O package Pandas – Uma 1ª visão

- O Pandas (*panel data analysis*), sendo um *package* mais recente, suporta-se no próprio NumPy.
- Um dos seus principais recursos são os DataFrames

  (tipo de estrutura já familiar para os programadores de R)

  - *tratam-se de objetos que representam dados bidimensionais devidamente rotulados ou etiquetados, um pouco à semelhança do que acontece com as tabelas que nos habituamos a usar noutros ambientes, como o Excel, SQL, etc.*
  - *proporcionam-nos, no fundo, uma interface para os dados, que facilita imenso a sua representação e manipulação*
- Um DataFrame assume o seguinte aspeto:

```
import pandas as pd
df=pd.DataFrame({'label':['A','B','C','A','B','C'], 'value':[1,2,3,4,5,6]})
df
```

	label	value
0	A	1
1	B	2
2	C	3
3	A	4
4	B	5
5	C	6

Repare-se que o conteúdo inicial é fornecido ao DataFrame na forma de dicionário

Uma 1ª visão do Pandas

- Com essa interface para os dados, é possível realizar de forma simples um conjunto enorme de operações, como, por exemplo,
  - *selecionar colunas pelo nome:*

```
df['label']
```

```
0    A
1    B
2    C
3    A
4    B
5    C
```

  - *selecionar linhas com o método loc()*

```
df.loc[2]
```

```
label    C
value    3
```

  - *aplicar operações de manipulação de strings a colunas com dados desse tipo:*

```
df['label'].str.lower()
```

```
0    a
1    b
2    c
3    a
4    b
5    c
```

Uma 1ª visão do Pandas

- *aplicar funções de agregação a colunas com dados numéricos:*

```
df['value'].sum()
```

```
21
```

- *e, não menos importante, realizar operações de agregação por categoria numa coluna específica:*

```
df.groupby('label').sum()
```

label	value
A	5
B	7
C	9

Neste último exemplo, calcula-se, em cada linha, as somas de todos os valores que compartilham um mesmo 'label', algo que seria bem mais trabalhoso (e menos eficiente) caso usássemos as funcionalidades nativas do Python ou até mesmo do NumPy.

Entendendo melhor o Pandas

- Na maior parte dos problemas tratados na Data Science, os dados sob análise apresenta-se na forma de tabelas
  1. *em que cada coluna tem um nome que a identifica e um tipo de dados próprio (i.e., cada coluna pode armazenar um tipo de dados diferente)*

     (esta caraterísticas das colunas não pode ser assegurada pelos arrays NumPy, como facilmente se percebe)
  2. *e em que cada linha contém os dados referentes a um objeto, instância, indivíduo, registo ou observação*

nome	idade	presencas	notaIA
Ana	20	20	12.2
Rui	25	3	5.3
Gil	27	28	15.7
Zé	23	17	15.9
Tó	20	28	19.2

- Para se lidar facilmente com dados na forma de tabelas, o Pandas introduz um novo tipo de dados: o DataFrame
  - *trata-se de uma estrutura de dados que permite manipular de uma forma simples, eficiente e flexível tabelas de dados em ambiente Python*

Séries

- Para além dos DataFrames, o Pandas introduz uma outra estrutura de dados: as séries
  - *uma série é semelhante a um array NumPy unidimensional*
  - *mas em que os seus elementos, sendo por omissão indexados por um inteiro de acordo com a sua posição (0, 1, 2, …), podem também ser indexados por índices personalizados pelo utilizador*

```
s0=pd.Series([0,10,20,30,40]); print(s0)
```

```
0     0
1    10
2    20
3    30
4    40
```

```
s1=pd.Series([0,10,20,30,40], index=['A','A','B','C','D']); print(s1)
```

```
A     0
A    10
B    20
C    30
D    40
```

imagine que define novos índices inteiros, mas que não iniciem em 0

- O acesso aos elementos de uma série é semelhante ao dos arrays
  - *pode ser usada a posição ou o rótulo, com os métodos iloc() e loc() se necessário*

```
print(s1['B'], s1.iloc[2], s1.loc['B'])
```

```
20 20 20
```

  - *e as operações de slicing também são possíveis*

```
print(s1[2:4])
```

```
B    20
C    30
```

```
print(s1.iloc[2:4])
```

```
B    20
C    30
```

