Meal Planner

TIA3: TERMINAL APP

By Kat Mountford

Key Features

1. **Add** a new recipe
2. **View** all recipes
3. **Choose** how many days to meal plan for
4. **Generate** a meal plan
5. **Optionally** regenerate a meal plan
6. **Save** the meal plan

```
Hey, welcome to Meal Planner! Please choose from one of the actions below to get started

What would you like to do?
a = Add a new recipe
v = View all recipes
n = Start a new meal plan
q = quit
```

FEATURE I

Adding recipes

There are two ways to add a new recipe to the recipes list (a CSV file called recipes.csv)

1. Add a new recipe via the main **user menu** in the application:

```
Hey, welcome to Meal Planner! Please choose from one of the actions below to get started

What would you like to do?
a = Add a new recipe
v = View all recipes
n = Start a new meal plan
q = quit
a
What is the name of the recipe?: Banana Split
What are the ingredients of the recipe? (For 1 serving): 2 Banana's, Cream, Sprinkles
How many calories in this recipe? (For 1 serving): 675
Your new recipe 'Banana Split' has successfully been added!
```

recipes.csv M

Users > kat > Documents > ca-projects > python > meal-planner > recipes.csv

```
275  Chai Latte,"Chai Tea, Soy Milk",210
276  Protein Smoothie,"Proten powder, milk, yoghurt, water",315
277  Banana Smoothie,"Banana, Milk, Ice",220
278  Banana Split,"2 Banana's, Cream, Sprinkles",675
279
```

```python
# Function to add recipe to CSV file
def add_recipe_to_CSV(new_recipe):
    # Append dictionary to the existing CSV file (all recipes)
    with open(r'recipes.csv', 'a', newline='') as f:
        fieldnames = ['title', 'ingredients', 'calories']
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writerow(new_recipe)

    # Confirmation for the user - recipe successfully added
    print(f"Your new recipe '{new_recipe['title']}' has successfully been added!")
```

Appends the new entry to recipes.csv using csv.DictWriter

FEATURE I

Adding recipes cont.

2. Add a new recipe via the **commandline** (through using argparse)

```
kat@Kats-MacBook-Air meal-planner % python3 main.py --title "Ravioli Carbonara" --ingredients "Ravioli, Carbonara Sauce, Bacon" --calories 812
Your new recipe 'Ravioli Carbonara' has successfully been added!
```

recipes.csv M

Users > kat > Documents > ca-projects > python > meal-planner > recipes.csv

```
277  Banana Smoothie,"Banana, Milk, Ice",220
278  Banana Split,"2 Banana's, Cream, Sprinkles",675
279  Ravioli Carbonara,"Ravioli, Carbonara Sauce, Bacon",812
```

Successfully **appends** to recipes.csv in the same way as previous process

Checks if any values have been passed from the cli before running the function to add the recipe - so it doesn't hinder the standard running of the application

```python
# Argparse to add recipe direct from the commandline
parser = argparse.ArgumentParser(description="Meal Planner: Meal planning made simple")

parser.add_argument("--title", help="Name of the recipe")
parser.add_argument("--ingredients", help="Ingredients for 1 serving")
parser.add_argument("--calories", type=int, help="Calories for 1 serving")

args = parser.parse_args()

# Check if a recipe has been added from the command line
if args.title or args.ingredients or args.calories:
    recipes.add_recipe_from_cli(args) # Call the add recipe via cli function and pass the args
```

FEATURE 2

View all recipes & navigation

Aside from accessing the CSV externally, the user can view all recipes **printed to their screen** within the application by pressing ‘v’ on the user menu:

After any completed action, the user is returned to the **user menu** so they can easily jump to the next task or quit the application

FEATURE 3

Choose how many days

When starting a new meal plan (User menu option = ‘n’), the user can choose **how many days** they want to get planned meals for:

```
What would you like to do?
a = Add a new recipe
v = View all recipes
n = Start a new meal plan
q = quit
n
How many days would you like to get a meal plan for?:
```

This is important for setting up a meal plan in the next step - as we loop through the meal plan process for the **range** of the number of days the user provides

```python
# Continuously loop until the user is happy with their meal plan
while True:
    all_meals = [] # Empty list to add meals to

    for i in range(num_days):
        day = Day(calorie_target) # Create a Day object and set the calorie range and daily calories
        day.set_meals() # Set the meals for the day
        all_meals.append(day) # Append set meals to all_meals list
```

FEATURE 4

Generating a meal plan

To generate a meal plan, a user next provides their daily **calorie target**.

