# Week 4 - Python Applications & String

We can create a variety of applications with Python. In week 4, we saw how to use Python to implement some Unix tools (uniq, cat, grep, wc). Those might appear to be boring to you because you don't use uniq in your daily life. How about creating an application that you really need every day? You can come up with your own ideas, but here we'll start with a money management application.

Let's start from the very beginning. Write a program that lets a user input the initial amount of money and one record of expense or income. The program will report the balance.

$ python3 pymoney.py

How much money do you have? 1000

Add an expense or income record with description and amount:  
breakfast -50

Now you have 950 dollars.

## Required Steps

1. Use **input()** function to accept the inputs.
2. Use **str.split()** method to deal with the input record.
3. Use **int()** function to convert the value.
4. Calculate the balance.
5. Use string formatting and **print()** to report the balance to the user.

## Notes

* The amount input by the user could be negative (representing expense) or positive (representing income). The "+" sign is optional for positive numbers. If you use the **int()** function, both strings "95" and "+95" are converted to the integer value 95.
* You might get two strings returned from the **split()** method. The first one is the description (e.g. "breakfast") and the second one is the amount of expense or income (e.g. "-50"). For now, your program doesn't need to handle the first string. Just convert the second string to integer and do the remaining operations.

## Related Knowledge

* input() and print() functions
* Operators
* Integers
* Variables
* Strings
* str.split() method
* String formatting

# Week 5 - Sequences, Sets, Dictionary

One record is never enough. Let's make it accept multiple records.

$ python3 pymoney.py

How much money do you have? 1000

Add some expense or income records with description and amount:

desc1 amt1, desc2 amt2, desc3 amt3, ...  
breakfast -50, lunch -70, dinner -100, salary 3500

Here's your expense and income records:

breakfast -50

lunch -70

dinner -100

salary 3500

Now you have 4280 dollars.

## Required Steps

1. Modify the prompt strings.
2. Split the input string by commas then spaces and build a data structure like [('breakfast', -50), ('lunch', -70), ...].
3. List the records and report the balance to the user.
4. Add some comments for your code.

## Related Knowledge

* str.split() method
* Conversion to list or tuple
* List comprehension
* str.join() method
* sum() function

