Assessment Information **1**

Task 1 Building a program for stamp selling **4**

Task 2: Wrangling and analysing sale data **6**

Finding and fixing the problems in the sale data 6

Sale data analysis 6

Documentation **6**

Grading Rubric **7**

Grading rubric 7

Penalties 7

Submission **7**

Assessment Information

In this assessment, you are required to complete the interactive program to be installed in a stamp-selling machine in Post Office.

In the previous assignment, you have completed most of the important functionalities of a stamp item, e.g., price calculation. In this assignment, the **Stamp** class is provided to you in the module **stamp.** The Stamp class has the following instance variables:

instance variables	**Type**	**description**
weight	float	the weight of a stamp item in kg
country_name	str	The country name
item_type	str	The type of stamp item, either “letter” or “parcel”
zone_number	int	the zone number for the given country name
unit_price	float	the price for a single stamp item

There are a list of implemented methods of the **Stamp** class:

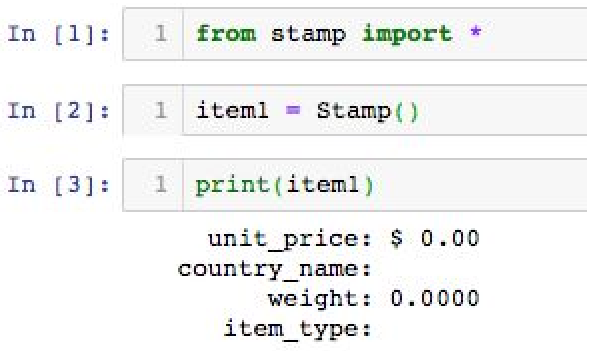
- __str__(self)

Return the string representation of the content of the current instance (self).

- __init__(self)

This is the constructor that is required for creating **Stamp** instances. It only does the following:

- Create a Stamp item instance
- Initialize the weight value as 0.0, initialize country_name and item_type values as an empty string ‘’.



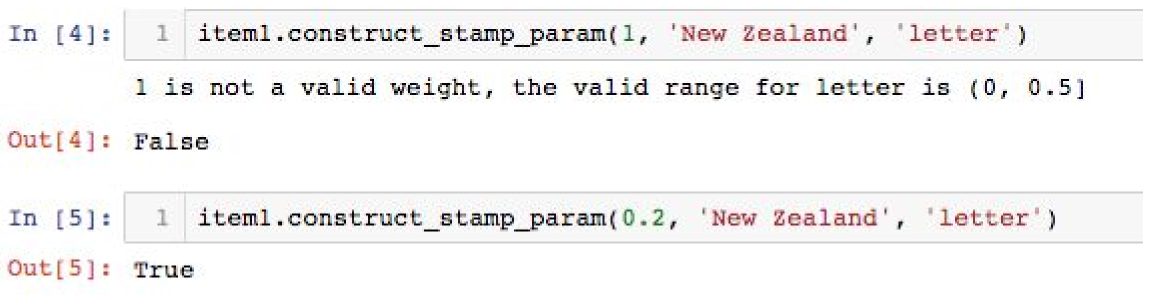
```
In [1]: from stamp import *

In [2]: item1 = Stamp()

In [3]: print(item1)

   unit_price: $ 0.00
 country_name:
       weight: 0.0000
    item_type:
```

- construct_stamp_param(self, weight = 0, country_name = ‘’, item_type = ‘’)

This is the method you could use to construct the created stamp item by setting up the values its weight, country_name and item_type, based on the input arguments:

Parameter	**type**	**Description**
weight	float	the weight of a stamp item in kg
country_name	str	The country name
item_type	str	The type of stamp item, either “letter” or “parcel”