At this stage a number of things happen; the first of which is that an **instance of the class Day** is created, and upon **initialisation**, the calorie target is stored and a function is called to automatically set the calories for each meal, based on the calorie target we just received.

```
How many days would you like to get a meal plan for?: 2
What is your daily calorie target? Please enter a target between 1400 - 3000 Calories: 1680
```

```python
for i in range(num_days):
    day = Day(calorie_target) # Create a Day object and set the calorie range and daily calories
    day.set_meals() # Set the meals for the day
    all_meals.append(day) # Append set meals to all_meals list
```

```python
class Day():
    def __init__(self, calorie_target):
        self.calorie_target = calorie_target
        self.meal_calories = {}  # Store meal calories based on the users calorie target
        # Set the meal calories (based on the calorie range) each time a day is initialized
        self.set_meal_calories()
        self.todays_meals = []  # Empty list for todays meals
        self.semantic_names = ["Breakfast","Lunch", "Dinner", "Snack 1", "Snack 2"]
        self.result = None
```

FEATURE 4

Meal calorie ranges

Based on the calorie target the user enters, this will determine what their **meal calories** are for each meal in a day.

This is both to provide the program the best chance at arriving at a meal plan within the daily calorie range for the user, and to ensure meal sizes make general sense (e.g. generally people's snacks are smaller and lower in calories than their dinners).

1. **Conditional** to check the range of the calorie target the user has entered

2. Once a match has been found, **set the calories for each meal** as a dictionary.

```
if 1400 <= self.calorie_target <= 1700:
    self.meal_calories = {
        "M1": 300,
        "M2": 350,
        "M3": 500,
        "M4": 200,
        "M5": 200,
    }
elif 1701 <= self.calorie_target <= 2000:
    self.meal_calories = {
        "M1": 400,
        "M2": 400,
        "M3": 600,
        "M4": 300,
        "M5": 200,
    }
```

M1 = Meal 1 (breakfast)
M2 = Meal 2 (Lunch)
M3 = Meal 3 (Dinner)
M4 = Meal 4 (Snack 1)
M5 = Meal 5 (Snack 2)

FEATURE 4

Meal calorie ranges cont.

Setting the meals in the meal plan is broadly done by looping through each of the meal calories that have been set, and then **finding a meal that matches** those calories from the recipe list and save those ‘matching’ meals to a new list titled todays_meals.

However, it’s very unlikely that there will be recipe in our recipe list with the exact calories that have been set for each meal. So to account for this, I’ve allowed a range of +- 75 calories each side of the meal calorie that’s been set - so we can get pretty close to the specified range, but allow some variation.

```
for calories in self.meal_calories.values():
    # For the value in that dict (cals), +- 75 either side
    min_cal = calories - 75
    max_cal = calories + 75
    meal_found = False

    # Iterate over the recipes to find a meal, and check it's in the calorie range.
    # While it's not, keep searching, if it is, add it to today's meals and set meal_found to True
    while meal_found == False:
        for recipe in recipes:
            # Find the 'calories' value
            value = int(recipe.get('calories'))
            if min_cal < value < max_cal:  # Check value within acceptable range
```

FEATURE 4

Searching for recipes

I also realised it was inefficient to continuously open and close the csv in order to search for a meal through each loop, so alternatively at the beginning of set_meals() a **copy of all the recipes** in the CSV is made and **stored in a new list**. This list is also **shuffled** to ensure that users are served a variety of meals each time rather than potentially getting the same ones over and over for a given calorie range.

```python
# Function to find and select meals for the specified calorie_ranges
def set_meals(self):
    attempts = 0  # Num of attempts to set a valid meal plan
    meal_completed = False
    recipes = self.shuffle_recipes()
```

```python
def shuffle_recipes(self):
    recipe_list = get_recipes()  # Get all recipes (as a dictionary)
    # Shuffle recipes so it's a random one being returned
    random.shuffle(recipe_list)
    return recipe_list
```

```python
for recipe in recipes:
    # Find the 'calories' value
    value = int(recipe.get('calories'))
    if min_cal < value < max_cal:  # Check value within acceptable range
        # Append to todays meals
        self.todays_meals.append(recipe)
        # Remove from the recipe list (so as to not duplicate on the same day)
        recipes.remove(recipe)
        meal_found = True
        break  # Exit the loop
```

Recipes are also removed from the list once they've been 'found' and added to todays_meals, to increase variation and prevent duplicate meals across multiple days.

