pandas101

January 25, 2022

[1]:
```
import pandas as pd
```

[3]:
```
students = ['Alice', 'Jack', 'Molly']
```

[4]:
```
#uma das formas mais fáceis de criar uma série com o pandas é usando pd.
 ↪Series()

pd.Series(students)
#o resultado é um objeto em série que o pandas automaticamente identifica o␣
 ↪tipo de dado da série com objeto
```

[4]:
```
0    Alice
1     Jack
2    Molly
dtype: object
```

[5]:
```
numbers = [1,2,3,4]
pd.Series(numbers)
```

[5]:
```
0    1
1    2
2    3
3    4
dtype: int64
```

[6]:
```
flo = [1.5,1.8,2.6]
pd.Series(flo)
```

[6]:
```
0    1.5
1    1.8
2    2.6
dtype: float64
```

[9]:
```
#NaN não é equialente a None
import numpy as np
np.nan === None
```

[9]:
```
True
```

[10]:
```
#para testar a presença de um NaN é preciso uma library do Numpy, isnan()
np.isnan(np.nan)
```

[10]:
```
True
```

[13]:
```
students_scores = {'Alice': 'Fisica',
                   'Jack': 'Química',
                   'Molly': 'Inglês'}
s = pd.Series(students_scores)
s
```

[13]:
```
Alice     Fisica
Jack     Química
Molly     Inglês
dtype: object
```

[17]:
```
s.values
```

[17]:
```
array(['Fisica', 'Química', 'Inglês'], dtype=object)
```

[18]:
```
#um objeto dtype não é só para strings, mas para objetos arbitrários
students = [('Jack', 'White'), ('Steve', 'Ray'), ('John', 'Cena')]
pd.Series(students)
```

[18]:
```
0    (Jack, White)
1     (Steve, Ray)
2     (John, Cena)
dtype: object
```

[19]:
```
#da pra criar seu próprio index, passando o index como uma lista explicita para␣
 ↪a série
s = pd.Series(['Física', 'Química', 'Inglês'], index = ['Jack', 'Steve',␣
 ↪'John'])
s
```

[19]:
```
Jack      Física
Steve    Química
John      Inglês
dtype: object
```

[20]:
```
#e daí o que acontece se a sua lista não está alinhada com as chaves do␣
 ↪dicionário?
#o pandas sobrescreve a criação automática da série para favorecer todos os␣
 ↪indices que você proveu
#então ele ignora todas as chaves do seu dicionári oque não estão no seu index,␣
 ↪e adiciona um NONE ou NAN
#para todos os tipos de indexes que você inseriu e que não estão na lista de␣
 ↪chaves do seu dicionário

students_scores = {'Alice': 'Fisica',
                   'Jack': 'Química',
                   'Molly': 'Inglês'}
s = pd.Series(students_scores, index = ['Alice', 'Molly', 'Sam'])
s
```

[20]:
```
Alice    Fisica
Molly    Inglês
Sam         NaN
dtype: object
```

1 Query and merge series objects together

[ ]:
```
#uma série do pandas pode ser consultada tanto pela posição do index quanto␣
 ↪pelo nome do index. se não der
#um index quando consultando, a posição e a legenda são a mesma. para fazer uma␣
 ↪consulta pela posição numérica,
#começando do zero 0, usa-se o atributo 'iloc'. para consultar pela legenda,␣
 ↪usase o atributo 'loc'

#começando com um exemplo de alunos matriculados em cursos
```

[2]:
```
students_classes = {'Alice': 'Fisica',
                    'Jack': 'Química',
                    'Molly': 'Inglês',
                    'Sam': 'History'}
s = pd.Series(students_classes)
s
```

[2]:
```
Alice     Fisica
Jack     Química
Molly     Inglês
Sam      History
dtype: object
```

[4]:
```
#para a série acima, podemos consultar o 4ř valor usado o atributo iloc com o␣
 ↪número 3
s.iloc[3]
```

[4]:
```
'History'
```

[5]:
```
#se quisermos ver qual o curso da Molly, usamos o atributo loc com o parâmetro␣
 ↪Molly
s.loc['Molly']
```

