目次へ

練習問題・残業時間2

また残業時間か。どんだけ残業好きなんだ。

いや、別に残業が好きなわけでもないんですが、例題を考えるのにいろいろと便利なので、まだこの話で引っ張ろうと思います。よろしく。

前回は、ファイルに一行ずつ記された退社時間を使って残業時間を算出しました。

今回も同じなのですが、下の様なデータを扱ってみようと思います。

```
Yamamoto 18:20
Ogawa 20:00
Yamamoto 22:15
Ogawa 18:00
Ogawa 23:30
Mizoguchi  0:50
Yamamoto 22:45
Ogawa 2:15
```

前回からの違いは、行ごとに従業員名とおぼしきものが含まれている点、また、午前０時を過ぎる残業が生じている点です。かわいそうに、なんという激務でしょう。

YamamotoさんやOgawaさんといった部分が、それぞれ従業員名です。半角スペース（空白）記号をひとつ挟んで、次に退社時刻が続いています。

前回のテクニックが通用しないところが少なくともひとつあります。「何文字目から、何文字目」という文字列の切り出し方ができませんね。名前の長さによっても時間部分を切り出す場所は変わってきますし、さらに0時とか2時とかのときに、「時間」の部分が1文字になっています。これは困ったぞ。

ってことで、今回は、splitという文字列用メソッドを新しく覚えるための練習でもあるのです。こいつを覚えれば、多少デコボコした文字列の扱いも怖るるに足らず。

split

上のようなデータが収まったファイルを一行ずつ読みながら処理することを考えます。

最初の一行は、下のようなものになるでしょう。

```
Yamamoto 18:20¥n
```

便宜上、「\n」という印を入れました。なんだか覚えていますか。改行記号です。一行読むというのは、最寄の改行記号までを読み込むということですので、扱う文字列は正確にはこういうことになりますよ。 \nはこれだけで一文字分です。（いわゆるエスケープ表記というものを使って、本来は見えないものを表現しています）

これをいったん適当な変数に入れて、いろいろな処理を試すことにします。対話シェルで。

```
>>> s = "Yamamoto 18:20¥n"
```

こいつを、空白記号で切り分けて、"Yamamoto" と "18:20" に分けたい。今までにやったのは、findを使って「空白文字はどこにあるか」を調べておいて、その位置をもとに[:]の指定を使って部分文字列を取り出す、というものでした。実は、こんなまどろっこしいことをすることは滅多にありません。splitという強力な機能があるからです。文字通り、文字列を「分割する」ためのものです。

下のような感じで書きます。試してみましょう。

```
>>> s.split(" ")
['Yamamoto', '18:20¥n']
```

この表記を見て、何が起こったのかお分かりでしょうか。文字列をsplitしてみたところ、ふたつの文字列でできた「リスト」が帰されたのです。splitの引数には、区切り文字とみなすものを与えます。ここでは空白記号ですね。

splitの引数は、空白記号でなくてもよいです。ためしに、変な例を。

```
>>> s.split("a")
['Y', 'm', 'moto 18:20¥n']
```

「a」という文字を区切りとみなしてsplitしてみたら、この行に「a」はふたつあったようで、三つの文字列でできたリストが帰ってきました。このように、区切り記号がたくさん見つかれば、そのぶんたくさんの要素からなるリストをつくることができるわけです。

ところで「\n」は扱う上でジャマなので、こいつは明示的に取り除きながら処理することを考えましょう。「最後の一文字」を取り除くという方法を使ってこれを処理できます。

```
>>> s
'Yamamoto 18:20¥n'
>>> s[:-1]
'Yamamoto 18:20'
```

だから、この処理済の文字列についてsplitを改めて行うということで、

```
>>> s[:-1].split(" ")
['Yamamoto', '18:20']
```

とすれば、処理に余計なものをはぶけます。

さて、帰ってくるのがリストだから、この中身をそれぞれ使うためには、いったんこれも変数に入れてから扱うのがいいでしょう。

```
>>> a = s[:-1].split(" ")
>>> a[0]
'Yamamoto'
>>> a[1]
'18:20'
```

aには、二要素から成るリストが入ったはずです。で、そのうち最初の値を使いたいときは a[0]、二番目の要素を使いたいときは a[1] という指定で値を使う。ここはよいですか。

さらに、'18:20'という文字列から'18'と'20'を切り分けるにも、今回はsplitを使いましょう。前回みたいに、最初の二文字を云々…とか言ってると、'0:50'とかいう文字列が現れたときに対処できません。

```
>>> a[1].split(":")     ←リストの中身を、さらにsplitしている。ここでの区切り文字はコロン記号
['18', '20']
```

これでsplitの使い方は示しました。文字列（または文字列の入った変数等）に、「.split」と続けて書くことで、文字列を簡単に分割することができるってことです。