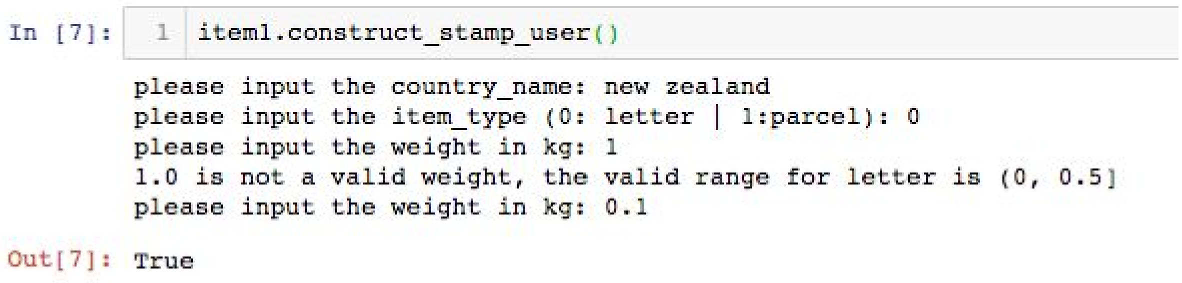
Note: This function will validate the input arguments. It returns True if the construction is successfully conducted. Otherwise, if any of the input arguments is invalid, it will return False. For example:



```
In [4]: item1.construct_stamp_param(1, 'New Zealand', 'letter')

        1 is not a valid weight, the valid range for letter is (0, 0.5]

Out[4]: False

In [5]: item1.construct_stamp_param(0.2, 'New Zealand', 'letter')

Out[5]: True
```

- construct_stamp_user(self):

This is the method you could use to construct the created stamp item by setting up the values of its weight, country_name and item_type, based on interactive user inputs. It will prompt the user to enter the valid information to fill in the data of the stamp object. For example:

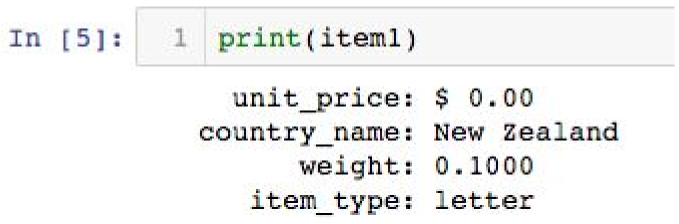


```
In [7]: item1.construct_stamp_user()

        please input the country_name: new zealand
        please input the item_type (0: letter | 1:parcel): 0
        please input the weight in kg: 1
        1.0 is not a valid weight, the valid range for letter is (0, 0.5]
        please input the weight in kg: 0.1

Out[7]: True
```

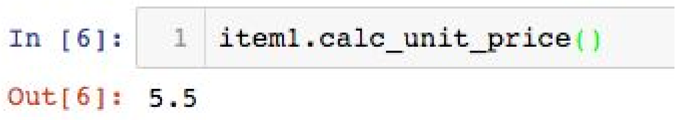
Note: This function will also validate the user inputs as shown in the example.

- calc_unit_price(self)

Return the price of the stamp object.



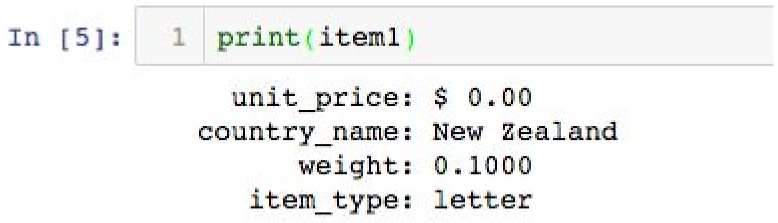
```
In [5]: print(item1)

             unit_price: $ 0.00
           country_name: New Zealand
                 weight: 0.1000
              item_type: letter
```



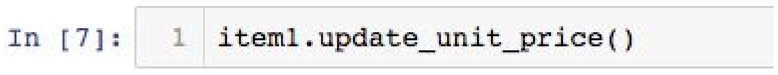
```
In [6]: item1.calc_unit_price()

Out[6]: 5.5
```

- update_unit_price(self)