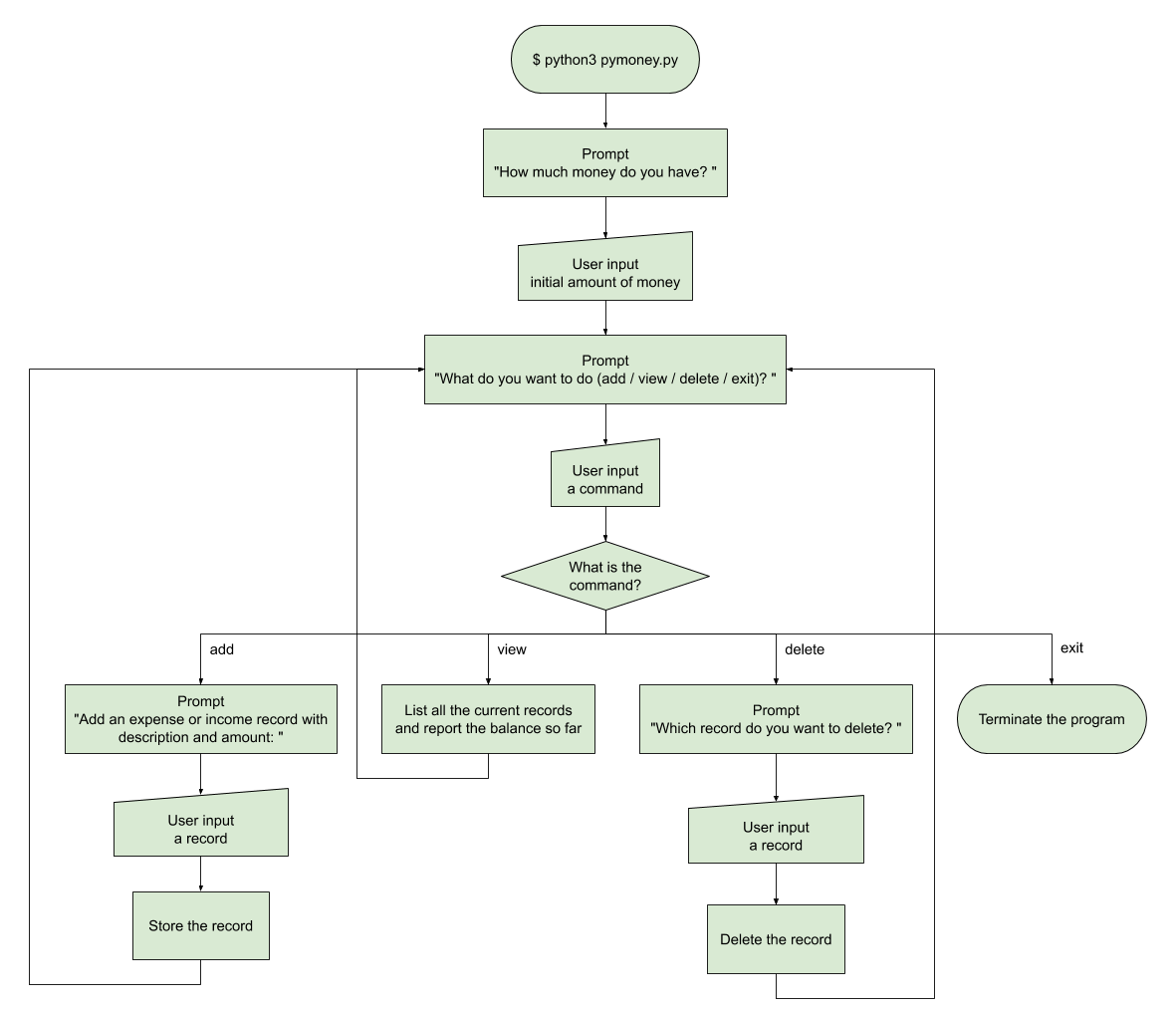
# Week 6 - Control Constructs

Currently our application only has one flow: user entering initial amount of money, user entering the records, program listing the records, and program reporting the balance.

Most of the time, the user should be able to decide what to do next. Let's provide 4 basic commands which the user can execute at any time.

* Add a record
* View the records and the balance
* Delete a record
* Exit the application

After asking for the initial amount of money, the program should keep prompting the user for one of the above commands until the user chooses to exit. The following flow chart illustrates the overall flow of the program.



$ python3 pymoney.py

How much money do you have? 1000

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -50

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

lunch -70

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

dinner -100

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records: you may design your own printing format

Description Amount

==================== ======

breakfast -50

lunch -70

dinner -100

==================== ======

Now you have 780 dollars.

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -50

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

salary 3500

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records: you may design your own printing format

Description Amount

==================== ======

breakfast -50

lunch -70

dinner -100

breakfast -50

salary 3500

==================== ======

Now you have 4230 dollars.

What do you want to do (add / view / delete / exit)? delete

Which record do you want to delete? design your own way to specify the "breakfast -50" record between "dinner -100" and "salary 3500"

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records: you may design your own printing format

Description Amount

==================== ======

breakfast -50

lunch -70

dinner -100

salary 3500

==================== ======

Now you have 4280 dollars.

What do you want to do (add / view / delete / exit)? exit

## Required Steps

1. Prompt the user for the initial amount of money.
2. Prepare a data structure (e.g. list of tuples, dictionary, class, etc.) to store the records.
3. Create a **while** loop. In the **while** loop,
   1. Prompt the user for a command.
   2. Create a **if-elif-else** statement to handle different commands.
      1. Handle the "add" command.
      2. Handle the "view" command. Try to print the records in a neat format.
      3. Handle the "delete" command. You should also remove the amount of this record when calculating the balance (e.g. 50 is added back to the balance after deleting "breakfast -50").
      4. Handle the "exit" command if necessary.
   3. Leave the **while** loop if the command is "exit".
4. Add appropriate comments for your code.

## Think and Solve

You might run into a question when designing the "delete" command: how should the user specify which record to delete?

Apparently by saying "breakfast" or "breakfast -50" is not enough because there are possibly more than one records with the same description and amount. It's not feasible to assume that the user is deleting the first, last, or all records as "breakfast -50".