[5]:
```
'Inglês'
```

[ ]:
```
#o loc e o iloc não são métodos, são atributos, portanto não usa-se o␣
 ↪partânteses () para chamá-los, mas colchetes []
#que são chamados de operadores de indexação
```

[6]:
```
#o pandas tenta manter o código bonitinho e usa vários truques para facilitar a␣
 ↪syntax
#então podemos chamar o operador de indexação da série de forma mais direta.
```

```
#por exemplo, se você passar um parâmetro integer, ele vai ver que você quer o␣
 ↪iloc
s[3]
```

[6]:
```
'History'
```

[7]:
```
#se você passar um parâmetro objeto, ele vai ver que você quer o 'loc'
s['Molly']
```

[7]:
```
'Inglês'
```

[9]:
```
#mas daí o que acontece se você criar uma série de números inteiros como␣
 ↪indexadores?
#o pandas não consegue determinar automaticamente se você quer fazer uma␣
 ↪consulta pelo label ou pelo index
#a opção mais segura é ser mais explicito e usar o iloc ou loc atributos com []␣
 ↪colchetes

#aqui um exemplo usando class e classcode, onde classes são indexadas pelo␣
 ↪classcode na forma de inteiros
class_code = {99: 'Fisica',
              100: 'Química',
              101: 'Inglês',
              102: 'History'}
s = pd.Series(class_code)
s
```

[9]:
```
99      Fisica
100    Química
101     Inglês
102    History
dtype: object
```

[14]:
```
s.iloc[3]
```

[14]:
```
'History'
```

[ ]:
```
#uma tarefa comum quando se trabalhando com os dados de uma série é considerar␣
 ↪todos os valores dentro de uma série
#e fazer albuma operação, tanto para achar um número,somar os valores ou␣
 ↪transformar os dados de alguma forma
```

[16]:
```
#uma abordagem de programação para isso seria iterar sobre todos os itens da␣
 ↪série e fazer uma operação desejada.
#por exemplo, podemos criar uma série de números inteiros representando a nota␣
 ↪de alunos e achar um valor médio

grades = pd.Series([90, 80, 70, 60])
total = 0
for grade in grades:
    total += grade
```

```
print(total/len(grades))
```

```
75.0
```

[21]:
```
#baixo é como se faria o código usando o método sum do numpy
import numpy as np

#agora chamamos np.sum e passamos como atributo a nossa série do pandas
total = np.sum(grades)
print(total/len(grades))     #essse código faz a mesma coisa que o de cima
```

```
75.0
```

[ ]:
```
#os dois métodos fazem a mesma coisa, mas é um mais rápido que o outro?
#o júpiter notebook tem uma função mágica que pode ajudar

#primeiro criamos uma série bem grande de números aleatórios, isso é usado␣
 ↪quando demostrando técnicas com o Pandas
```

[42]:
```
numbers = pd.Series(np.random.randint(0,1000,10000)) #me crie dez mil números␣
 ↪de 0 a 1000
numbers.head()
```

[42]:
```
0    352
1    423
2    185
3    131
4    439
dtype: int64
```

[40]:
```
len(numbers)
```

[40]:
```
10000
```

[ ]:
```
#o interpretador ipython tem uma coisa chamada funções mágicas que começam com␣
 ↪um símbolo de %
#se digitarmos esse código e apertarmos o botão tab, veremos uma lista de␣
 ↪funções mágiocas que podemos usar
```

[ ]:
```
#aqui vamos usar uma função mágica celular. ela começa com dois %%. o nome dela␣
 ↪é timeit. Essa função vai rodar
#o código algumas vezes para determinar o tempo médio de execução
```

[ ]:
```
#vamos rodar primeiro com o primeiro código. podemos dar para a timeit o número␣
 ↪de loops que quisermos, por default é 1mil
#para rodar, ela precisa ser escrita na primeira linha da célula
```

[31]:
```
%%timeit -n 1000
total = 0
for number in numbers:
```

```
    total += number
total/len(numbers)
```

```
1.11 ms ś 19.7 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

[ ]:
```
#agora vamos tentar com vetorização
```

[32]:
```
%%timeit -n 1000
total = np.sum(numbers)
total/len(numbers)     #essse código faz a mesma coisa que o de cima
```