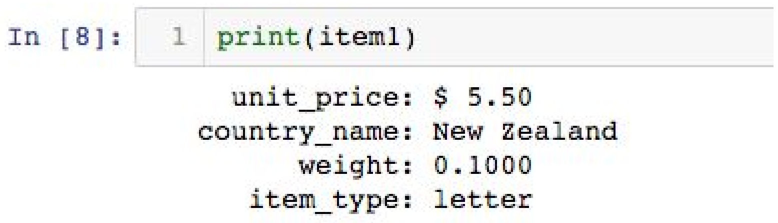
This method first calculates the unit price of the stamp item, and then updates the value of its instant variable **unit_price**.



```
In [5]: print(item1)

             unit_price: $ 0.00
           country_name: New Zealand
                 weight: 0.1000
              item_type: letter
```



```
In [7]: item1.update_unit_price()
```

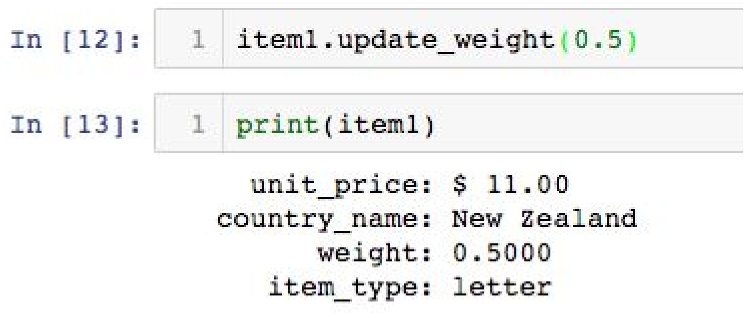


```
In [8]: print(item1)

             unit_price: $ 5.50
           country_name: New Zealand
                 weight: 0.1000
              item_type: letter
```

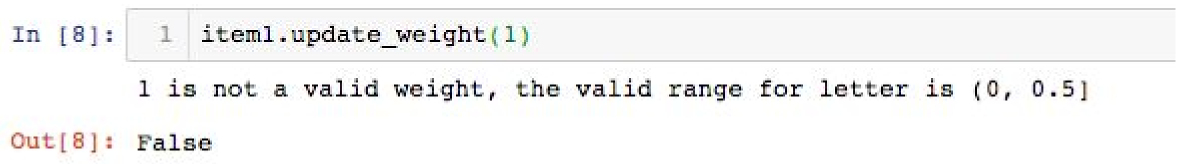
- update_weight(self, weight)

This is the method you could use to update the weight value of a Stamp item.

Note: This method will validate the input argument. Only if the input argument weight is in the valid range, it will update the weight and the (recalculated) unit_price of the Stamp item instance; Otherwise it will return False.



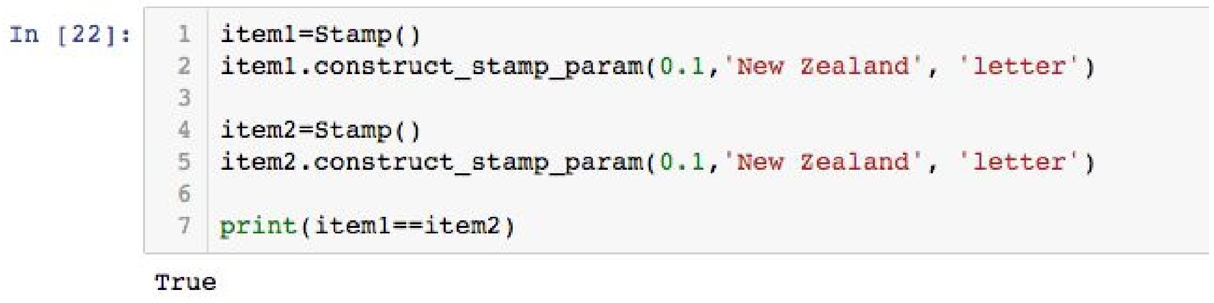
```
In [12]: item1.update_weight(0.5)

In [13]: print(item1)

     unit_price: $ 11.00
   country_name: New Zealand
         weight: 0.5000
      item_type: letter
```



```
In [8]: item1.update_weight(1)

        1 is not a valid weight, the valid range for letter is (0, 0.5]

Out[8]: False
```