As a software developer, you need to come up with a solution and implement it in your code.

## Related Knowledge

* while loop
* if-elif-else statements
* list, tuple, dictionary and their methods
* List comprehension
* enumerate() function
* str.split() and str.join() methods
* string formatting

# **Week 7 - Exceptions & Files**

You might notice that no record is left after the user exits from the application then starts it again. That's because all the variables in Python are stored in the memory (記憶體). Once a program terminates, it releases the memory space it allocated. Our application is far from useful. You can't ask the user to run the application for days, months, or even years. Even if a user is willing to run it forever, computers power off accidentally from time to time.

However, things stored in files stay there no matter how many times the computer restarts, unless the hard disk (硬碟) is broken. This leads to our solution: **write the records to a file before the program stops and read from it when the program restarts**.

$ python3 pymoney.py

How much money do you have? 1000

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -50

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

lunch -70

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

salary 3500

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records:

Description Amount

==================== ======

breakfast -50

lunch -70

salary 3500

===========================

Now you have 4380 dollars.

What do you want to do (add / view / delete / exit)? exit

$ python3 pymoney.py

Welcome back!

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records:

Description Amount

==================== ======

breakfast -50

lunch -70

salary 3500

===========================

Now you have 4380 dollars

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -50

What do you want to do (add / view / delete / exit)? delete

Which record do you want to delete? design your own way to specify "lunch -70"

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

dinner -120

What do you want to do (add / view / delete / exit)? exit

$ python3 pymoney.py

Welcome back!

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records:

Description Amount

==================== ======

breakfast -50

salary 3500

breakfast -50

dinner -120

===========================

Now you have 4280 dollars.

What do you want to do (add / view / delete / exit)? exit

Since your code is getting longer and more complicated, many things can go wrong and crash your program. Besides some mistakes you might make in your code, more errors can occur as the user inputs something out of your expectation. Now that we included file operations in our program, we also have to consider things happen outside the program: files created by the program being modified or deleted outside the program, for example.

Here we list some exceptions that may occur in our current program:

* (1) When prompted for the initial amount of money, the user inputs a string that cannot be converted to integer.
* (2) When prompted for a command (add / view / delete / exit), the user inputs a string that is not one of the four above.
* When prompted to add a record,
* (3) the user inputs a string that cannot be split into a list of two strings, or
* (4) the second string after splitting cannot be converted to integer.
* When prompted to delete a record,
* (5) the user inputs in an invalid format in respect of your design, or
* (6) the specified record does not exist.
* When loading the records from the file,
* (7) the file does not exist,
* (8) no line is in the file,
* (9) the first line cannot be interpreted as initial amount of money (i.e. cannot be converted to integer), or
* (10) any of the other lines cannot be interpreted as a record (i.e. cannot be split into a list of two strings or the second string after splitting cannot be converted to integer).

$ python3 pymoney.py

How much money do you have? abc

Invalid value for money. Set to 0 by default.

What do you want to do (add / view / delete / exit)? hello

Invalid command. Try again.

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

salary3000

The format of a record should be like this: breakfast -50.

Fail to add a record.

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

salary 3500

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -abc

Invalid value for money.

Fail to add a record.

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records:

Description Amount

==================== ======

salary 3500

===========================

Now you have 3500 dollars.

What do you want to do (add / view / delete / exit)? delete

Which record do you want to delete? invalid in respect to your design

Invalid format. Fail to delete a record.

What do you want to do (add / view / delete / exit)? delete

Which record do you want to delete? specify a record that doesn't exist

There's no record with xxxxxxxx. Fail to delete a record.

What do you want to do (add / view / delete / exit)? exit

(modify the content of records.txt to make it invalid)

$ python3 pymoney.py

Invalid format in records.txt. Deleting the contents.

How much money do you have? 1000

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with description and amount:

breakfast -50

What do you want to do (add / view / delete / exit)? view

Here's your expense and income records:

Description Amount

==================== ======

breakfast -50

===========================

Now you have 950 dollars.

