Aula 3B - Matemática Discreta: Parte 2

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Junho de 2020

1 Fundamentos de Matemática Discreta com Python - Parte 2

1.1 Controle de fluxo: condicionais `if`, `elif` e `else`

Em Python, como na maioria das linguagens, o operador `if` ("se") serve para tratar situações quando um bloco de instruções de código precisa ser executado apenas se uma dada condição estabelecida for avaliada como verdadeira. Um bloco condicional é escrito da seguinte forma:

```
if condição:
    # faça algo
```

Este bloco diz basicamente o seguinte: "faça algo se a condição for verdadeira". Vejamos alguns exemplos.

[1]:
```
if 2 > 0: # a condição é 'True'
    print("2 é maior do que 0!")
```

```
2 é maior do que 0!
```

[2]:
```
2 > 0 # esta é a condição que está sendo avaliada
```

[2]:
```
True
```

[3]:
```
if 2 < 1: # nada é impresso porque a condição é 'False'
    print("2 é maior do que 0!")
```

[4]:
```
2 < 1 # esta é a condição que está sendo avaliada
```

[4]:
```
False
```

A condição pode ser formada de diversas formas desde que possa ser avaliada como `True` ou `False`.

[5]:
```
x, y = 2, 4
if x < y:
    print(f'{x} < {y}')
```

```
2 < 4
```

A estrutura condicional pode ser ampliada com um ou mais `elif` ("ou se") e com `else` (senão). Cada `elif`, uma redução de *else if*, irá testar uma condição adicional se a condição relativa a `if` for `False`. Se alguma delas for testada como `True`, o bloco de código correspondende será executado. Caso contrário, a decisão do interpretador será executar o bloco que acompanhará `else`.

Exemplo: teste da tricotomia. Verificar se um número é $>$, $<$ ou $=0$.

[6]:
```
x = 4.1 # número para teste

if x < 0: # se
    print(f'{x} < 0')
elif x > 0: # ou se
    print(f'{x} > 0')
else: # senão
    print(f'{x} = 0')
```

```
4.1 > 0
```

Exemplo: Considere o conjunto de classificações sanguíneas ABO (+/-)

$$S = \{\text{A+}, \text{A-}, \text{B+}, \text{B-}, \text{AB+}, \text{AB-}, \text{O+}, \text{O-}\}$$

Se em um experimento aleatório, n pessoas ($n \geq 500$) diferentes entrassem por um hospital em um único dia, qual seria a probabilidade de p entre as n pessoas serem classificadas como um(a) doador(a) universal (sangue O-) naquele dia? Em seguida, estime a probabilidade das demais.

[7]:
```
# 'randint' gera inteiros aleatoriamente
from random import randint

# número de pessoas
n = 500

# associa inteiros 0-7 ao tipo sanguíneo
tipos = [i for i in range(0,8)]
sangue = dict(zip(tipos,['A+','A-','B+','B-','AB+','AB-','O+','O-']))

# primeira pessoa
i = randint(0,8)

# grupo sanguíneo
s = []

# repete n vezes
for _ in range(0,n):
    if i == 0:
        s.append(0)
    elif i == 1:
        s.append(1)
    elif i == 2:
        s.append(2)
    elif i == 3:
```

```
        s.append(3)
    elif i == 4:
        s.append(4)
    elif i == 5:
        s.append(5)
    elif i == 6:
        s.append(6)
    else:
        s.append(7)

    i = randint(0,7) # nova pessoa

# calcula a probabilidade do tipo p em %.
# Seria necessário definir uma lambda?
prob = lambda p: p/n*100

# armazena probabilidades no dict P
P = {}
for tipo in tipos:
    P[tipo] = prob(s.count(tipo))
    if sangue[tipo] == 'O-':
        print('A probabilidade de ser doador universal é de {0:.2f}%.'.
 ↪format(P[tipo]))
    else:
        print('A probabilidade de ser {0:s} é de {1:.2f}%.'.
 ↪format(sangue[tipo],P[tipo]))
```

```
A probabilidade de ser A+ é de 11.60%.
A probabilidade de ser A- é de 12.80%.
A probabilidade de ser B+ é de 10.80%.
A probabilidade de ser B- é de 12.60%.
A probabilidade de ser AB+ é de 14.80%.
A probabilidade de ser AB- é de 12.20%.
A probabilidade de ser O+ é de 13.80%.
A probabilidade de ser doador universal é de 11.40%.
```

1.2 Conjuntos

As estruturas `set` (conjunto) são úteis para realizar operações com conjuntos.