```
66.8 ţs ś 564 ns per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

[ ]:
```

```

[43]:
```
#uma coisa relacionada entre pandas e numpy é o broadcasting, com ele podemos␣
 ↪aplicar uma operação para cada valor da série,
#mudando a série. por exemplo, se quisermos aumentar em 2 cada valor aleatório,␣
 ↪poderíamos fazer rapidamente
#usando o operador += diretamente na série do objeto

numbers.head()
#agora vamos adicionar 2 números a todos os valores
```

[43]:
```
0    352
1    423
2    185
3    131
4    439
dtype: int64
```

[44]:
```
numbers+=2
```

[45]:
```
numbers.head()
```

[45]:
```
0    354
1    425
2    187
3    133
4    441
dtype: int64
```

[48]:
```
#uma maneira de fazer isso seria iterando sobre os valores de todos os números␣
 ↪da série acrescentando 2 a cada um deles
#o panda suporta fazer isso atrvés de uma série como um dicionário, permitindo␣
 ↪desenpacotar valores facilmente

#oderíamos usar a função iteritems() que retorna um label e valor
for label, value in numbers.iteritems():
    numbers.set_value(label, value+2)
```

```
numbers.head()
```

[48]:
```
0    356
1    427
2    189
3    135
4    443
dtype: int64
```

[ ]:
```
#agora vamos ver a velocidade entre as duas opções
```

[69]:
```
%%timeit -n -10
s = pd.Series(np.random.randint(0, 1000, 10000))
for label, value in s.iteritems():
    s.loc[label] = value + 2
```

```
-15.3 ns ś 13.3 ns per loop (mean ś std. dev. of 7 runs, -10 loops each)
```

[70]:
```
%%timeit -n -10
s = pd.Series(np.random.randint(0, 1000, 10000))
s+=2

#agora usando o broadcasting
```

```
-11.9 ns ś 6.92 ns per loop (mean ś std. dev. of 7 runs, -10 loops each)
```

[ ]:
```

```

[ ]:
```
#o atributo .loc[] deixa você, não apenas modificar os dados num lugar, ams␣
 ↪também adicionar novos dados
#se o valor que você passar não existir então uma nova entrada é adicionada.
#indices podem ter tipos misturados
```

[71]:
```
#criando um exemplo usando Serie e alguns números
s = pd.Series([1,2,3])

#agora dicionamos um novo valor que não existe na série
s.loc['História'] = 102


s
```

[71]:
```
0             1
1             2
2             3
História    102
dtype: int64
```

[ ]:
```

```

[ ]:
```
#como 'história' não estava na série original, ele cria uma nova entrada na␣
 ↪série com o valor que passamos 102
```

[73]:
```
#aqui vai um exemplo onde os indexadores da série não são únicos, isso daí␣
 ↪torna a série do pandas um pouco diferente
#de um banco de dados relacional

#vamos criar uma série onde estudantes e os cursos que fazem

students_classes = pd.Series({'Alice': 'Fisica',
                  'Jack': 'Química',
                  'Molly': 'Inglês',
                  'Sam': 'History'})
students_classes
```

[73]:
```
Alice     Fisica
Jack     Química
Molly     Inglês
Sam      History
dtype: object
```

[74]:
```
#agora vamos criar uma nova série só para a aluna Kelly, que lista todos os␣
 ↪cursos que ela já fez, vamos setar o index
#para Kelly e os dados para ser o nome dos cursos
kelly_classes = pd.Series(['Filosofia', 'Artes', 'Math'], index=['Kelly',␣
 ↪'Kelly', 'Kelly'])
kelly_classes
```

[74]:
```
Kelly    Filosofia
Kelly        Artes
Kelly         Math
dtype: object
```

[75]:
```
#agora podemos dar um .append() nessa ultima série para a primeira série
all_students = students_classes.append(kelly_classes)
all_students
```

[75]:
```
Alice       Fisica
Jack       Química
Molly       Inglês
Sam        History
Kelly    Filosofia
Kelly        Artes
Kelly         Math
dtype: object
```

[76]:
```
#agora se formos consultar a série all_students por Kelly, teremos de volta␣
 ↪todos os valores referentes a ela
all_students.loc['Kelly']
```

[76]:
```
Kelly    Filosofia
Kelly        Artes
Kelly         Math
```

```
dtype: object
```

[ ]:

[ ]: