# Ficha 5 - Introdução às tabelas resolvido

Jorge Gustavo Rocha

May 30, 2020

## 1 Tabelas

Neste módulo vamos introduzir a noção de tabela do `pandas`. Estas tabelas são muito usadas.

### 1.1 Tabelas

Uma tabela, do tipo `pandas.DataFrame` é uma estrutura de dados multidimensional. Este tipo é oferecido pelo módulo `pandas`, pelo que temos que começar por importar o módulo `pandas` no início do notebook.

Uma nova tabela é criada com `pandas.DataFrame()`. Vamos ilustrar a criação de uma tabela com os dados da Taxa bruta de Natalidade e da Taxa Bruta de Mortalidade, dos anos mais recentes. Estas taxas dizem-nos quantos bebés nasceram ou quantos óbitos foram registados por 1000 habitantes.

As tabelas estão organizadas por linhas e colunas. Vamos organizar a informação em três colunas:

1. A primeira coluna (`'Ano'`) refere-se ao ano
2. A segunda coluna tem a correspondente taxa bruta de natalidade (`'Natalidade'`)
3. A terceira coluna tem a correspondente taxa bruta de mortalidade (`'Mortalidade'`).

[44]:
```
import pandas

população = pandas.DataFrame({
    'Ano': [ 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018 ],
    'Natalidade': [ 9.2, 8.5, 7.9, 7.9, 8.3, 8.4, 8.4, 8.5 ],
    'Mortalidade': [ 9.7, 10.2, 10.2, 10.1, 10.5, 10.7, 10.7, 11.0 ]
})
```

Vamos usar esta tabela nos exemplos seguintes.

Antes de continuar, uma pergunta retórica: qual é o tipo de objecto que estamos a passar ao método `pandas.DataFrame()`? É um objecto do tipo `dict`, que é um dicionário em Python. Esta estrutura de dados já foi abordada em Estruturas de dados.ipynb. Ou seja, este método do `pandas` pega num `dict` e cria uma tabela (que é do tipo `pandas.core.frame.DataFrame`).

#### 1.1.1 Consultar a tabela

Basta escrever `população` e é-nos apresentado o conteúdo da tabela. Esta visualização funciona bem, porque a tabela é muito pequena.

[45]:
```
população
```

[45]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 2 | 2013 | 7.9 | 10.2 |
| 3 | 2014 | 7.9 | 10.1 |
| 4 | 2015 | 8.3 | 10.5 |
| 5 | 2016 | 8.4 | 10.7 |
| 6 | 2017 | 8.4 | 10.7 |
| 7 | 2018 | 8.5 | 11.0 |

Geralmente as tabelas são grande. Para visualizar ou processar apenas uma parte de uma tabela, temos muitas possibilidades para extrair apenas uma parte da tabela.

Pode-se fazer `população.head(3)` para ver as primeiras 3 linhas. Se não for indicado um valor, `população.head()` apresenta as primeiras 5 linhas.

[46]:
```
população.head()
```

[46]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 2 | 2013 | 7.9 | 10.2 |
| 3 | 2014 | 7.9 | 10.1 |
| 4 | 2015 | 8.3 | 10.5 |

### 1.1.2 Ver as últimas linhas

`população.tail()` apresenta as últimas linhas da tabela. Pode ter como argumento o número de linhas que se pretende visualizar. `população.tail(1)` mostra apenas a última linha.

[47]:
```
população.tail(1)
```

[47]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 7 | 2018 | 8.5 | 11.0 |

### 1.1.3 Por colunas

Pode-se mostrar apenas uma ou mais colunas, com as seguintes formas:

1. `população.Natalidade`, só a coluna `Natalidade`
2. `população['Natalidade']`, igual ao anterior
3. `população[['Natalidade', 'Ano']]`, para apresentar duas colunas específicas.

[48]:
```
população[[ 'Natalidade', 'Ano', 'Mortalidade' ]]
```

[48]:

| | Natalidade | Ano | Mortalidade |
|---|---|---|---|
| 0 | 9.2 | 2011 | 9.7 |
| 1 | 8.5 | 2012 | 10.2 |
| 2 | 7.9 | 2013 | 10.2 |

```
3          7.9  2014          10.1
4          8.3  2015          10.5
5          8.4  2016          10.7
6          8.4  2017          10.7
7          8.5  2018          11.0
```

### 1.1.4 Por coordenadas

Pode-se consultar os valores em posições específicas, com o médoto `DataFrame.iloc()`. Os índices começam no 0 (zero).