What do you want to do (add / view / delete / exit)? exit

## **Required Steps**

1. Before the program terminates, write the initial amount of money and records into a file named **'records.txt'** in a **with-as** statement.
2. Use the **write()** method to write the initial amount of money into the file.
3. Convert the data structure storing the records into a list of strings and use the **writelines()** method to write the list of strings into the file.
4. Put new line characters properly so that you can read the file by **readline()** and **readlines()** later.
5. At the beginning of the program, try to open 'records.txt' by calling the **open()** function in a **try-except** statement.
6. If the file exists, use the **readline()** method to read the first line, which is the initial amount of money, into a variable. Then use the **readlines()** method to read the records and build the data structure you used before (e.g. list of tuples, dictionary, class, etc.) to store them. Remember to **close()** the file after reading.
7. If the file doesn't exist (i.e. a **FileNotFoundError** is raised), prompt the user for the initial amount of money and initialize the variables needed in the coming operations.
8. Handle 10 possible exceptions apart from the already handled **FileNotFoundError**.
9. Use **try-except** (and probably **try-except-finally**) statements.
10. Specify the error type if possible. (e.g. except **ValueError**:)
11. Use **sys.stderr.write()** to report some error messages to the user if necessary.
12. Prompt the user again or set related variables directly, depending on the situation.
13. Add appropriate comments for your code.

## **Notes**

* In the 10 exceptions we list above, "the file does not exist" is not really an unexpected condition. It inevitably happens when this application runs for the first time. Therefore, it doesn't make sense to report an error using **sys.stderr.write()**. You can simply assume that there is no record at all and prompt the user for the initial amount of money.
* In some cases, you may use the **if-else** statement instead, if it's more reasonable (and more convenient) than using the **try-except** statement.

## **Related Knowledge**

* File operations
* open() function
* readline() and readlines() method
* write() and writelines() method
* close() method
* with-as statements
* Stand error output (sys.stderr)
* Exception handling
* try-except and try-except-finally statements
* Common error types

# **Week 8 - Functions**

Over the past weeks, the program we build has grown to almost or over one hundred lines of code. Let's move the detailed works into functions and make the main flow clearer.

Here are the functions we'll extract:

* initialize: read the file 'records.txt' or prompt for initial amount of money.
* add: prompt for a record and add it into the record list (or other data structure you're using).
* view: print the records.
* delete: prompt for a record to delete and delete it from the record list.
* save: write the records to the file 'records.txt'.

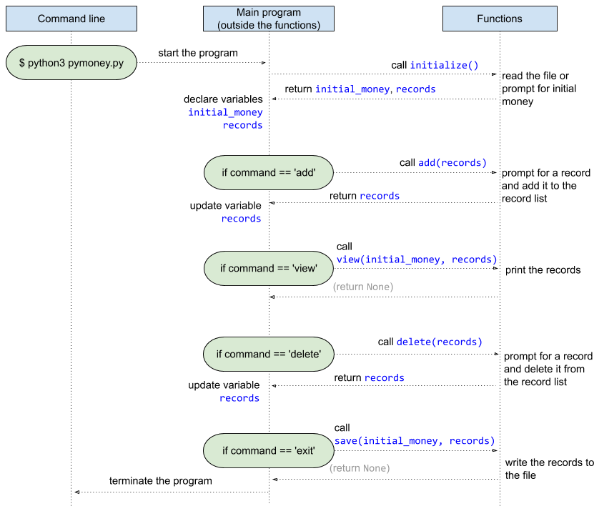
After defining the functions, the remaining part (not in functions) of your code should be like this:

| import sys    # The 5 function definitions here    initial\_money, records = initialize()  while True:  command = input('\nWhat do you want to do (add / view / delete / exit)? ')  if command == 'add':  records = add(records)  elif command == 'view':  view(initial\_money, records)  elif command == 'delete':  records = delete(records)  elif command == 'exit':  save(initial\_money, records)  break  else:  sys.stderr.write('Invalid command. Try again.\n') |
| --- |

Here we want to make the functions independent of other variables defined in the global scope. That is, none of the functions should access **initial\_money** or **records** directly. Instead, needed variables should be passed into the functions as parameters and reassigned as the returned value from the functions.

