Introduction to Programming

Lab 4: Dictionaries

Fatmanur Gençer, Christian Callau

2018-19 First Trimester

0. Introduction

The objective of this lab is to implement an agenda using lists and dictionaries. This agenda, which must be implemented using a dictionary, will have a key (the user's name), and will containing the following information for each user:

N phone numbers (where N can be any integer >= 0)

N email accounts (where N can be any integer >= 0)

1 postal address, address will be a dictionary, with 4 keys ("street", "number", "zip code" and "city")

1. Design and Data Structures

We followed the data structure model given by the practice instructions. We made an object called agenda where we will have all the data. This object will be a dictionary, where the key is the name of the contact and the value the rest of the information about that contact.

The rest of the information besides the contact name will be a list of phone numbers, a list of email addresses and an address object. We will store these three variables into a list and that will be the value of each entry in the agenda dictionary. The address will be another dictionary with the fields street, number, zipcode and city.

This is an example of a valid entry of the agenda dictionary:

```
agenda = {
  'Christian Callau': [
    [
      '612345678',
      '631415926'
    ],
    [
      'christian.callau01@estudiants.upf.edu',
      'christiancallau@gmail.com'
    ],
    {
      'street': 'Fake Street',
      'number': 123,
      'zipcode': '45678',
      'city': 'Springfield'
    }
  ]
}
```

For testing purpose while developing the practice we generated some random entries and put them on the code, so we don't have to put them manually. For generating the entries randomly, we wrote the following code:

```
# Data for generating the contacts
names = (
  'Alejandro Fernandino', 'Francisco Rubio', 'Elsa Chicote', 'Juan David Casas',
  'Mariangel Collazo', 'Santino Cotilla', 'Isabela Alvarado', 'Santiago Aguayo',
  'Eduardo Cadaval', 'Manuela Saavedra'
)
strets = (
  'Forest Road', 'Avondale Road', 'Adelaide Street', 'Green Street',
  'Alexandra Road', 'Manse Road', 'Vicarage Lane', 'Dale Street', 'Cecil Road',
  'Water Street'
)
emails = ('gmail.com', 'outlook.com', 'hotmail.es', 'yahoo.es', 'upf.edu')
cities = (
  ('Barcelona', ('08006', '08005')),
  ('Tarragona', ('04007', '04007')),
  ('Madrid', ('28001', '28002'))
)
agenda = {'Christian Callau': [['612345678', '631415926'], ['christian.callau01@estudiants.upf.edu', 'chris@gmail.com'], {'street': 'Fake Street', 'number': 123, 'zip code': '08006', 'city': 'Springfield'}]}

# For every name in the name list
for name in names:
  phones = []
  # Generate between 0 and 4 phone numbers
  for _ in range(randint(0, 4)):
    # Generate a random phone number
    phones.append(str(randint(600000000, 999999999)))
  emils = []
  # Generate between 0 and 4 email accounts
  for _ in range(randint(0, 4)):
    # Email name is the name without spaces and in non capital letters
    aux = name.replace(' ', '').lower()
    # 50/50 chance to generate a numerical suffix
    num = ''
    if randint(0, 1):
      num = '{:02d}'.format(randint(1, 99))
    # Choose a random domain
    dom = emails[randint(0, 4)]
    # Add the email to the emails list
    emils.append('{}{}@{}'.format(aux, num, dom))
  # Choose a random city and a random zipcode for that city
  city_index = randint(0, 2)
  zip_index = randint(0, 1)
  # Generate the whole address
  address = {
    'street': strets[randint(0, 9)],
    'number': str(randint(1, 999)),
    'zipcode': cities[city_index][1][zip_index],
    'city': cities[city_index][0]
  }
  # Add the contact to the agenda
  agenda[name] = [phones, emils, address]

# Print the result
print(agenda)
```

With that code we generated 10 additional entries that even though are random had some coherence.

For the main function we put the menu options into a tuple like in previous practices. The last option corresponds to exit the menu and the option before that corresponds to print the menu since we are not printing the menu each time an option is executed. We also store the functions in a tuple in the same order as the options so when the user enters an option we just had to extract the function from the tuple using the option as the index (minus 1, because the options start from 1 and the list index start from 0) and execute the function giving it the correct parameters.