E quando há rótulos repetidos?

```
print(s1['A'])
```

```
A     0
A    10
```

Séries temporais

- Uma série temporal pode ser entendida como sendo uma série em que os seus elementos são indexados por índices que representam instantes de tempo
- Suponhamos que pretendemos registar o nosso peso diário ao longo duma semana
  - *Podemos então construir uma série temporal, começando pela definição do conjunto de índices a usar, através da função date_range()*

```
dias = pd.date_range('26-10-2020', periods=7); print(dias)
```

```
DatetimeIndex(['2020-10-26', '2020-10-27', '2020-10-28', '2020-10-29',
               '2020-10-30', '2020-10-31', '2020-11-01'],
              dtype='datetime64[ns]', freq='D')
```

nº de itens a criar

1º instante

'D': frequência diária. Existem muitas outras periodicidades que podem ser escolhidas

  - *Assumindo que os seguintes valores correspondem às 7 pesagens*

```
pesagens=np.round(np.random.random(7)+75,2) #simulação de valores
print(pesagens)
```

```
[75.48 75.71 75.17 75.73 75.21 75.59 75.57]
```

  - *facilmente criamos a série temporal correspondente*

```
pesos=pd.Series(pesagens, index=dias); print(pesos)
```

```
2020-10-26    75.48
2020-10-27    75.71
2020-10-28    75.17
2020-10-29    75.73
2020-10-30    75.21
2020-10-31    75.59
2020-11-01    75.57
Freq: D, dtype: float64
```

Índices com data/hora

- Para além da data, é também possível incluir a hora nos índices de uma série temporal
  - *Como exemplo, criemos um conjunto de índices com data/hora*

```
horas = pd.date_range('26-10-2020 15:20', periods=7); print(horas)
```

```
DatetimeIndex(['2020-10-26 15:20:00', '2020-10-27 15:20:00',
               '2020-10-28 15:20:00', '2020-10-29 15:20:00',
               '2020-10-30 15:20:00', '2020-10-31 15:20:00',
               '2020-11-01 15:20:00'],
              dtype='datetime64[ns]', freq='D')
```

Para além deste tipo de formato, o Pandas consegue interpretar vários outros formatos de data e hora

  - *Podemos agora atribuir esse novo conjunto de índices à série de pesos existente*

```
pesos.index=horas; print(pesos)
```

```
2020-10-26 15:20:00    75.48
2020-10-27 15:20:00    75.71
2020-10-28 15:20:00    75.17
2020-10-29 15:20:00    75.73
2020-10-30 15:20:00    75.21
2020-10-31 15:20:00    75.59
2020-11-01 15:20:00    75.57
Freq: D, dtype: float64
```

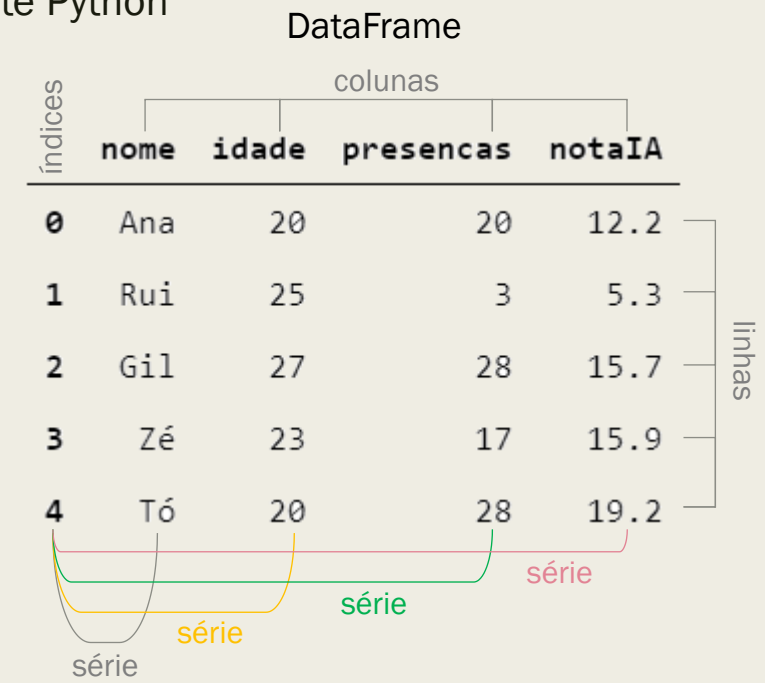
- Repare-se que enquanto pesos é uma série, dias e horas são apenas índices

```
print(type(pesos),type(dias),type(horas), sep='\n')
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
```