[8]:
```
set(['a','b','c']) # criando por função
```

[8]:
```
{'a', 'b', 'c'}
```

[9]:
```
{'a','b','c'} # criando de modo literal
```

[9]:
```
{'a', 'b', 'c'}
```

[10]:
```
{1,2,2,3,3,4,4,4} # 'set' possui unicidade de elementos
```

[10]:
```
{1, 2, 3, 4}
```

1.2.1 União de conjuntos

Considere os seguintes conjuntos.

[11]:
```
A = {1,2,3}
B = {3,4,5}
C = {6}
```

[12]:
```
A.union(B) # união
```

[12]:
```
{1, 2, 3, 4, 5}
```

[13]:
```
A | B # união com operador alternativo ('ou')
```

[13]:
```
{1, 2, 3, 4, 5}
```

1.2.2 Atualização de conjuntos (união)

A união *in-place* de dois conjuntos pode ser feita com `update`.

[14]:
```
C
```

[14]:
```
{6}
```

[15]:
```
C.update(B) # C é atualizado com elementos de B
C
```

[15]:
```
{3, 4, 5, 6}
```

[16]:
```
C.union(A) # conjunto união com A
```

[16]:
```
{1, 2, 3, 4, 5, 6}
```

[17]:
```
C # os elementos de A não foram atualizados em C
```

[17]:
```
{3, 4, 5, 6}
```

A atualização da união possui a seguinte forma alternativa com `|=`.

[18]:
```
C |= A # elementos de A atualizados em C
C
```

[18]:
```
{1, 2, 3, 4, 5, 6}
```

1.2.3 Interseção de conjuntos

[19]:
```
A.intersection(B) # interseção
```

[19]:
```
{3}
```

[20]:
```
A & B # interseção com operador alternativo ('e')
```

[20]:
```
{3}
```

1.2.4 Atualização de conjuntos (interseção)

A interseção *in-place* de dois conjuntos pode ser feita com `intersection_update`.

[21]:
```
D = {1, 2, 3, 4}
E = {2, 3, 4, 5}
```

[22]:
```
D.intersection(E) # interseção com E
```

[22]:
```
{2, 3, 4}
```

[23]:
```
D # D inalterado
```

[23]:
```
{1, 2, 3, 4}
```

[24]:
```
D.intersection_update(E)
D # D alterado
```

[24]:
```
{2, 3, 4}
```

A atualização da interseção possui a seguinte forma alternativa com `&=`.

[25]:
```
D &= E
D
```

[25]:
```
{2, 3, 4}
```

1.2.5 Diferença entre conjuntos

[26]:
```
A
```

[26]:
```
{1, 2, 3}
```

[27]:
```
D
```

[27]:
```
{2, 3, 4}
```

[28]:
```
A.difference(D) # apenas elementos de A
```

[28]:
```
{1}
```

[29]:
```
D.difference(A) # apenas elementos de D
```

[29]:
```
{4}
```

[30]:
```
A - D # operador alternativo
```

[30]:
```
{1}
```

[31]:
```
D - A
```

[31]:
```
{4}
```

1.2.6 Atualização de conjuntos (diferença)

A interseção *in-place* de dois conjuntos pode ser feita com `difference_update`.

[32]:
```
D = {1, 2, 3, 4}
E = {1, 2, 3, 5}
```

[33]:
```
D
```

[33]: {1, 2, 3, 4}

[34]:
```
D.difference(E)
D
```

[34]: {1, 2, 3, 4}

[35]:
```
D.difference_update(E)
D
```

[35]: {4}

A atualização da diferença possui a seguinte forma alternativa com -=.

[36]:
```
D -= E
D
```

[36]: {4}

1.2.7 Adição ou remoção de elementos

[37]:
```
A
```

[37]: {1, 2, 3}

[38]:
```
A.add(4) # adiciona 4 a A
A
```

[38]: {1, 2, 3, 4}

[39]:
```
B
```

[39]: {3, 4, 5}

[40]:
```
B.remove(3) # remove 3 de B
B
```

[40]: {4, 5}

1.2.8 Reinicialização de um conjunto (vazio)

Podemos remover todos os elementos de um conjunto com `clear`, deixando-o em um estado vazio.