FEATURE 4

The set_meals() function

With a shuffled recipe list and meal_calories set, for each of the meals, we can loop through the list of recipes and **search** for recipes where the **calories for that recipe** are **within the meal_calories range**

for Loop to iterate through each value of meal_calories

If calories in the recipe are within the range of the meal_calories - append the meal to today's meals and move onto the next meal by setting meal_found = True

```python
# Function to find and select meals for the specified calorie_ranges
def set_meals(self):
    attempts = 0  # Num of attempts to set a valid meal plan
    meal_completed = False
    recipes = self.shuffle_recipes()

    # While todays meals is incomplete
    while meal_completed == False and attempts < 1000:

        for calories in self.meal_calories.values():
            # For the value in that dict (cals), +- 75 either side
            min_cal = calories - 75
            max_cal = calories + 75
            meal_found = False

            # Iterate over the recipes to find a meal, and check it's in the calorie range.
            # While it's not, keep searching, if it is, add it to today's meals and set meal_found to True
            for recipe in recipes:
                # Find the 'calories' value
                value = int(recipe.get('calories'))
                if min_cal < value < max_cal:  # Check value within acceptable range
                    # Append to todays meals
                    self.todays_meals.append(recipe)
                    # Remove from the recipe list (so as to not duplicate on the same day)
                    recipes.remove(recipe)
                    meal_found = True
                    break  # Exit the loop

            if meal_found == False:
                no_meal_err = {}
                no_meal_err = {"title":"Oops! No meal found. This can happen if there aren't enough recipes calorie range for
                your target. Try adding some more recipes!", "ingredients": "None", "calories": 0}
                self.todays_meals.append(no_meal_err)
```

Nested for loop in order to iterate through each recipe in the list to find a matching meal within the range of the meal_calories

FEATURE 4

The set_meals() function

Depending on the number of meal plans being generated, the calorie_target that has been set from the user and how many recipes of varying calorie ranges are in the recipes.csv; there is a chance that for some meals the program **may not be able to find a match within suitable range**.

I wanted to ensure that in this case, the program continued to run and attempt to still match other meals. For example in low calorie ranges, the daily calorie target is often hit with simply 3 main meals and 1 snack (or 4 meals total, rather than 5) - this is reasonable and should still count as a valid meal plan.

```
        # Iterate over the recipes to find a meal, and check it's in the calorie range.
        # While it's not, keep searching, if it is, add it to today's meals and set meal_found to True
        for recipe in recipes:
            # Find the 'calories' value
            value = int(recipe.get('calories'))
            if min_cal < value < max_cal:  # Check value within acceptable range
                # Append to todays meals
                self.todays_meals.append(recipe)
                # Remove from the recipe list (so as to not duplicate on the same day)
                recipes.remove(recipe)
                meal_found = True
                break  # Exit the loop

        if meal_found == False:
            no_meal_err = {}
            no_meal_err = {"title":"Oops! No meal found. This can happen if there aren't enough recipes calorie range for
            your target. Try adding some more recipes!", "ingredients": "None", "calories": 0}
            self.todays_meals.append(no_meal_err)
```

Message is appended to todays_meals in the same structure (dictionary) as valid meals are in order to continue the iteration seamlessly for remaining meals yet still clearly state to the user what has happened.

FEATURE 4

The set_meals() function cont.

Once we have today's meals completed (1 for each meal - 5 meals total), then another check occurs to sum the calorie values across all meals for the day (todays_meals); and compare if the calorie total is close to the calorie target the user set initially. If it's **within +- 100 of the users calorie target, then it's accepted**.

```
# Check that meal total is acceptable close to the users target calories
total_calories = 0
# Sum each meals calories together to find the total
for meal in self.todays_meals:
    total_calories += int(meal['calories'])

# Set the range to be acceptable based on the daily calories goal
daily_min_cal = self.calorie_target - 100
daily_max_cal = self.calorie_target + 100

# If within target, mark meal as completed
if daily_min_cal < total_calories < daily_max_cal:
    meal_completed = True

# Otherwise, start again
else:
    self.todays_meals = []
    self.shuffle_recipes()
    attempts += 1
```

If it's not within the acceptable range, then the process is **repeated** with new meals until the summed calories ***are*** within an acceptable range.