| **O** | **X** |
| --- | --- |
| def foo(L):  s = input()  L.append(s)  return L    L = ['abc', 'def']  L = foo(L) | def foo():  s = input()  L.append(s)    L = ['abc', 'def']  foo() |
| The function foo does not know there is a variable L outside. It only sees the passed-in parameter L and simply returns it after modification. | Yes, this works the same as the code at the left. But now foo is dependent on the outside variable L, which is not what we want here. |

This diagram illustrates the data flow between the main program and the functions:



## **Required Steps**

1. Identify the variables that are used and maintained throughout the main program. Generally you should find only **initial\_money** and **records** (maybe under other names). You may have other ones but make sure they are necessary. Eliminate unnecessary ones and keep those kind of variables as few as possible.
2. Define the 5 functions: **initialize**, **add**, **view**, **delete**, and **save**.
3. Define the functions with proper formal parameters and move the code originally in **if-elif** into respective functions.
4. Adjust the code inside the functions so that they're only using parameters and not accessing outside variables.
5. Return values for the caller (i.e. main program) to update related variables.
6. Rearrange the remaining code as the example in purple border above.
7. Call the functions and pass needed variables as parameters.
8. Declare or update related variables after a function returns.

## **Related Knowledge**

* Functions, parameters, and return values
* Scope of identifiers

# Week 9 - Plug-in Functions & Recursion

After restructuring our code, let's introduce a new little feature to this application -- categorizing the records.

You're going to provide some categories and subcategories, such as

* expense
  + food
    - meal
    - snack
    - drink
  + transportation
    - bus
    - railway
* income
  + salary
  + bonus

There is a new command "view categories" for the user, which list all the provided categories hierarchically. Once they know the available categories, they can specify a category when adding a record. When the user "view" all the records, the table also shows the categories for each record.

Then we provide another new command, "find", for the user. They can input a category name, and the program should find all records in the specified category or in a subcategory under that one. For example, if the user is finding "food", then all the records with category "food", "meal", "snack", and "drink" should be listed out.

$ python3 pymoney.py

How much money do you have? 1000

What do you want to do (add / view / delete / view categories / find / exit)? view categories

- expense

- food

- meal

- snack

- drink

- transportation

- bus

- railway

- income

- salary

- bonus

What do you want to do (add / view / delete / view categories / find / exit)? add

Add an expense or income record with category, description, and amount (separate by spaces):

meal breakfast -50

... (add more records with different categories)

What do you want to do (add / view / delete / exit)? add

Add an expense or income record with category, description, and amount (separate by spaces):

clothing pants -350

The specified category is not in the category list.

You can check the category list by command "view categories".

Fail to add a record.

What do you want to do (add / view / delete / view categories / find / exit)? view

Here's your expense and income records:

Category Description Amount

=============== ==================== ======

meal breakfast -50

drink coffee -100

food bread -80

food sausage -40

meal lunch -120

railway MRT -45

transportation taxi -170

bus 902 -20

income lottery 50

salary part-time 1200

snack cake -35

meal lunch -70

===========================================

Now you have 1520 dollars.

What do you want to do (add / view / delete / view categories / find / exit)? find

Which category do you want to find? food

Here's your expense and income records under category "food":

Category Description Amount

=============== ==================== ======

meal breakfast -50

drink coffee -100

food bread -80

food sausage -40

meal lunch -120

snack cake -35

meal lunch -70

===========================================

The total amount above is -495.

What do you want to do (add / view / delete / view categories / find / exit)? find

Which category do you want to find? transportation

Here's your expense and income records under category "transportation":

Category Description Amount

=============== ==================== ======

railway MRT -45

transportation taxi -170

bus 902 -20

===========================================

The total amount above is -235.

What do you want to do (add / view / delete / view categories / find / exit)? find

Which category do you want to find? income

Here's your expense and income records under category "income":

Category Description Amount

=============== ==================== ======

bonus lottery 50

salary part-time 1200

===========================================

The total amount above is 1250.