[41]:
```
A
```

[41]: {1, 2, 3, 4}

[42]:
```
A.clear()
A # A é vazio
```

[42]: set()

[43]:
```
len(A) # 0 elementos
```

[43]: 0

1.2.9 Diferença simétrica

A diferença simétrica entre dois conjuntos A e B é dada pela união dos complementares relativos:

$$A \triangle B = A \backslash B \cup B \backslash A$$

Logo, em $A \triangle B$ estarão todos os elementos que pertencem a A ou a B mas não aqueles que são comuns a ambos.

Nota: os complementares relativos $A \backslash B$ e $B \backslash A$ aqui podem ser interpretados como $A - B$ e $B - A$. Os símbolos $\backslash$ e $-$ em conjuntos podem ter sentidos diferentes em alguns contextos.

[44]:
```
G = {1,2,3,4}
H = {3,4,5,6}
```

[45]:
```
G.symmetric_difference(H) # {3,4} ficam de fora, pois são interseção
```

[45]:
```
{1, 2, 5, 6}
```

[46]:
```
G ^ H # operador alternativo
```

[46]:
```
{1, 2, 5, 6}
```

1.2.10 Atualização de conjuntos (diferença simétrica)

A diferença simétrica *in-place* de dois conjuntos pode ser feita com `symmetric_difference_update`.

[47]:
```
G
```

[47]:
```
{1, 2, 3, 4}
```

[48]:
```
G.symmetric_difference_update(H)
G # alterado
```

[48]:
```
{1, 2, 5, 6}
```

[49]:
```
G ^= H # operador alternativo
G
```

[49]:
```
{1, 2, 3, 4}
```

1.2.11 Continência

Podemos verificar se um conjunto A é subconjunto de (está contido em) outro conjunto B ($A \subseteq B$) ou se B é um superconjunto para (contém) A ($B \supseteq A$) com `issubset` e `issuperset`.

[50]:
```
B
```

[50]:
```
{4, 5}
```

[51]:
```
C
```

[51]:
```
{1, 2, 3, 4, 5, 6}
```

[52]:
```
B.issubset(C) # B está contido em C
```

[52]:
```
True
```

[53]:
```
C.issuperset(B) # C contém B
```

[53]:
```
True
```

1.3 Subconjuntos e subconjuntos próprios

Podemos usar operadores de comparação entre conjuntos para verificar continência.

- $A \subseteq B$: A é subconjunto de B
- $A \subset B$: A é subconjunto próprio de B (A possui elementos que não estão em B)

[54]:
```
{1,2,3} <= {1,2,3} # subconjunto
```

[54]:
```
True
```

[55]:
```
{1,2} < {1,2,3} # subconjunto próprio
```

[55]:
```
True
```

[56]:
```
{1,2,3} > {1,2}
```

[56]:
```
True
```

[57]:
```
{1,2} >= {1,2,3}
```

[57]:
```
False
```

1.3.1 Disjunção

Dois conjuntos são disjuntos se sua interseção é vazia. Podemos verificar a disjunção com `isdisjoint`

[58]:
```
E
```

[58]:
```
{1, 2, 3, 5}
```

[59]:
```
G
```

[59]:
```
{1, 2, 3, 4}
```

[60]:
```
E.isdisjoint(G) 1,2,5 são comuns
```

```
  File "<ipython-input-60-89ce3afb7e04>", line 1
    E.isdisjoint(G) 1,2,5 são comuns
                    ^
SyntaxError: invalid syntax
```

[61]:
```
D
```

[61]:
```
{4}
```

[62]:
```
E.isdisjoint(D)
```

[62]:
```
True
```

[63]:
```
A
```

[63]:
```
set()
```

[64]:
```
E.isdisjoint(A)
```

[64]:
```
True
```

1.3.2 Igualdade entre conjuntos

Dois conjuntos são iguais se contém os mesmos elementos.

[65]:
```
H = {3,'a', 2}
I = {'a',2, 3}
J = {1,'a'}
```

[66]:
```
H == I
```

[66]:
```
True
```

[67]:
```
H == J
```

[67]:
```
False
```

[68]:
```
{1,2,2,3} == {3,3,3,2,1} # lembre-se da unicidade
```

[68]:
```
True
```

1.4 Compreensão de conjunto

Podemos usar `for` para criar conjuntos de maneira esperta do mesmo modo que as compreensões de lista e de dicionários. Neste caso, o funcionamento é como `list`, porém, em vez de colchetes, usamos chaves.

[69]:
```
{e for e in range(0,10)}
```

[69]:
```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

[70]:
```
{(i,v) for (i,v) in enumerate(range(0,4))}
```

[70]:
```
{(0, 0), (1, 1), (2, 2), (3, 3)}
```

1.5 Sobrecarga de operadores

Em Python, podemos realizar alguns procedimentos úteis para laços de repetição.

[71]:
```
x = 2
x += 1 # x = 2 + 1 (incrementação)
x
```

[71]:
```
3
```

[72]:
```
y = 3
y -= 1 # y = 3 - 1 (decrementação)
y
```

[72]:
```
2
```

[73]:
```
z = 2
z *= 2 # z = 2*2
z
```

[73]:
```
4
```

[74]:
```
t = 3
t /= 3 # t = 3/3
t
```

[74]:
```
1.0
```

Exemplo: verifique se a soma das probabilidades no `dict P` do experimento aleatório é realmente 100%.

[75]:
```
s = 0
for p in P.values(): # itera sobre os valores de P
    s += p # soma cumulativa
print(f'A soma de P é {s}%')
```

```
A soma de P é 100.0%
```

De modo mais Pythônico:

[76]:
```
sum(P.values()) == 100
```

[76]:
```
True
```

Ou ainda:

[77]:
```
if sum(P.values()) == 100:
    print(f'A soma de P é {s}%')
else:
    print(f'Há erro no cálculo!')
```

```
A soma de P é 100.0%
```

1.6 Controle de fluxo: laço `while`

O condicional `while` permite que um bloco de código seja repetidamente executado até que uma dada condição seja avaliada como `False`, ou o laço seja explicitamente terminado com a keyword `break`. Em laços `while`, é muito comum usar uma linha de atualização da condição usando sobrecarga de operadores.

A instrução é como segue:

```
while condicao:
    # faça isso
    # atualize condicao
```

[78]:
```
x = 10
boom = 0
while x > boom: # não leva em conta igualdade
```

```
    print(x)
    x -= 1 # atualizando por decrementação
print('Boom!')
```

```
10
9
8
7
6
5
4
3
2
1
Boom!
```

[79]:
```
x = 5
boom = 10
while x <= boom: # leva em conta igualdade
    print(x)
    x += 0.5 # atualizando por incrementação
```

```
5
5.5
6.0
6.5
7.0
7.5
8.0
8.5
9.0
9.5
10.0
```

[80]:
```
from math import sin,pi
x = 1.0
i = 1
while x**3 > 0:
    if i % 100 == 0: # imprime apenas a cada 1000 repetições
        print(f'Repeti {i} vezes e x = {x**3}. Contando...')
    x -= 1e-3  # atualiza o decremento
    i += 1 # contagem de repetição
print(f'x = {x**3}')
```

```
Repeti 100 vezes e x = 0.7314327009999998. Contando...
Repeti 200 vezes e x = 0.5139224009999996. Contando...
```

```
Repeti 300 vezes e x = 0.3444721009999996. Contando...
Repeti 400 vezes e x = 0.2170818009999996. Contando...
Repeti 500 vezes e x = 0.12575150099999965. Contando...
Repeti 600 vezes e x = 0.06448120099999974. Contando...
Repeti 700 vezes e x = 0.02727090099999983. Contando...
Repeti 800 vezes e x = 0.008120600999999913. Contando...
Repeti 900 vezes e x = 0.0010303009999999755. Contando...
Repeti 1000 vezes e x = 9.999999999973564e-10. Contando...
x = -6.8435572439409775e-46
```

[81]:
```python
from math import sin,pi
x = 1.0
i = 1
while x**3 > 0:
    if i % 100 == 0: # imprime apenas a cada 1000 repetições
        print(f'Repeti {i} vezes e x = {x**3}. Contando...')
    if i == 500:
        print(f'Repeti demais. Vou parar.')
        break # execução interrompida aqui
    x -= 1e-3  # atualiza o decremento
    i += 1 # contagem de repetição
print(f'x = {x**3}')
```