Convém recordar a notação do Python para selecionar partes de uma lista.

1. `população.iloc[4]`, selecionar a quinta linha
2. `população.iloc[4:6]`, selecionar a quinta e sexta linha
3. `população.iloc[0:5]`, selecionar as 5 primeiras linhas (linhas 0 a 4), equivalente a `população.head()`
4. `população.iloc[0,1]`, primeira linha, segunda coluna
5. `população.iloc[0:3, 0:2]`, primeiras 3 linhas (linhas 0, 1 e 2) e primeiras duas colunas (colunas 0 e 1)

[49]:
```
# Selecionar a quinta linha
# população.iloc[4]
# Selecionar a quinta e sexta linha
população.iloc[4:6]
# Selecionar as 5 primeiras linhas (linhas 0 a 4)
# população.iloc[0:5]
# população.head()
# Primeira linha, segunda coluna
# população.iloc[0,1]
# As primeiras 3 linhas (linhas 0, 1 e 2) e as primeiras duas colunas (colunas
↪0 e 1)
# população.iloc[0:3, 0:2]
```

[49]:
```
    Ano  Natalidade  Mortalidade
4  2015         8.3         10.5
5  2016         8.4         10.7
```

### 1.1.5 Exercício

Usando o método `população.iloc()`, mostra apenas a primeira e última linha da tabela.

[87]:
```
população.iloc[ [0, -1] ]
```

[87]:
```
    Ano  Natalidade  Mortalidade  Diferença
0  2011         9.2          9.7       -0.5
7  2018         8.5         11.0       -2.5
```

### 1.1.6 Metainformação

Os métodos anteriores permitem-nos consultar a informação que está contida na tabela. Ou seja, o seu conteúdo. A metainformação dá-nos as propriedades da tabela e não o conteúdo propriamente dito.

Estude as propriedades que são reportadas pelos seguintes métodos/funções:

[50]:
```
população.shape
```

[50]:
```
(8, 3)
```

[51]:
```
len(população)
```

[51]:
```
8
```

[52]:
```
população.columns
```

[52]:
```
Index(['Ano', 'Natalidade', 'Mortalidade'], dtype='object')
```

[53]:
```
população.dtypes
```

[53]:
```
Ano              int64
Natalidade     float64
Mortalidade    float64
dtype: object
```

[54]:
```
população.describe()
```

[54]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| count | 8.00000 | 8.000000 | 8.000000 |
| mean | 2014.50000 | 8.387500 | 10.387500 |
| std | 2.44949 | 0.408613 | 0.415546 |
| min | 2011.00000 | 7.900000 | 9.700000 |
| 25% | 2012.75000 | 8.200000 | 10.175000 |
| 50% | 2014.50000 | 8.400000 | 10.350000 |
| 75% | 2016.25000 | 8.500000 | 10.700000 |
| max | 2018.00000 | 9.200000 | 11.000000 |

### 1.1.7 Ordenar a tabela

A tabela está organizada por linhas e colunas. Tal como foi declarada a tabela já está ordenada por linhas, pelo coluna `'Ano'`, certo?

Podemos ordenar a tabela por linhas, mas usando a coluna `'Natalidade'`, por exemplo:

[55]:
```
população.sort_values(by=['Natalidade'])
```

[55]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 2 | 2013 | 7.9 | 10.2 |
| 3 | 2014 | 7.9 | 10.1 |
| 4 | 2015 | 8.3 | 10.5 |
| 5 | 2016 | 8.4 | 10.7 |
| 6 | 2017 | 8.4 | 10.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 7 | 2018 | 8.5 | 11.0 |
| 0 | 2011 | 9.2 | 9.7 |

A ordenação por ordem descrecente faz-se adicionando o parâmetro `ascending=False`:

[56]:
```
população.sort_values(by=['Natalidade'], ascending=False)
```

[56]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 7 | 2018 | 8.5 | 11.0 |
| 5 | 2016 | 8.4 | 10.7 |
| 6 | 2017 | 8.4 | 10.7 |
| 4 | 2015 | 8.3 | 10.5 |
| 2 | 2013 | 7.9 | 10.2 |
| 3 | 2014 | 7.9 | 10.1 |

O método `DataFrame.sort_index()` permite ordenar pelo índice das linhas (`axis=0`) ou pelo índice das colunas (`axis=1`).