What do you want to do (add / view / delete / view categories / find / exit)? exit

## Required Steps

1. Define a function **initialize\_categories**.
   1. In the function, define a multi-level list  
      ['expense', ['food', ['meal', 'snack', 'drink'], 'transportation', ['bus', 'railway']], 'income', ['salary', 'bonus']]  
      (you may add more subcategories as you want)  
      and simply return it.
   2. Call **initialize\_categories** before the while loop and assign the returned value to a variable **categories**.
2. Define a recursive function **view\_categories**.
   1. In the function, handle the base case and recursive case.
      1. Print a category with proper indentation in the base case.
      2. Iterate through the list and call **view\_categories** in the recursive case.
   2. Add a condition **command == 'view categories'** in the **if-elif** construct and call **view\_categories** with **categories** as the parameter.
3. In the **add** function, prompt the user to specify a category for a record in addition to description and amount.
   1. Adjust your data structure for a record to store the category in addition to description and amount.
   2. Modify the code of **view** to add the categories of the records to the printed table.
   3. Modify the code of file reading and file writing to add the categories of the records to the file.
   4. Go through your code to make other necessary modifications due to this change.
4. Check if the specified category is in the predefined list **categories** when adding a record.
   1. Define a recursive function **is\_category\_valid(category, categories)** that returns **True** if **category** is in **categories** and **False** otherwise.
   2. Call **is\_category\_valid** in the **add** function to prevent a record from being added if the specified category is not valid.
   3. You might have to pass the global variable **categories** as a parameter into the **add** function so that it can be passed into **is\_category\_valid** in the **add** function.
5. Define a **find** function.
   1. In the **find** function, prompt for a category name to find.
   2. (If you find it too difficult, you could look for the hint below.)  
      Define a function **find\_subcategories** (may be recursive) that takes a category name to find and the predefined list **categories** as parameters, and returns a non-nested list containing the specified category and all the subcategories under it (if any).
      1. For example, **find\_subcategories('expense', categories)** returns  
         ['expense', 'food', 'meal', 'snack', 'drink', 'transportation', 'bus', 'railway']
      2. **find\_subcategories('transportation', categories)** returns  
         ['transportation', 'bus', 'railway']
      3. **find\_subcategories('salary', categories)** returns ['salary']
      4. **find\_subcategories('not-in-the-list', categories)** returns []
   3. In the **find** function, call **find\_subcategories** with the category name input by the user.
   4. Use the built-in **filter** function with lambda expression to filter the records whose category is in the list returned from **find\_subcategories**.
   5. Print out the filtered records and report the total amount of money of the listed records.
   6. Add a condition **command == 'find'** in the **if-elif** construct and call the **find** function with **records** and **categories** as the parameters.
6. Add at least one line of triple-quote docstring to each function you defined.
7. Make sure all of the variable and function names are in snake\_case.

## Notes

* You may skip exception handling for the new functions. However, the 4th required step (check if the specified category is in the predefined list **categories** when adding a record) is, of course, required.

## Hint for find\_subcategories

Basically there are 2 steps in this function: find the target category and flatten the subcategories under it.

1. The first step is similar to the **rec\_find** function in the lecture of week 9.

| def find\_subcategories(category, categories):  if type(categories) == list:  for i, v in enumerate(categories):  p = find\_subcategories(category, v)  if p == True:  return (i,)  if p != False:  return (i,) + p  return categories == category |
| --- |

Instead of returning a tuple of indices (the two lines in red highlight), it should return a list of the target category and its subcategories.

| def find\_subcategories(category, categories):  if type(categories) == list:  for v in categories:  p = find\_subcategories(category, v)  if p == True:  # if found, return the flatten list including itself  # and its subcategories  index = categories.index(v)  return flatten(categories[index:index + 2])  if p != False:  # p is a list returned from flatten  return p  return categories == category  def flatten(L):  # return a flat list that contains all element in the nested list L  # for example, flatten([1, 2, [3, [4], 5]]) returns [1, 2, 3, 4, 5] |
| --- |

Also some boundary conditions are (1) the target category does not have subcategories and (2) the target category is not found.

| def find\_subcategories(category, categories):  if type(categories) == list:  for v in categories:  p = find\_subcategories(category, v)  if p == True:  index = categories.index(v)  if index + 1 < len(categories) and \  type(categories[index + 1]) == list:  return flatten(categories[index:index + 2])  else:  # return only itself if no subcategories  return [v]  if p != []:  return p  return True if categories == category else []  # return [] instead of False if not found  def flatten(L):  # return a flat list that contains all element in the nested list L  # for example, flatten([1, 2, [3, [4], 5]]) returns [1, 2, 3, 4, 5] |
| --- |