～ごとに、とあったら辞書

これで、まずはデータの一行ごとに残業時間を算出することまではできるんじゃないでしょうか。

今回は、さらにそれを「人ごとに」合計しなくちゃいけません。Yamamotoさんは何時間何分、Ogawaさんは何時間何分…という風に。こうあったら、「ああ、辞書使おう」と思いつくべきです。辞書がなんだか覚えていますでしょうか。何かを「キー」にとって、それに対応する値をそれぞれ保持しているモノです。

たとえば下のような辞書を考えましょう。

```
>>> gd = {'Yamamoto':0, 'Ogawa':0}
>>> gd['Yamamoto']
0
>>> gd['Yamamoto'] = gd['Yamamoto'] + 20
>>> gd['Yamamoto']
20
```

gdという名前の辞書を作ってみました。最初は誰も0から始めて、順々に足し算されていくという想定です。足し算を行うときの書き方は長ったらしくなりますが、今までに見たような「a = a + 1」と同じことをやっているだけです。（これをもうちょっとだけ楽に書く方法がありますが、今のところはまだいいかな）

このテクニックをつかえば、人ごとに合計を管理することができますね。

でもちょっとワナがありますので、もう一つ説明しておきます。ここではgdという変数に入った辞書を使っていてYamamotoさんとOgawaさんを扱っていますが、あるときSegawaさんのデータが入ってきたらどうなるでしょう。

```
>>> gd['Segawa']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Segawa'
>>> gd['Segawa'] + gd['Segawa'] + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Segawa'
```

怒られちゃいます。'Segawa'なんてキーは登録されていないから、処理はできないよ、というのです。

融通利かせて、まだ入れていないキーがあったら自動的に　0 なんかが入ってくれればいいのにね。そうしないと、あらかじめ出現するかもしれない名前を最初からみんな含んだ辞書をつくっておかなくちゃいけないわけで、そんなことはやりたくもありません。

で、そんな書き方は可能です。下のような書き方をひとつ覚えましょう。

```
>>> gd.setdefault('Segawa', 0)
```

これは、gdに入っている辞書について、今から'Segawa'というキーで問い合わせをするつもりなんだけど、もし**そういうキーがない場合には**0を入れておいてくれ、という意味です。キーがすでにある場合には何もしません。なので、すでに対応するキーに10とか20とかいう値が入っていればなにも行われません。「キーがなければ初期値はコレ」という指示をするために、なかなか都合のいい書き方です。

これを使うなら、最初に合計管理のために辞書を用意するときにも、すっかり空っぽの辞書だけ準備しておいてもよいですね。

```
>>> gd = {}
```

これが、どんなキーもあらかじめ持っていない辞書をつくるときの書き方です。setdefaultで必要なたびにキーを作ることができるようになったなら、初期値はこれで十分です。

「午前サマ」の処理

こいつは、ifを使って簡単に処理してしまえるでしょう。

残業開始が17:20、退社が零時をまわった1:10だったとします。

今までの方法で残業時間を計算してみるなら、下のようになりますね。退社時間から残業開始時間を引き算するわけなので。

```
>>> (1*60 + 10) - (17*60 + 20)
-970
```

マイナス値になってしまいます。が、ちょっと考えてみれば、これの補正方法はわかるでしょう。「1時帰宅」と言われたとして、これを文字通り計算したからこんなことになるのであって、あらかじめ24時間を足して「25時帰宅」と読み替えておけばよかったのです。だから、マイナスになった値に、24時間（1440分）を足すことにすれば期待通りの値になります。つまり、スクリプトの一部はこんな風になると予想できます。

```
zangyo = (.....)   ←なんらかの方法で残業時間を「分」で算出した
if zangyo < 0:
  zangyo = 1440 + zangyo   ←負の値になってたら、補正する
..つづきの処理..
```

こんなところで、今回の練習問題をやるために必要な要素はある程度説明できたかな。あとはこれらをどう組み立てて一本のスクリプトとして表現するかですね。

ファイルを全部処理し終わって、最終的な合計情報が見たいときは、たとえば g という変数に辞書を入れていた場合は、今回は単に print g として表示させればよいでしょう。多少見づらい表示になるでしょうがが、注意して読めば内容はわかるでしょう。

練習問題02

【練習問題：02】今から示す退社時間ファイル・バージョン2(leavetime2.txt)をすべて読み込んで、合計で一番長時間の残業した者を答えよ。また、その残業時間はどれだけか。何分、という答え方でも構わない。また、python自身に最長残業者を判断させる必要はなく、全体の結果をもとに人間が判断すればよい。
後の模範解答では、スクリプト名を zangyo_sum2.py とする予定である。

　leavetime2.txt ←右クリックから「リンク先を保存」「対象を保存」などを選んで保存してください。

※この残業時間表はフィクションです。