Voltemos aos DataFrames

- Comos se disse anteriormente, um DataFrame permite representar uma tabela de dados em ambiente Python



Como ilustrado, cada coluna individual forma, com a coluna de índices, uma série

- O DataFrame é a estrutura ideal para representar os dados da Data Science e da ML
  - *modela, fielmente, a forma como os dados são representados na vida real*
    - é habitual os dados tratados na ML encontrarem-se em tabelas SQL, em tabelas guardadas em ficheiros CSV ou até mesmo em tabelas do Excel
- Sendo por essa razão que os DataFrames do Pandas são tão usados na ML
  - *É nesse formato ou em arrays NumPy que os algoritmos de ML do package Scikit-learn esperam receber os dados*

Como criar um DataFrame

- O DataFrame ilustrado anteriormente pode ser criado instanciando a classe DataFrame do Pandas,
  - *fornecendo o conteúdo da tabela em forma de dicionário*

```
alunos = pd.DataFrame({'nome': ['Ana', 'Rui', 'Gil', 'Zé', 'Tó'],
'idade': [20,25,27,23,20], 'presencas': [20,3,28,17,28], 'notaIA':
[12.2,5.3,15.7,15.9,19.2]})
```

  - *ou fornecendo os dados e os nomes das colunas em listas separadas*

```
data=[['Ana',20,20,12.2], ['Rui',25,3,5.3], ['Gil',27,28,15.7],
['Zé',23,17,15.9], ['Tó',20,28,19.2]]
lables=['nome', 'idade', 'presencas', 'notaIA']
```

```
alunos=pd.DataFrame(data, columns=lables)
```

- Porém, como os *datasets* usados na ML são, por norma, bastante volumosos, os DataFrames acabam, na maior parte das vezes, por ser carregados automaticamente a partir de uma fonte externa
  - *é frequente, por exemplo, carregar-se um DataFrame a partir de um ficheiro de texto CSV, usando a função read_csv()*

```
alunos=pd.read_csv('dados.csv')
```

Para o caso do conteúdo da tabela se encontrar no ficheiro 'dados.csv'

Personalizar os índices num DataFrame (DF)

- Tal como com as séries, é possível personalizar os índices de um DF
  - *No DF alunos podemos, por exemplo, usar para indexação das linhas os números mecanográficos dos respetivos estudantes*

```
alunos.index=[31234,33333,40000,44444,30000]
```

```
alunos
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

- Se mostrarmos novamente o DF alunos, percebe-se que as linhas passaram efetivamente a ter uma nova forma de indexação

- Através dos atributos 'índex', 'columns' e 'values' conseguimos facilmente consultar (ou alterar) as 3 componentes importantes da DF

```
alunos.index
```

```
Int64Index([31234, 33333, 40000, 44444, 30000], dtype='int64')
```

```
alunos.columns
```

```
Index(['nome', 'idade', 'presencas', 'notaIA'], dtype='object')
```

```
alunos.values
```

```
array([['Ana', 20, 20, 12.2],
       ['Rui', 25, 3, 5.3],
       ['Gil', 27, 28, 15.7],
       ['Zé', 23, 17, 15.9],
       ['Tó', 20, 28, 19.2]], dtype=object)
```

Consulta rápida dum DataFrame

- Como os *datasets* tratados na ML são de grande dimensão, muito raramente se opta por mostrar a totalidade das linhas que compõem o DF correspondente
  - *Muitas vezes é suficiente consultarmos algumas das suas linhas para uma primeira verificação e validação do conteúdo da tabela*
  - *Para esse efeito, os DF dispõem dos dois métodos, head() e tail(), que nos permitem rapidamente consultar o conteúdos das primeiras e últimas linhas, respetivamente.*

```
alunos.head(2)
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3

```
alunos.tail(2)
```

	nome	idade	presencas	notaIA
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

  - *Quer o head() quer o tail(), quando invocados sem argumentos, apresentam, por defeito, 5 linhas.*