1. The second step is to define the **flatten** function. Here we give you a reference answer, but you are encouraged to try it yourself.

| def flatten(L):  if type(L) == list:  result = []  for child in L:  result.extend(flatten(child))  return result  else:  return [L] |
| --- |

## Related Knowledge

* Lambda expressions
* filter() function
* Docstring
* Python style guide
* Recursion
* Recursive finding
* Indentation

# **Week 11 - 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 an expense or income record with ...:\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.
2. Instantiate a **Records** before the **while** loop and call the methods in **if-elif** instead of calling the functions you defined last time.
3. Run the code with all commands and different cases to make sure it works so far.
4. After that, you may remove the functions that are no longer being called.
5. Define the **Categories** class according to the given template.
6. Instantiate a **Categories** before the while loop and call the methods in **if-elif** instead of calling the functions you defined last time.
7. Run the code with all commands and different cases to make sure it works.
8. After that, you may remove the functions that are no longer being called.
9. Add at least one line of triple-quote docstring to each method.
10. 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

# Week 12 - Iterable, Iterator, and Generator

In week 9, we implemented the recursive function **find\_subcategories** by finding the user-input category, flattening its subcategories, and recursively returning the result. Now that you have learned about generators in Python, you might find that a generator could be more concise and straightforward for this function.

## Required Steps

1. Rewrite the **find\_subcategories** method of the **Categories** class using a generator.
   1. Define an inner function **find\_subcategories\_gen**, which is a recursive generator, in the **find\_subcategories** method.
   2. For **find\_subcategories** itself, simply return a list generated by **find\_subcategories\_gen**.
2. Remove the **\_flatten** method, which is no longer needed.

## Hint for find\_subcategories\_gen

You are encouraged to try by yourself before referring to this hint.

Here's the template:

| class Categories:  def \_\_init\_\_(self):  ...  def view(self, ...):  ...  def is\_category\_valid(self, ...):  ...    def find\_subcategories(self, category):  def find\_subcategories\_gen(category, categories):  # A generator that yields the target category and its subcategories    return \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  # A list generated by find\_subcategories\_gen(category, self.\_categories)  # The \_flatten method is no longer needed. Remove it! |
| --- |

1. First of all, you can refer to the **atom\_gen()** example in the lecture of week 12, which generates all the elements in a nested list.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  for atom in find\_subcategories\_gen(category, child):  yield atom  else: # base case  yield categories |
| --- |

For now, this generator simply yields all of the categories in the nested list, no caring which **category** is to be found (i.e. the parameter **category**).

1. Before going further, here's a tip to simplify the above code: use a **yield from** statement instead of the inner for-loop.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  yield from find\_subcategories\_gen(category, child)  else: # base case  yield categories |
| --- |

Same as the inner for-loop did, **yield from** yields the results yielded from **find\_subcategories\_gen(...)** one by one.

1. Since we don't want to yield every category, we have to set some conditions before the **yield** statement in the base case. The first condition is when **categories** is the same as the **category** we want to find.

| def find\_subcategories\_gen(category, categories):  if type(categories) == list: # recursive case  for child in categories:  yield from find\_subcategories\_gen(category, child)  else: # base case  if categories == category:  yield categories |
| --- |

This way, the generator only yields the target **category**, and all the other categories are skipped.

1. We also want to yield all the subcategories under **category**. The idea is to make a boolean flag **found** in the parameter list. The flag stays as **False** at the beginning and is set to **True** when the function recurs to the subcategories of **category**. In the base case, the category can also be yielded when the flag **found** is True.

| def find\_subcategories\_gen(category, categories, found=False):  if type(categories) == list:  for index, child in enumerate(categories):  yield from find\_subcategories\_gen(category, child, \_\_\_\_\_)  if child == category and index + 1 < len(categories) \  and type(categories[index + 1]) == list:  # When the target category is found,  # recursively call this generator on the subcategories  # with the flag set as True.  yield from \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  else:  if categories == category or \_\_\_\_\_:  yield categories |
| --- |

Fill the blanks yourself!

## Related Knowledge

* Recursive generator