目次へ

残業時間編・オマケ2

ここまで練習問題をこなしてみましたが、せっかく最初のテキストで使い方を覚えた「関数」を使うチャンスがないなあと思ってました。

なので、さっきまでの「残業時間編・ファイナル」の課題について、関数というものを導入しながらちょっといい感じに解決していけることも示しておこうと思います。

今回は、「帰宅時間」から「残業時間」を計算するところを関数に仕立てるとどんな具合になるかな、というところから入りましょう。

思い出して欲しいですが、pythonでいう関数ってのは、なんかが「引数」として入力されたときに、これについて「返り血」、もとい、「返り値」が帰ってくるという仕掛けです。

ほしい関数を、自分に発注する

どんな「引数」を与えて、どんな「返り値」を戻してくれる関数があったら幸せだろう…と想像してみるところから、関数の実装計画が決まります。もっとおおげさに、例えば、なんでも屋さんのカウンターに向かって、「こんな関数欲しいんだけどナー」と注文している自分たちを想像してみましょう。どんな関数があったらさっきの「残業時間」の課題が楽になったでしょうか。

こんなのはどうでしょう。

```
引数として、帰宅時間の「時」の数字と「分」の数字を渡すから、それを受け取って残業時間を「分」で返すような関数をちょうだ
```

まあ、いいでしょう。残業時間を計算するための定時ってのは17:20だということは…たぶん、この関数の中に直接書き込んであることになるでしょうか。

または、残業時間も引数として入れるようにすると、より使い道が広がるんじゃないかなとも想像されますね。なので下のような注文に変更しましょうか。

```
引数として、帰宅時間の「時」と「分」を、また定時の「時」と「分」をそれぞれ数字として渡すから、それを受け取って残業時間
返すような関数をちょうだいな！
```

こんな関数なら、例え定時が17:10になったときも、関数自体を変更しないで使いまわせそうですね。

で、こんな関数があったとして、これを使うためにどんなスクリプトを書くことになるかを想像してみましょう。関数の名前は、仮に cal_zangyo_minutes とでもして、

```
>>> kitaku = "23:45"
>>> （ここで、いろいろと処理して「時」と「分」をそれぞれ導き出す…）
>>> k_hour
23
>>> k_minute
45
>>> cal_zangyo_minutes(k_hour, k_minute, 17, 20)　←この関数は、（帰宅時、帰宅分、定時時、定時分）という順番に引数をと
385
※まだこんな関数は作っていませんから、あくまで実行予想みたいなものですよ
```

"23:45"という文字列を関数に放り込むまで、結構手順を踏む必要があるみたいですね。大したことないといえば大したことないんですけど。

もっと楽できるとうれしいなあ。いっそ、もう一歩踏み込んで、こんな関数をつくってもらうことにしましょうか？

```
引数として、"17:20"みたいな文字列を（定時、帰宅時）の順で渡すから、適当に処理して、残業時間を「分」でよこすような関数
ちょうだいな！
```

こんな関数があったら、使うほうはいいでしょうねえ。きっと下のような感じで済みます。

```
>>> kitaku ="23:45"
>>> teiji = "17:20"
>>> cal_zangyo_minutes(teiji, kitaku)
385
```

おお、こいつはシンプルな感じでいいじゃないか。よーしオヤジ、この関数をさっそく一つこしらえてくれよ！

…

お気づきでしょうが、この関数を作るのは自分たちです。こんな小規模な開発なら、関数を使いたい人と関数を作らされる人は一人二役です。実際、プログラムを作る人で、どんな関数が必要かを考えるようなときに、上みたいな一人芝居を密かに演じている人は多いんですよ。（根拠はありません。でもきっとそうだよね。）

じゃあ関数を作り始めてみましょうかね。いきなりですが、下のスクリプトが、関数部分だけを書いたときのスクリプトです。ダウンロードして実行を試してみてもいいですよ。

3_sample_func.py

```
def cal_zangyo_minutes(teiji, kitaku):
    t = teiji.split(":")
    t_hour = int(t[0])
    t_min = int(t[1])
    k = kitaku.split(":")
    k_hour = int(k[0])
    k_min = int(k[1])

    z_min = (k_hour * 60 + k_min) - (t_hour * 60 + t_min)
    if z_min < 0:
      z_min += 24 * 60

    return z_min

# test
print cal_zangyo_minutes("17:20", "23:45")
```

最後の一行は、作った関数が期待どおりに動作するかを試すために書き足しました。（行頭が # ではじまる行は、ただのコメントとして何か書くときに使います。ここはpythonには無視されて、人間が読むためだけに使われます。）

これの実行結果を見ると、ちゃんと385と表示されました。一応もくろみどおりのようですね。"23:45"を「午前サマ」な時間にしたときもそれなりの値が出てくるか、試しておいてください。

この関数自体はあまりうまい書き方でもないのですが、とりあえずこれでよしとしておいてください。

ってことで、関数いっちょう、お待ち！

さあ、この関数を自由に使えるんだぞ、という前提で、「残業時間・ファイナル」の回答をもういちど書いてみます。指南編で引用させてもらったうちの一つを、ちょっと書き換えました。

3_sample_func.py

```
def cal_zangyo_minutes(teiji, kitaku):
    t = teiji.split(":")
    t_hour = int(t[0])
    t_min = int(t[1])
    k = kitaku.split(":")
    k_hour = int(k[0])
    k_min = int(k[1])

    z_min = (k_hour * 60 + k_min) - (t_hour * 60 + t_min)
    if z_min < 0:
      z_min += 24 * 60

    return z_min


ruikeibo = {}

for my_line in open ("leavetime3.txt"):
    my_triplet = my_line[:-1].split(",")
    year_month = my_triplet[0][:6]
    my_key = year_month + "," + my_triplet[1]
    taishaJikoku = my_triplet[2]
    zangyo  = cal_zangyo_minutes("17:20", taishaJikoku)        #<- ここらへんがシンプル化
    ruikeibo.setdefault(my_key, 0)
    ruikeibo[ my_key ] += zangyo

（つづきは省略）
```

回答が短くなった…とは言いがたいですが、cal_zangyo_minutes関数を一生懸命つくっておいたおかげで、ファイルを読み込みながら処理する部分はちょっとシンプルに直すことができました。

さらに、読みやすくなったとも感じませんか。関数の名前を cal_zangyo_minutes としましたが、ここを見て、「ああ、残業時間を、きっと分単位で出すんだね」と理解させることができるため、あとでプログラムを直したいときに容易になるわけです。関数の名前をつけるときは、このように、あとで理解をしやすくする名前にするよう強く努力すべきです。（あ、cal ってのは、calicurate（算出する）のつもりでつけました。あんまり分かりやすくもなかったかも。でも長すぎる関数名もアレですしねえ…）

関数を使いながら課題を解く、というのはこんな感じのものです。

ここでは、作った関数の使いどころが一箇所だけだったので別にスクリプトを短くすることもできませんでしたが、うまい関数を自分に「発注」してこしらえ、これが何ヶ所にも使えるようにうまくハマっていると、スクリプトは総じて短くなり、さらに関数を使っている部分もシンプルに読めるようになり、まことにいいものです。これぞプログラミングの醍醐味です。いやそれは言いすぎた。