- Os DF contêm também um método muito útil, o describe(), que permite apresentar vários dados estatísticos relacionados com os valores numéricos guardados:
  - *quantidade de itens, média, desvio padrão, valor mínimo, 1º quartil, mediana, 3º quartil e valor máximo*

```
alunos.describe()
```

	idade	presencas	notaIA
count	5.000000	5.000000	5.000000
mean	23.000000	19.200000	13.660000
std	3.082207	10.281051	5.288951
min	20.000000	3.000000	5.300000
25%	20.000000	17.000000	12.200000
50%	23.000000	20.000000	15.700000
75%	25.000000	28.000000	15.900000
max	27.000000	28.000000	19.200000

Como selecionar colunas ou linhas específicas

- Uma ou mais colunas específicas podem ser selecionadas, usando como índice os seus nomes

```
alunos['idade']
```

```
31234    20
33333    25
40000    27
44444    23
30000    20
```

```
alunos[['nome','idade']]
```

	nome	idade
31234	Ana	20
33333	Rui	25

- Também podemos aceder à coluna através do respetivo atributo

```
alunos.idade
```

```
31234    20
33333    25
40000    27
44444    23
30000    20
```

  - *Quando se seleciona uma única coluna, obtém-se como resultado uma série; mas quando se selecionam várias o resultado já é um DF*

- A seleção de linhas específicas requer que se usem os métodos loc() e iloc()
  - *loc() caso se use como índice o respetivo rótulo*
  - *iloc() caso se use como índice a posição relativa*

```
alunos.loc[40000] # o mesmo que alunos.iloc[2]
```

```
nome          Gil
idade          27
presencas      28
notaIA       15.7
```

```
alunos.iloc[[1,3]] #o mesmo que alunos.loc[[33333,44444]]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
44444	Zé	23	17	15.9

Slicing em DataFrames

- A operação de *slicing* em DF pode ser realizada com base,

quer nas linhas,

```
alunos.iloc[1:3,:]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7

quer em ambas

```
alunos.iloc[1:3,2:]
```

	presencas	notaIA
33333	3	5.3
40000	28	15.7

quer nas colunas

```
alunos.iloc[:,0:2]
```

	nome	idade
31234	Ana	20
33333	Rui	25
40000	Gil	27
44444	Zé	23
30000	Tó	20

- Na operação de *slicing* também podem ser usados, em vez das posições, os rótulos dos índices em ambos os eixos (com o método loc(), claro)

```
alunos.loc[33333:40000,'presencas':]
```

	presencas	notaIA
33333	3	5.3
40000	28	15.7

Mas cuidado: usando-se rótulos na operação de *slicing*, o limite superior é incluído na seleção

Indexação booleana e células individuais

- Se o objetivo for selecionar linhas dum DF com base no valor das células, podemos sempre usar indexação booleana
  - *Por exemplo, os alunos com mais de 20 anos com menos de 25 presenças, são os seguintes*

```
alunos[(alunos.presencas<25) & (alunos.idade>20)]
```

	nome	idade	presencas	notaIA
33333	Rui	25	3	5.3
44444	Zé	23	17	15.9

- Naturalmente, também é possível aceder ao valor de uma única célula
  - *Por exemplo, se pretendermos saber a nota do aluno 40000, podemos usar o método at()*

```
alunos.at[40000,'notaIA']  #o mesmo que alunos.loc[40000,'notaIA'] e que alunos.iloc[2,3]
```

```
15.7
```

Relativamente aos restantes, o método at() é mais eficiente, pois só permite aceder a um elemento

DataFrame transposta

- Á semelhança do que acontecia nos arrays NumPy, é possível usar o método transpose() num DF caso se pretenda trocar as linhas pelas colunas
  - *Também se pode aceder a esse método através da propriedade T*

```
alunos.T  #o mesmo que alunos.transpose()
```

	31234	33333	40000	44444	30000
nome	Ana	Rui	Gil	Zé	Tó
idade	20	25	27	23	20
presencas	20	3	28	17	28
notaIA	12.2	5.3	15.7	15.9	19.2

Ordenar um DataFrame pelos índices

1. Com o método sort_index(), podemos ordenar as linhas dum DF pela coluna de índices

```
alunos.sort_index(axis=0)
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

2. Mas podemos também ordenar as colunas do DF pela linha de cabeçalho (nomes das colunas)

