# Week 10 - Object-Oriented Programming

We have been using a built-in data structure (e.g. list of tuples) to store the records. However, using class **Record** is a better way since it can represent "amount of a record" rather than "the third item of a tuple".

In addition, we have many functions dealing with the records, and many functions dealing with the categories. We can use two different classes, **Records** and **Categories**, to hold the functions as their methods.

Below are the templates for the three classes, **Record**, **Records**, and **Categories**.

(Comments in green are implementation instructions for you, not docstrings.)

| class Record:  """Represent a record."""  def \_\_init\_\_(self, ...):  # 1. Define the formal parameters so that a Record can be instantiated  # by calling Record('meal', 'breakfast', -50).  # 2. Initialize the attributes from the parameters. The attribute  # names should start with an underscore (e.g. self.\_amount)  # Define getter methods for each attribute with @property decorator.  # Example usage:  # >>> record = Record('meal', 'breakfast', -50)  # >>> record.amount  # -50 |
| --- |
| class Records:  """Maintain a list of all the 'Record's and the initial amount of money."""  def \_\_init\_\_(self):  # 1. Read from 'records.txt' or prompt for initial amount of money.  # 2. Initialize the attributes (self.\_records and self.\_initial\_money)  # from the file or user input.  def add(self, ...):  # 1. Define the formal parameter so that a string input by the user  # representing a record can be passed in.  # 2. Convert the string into a Record instance.  # 3. Check if the category is valid. For this step, the predefined  # categories have to be passed in through the parameter.  # 4. Add the Record into self.\_records if the category is valid.  def view(self):  # 1. Print all the records and report the balance.  def delete(self, ...):  # 1. Define the formal parameter.  # 2. Delete the specified record from self.\_records.  def find(self, ...):  # 1. Define the formal parameter to accept a non-nested list  # (returned from find\_subcategories)  # 2. Print the records whose category is in the list passed in  # and report the total amount of money of the listed records.  def save(self):  # 1. Write the initial money and all the records to 'records.txt'. |
| class Categories:  """Maintain the category list and provide some methods."""  def \_\_init\_\_(self):  # 1. Initialize self.\_categories as a nested list.  def view(self, ...):  # 1. Define the formal parameters so that this method  # can be called recursively.  # 2. Recursively print the categories with indentation.  # 3. Alternatively, define an inner function to do the recursion.  def is\_category\_valid(self, ...):  # 1. Define the formal parameters so that a category name can be  # passed in and the method can be called recursively.  # 2. Recursively check if the category name is in self.\_categories.  # 3. Alternatively, define an inner function to do the recursion.  def find\_subcategories(self, ...):  # 1. Define the formal parameters so that a category name can be  # passed in and the method can be called recursively.  # 2. Recursively find the target category and call the  # self.\_flatten method to get the subcategories into a flat list.  # 3. Alternatively, define an inner function to do the recursion.  def \_flatten(self, ...):  # 1. Define the formal parameters so that this method  # can be called recursively.  # 2. Recursively call self.\_flatten and return the flat list.  # 3. (FYI) The method name starts with an underscore to indicate that  # it is not intended to be called outside the class.  # 4. Alternatively, put flatten as an inner function of  # find\_subcategories. |

With the classes defined, we can write the remaining codes in a more object-oriented way.

| import sys  # class definitions here  categories = Categories()  records = Records()  while True:  command = input('\nWhat do you want to do (add / ...)? ')  if command == 'add':  record = input('Add some expense or income records with category, description, and amount (separate by spaces):\ncat1 desc1 amt1, cat2 desc2 amt2, cat3 desc3 amt3, ...\n')  records.add(record, categories)  elif command == 'view':  records.view()  elif command == 'delete':  delete\_record = input("Which record do you want to delete? ")  records.delete(delete\_record)  elif command == 'view categories':  categories.view()  elif command == 'find':  category = input('Which category do you want to find? ')  target\_categories = categories.find\_subcategories(category)  records.find(target\_categories)  elif command == 'exit':  records.save()  break  else:  sys.stderr.write('Invalid command. Try again.\n') |
| --- |

## Required Steps

1. Define the **Record** and **Records** class according to the given templates.
   1. Instantiate a **Records** before the **while** loop and call the methods in **if-elif** instead of calling the functions you defined last time.
   2. Run the code with all commands and different cases to make sure it works so far.
   3. After that, you may remove the functions that are no longer being called.
2. Define the **Categories** class according to the given template.
   1. Instantiate a **Categories** before the while loop and call the methods in **if-elif** instead of calling the functions you defined last time.
   2. Run the code with all commands and different cases to make sure it works.
   3. After that, you may remove the functions that are no longer being called.
3. Add at least one line of triple-quote docstring to each method.
4. Make sure all of the variable and method names are in snake\_case, and the attribute names start with an underscore.

## Related Knowledge

* Constructor
* Attributes
* Methods
* @property decorator