Por exemplo, se quiseremos apresentar a coluna `'Natalidade'` em primeiro lugar, podemos fazer o seguinte:

[57]:
```
população.sort_index(axis=1, ascending=False)
```

[57]:

| | Natalidade | Mortalidade | Ano |
|---|---|---|---|
| 0 | 9.2 | 9.7 | 2011 |
| 1 | 8.5 | 10.2 | 2012 |
| 2 | 7.9 | 10.2 | 2013 |
| 3 | 7.9 | 10.1 | 2014 |
| 4 | 8.3 | 10.5 | 2015 |
| 5 | 8.4 | 10.7 | 2016 |
| 6 | 8.4 | 10.7 | 2017 |
| 7 | 8.5 | 11.0 | 2018 |

### 1.1.8 Filtar a tabela com expressões

Para além das variadas consultas já apresentadas, é muito prático filtar a o conteúdo com base em expressões sobre os valores.

[58]:
```
# população.Ano >= 2015

população[ população.Ano >= 2015 ]
```

[58]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 4 | 2015 | 8.3 | 10.5 |
| 5 | 2016 | 8.4 | 10.7 |
| 6 | 2017 | 8.4 | 10.7 |
| 7 | 2018 | 8.5 | 11.0 |

[59]:
```
população[ população.Ano.isin( [ 2012, 2016] )]
```

[59]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 1 | 2012 | 8.5 | 10.2 |
| 5 | 2016 | 8.4 | 10.7 |

[60]:
```
população[ (população.Natalidade >= 8.0) & (população.Mortalidade <= 10.0) ]
```

[60]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |

### 1.1.9 Valores agregados de uma coluna

[61]:
```
população.Natalidade.sum()
```

[61]: 67.1

[62]:
```
população.Natalidade.mean()
```

[62]: 8.3875

[63]:
```
população.Natalidade.max()
```

[63]: 9.2

[64]:
```
população.Natalidade.min()
```

[64]: 7.9

### 1.1.10 Exercício

Calcule o(s) ano(s) em que se verificou a taxa bruta de natalidade mínima anteriormente calculada.

[65]:
```
população[ população.Natalidade == população.Natalidade.min() ]
```

[65]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 2 | 2013 | 7.9 | 10.2 |

```
3  2014         7.9         10.1
```

### 1.1.11 Exercício

Calcule a taxa bruta de natalidade **média** entre 2015 e 2018.

[66]:
```
população[ população.Ano.isin( [ 2015, 2016, 2017, 2018] )]['Natalidade'].mean()
```

[66]:
```
8.4
```

### 1.1.12 Máximo por coluna

O método `DataFrame.max()` aplicado à tabela, retorna o máximo para cada uma das colunas.

[67]:
```
população.max()
```

[67]:
```
Ano            2018.0
Natalidade        9.2
Mortalidade      11.0
dtype: float64
```

### 1.1.13 Trocar linhas por colunas (tópico avançado)

A tabela `população` que temos estado a usar tem 8 linhas e 3 colunas: `Ano`, `Natalidade` e `Mortalidade`.

Podemos criar uma nova tabela trocando as linhas por colunas. Usando o método `DataFrame.transpose()` todas as colunas viram linhas, como no exemplo seguinte.

[68]:
```
p1 = população.transpose()
p1
```

[68]:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Ano | 2011.0 | 2012.0 | 2013.0 | 2014.0 | 2015.0 | 2016.0 | 2017.0 | 2018.0 |
| Natalidade | 9.2 | 8.5 | 7.9 | 7.9 | 8.3 | 8.4 | 8.4 | 8.5 |
| Mortalidade | 9.7 | 10.2 | 10.2 | 10.1 | 10.5 | 10.7 | 10.7 | 11.0 |

Como se vê, passamos a ter 8 colunas, indexadas de 0 a 7. Nalgumas situações, dá jeito que uma das colunas passe a ser o índice das colunas. Para tal, usa-se o método `DataFrame.set_index()` antes de `transpose()`, como se faz no exemplo seguinte:

[69]:
```
pop = população.set_index('Ano').transpose()
pop
```

[69]:

| Ano | 2011 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 | 2018 |
|---|---|---|---|---|---|---|---|---|
| Natalidade | 9.2 | 8.5 | 7.9 | 7.9 | 8.3 | 8.4 | 8.4 | 8.5 |
| Mortalidade | 9.7 | 10.2 | 10.2 | 10.1 | 10.5 | 10.7 | 10.7 | 11.0 |

Como a coluna `Ano` era do tipo `int`, os índices das colunas são também do tipo `int`.

[70]:
```
pop[[2011, 2012]]
```

[70]:
```
Ano          2011  2012
Natalidade    9.2   8.5
Mortalidade   9.7  10.2
```

### 1.1.14 Exercício (resolvido)

Este exercício demostra a facilidade com que se pode rearranjar uma tabela em `pandas`.

