Theory: Functional decomposition

27 minutes 5 / 7 problems solved

Start practicing

At this point, you already know how to declare functions in Python. This is a very useful skill, no doubt about that, but to make the most of it, we need to know when to declare them. In this topic, we'll see how to decompose the solution of a particular problem into functions.

§1. Real-life example

Before we go to the actual decomposing, let's figure out what it is that we want to decompose.

Suppose, we are writing a program that simulates an ATM. How do real-life ATMs work? Well, usually a client inserts the card, enters the pin, and, if the pin is correct, performs some operations, for example, withdraws money or deposits money to an account. We can reimagine these actions as parts of a computer program. This is how the algorithm can be described in general:

1. Parse the input data (card and entered pin);
2. Check that the pin is correct;
3. Ask the client what they want to do;
4. If the operation is supported, perform it.

Before we program this algorithm, let's settle a few things. Obviously, a real bank has a database that stores all necessary data, like the encrypted correct pin or the current card balance. Here we are creating a very simple version of an ATM, so we're not going to include database checkups. Instead, we will define variables `card_pin` and `card_balance`. These variables will represent the correct pin and card balance that we would've gotten from a database.

We also need to determine which operations we'll allow. Let's settle on three: displaying the card balance, adding money to the account and withdrawing money from the account.

Now let's see the code:

Current topic:

✓ Functional decomposition Stage 1 7★ ···

Topic depends on:

✓ Components of computational thinking Stage 1 7★ ···

✓ Declaring a function Stage 1 10★ ···

✓ Elif statement Stage 1 15★ ···

Topic is required for:

✓ Algorithms in Python ···

Table of contents:

↑ Functional decomposition

§1. Real-life example

§2. Functional decomposition

§3. The result

§4. Summary

Feedback & Comments

```
# operations
DEPOSIT = "DEPOSIT"
WITHDRAW = "WITHDRAW"
DISPLAY = "DISPLAY"

# read the data
card_number = input("Enter card number: ")
input_pin = input("Enter PIN: ")


# card_pin and card_balance are read from the database



# check that the pin is correct

if card_pin == input_pin:

    # ask the client what they want to do

    action = input("Enter desired action: ")

    if action == DEPOSIT:

        money = float(input("Enter the sum of money to DEPOSIT: "))

        card_balance += money

        print("Deposited: $", money)

        print("Current balance:", card_balance)

    elif action == WITHDRAW:

        money = float(input("Enter the sum of money to WITHDRAW: "))

        card_balance -= money

        print("Withdrawn: $", money)

        print("Current balance:", card_balance)

    elif action == DISPLAY:

        print("Current balance:", card_balance)

    else:

        ...

else:

    print("Incorrect pin!")
```

As you can see, a lot is going on and it's a bit hard to follow. The main logic is the same we've described above. This code works perfectly fine for our problem and we could leave it like that.

However, what if we want this script to work for many users and not just one? What if we want to process other cases and perform other actions, for instance, check if the card is in the database or change the pin? Some parts of this code will be useful, other parts we'll have to comment or delete. We would also need to track all places where we're introducing changes to make sure that everything runs smoothly. Now it starts to sound like we may have a problem with our code. The solution, as you may have guessed, is decomposition.

§2. Functional decomposition

Functional decomposition is simply a process of decomposing the problem into several functions. Each function does a particular task and we can perform these functions in a row to get the results we need.

When we look at a problem, we need to think about which actions we may want to repeat multiple times or, alternatively, perform separately. This is how we can get the desired functions. Let's look at our ATM simulation again and figure out which steps can be turned into separate functions.

First, an action that we do frequently is reading the input with a particular message displayed. Second, we perform a certain sequence of actions when the pin is correct, specifically we ask what we should do next. Third, depending on the answer from the client, we either perform certain actions to deposit the sum to the account or withdraw them from the account. And lastly, whatever the action, we always print out the current balance.

Some of these actions can be converted to separate functions to make the program simpler.

Let's go over them step by step. First, let's separate our main operations into functions.

```
def deposit_money(amount, card_balance):
    """Deposit given amount of money to the account."""
    card_balance += amount
    # save new balance to the database
    return card_balance


def withdraw_money(amount, card_balance):
    """Withdraw given amount of money from the account."""

    card_balance -= amount

    # save new balance to the database

    return card_balance
```

You may have noticed that in the original program we print the current balance regardless of what we've done before. This means that we can also create a separate function that would log everything.

```
def log_transaction(action, money, card_balance):
    if action in (DEPOSIT, WITHDRAW):
        print(action + ": $", money)
    print("Current balance:", card_balance)
```

This function is going to be called after we've done something and it will display information about the current balance and the changes that have been made.

Next, it makes sense to create a function that would manage these operations:

```
def move_money(action, money, card_balance):
    if action == DEPOSIT:
        return deposit_money(money, card_balance)
    elif action == WITHDRAW:
        return withdraw_money(money, card_balance)
    elif action == DISPLAY:
        return card_balance
```

You can see that this function returns the card balance that we get after our manipulations. This is helpful because, as we've seen before, we always want to know how much money we end up with. The main purpose of this function, however, is to simplify the process of revising the functionality of our program. If we want to add some other action, we just add another option to the `if - else` statement and specify the function that would carry out this task. Removing is similar.

One important part that we haven't covered yet is getting the information about the money we'll be moving somewhere. We know that we don't need this information for display, but it is necessary for other operations.

```
def get_amount_of_money(action):
    if action == DISPLAY:
        return 0.0
    money = input("Enter the sum of money to " + action + ": ")
    return float(money)
```

At this moment, we only have bits and pieces of our final program. Another important step is creating a function that would put it all together.

```
def make_transaction(action, card_balance):
    money = get_amount_of_money(action)
    card_balance = move_money(action, money, card_balance)
    log_transaction(action, money, card_balance)
```

This is when the main logic takes place. We have a single entry point that determines the order of operations and calls necessary functions.

§3. The result

Now, let's rewrite the program above using these functions:

```
card_number = input("Enter card number: ")
input_pin = input("Enter PIN: ")

# card_pin and card_balance are read from the database

if card_pin == input_pin:
    action = input("Enter desired action: ")
    make_transaction(action, card_balance)
else:

    print("Incorrect pin!")
```

That's it! Sure, together with the functions, the code is much bigger, but this provides us with more advantages than disadvantages. We can understand the general direction of the program and can easily introduce changes if needed. Now, for example, if we want to add another action, we just need to define its function and modify the `move_money` function. We can also easily test separate components since they are determined in separate functions. All in all, our program now is a real functioning program that won't fall apart when we decide to change it a bit.

§4. Summary

In this topic, we've covered the concept of functional decomposition, dividing the process into several functions.

Among other things, decomposing allows us to:

- structure code better;
- see the general logic of the program;
- introduce changes easily;
- test separate functions.

Obviously, functional decomposition is not a universal solution. However, if you can think of your problem in terms of a sequence of some functions, it can be of great help to you!

Report a typo

523 users liked this theory. **10** didn't like it. **What about you?**

Start practicing

Comments (14) Hints (0) Useful links (0) Show discussion