Each time resetting todays_meals and getting a refreshed list of shuffled recipes.

FEATURE 4

The set_meals() function cont.

The number of attempts at getting a meal plan within an acceptable range is tracked and incremented each time the program has to re-try

```
# Function to find and select meals for the specified calorie_ranges
def set_meals(self):
    attempts = 0  # Num of attempts to set a valid meal plan
    meal_completed = False
    recipes = self.shuffle_recipes()

    # While todays meals is incomplete
    while meal_completed == False and attempts < 1000:
```

Variable to store the number of attempts made

Only continue whilst attempts are under 1000

```
    # If within target, mark meal as completed
    if daily_min_cal < total_calories < daily_max_cal:
        meal_completed = True

    # Otherwise, start again
    else:
        self.todays_meals = []
        self.shuffle_recipes()
        attempts += 1
```

Increment attempts upon re-try

If any of the full day meal plans returned are blank (e.g. after 1000 tries there weren't any valid combinations to meet the accepted range for the users calorie target) then a clear message is returned to the user explaining what has happened

```
# Continuously loop until the user is happy with their meal plan
while True:
    all_meals = [] # Empty list to add meals to

    for i in range(num_days):
        day = Day(calorie_target) # Create a Day object and set the calorie range and daily calories
        day.set_meals() # Set the meals for the day
        all_meals.append(day) # Append set meals to all_meals list

    if day.todays_meals == []:
        print(f"\nSorry, we weren't able to find any suitable meals for you this time. This can happen if there
        aren't enough meals in your calorie range.\nPlease try again with a different calorie range or try adding
        some more recipes!\n")
        break
```