Imagine que nos davam os dados sobre a população no seguinte formato, em que as taxas são dadas por um par `( natalidade, mortalidade )`.

[71]:
```
dados = {
    '2011': (9.2, 9.7),
    '2012':        (8.5, 10.2),
    '2014':        (7.9, 10.1),
    '2015':        (8.3, 10.5),
    '2016':        (8.4, 10.7),
    '2017':        (8.4, 10.7),
    '2018':        (8.5, 11.0)
}
evolução = pandas.DataFrame(dados)
evolução
```

[71]:

| | 2011 | 2012 | 2014 | 2015 | 2016 | 2017 | 2018 |
|---|---|---|---|---|---|---|---|
| 0 | 9.2 | 8.5 | 7.9 | 8.3 | 8.4 | 8.4 | 8.5 |
| 1 | 9.7 | 10.2 | 10.1 | 10.5 | 10.7 | 10.7 | 11.0 |

Dado este formato, queremos voltar a ter o formato usado no início desta ficha, com oito linhas e três colunas:

[72]:
```
população
```

[72]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 2 | 2013 | 7.9 | 10.2 |
| 3 | 2014 | 7.9 | 10.1 |
| 4 | 2015 | 8.3 | 10.5 |
| 5 | 2016 | 8.4 | 10.7 |
| 6 | 2017 | 8.4 | 10.7 |
| 7 | 2018 | 8.5 | 11.0 |

**Resolução** Vamos começar por calcular a tabela transposta:

[73]:
```
transposta = evolução.transpose()
transposta
```

[73]:

| | 0 | 1 |
|---|---|---|
| 2011 | 9.2 | 9.7 |
| 2012 | 8.5 | 10.2 |
| 2014 | 7.9 | 10.1 |
| 2015 | 8.3 | 10.5 |
| 2016 | 8.4 | 10.7 |
| 2017 | 8.4 | 10.7 |
| 2018 | 8.5 | 11.0 |

A transposta fica com a geometria que queremos (as linhas passaram para colunas).

No entanto, queremos que os índices desta nova tabela (os anos) passem a ser uma coluna também. Para isso, fazemos `transposta.reset_index()`.

[74]:
```
transposta.reset_index()
```

[74]:

| | index | 0 | 1 |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |
| 2 | 2014 | 7.9 | 10.1 |
| 3 | 2015 | 8.3 | 10.5 |
| 4 | 2016 | 8.4 | 10.7 |
| 5 | 2017 | 8.4 | 10.7 |
| 6 | 2018 | 8.5 | 11.0 |

Depois, vamos renomear as colunas. Em vez de `['index', 0, 1]`, queremos que fique `[ 'Ano', 'Natalidade', 'Mortalidade' ]`.

Juntando o código todo necessário à resolução do exercício:

[75]:
```
dados = {
    '2011': (9.2, 9.7),
    '2012':        (8.5, 10.2),
    '2014':        (7.9, 10.1),
    '2015':        (8.3, 10.5),
    '2016':        (8.4, 10.7),
    '2017':        (8.4, 10.7),
    '2018':        (8.5, 11.0)
}
evolução = pandas.DataFrame(dados)
transposta = evolução.transpose()
transposta.reset_index()
nova = transposta.reset_index()
nova.columns = [ 'Ano', 'Natalidade', 'Mortalidade' ]
nova
```

[75]:

| | Ano | Natalidade | Mortalidade |
|---|---|---|---|
| 0 | 2011 | 9.2 | 9.7 |
| 1 | 2012 | 8.5 | 10.2 |

```
2  2014          7.9          10.1
3  2015          8.3          10.5
4  2016          8.4          10.7
5  2017          8.4          10.7
6  2018          8.5          11.0
```

### 1.1.15 Modificar a tabela

A tabela pode ser modificada.

As duas formas mais simples de o fazer são usando o método `DataFrame.at()` ou `DataFrame.iat()`.

Os dois exemplos seguinte são equivalentes: alteram a taxa bruta da natalidade da primeira linha da tabela.

[76]:
```
população.at[ 0, 'Natalidade' ] = 9.2
```

[77]:
```
população.iat[ 0, 1 ] = 9.2
```

[78]:
```
população.iloc[0, 1]
```

[78]:
```
9.2
```

### 1.1.16 Acrescentar uma coluna

Vamos criar uma coluna `'Diferença'` que resulta da diferenças entras as duas taxas representadas na tabela.