- __eq__(self, other_item)

This is the magic method __eq__.
It returns True, if and only if both the current instance (self) and the other_item have the same country_name, item_type and weight, otherwise it returns False.



```
In [22]: item1=Stamp()
         item1.construct_stamp_param(0.1,'New Zealand', 'letter')

         item2=Stamp()
         item2.construct_stamp_param(0.1,'New Zealand', 'letter')

         print(item1==item2)

         True
```

Task 1 Building a program for stamp selling

In this task, you are required to write a program for stamp selling as an basic interface to a user by using the provided Stamp class.
Your program should support continuous shopping activities in Post Office. If you demand the next user to restart the program each time they do shopping, your program does not fulfil this requirement. However, you could assume that it serves only one customer at a time, i.e., the next customer will only start shopping after the previous customer checkouts or quits the program.

This program should provide the following functionalities as a stamp-selling machine.

- [Fun 1] Add an item to the shopping cart

Your program should allow the user to add items to his/her shopping cart. If a user tries to add a duplicate item into the shopping cart, your program should prompt the user to confirm whether or not he/she wants to continue this action.

- [Fun 2] View shopping cart

Users should be able to view their shopping cart. Once requested, your program should print out the list of items & costs, as well as the total cost of the items in the shopping cart.

- [Fun 3] Amend an item

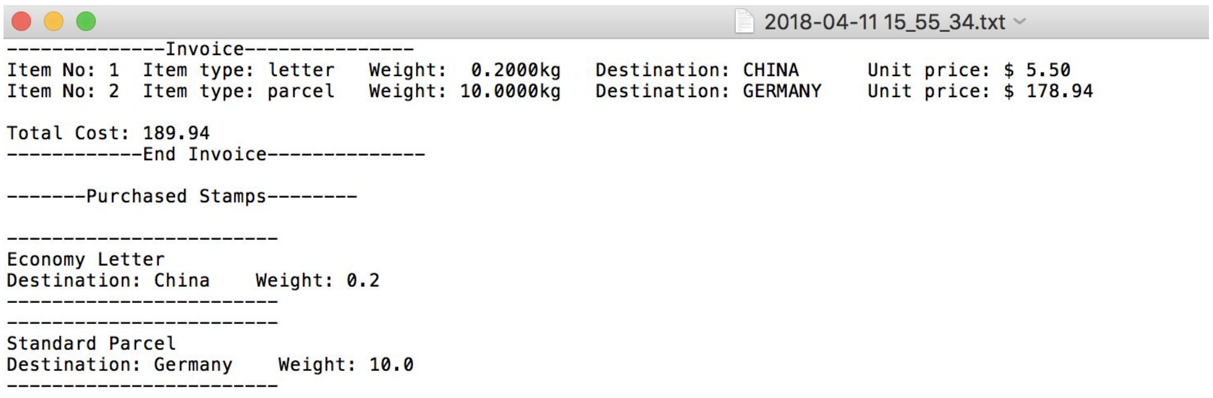
Your program should allow the user to amend an item in the shopping cart, limited to its weight only.

- [Fun 4] Remove an item from the shopping cart

Your program should allow the user to delete an item in the shopping cart. It should ask the users to confirm the deletion before removing an item from the cart.