FEATURE 4

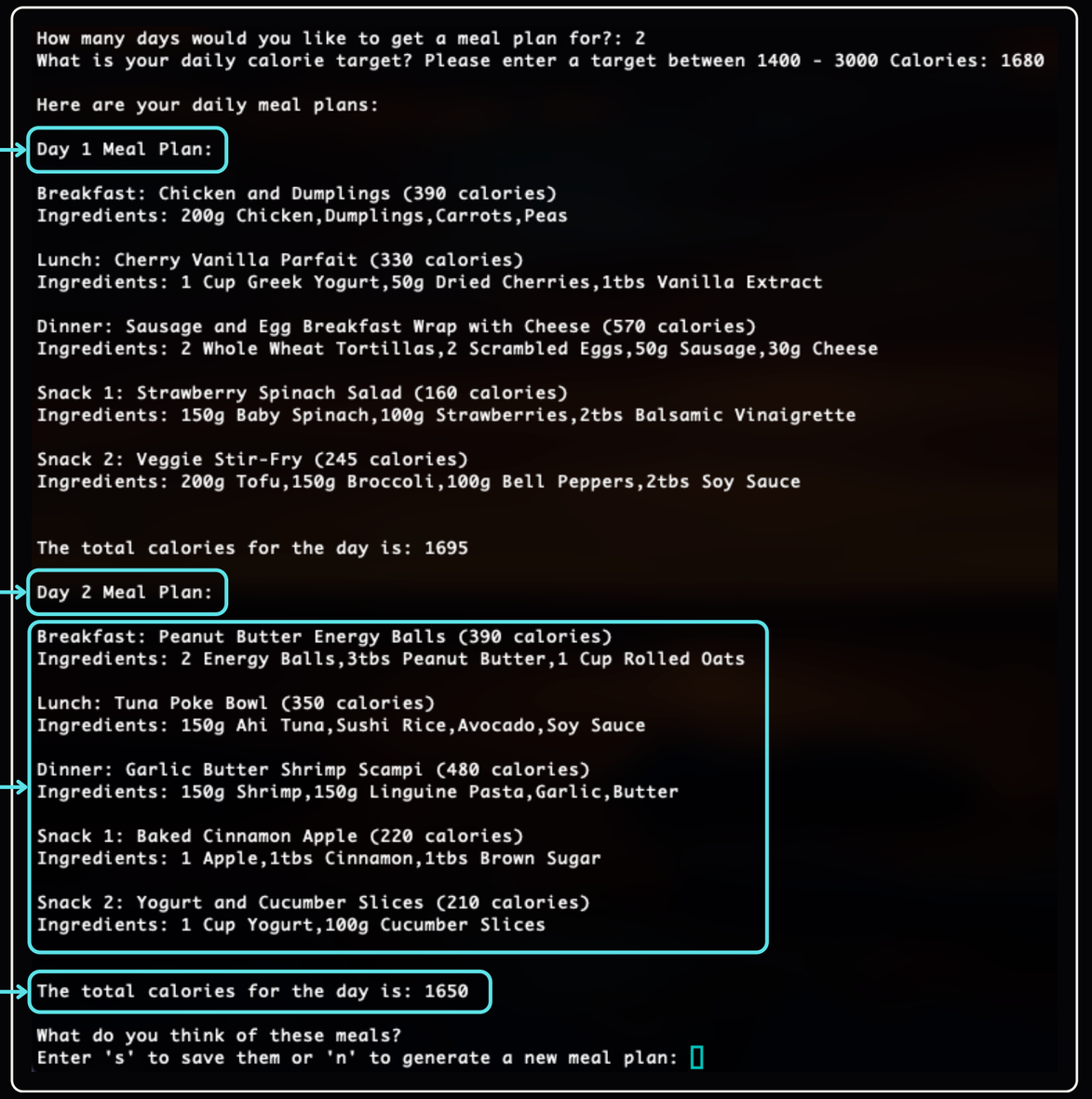
Displaying the meal plan

Assuming a valid combination of meals is found that is within the acceptable range to the users calorie target, then the meals are displayed to the user

Day number

Meal information, broken down into each meal; with semantic meal names for easy readbility