[79]:
```
população['Diferença'] = população.Natalidade - população.Mortalidade
população
```

[79]:
```
    Ano  Natalidade  Mortalidade  Diferença
0  2011          9.2          9.7       -0.5
1  2012          8.5         10.2       -1.7
2  2013          7.9         10.2       -2.3
3  2014          7.9         10.1       -2.2
4  2015          8.3         10.5       -2.2
5  2016          8.4         10.7       -2.3
6  2017          8.4         10.7       -2.3
7  2018          8.5         11.0       -2.5
```

### 1.1.17 Ler tabelas em arquivos

Frequentemente as tabelas que manipulamos em Python vêem de arquivos e contêm muitos valores. Por isso, convém dominar os métodos anteriormente apresentados, para podermos explorar e processar tabelas com muitos dados.

Considere a seguinte tabela (com poucas dezenas de linhas e colunas), disponível no repositório github. Como pode ver, o `pandas` lê sem problemas uma tabela remota, se lhe passarmos um endereço válido.

Explore o conteúdo da tabela, para perceber melhor o conteúdo, para depois fazer os exercícios pedidos.

[80]:
```
pandemia = pandas.read_csv('https://raw.githubusercontent.com/jgrocha/covid-pt/
↪master/situacao_epidemiologica.csv')
```

Esta não é considerada uma tabela grande. Mesmo assim, veja que é prático saber compor os métodos que se aprenderam para, por exemplo, mostrar os casos confirmados de COVID-19 nos últimos 10 dias.

Na mesma linha estamos a usar:

1. um método para ordenar `data_relatorio` por ordem descrecente;
2. a restingir apenas a duas colunas específicas;
3. a aproveitar apenas as 10 primeiras linhas.

[81]:
```
pandemia.sort_values(by=['data_relatorio'], ascending=False)[['data_relatorio',␣
↪'confirmados']].head(10)
```

[81]:

| | data_relatorio | confirmados |
|---|---|---|
| 64 | 2020-05-06 | 26182 |
| 63 | 2020-05-05 | 25702 |
| 14 | 2020-05-03 | 25282 |
| 46 | 2020-05-02 | 25190 |
| 45 | 2020-05-01 | 25351 |
| 43 | 2020-04-30 | 25056 |
| 42 | 2020-04-29 | 24322 |
| 38 | 2020-04-28 | 24322 |
| 15 | 2020-04-27 | 24027 |
| 11 | 2020-04-26 | 23864 |

### 1.1.18 Selecionar algumas colunas

Já vimos em exemplos anteriores que se podem selecionar algumas colunas por extensão, isto é, através de uma lista.

Por vezes, nestas tabelas maiores, dá muito jeito selecionar colunas com base numa expressão (por compreensão).

Por exemplo, temos casos confirmados para o género masculino e feminino divididos por grupos etários. O mesmo acontece com o registo de óbitos: há colunas para cada um dos géneros, por grupos etários.

Usando expressões regulares (um tópico avançado), podemos indicar um **padrão** e só as colunas que obedecem a esse padrão são apresentadas.

Experimente diferentes expressões.

[82]:
```
# pandemia.filter(regex=("obitos_")).head()
pandemia.filter(regex=("80_sup")).head()
```

[82]:
```
   confirmados_masculino_80_sup  confirmados_feminino_80_sup  \
0                           NaN                          NaN
1                           NaN                          NaN
2                           NaN                          NaN
3                           2.0                          0.0
4                           2.0                          1.0

   obitos_masculino_80_sup  obitos_feminino_80_sup
0                      NaN                     NaN
1                      NaN                     NaN
2                      NaN                     NaN
3                      NaN                     NaN
4                      NaN                     NaN
```

### 1.1.19 Exercício

Temos muitos registos, um para cada dia. Diga qual é o primeiro dia dos registos.

[83]:
```
pandemia.sort_values(by=['data_relatorio']).head(1)['data_relatorio']
```

[83]:
```
10    2020-03-03
Name: data_relatorio, dtype: object
```

### 1.1.20 Exercício

Os óbitos estão registados comulativamente. Isto é, para cada dia, estão indicados todos os óbitos registados até esse dia e não apenas os óbitos desse dia. Diga em que dia se atingiram os mil óbitos.

[84]:
```
pandemia[ pandemia.obitos >= 1000 ][[ 'data_relatorio', 'obitos']].
↪sort_values(by=['obitos']).head(1).data_relatorio
```

[84]:
```
45    2020-05-01
Name: data_relatorio, dtype: object
```