The most useful functions of the practice are the ones that return a field parameter like the email, the phone number, the zip code, etc. These functions use regular expressions to verify that the user input matches them and then returns the value entered by the user. These functions are used through all the code. For example, the “get_valid_phone_number” function and their helper function:

```
1.  # Returns a valid string from input that matches the given regular expression
2.  def get_input_matches_regex(msg, regex):
3.    pattern = re.compile(regex)
4.    user_input = input('Enter the {}: '.format(msg))
5.    while not pattern.match(user_input):
6.      user_input = input('Invalid {}. Try again: '.format(msg))
7.    return user_input
8.
9.  # The telephone number (which should be a string), contains exactly 9 digits
10. def get_valid_phone_number():
11.   return get_input_matches_regex('phone number', r'^[1-9]\d{8}$')
```

For instance, the helper function “get_phone_list” when introducing a new contact asks for how many phones are to be introduced and then calls the function “get_valid_phone_number” which returns valid formatted phone numbers from the user input, if the user doesn’t type a valid phone number the error is handled and is asked to try again.

Another example is the helper function “get_valid_address” which only must create the address object by calling the helper functions that returns each of the fields with the proper format:

```
1.  # Return a valid address
2.  def get_valid_address():
3.    return {
4.      'street': get_valid_street(),
5.      'number': get_valid_street_number(),
6.      'zipcode': get_valid_zipcode(),
7.      'city': get_valid_city_name()
8.    }
```

For printing the contact information, we use the str.join() method that given a list of strings joins them using the “str” string as a separator. With the address we have to take the values of the dictionary not the keys, so for that we have to call the method dic.values(), and because one of the fields, the street number, is an integer we create a new list to pass to the str.join() method by casting every element of the address.values() list to a string using the method str(). Finally, we create a final string with the string of the phones, emails and address adding the contact name and some line breaks and we print it:

```
1.  # Prints in a visual format the given contact information
2.  def print_contact(name, info):
3.    phones  = ', '.join(info[0])
4.    emails  = ', '.join(info[1])
5.    address = ', '.join([str(val) for val in info[2].values()])
6.    print('Contact: {}\n  Phones: {}\n  Emails: {}\n  Address: {}\n'
7.      .format(name, phones, emails, address))
```

For modifying a contact, we made a submenu that ask if the user wants to modify the phone list, the email list or the address. In the first and second case we call a sub function that has two more options, adding a new phone or email, or removing an existing one, for this last option we will print the whole list of existing phones or emails and ask witch one to delete, the option will be the index of the item to remove, we will use the function list.pop() to remove it and save it to a variable so we can print an informative line saying that the value has been removed successfully.

For modifying the address, the submenu shows the available address fields to modify, selecting one of them will just call one of the helper functions we talked about earlier and assign the new value to the corresponding key field. In this modify functions we are using two tricks worth mention:

The first one is in “modify_phone_list” and “modify_email_list”, we are calling the function “print_menu” that we were using to print the items of the menu options tuple but passing the list of phones / emails, resulting to printing the values of the list as options. Then the option given by the user (minus 1) will correspond to the index of the specific item.

The second one is in “modify_address”, we store the fields of the address dictionary in a tuple, when the user inputs an option that option (minus 1) corresponds to the field in the tuple, so we extract it and use it as the key of the address field:

```
1.  # Address fields
2.  fields = ('street', 'number', 'zipcode', 'city')
3.  ...
4.    option = get_option(1, total_options)
5.  ...
6.      value = menu_functions[option-1]()
7.      agenda[name][2][fields[option-1]] = value
```

Moving one, to the easier functions, for printing the whole agenda we just had to call the previous function that prints a single contact but for all the contacts of the agenda.

For getting the contacts of a specific city, we ask the user the wanted city, then for all the contacts in the agenda we compare the city field, if its equal we print the contact and move on to the next.

For getting the contacts of a specific zipcode is the same as the previous function.

For getting the contacts with the name starting with a specific character we do the same as the previous function but in this case, we are comparing only the first letter of the name, which is the key of the entry in the dictionary.

For getting the contacts with a substring present in the name we do the same as the previous function but in this case, we use the keyword “in” and we convert both the substring and the name to lower case letters.

For getting the contacts that contain a substring in any of the phone numbers we do the same as the previous function but in this case, we must add an additional nested for loop to iterate over each phone of each contact in the agenda.

For getting the contacts that have a certain domain in any of the email addresses we do the same as the previous function, to get the domain of an email we split it by the ‘.’ (dot) character and get the last element.

For getting the contacts that contain exactly N phone numbers or email addresses we ask the user to enter the N and then compare it to the length of the list of either phones or emails.