Full Graph" button.

\* To add a csv file with new data, type in the full directory for the file

and press the "Add File" button. If successful, a message should appear

saying "success".

Batch Status and Tank Status:

![Status Image](Images/screenshot2.JPG)

\* The Batch Status shows all batches currently being produced.

\* To move a batch to the next stage in production, click on the "Next Step"

button next to it. If a tank is required for the next stage, a drop down

menu will appear to choose the tank from.

\* The Tank Status shows any tanks that is currently processing a batch.

Adding a new batch and refreshing the page:

![Add Image](Images/screenshot3.JPG)

\* To add a new batch manually, choose the beer type from the drop down menu,

enter the volume in integers and press the "Add Batch" button. Only batches

of up to 1000L can be entered as there is no tank that can handle more than

1000L.

\* The refresh button updates all the information being shown in the interface

such as the time each batch has been processing.

Suggest To Do:

![Suggestion Image](Images/screenshot4.JPG)

\* This is a list of actions that are suggested. There are two types of

suggestions that will be made. One to suggest to start brewing and one to

suggest to move a batch to the next stage.

\* Starting a batch will be suggested by looking at the beer with the highest

demand ten weeks into the future and the available facilities. Pressing the

"Execute" button next to the suggestion will add the batch with the

suggested volume automatically.

\* Moving a batch to the next stage will be suggested if the batch has been

processing in that stage for longer than the maximum time for that stage.

A button to do this automatically will appear if the next stage does not

require a tank.

Adding an order:

![Order Image](Images/screenshot5.JPG)

\* To add a new order, choose the type of beer, number of bottles ordered

and the due date for the order, then press the "Add Order" button.

\* The maximum number of bottles that can be ordered is 1000.

"Orders" and "Bottled and ready":

![Order and Ready Image](Images/screenshot6.JPG)

\* The "Orders" section shows all the orders that have not yet been

delivered.

\* The "Deliver" button will remove the corresponding number of bottles from

the inventory.

\* The "Bottled and ready" section shows each type of beer and how many bottles

bottles of it are ready to be delivered.

## Log file

Logs of what happened is recorded in a file called log\_file.log.