```
Repeti 100 vezes e x = 0.7314327009999998. Contando...
Repeti 200 vezes e x = 0.5139224009999996. Contando...
Repeti 300 vezes e x = 0.3444721009999996. Contando...
Repeti 400 vezes e x = 0.2170818009999996. Contando...
Repeti 500 vezes e x = 0.12575150099999965. Contando...
Repeti demais. Vou parar.
x = 0.12575150099999965
```

Exemplo: construa seu próprio gerador de números aleatórios para o problema da entrada de pessoas no hospital.

[82]:
```python
# exemplo simples
def meu_gerador():
    nums = []
    while True: # executa indefinidamente até se digitar ''
        entr = input() # entrada do usuário
        nums.append(entr) # armazena
        if entr == '': # pare se nada mais for inserido
            return list(map(int,nums[:-1])) # converte para int e remove '' da␣
 ↪lista
```

[83]:
```python
# execução:
# 2; shift+ENTER; para 2
# 3; shift+ENTER; para 3
# 4; shift+ENTER; para 4
# shift+ENTER; para nada
```

```
nums = meu_gerador()
nums
```

```
2
3
4
```

[83]: `[2, 3, 4]`

Exemplo: verifique se a soma das probabilidades no `dict P` do experimento aleatório é realmente 100%.

[84]:
```
sum(P.values())
```

[84]: `100.0`

1.7 `map`

A função `map` serve para construir uma função que será aplicada a todos os elementos de uma sequencia. Seu uso é da seguinte forma:

```
map(funcao,sequencia)
```

No exemplo anterior, as entradas do usuário são armazenadas como `str`, isto é, '2', '3' e '4'. Para que elas sejam convertidas para `int`, nós executamos um *casting* em todos os elementos da sequencia usando `map`.

A interpretação é a seguinte: para todo `x` pertencente a `sequencia`, aplique `funcao(x)`. Porém, para se obter o resultado desejado, devemos ainda aplicar `list` sobre o `map`.

[85]:
```
nums = ['2','3','4']
nums
```

[85]: `['2', '3', '4']`

[86]:
```
m = map(int,nums) # aplica a função 'int' aos elementos de 'num'
m
```

[86]: `<map at 0x7fbabebbb550>`

Observe que a resposta de `map` não é *human-readable*. Para lermos o que queremos, fazemos:

[87]:
```
l = list(m) # aplica 'list' sobre 'map'
l
```

[87]: `[2, 3, 4]`

Podemos substituir `funcao` por uma função anônima. Assim, suponha que você quisesse enviezar os valores de entrada somando 1 a cada número. Poderíamos fazer isso como:

[88]:
```
list(map(lambda x: x**2,l)) # eleva elementos ao quadrado
```

[88]: `[4, 9, 16]`

1.8 `filter`

Podemos aplicar também como uma espécie de "filtro" para valores usando a função `filter`. No caso anterior, digamos que valores acima de 7 sejam inseridos erroneamente no gerador de números (lembre-se que no sistema sanguíneo ABO, consideramos um `dict` cujo valor das chaves é no máximo 7). Podemos, ainda assim, filtrar a lista para coletar apenas valores menores do que 7. Para tanto, definimos uma função `lambda` com este propósito.

[89]:
```
lista_erronea = [2,9,4,6,7,1,9,10,2,4,5,2,7,7,11,7,6]
lista_erronea
```

[89]:
```
[2, 9, 4, 6, 7, 1, 9, 10, 2, 4, 5, 2, 7, 7, 11, 7, 6]
```

[90]:
```
f = filter(lambda x: x <= 7, lista_erronea) # aplica filtro
f
```

[90]:
```
<filter at 0x7fbabebdf940>
```

[91]:
```
lista_corrigida = list(f) # valores > 7 excluídos
lista_corrigida
```

[91]:
```
[2, 4, 6, 7, 1, 2, 4, 5, 2, 7, 7, 7, 6]
```

1.9 Exemplos com maior complexidade

Exemplo: Podemos escrever outro gerador de forma mais complexa. Estude este caso (pouco Pythônico).

[92]:
```
import random

la = random.sample(range(0,1000),1000) # escolhe 1000 números numa lista␣
 ↪aleatória de 0 a 1000
teste = lambda x: -1 if x >= 8 else x # retorna x no intervalo [0,7], senão o␣
 ↪próprio número
f = list(map(teste,la))
final = list(filter(lambda x: x != -1,f)) # remove > 8
final
```

[92]:
```
[0, 7, 5, 1, 3, 2, 4, 6]
```

Exemplo: Associando arbitrariamente o identificador de uma pessoa a um tipo sanguíneo com compreensão de `dict`.

[93]:
```
id_pessoas = {chave:x for chave,x in enumerate(f) if x > -1} # compreensão de␣
 ↪dicionário com if
id_pessoas
```

[93]:
```
{123: 0, 312: 7, 355: 5, 460: 1, 584: 3, 747: 2, 778: 4, 844: 6}
```