Total calories for the day

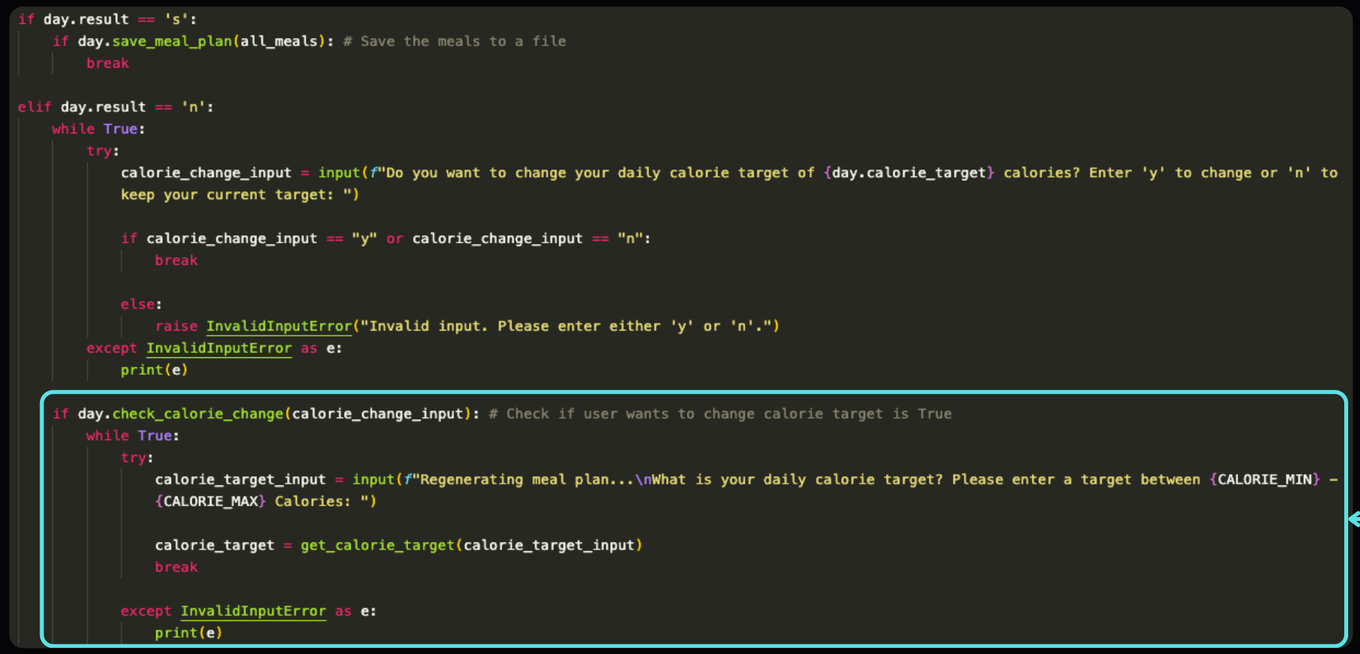


FEATURE 5

Optional regeneration

The user is then asked if they're happy with the meals and can choose to either save them or generate a new meal plan. If opting to regenerate, the user can additionally optionally reset their daily calorie target

```
What do you think of these meals?
Enter 's' to save them or 'n' to generate a new meal plan: n
Do you want to change your daily calorie target of 1680 calories? Enter 'y' to change or 'n' to keep your current target: y
Regenerating meal plan...
What is your daily calorie target? Please enter a target between 1400 - 3000 Calories:
```



```python
if day.result == 's':
    if day.save_meal_plan(all_meals): # Save the meals to a file
        break

elif day.result == 'n':
    while True:
        try:
            calorie_change_input = input(f"Do you want to change your daily calorie target of {day.calorie_target} calories? Enter 'y' to change or 'n' to
            keep your current target: ")

            if calorie_change_input == "y" or calorie_change_input == "n":
                break

            else:
                raise InvalidInputError("Invalid input. Please enter either 'y' or 'n'.")
        except InvalidInputError as e:
            print(e)

    if day.check_calorie_change(calorie_change_input): # Check if user wants to change calorie target is True
        while True:
            try:
                calorie_target_input = input(f"Regenerating meal plan...\nWhat is your daily calorie target? Please enter a target between {CALORIE_MIN} -
                {CALORIE_MAX} Calories: ")

                calorie_target = get_calorie_target(calorie_target_input)
                break

            except InvalidInputError as e:
                print(e)
```

Nested if statement, so only if the user wants to regenerate their plan AND chooses to reset their target, will they be prompted to set a new calorie_target, otherwise the meal plan will be regenerated using the original calorie goal

FEATURE 6

Saving the meal plan

Assuming the user is happy with their meal plan, they can then choose to save it. This will save the meal plan to a text file in the source directory for them, with a name of their choosing.

```
What do you think of these meals?
Enter 's' to save them or 'n' to generate a new meal plan: s
Great, let's save these for you. What should the save file be called?: my_meal_plan

Your meals have been saved!
```

```
kat@Kats-MacBook-Air meal-planner %
 ls
README.md
__pycache__
docs
main.py
meals.py
my_meal_plan.txt
recipes.csv
recipes.py
requirements.txt
setup.sh
test_main.py
test_meals.py
```

my_meal_plan.txt

```
Retrieved on 25-10-2023 at 14:21.

Day 1 Meal Plan:
Breakfast: Rice Cakes with Nutella (360 calories)
Ingredients: 2 Rice Cakes,2tbs Nutella
Lunch: Fruit and Cheese Quesadilla (350 calories)
Ingredients: 1 Whole Wheat Tortilla,50g Cheese,100g Sliced Fruit
Dinner: Chicken Spinach Salad (630 calories)
Ingredients: 100g Grilled Chicken, 50g Baby Spinach
Snack 1: Chocolate Banana Smoothie (280 calories)
Ingredients: 1 Banana,2tbs Cocoa Powder,1 Cup Almond Milk
Snack 2: Fruit Salad (180 calories)
Ingredients: 100g Mixed Fruit,1tbs Lemon Juice,1tbs Honey
The total calories for the day is: 1800

Day 2 Meal Plan:
Breakfast: Portobello Mushroom Burger (340 calories)
Ingredients: 2 Portobello Mushrooms,Swiss Cheese,Brioche Bun
Lunch: Blueberry Pancakes (390 calories)
Ingredients: 2 Blueberry Pancakes,2tbs Maple Syrup
Dinner: Peanut Butter Banana Toast (620 calories)
Ingredients: 2 Slices Whole Wheat Bread, 2tbsp Peanut Butter
Snack 1: Cinnamon Roll Oatmeal (280 calories)
Ingredients: 1 Cup Oatmeal,1tbs Cinnamon,1tbs Brown Sugar
Snack 2: Rice Cakes with Hummus (240 calories)
Ingredients: 2 Rice Cakes,3tbs Hummus
The total calories for the day is: 1870
```

Date and time the meal plan was saved

Meal and calorie details (same as displayed on screen to the user)

Thoughts & Challenges

Favourite Parts

- Learning how to write scripts! Never done this before so was fun to learn
- Researching and trying python packages I'd not yet heard of; there's so much you can do with them!

Challenges

- Initially not having written easily testable code and so having to refactor in order to improve testability
- Managing potential exceptions when dealing with random data was challenging; particularly to replicate