```
alunos.sort_index(axis=1)
```

	idade	nome	notaIA	presencas
31234	20	Ana	12.2	20
33333	25	Rui	5.3	3
40000	27	Gil	15.7	28
44444	23	Zé	15.9	17
30000	20	Tó	19.2	28

3. Porém, o sort_index() devolve um DF ordenado, não alterando o DF original

```
alunos
```

	nome	idade	presencas	notaIA
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9
30000	Tó	20	28	19.2

4. Para alterar o DF original, nesta, como noutras funções, devemos usar ‘inplace’

```
alunos.sort_index(axis=0, inplace=True)
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

Ordenar um DataFrame pelos valores

- Com o método sort_values() conseguimos ordenar um DF pelos valores das células
  - *Para ordenar as linhas dum DF pelos valores duma coluna específica, só temos que fornecer o nome dessa coluna*

```
alunos.sort_values('notaIA', ascending=False)
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
44444	Zé	23	17	15.9
40000	Gil	27	28	15.7
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3

neste exemplo, ordenam-se os alunos por ordem decrescente da sua nota

Neste, como nos restantes métodos de ordenação, podemos inverter a ordem com o parâmetro 'ascending'

  - *Já na ordenação das colunas dum DF pelos valores duma linha específica, devemos indicar o respetivo rótulo e escolher o segundo eixo (axis=1)*
    - Mas só fará sentido esta ordenação se todas as colunas forem de um mesmo tipo

anonimizemos o DF alunos para que fiquem apenas colunas numéricas, e assim podermos ordenar

```
anonimos=alunos.iloc[:,1:]
```

	idade	presencas	notaIA
30000	20	28	19.2
31234	20	20	12.2
33333	25	3	5.3
40000	27	28	15.7
44444	23	17	15.9

```
anonimos.sort_values(33333, axis=1)
```

E, se calhar, nem assim...

	presencas	notaIA	idade
30000	28	19.2	20
31234	20	12.2	20
33333	3	5.3	25
40000	28	15.7	27
44444	17	15.9	23

mesmo assim, não fará muito sentido ordenar valores com diferentes significados

Aplicar uma função a um DataFrame

Com o método apply(), é possível aplicar uma função aos valores de uma coluna dum DF,

1. quer se trate de uma função nossa

```
quad = lambda x: x**2
```

```
alunos.presencas.apply(quad)
```

```
30000    784
31234    400
33333      9
40000    784
44444    289
```

2. quer se trate de uma função do Python

```
alunos.notaIA.apply(round)
```

```
30000    19
31234    12
33333     5
40000    16
44444    16
```

Se a função tiver parâmetros adicionais, podem ser fornecidos através dum tuplo. Por exemplo, para se arredondar com 1 casa decimal:

```
alunos.notaIA.apply(round, args=(1,))
```

3. Note-se, porém, que a DF não é alterada

```
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

4. Mas se o objetivo for mesmo alterar o DF, refletindo em si os resultados da aplicação da função, basta atribuir esses resultados à respetiva coluna
   - *comecemos por duplicar o DF alunos, através de uma cópia em profundidade*

```
al=alunos.copy()
```

Caso pretendêssemos uma cópia superficial, podíamos usar o parâmetro 'deep':

```
#al=alunos.copy(deep=False)
```

Aplicar uma função a um DataFrame (continuação)

5. O arredondamento das notas do DF 'al', com alteração do próprio DF, far-se-ia da seguinte forma

```
al.notaIA=al.notaIA.apply(round)
```

```
al
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19
31234	Ana	20	20	12
33333	Rui	25	3	5
40000	Gil	27	28	16
44444	Zé	23	17	16

6. Se pretendermos aplicar uma função a várias colunas dum DF, ou a todas, podemos fazê-lo iterando

```
for c in alunos.columns[1:]:
    al[c]=al[c].apply(quad)
```

```
al
```

	nome	idade	presencas	notaIA
30000	Tó	400	784	361
31234	Ana	400	400	144
33333	Rui	625	9	25
40000	Gil	729	784	256
44444	Zé	529	289	256

7. Refira-se por fim que também é possível aplicar uma função a uma linha, em vez de ser a uma coluna, bastando, para isso, indexar a linha com o método loc (fornecendo o rótulo) ou iloc (fornecendo a posição)