- [Fun 5] Checkout

Once a customer chooses to check out, the program should print out the invoice along with a list of purchased stamps to a .txt file. This file should have a unique name, indicating the shopping date and time (see the example in the following figure). It is assumed that the list of stamps can be divided and used by the users in their post items. Then, staff in Post Office can check the stamps and arrange the delivery. The format of the invoice should looks like the following figure.



```
--------------Invoice---------------
Item No: 1  Item type: letter   Weight:  0.2000kg   Destination: CHINA     Unit price: $ 5.50
Item No: 2  Item type: parcel   Weight: 10.0000kg   Destination: GERMANY   Unit price: $ 178.94

Total Cost: 189.94
------------End Invoice--------------

-------Purchased Stamps--------

------------------------
Economy Letter
Destination: China    Weight: 0.2
------------------------
------------------------
Standard Parcel
Destination: Germany    Weight: 10.0
------------------------
```

Then, your program will also need to **append** the record to the given "**sales_history.csv**" file. Each record contains the information for each sold item:

Field	Description
sale_id	this is a unique identifier for a sale record. You can choose to use an incremental key for the sale_id, e.g., the id for the current sale = the id of the previous sale + 1. Note that a sale only occurs when a customer successfully checks out. A sale (to a customer) could contain multiple stamp items, but they are of the same sale_id.
date_time	date & time of the purchased in *yyyy-mm-dd hh:mm:ss* format
weight	The weight of the item (in kg)
destination country	The destination country of where the item will be sent to
item type	The type of item is one of the following values: letter or parcel.
cost	The cost of the item

Note that you should not change the name and format of this file.

Task 2: Wrangling and analysing sale data

In this task, you are required to find out a list of data problems in the "**sales_history.csv**" file provided to you and fix those problems. You are also require to do some analysis based on the cleaned "**sales_history.csv**" file.

Finding and fixing the problems in the sale data

In this subtask, you will need to identify the following data problems in the "**sales_history.csv**" file and fix these problems.

- date format mismatch
- misspelling of item type
- country name inconsistency

The records in the cleaned data file should be formatted in the same way as that mentioned in Task 1

Sale data analysis

Then analyse the sale data and provide the basic statistics on the following:

- The gross sales amount of different years
- The customer flow during a day, e.g., you can compare the number of sales occur at different time (hours) of a day
- The popularity of item type for letter and parcel, e.g., you could count the number of purchased stamps with different item types.
- The top 5 most popular destination countries including any postage.

Wherever suits, you should produce graphs or plots to present these results (You may want to consider the pandas and 'matplotlib' libraries in this part of the implementation.)

Documentation

Precise and concise comments/documents of your code are essential as part of assessment criteria. Do Not include things that are irrelevant in your code as that will reduce your code readability.

On the other hand, you are also required to provide the user manual for your program.

Grading Rubric

Grading rubric

Task	**mark allocation**
Task 1 Building a program for stamp selling	40%
Task 2 Wrangling and analysing sale data	55%
User manual	5%

For task 1

Parts	**mark allocation**
Functionality	70%
Testing	20%
Code style & architect	10%

For task 2

Parts	**mark allocation**
Functionality	70%
Code style & architect & documentation	30%

Penalties

- Late submission: -5% per day late, no submission accepted after 4 days.

For further details on late submission, **read the policy here**.

Submission

- Rename "rename_me" directory to your student ID. This directory should contain Stamp module, stamp related data, sale history data, and **your source code, testing code and user manual**.
- For task 1:
  - Write all the **source code** in python files **(*.py)** instead of jupyter notebooks
  - Write the **testing code** in the jupyter notebooks named **test_stamp_selling.ipynb**
- For task 2:
  - Write all the code including the documentation in a jupyter notebook named **wrangling_and_analysis.ipynb**
- the user manual should be in pdf format named **'user manual.pdf'**
- Do not include any unnecessary file in this folder
- Zip the containing folder with the same name (ie <student_numnber>.zip) and upload it

Note:

- Up to 20% penalties will be applied if unable to follow the above submission instructions. Feel free to contact the teaching team if there is anything unclear.
- The stamp module (stamp.pyc) is only compatible with **python version 3.5.2**. If you don't have this python version installed in your own computer, you could upload all the files and run your program in the jupyterhub to do testing.