```
anonimos.loc[40000].apply(quad)
```

```
idade        729.00
presencas    784.00
notaIA       246.49
```

```
anonimos.iloc[3].apply(quad)
```

```
idade        729.00
presencas    784.00
notaIA       246.49
```

Inserir ou remover uma coluna num DataFrame

1. Suponha-se que se pretende acrescentar ao DF uma nova coluna, indicando se o aluno tem ou não frequência do ano anterior

```
alunos['freqAnt']=[False,False,True,False,True]
```

	nome	idade	presencas	notaIA	freqAnt
30000	Tó	20	28	19.2	False
31234	Ana	20	20	12.2	False
33333	Rui	25	3	5.3	True
40000	Gil	27	28	15.7	False
44444	Zé	23	17	15.9	True

2. E se se pretender que a coluna seja inserida antes da coluna das notas?
   - *Para isso usamos o método insert()*

   Mas comecemos por voltar a retirar a coluna que foi inserida

```
alunos.drop('freqAnt', axis=1, inplace=True)
alunos
```

	nome	idade	presencas	notaIA
30000	Tó	20	28	19.2
31234	Ana	20	20	12.2
33333	Rui	25	3	5.3
40000	Gil	27	28	15.7
44444	Zé	23	17	15.9

3. Voltemos então a inserir a coluna, mas agora na posição 3

```
alunos.insert(3,'freqAnt',[False,False,True,False,True])
alunos
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9

O método drop(), à semelhança de outros, limita-se a devolver um novo DF, como resultado.

- Caso se pretenda que o DF original seja alterado, é necessário usar o parâmetro 'inplace'

E se se pretender eliminar várias colunas, basta fornecer ao método drop() uma lista com os nomes dessas colunas

Inserir ou remover uma linha num DataFrame

1. A inserção de uma linha, por indexação, requer que se use a o método loc() com o novo rótulo

```
alunos.loc[34567]=['Ivo',21,27,False,14]
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

2. Já na remoção de uma linha é suficiente passar ao método drop() o rótulo da mesma

```
alunos.drop(33333)
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
31234	Ana	20	20	False	12.2
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

- O método drop(), por defeito, remove linhas (axis=0 opcional)
- Ao não usarmos 'inplace', a remoção não se efetiva no próprio DF

3. Por vezes é necessário remover linhas pelos valores das suas células
   - *Suponha-se, por exemplo, que se pretende retirar do DF os alunos reprovados por faltas*
   - *que será equivalente a selecionar apenas os restantes*
   - *ou seja, que tenham mais que 24 presenças ou que já possuam frequência do ano anterior*

```
alunos[(alunos.presencas>=25) | alunos.freqAnt]
```

	nome	idade	presencas	freqAnt	notaIA
30000	Tó	20	28	False	19.2
33333	Rui	25	3	True	5.3
40000	Gil	27	28	False	15.7
44444	Zé	23	17	True	15.9
34567	Ivo	21	27	False	14.0

Tabela de frequências cruzadas com DataFrames

- Uma tabela de frequências cruzadas mostra a distribuição de duas ou mais variáveis categóricas de forma agrupada, possibilitando uma consulta rápida das possíveis relações entre elas
  - *Para vermos a aplicabilidade desse tipo de tabela, comecemos por acrescentar duas novas colunas de valores categóricos (não numéricos) ao DF alunos*

```
alunos.insert(1,'freq',['ordin','trab','erasm','ordin','ordin','trab'])
alunos['classif']=['Aprovado','reprovado','reprovado','Aprovado','Aprovado','Aprovado']
   alunos
```

	nome	freq	idade	presencas	freqAnt	notaIA	classif
30000	Tó	ordin	20	28	False	19.2	Aprovado
31234	Ana	trab	20	20	False	12.2	reprovado
33333	Rui	erasm	25	3	True	5.3	reprovado
40000	Gil	ordin	27	28	False	15.7	Aprovado
44444	Zé	ordin	23	17	True	15.9	Aprovado
34567	Ivo	trab	21	27	False	14.0	Aprovado

- Usando essas duas colunas no método crosstab(), conseguimos rapidamente perceber como o tipo de frequência do aluno estará relacionado com a aprovação

```
pd.crosstab(alunos.freq,alunos.classif)
```

classif	Aprovado	reprovado
freq		
erasm	0	1
ordin	3	0
trab	1	1