Aula 3A - Matemática Discreta com Python: Parte 1

Gustavo Oliveira[1] e Andrea Rocha[1]

[1]Departamento de Computação Científica / UFPB

Junho de 2020

1 Fundamentos de Matemática Discreta com Python

1.1 Matemática Discreta

A matemática é uma ciência fundamental para qualquer formação ligada à computação, à informática e, em nosso caso, à ciência e análise de dados. Porém, a característica distintiva da matemática realizada pelas máquinas é a finitude. Isto é, enquanto a matemática tradicional que aprendemos é capaz de lidar com quantidades infinitas e incontáveis, as máquinas possuem limitações em suas capacidades, seja em armazenamento, seja em memória. Embora tais capacidades sejam expansíveis de alguma forma, devemos compreender que os recursos computacionais são finitos.

Chamamos de *Matemática Discreta* (ou *Álgebra Abstrata*) a área da Matemática que lida com objetos discretos, a saber, conjuntos, sequencias, listas, coleções ou quaisquer entidades *contáveis*. Por exemplo, diz-se que o conjunto dos números reais é incontável, ou não enumerável, pelo fato de não conseguirmos obter um paralelo entre seus elementos e o conjunto dos números naturais. Em outras palavras, não podemos determinar o número de elementos do conjunto $\mathbb{R}$. Por outro lado, isto não acontece com uma variedade de outros conjuntos que podemos encontrar na vida real. Observemos os exemplos:

- O conjunto das vogais da língua portuguesa;
- O conjunto dos times de futebol brasileiros da série A em 2020;
- O conjunto de nomes das estações do ano;
- O conjunto das personagens que formam o quarteto principal do filme *Os Pinguins de Madagascar* e;
- O conjunto dos números pares positivos menores ou iguais a dez.

Cada conjunto desses possui um número finito de elementos. Isto quer dizer que são contáveis, ou enumeráveis. Podemos defini-los, em linguagem matemática, por meio da *extensão*, quando listamos seus elementos, ou por meio da *compreensão*, quando usamos uma propriedade que distingue seus elementos. Ao longo do ensino básico, você já deparou com isto. Vamos apenas relembrar.

Reescritos por extensão, esses conjuntos, em ordem, são lidos como:

- $\{a, e, i, o, u\}$
- $\{$Atlético-PR, ..., Bahia, Botafogo, ..., Coritiba, ..., Fortaleza, ..., Internacional, ..., São Paulo, Sport, Vasco
- $\{$Primavera, Verão, Outono, Inverno$\}$

- $\{$Capitão, Kowalski, Recruta, Rico$\}$
- $\{2, 4, 6, 8, 10\}$

Já por compreensão, poderiam ser lidos como:

- $\{c \in \mathbb{A}\ ;\ c$ é vogal$\}$
- $\{t \in \mathbb{T}\ ;\ t$ é da Série A$\}$
- $\{x\ ;\ x$ é uma estação do ano$\}$
- $\{p\ ;\ p$ é personagem do quarteto principal do filme Os Pinguins de Madagascar$\}$
- $\{e\ ;\ e$ é estação do ano$\}$
- $\{n \in \mathbb{Z} \mid n = 2k \wedge 2 \leq n \leq 10 \wedge k \in \mathbb{Z}\}$

Por livre conveniência, chamamos de $\mathbb{A}$ o conjunto de todas as letras de nosso alfabeto e de $\mathbb{T}$ o conjunto de todos os times de futebol do Brasil. Adicionalmente, vale ressaltar que poderíamos usar diferentes formas de denotá-los por compreensão além dessas. Tal liberdade de escolha, desde que coerente, transmite exatamente o caráter abstrato que a Matemática Discreta possui.

1.2 Estruturas de dados em Python para lidar com objetos discretos

A linguagem oferece diversos objetos para operarmos com quantidades discretas em formas de sequencias, listas ou coleções. De forma genérica, você pode interpretá-las como "conjuntos" que contém zero ou mais elementos. Um conjunto com zero elementos é chamado de *vazio*. Em Python, também temos meios para representar o "vazio" também, como veremos adiante.

As principais estruturas que aprenderemos serão:

- `list`: estrutura cujo conteúdo é modificável e o tamanho variável. Listas são caracterizadas por *mutabilidade* e *variabilidade*. Objetos `list` são definidos por um par de colchetes e vírgulas que separam seus elementos: `[., ., ... ,.]`.
- `tuple`: estrutura cujo conteúdo não é modificável e o tamanho fixado. Tuplas são caracterizadas por *imutabilidade* e *invariabilidade*. Objetos `tuple` são definidos por um par de colchetes e vírgulas que separam seus elementos: `(., ., ... ,.)`.
- `dict`: estruturas contendo uma coleção de pares do tipo *chave-valor*. Dicionários são caracterizados por *arrays associativos* (*tabelas hash*). Objetos `dict` são definidos por um par de chaves e agrupamentos do tipo `'chave':valor` (*key:value*), separados por vírgula: `{'chave1':valor1, 'chave2':valor2, ... ,'chaven':valorn}`. As chaves (*keys*) são do tipo `str`, ao passo que os valores podem ser de tipos arbitrários.
- `set`: estruturas similares a `dict`, porém não possuem chaves e contêm objetos únicos. Conjuntos são caracterizadas por *unicidade* de elementos. Objetos `set` são definidos por um par de chaves e vírgulas que separam seus elementos: `{., ., ... ,.}`.

1.3 Listas

Estruturas `list` formam uma coleção de objetos arbitrários e podem ser criadas de modo sequenciado com operadores de pertencimento ou por expressões geradoras, visto que são estruturas iteráveis.

[1]:
```
vogais = ['a','e','i','o','u']
vogais
```

[1]:
```
['a', 'e', 'i', 'o', 'u']
```

[2]:
```
times = ['Bahia', 'Sport', 'Fortaleza', 'Flamengo']
times
```

[2]:
```
['Bahia', 'Sport', 'Fortaleza', 'Flamengo']
```

[3]:
```
pares10 = [2,4,6,8,10]
pares10
```

[3]:
```
[2, 4, 6, 8, 10]
```

[4]:
```
mix = ['Bahia',24,6.54,[1,2]] # vários objetos na lista
mix
```

[4]:
```
['Bahia', 24, 6.54, [1, 2]]
```

1.3.1 Listas por geração

Exemplo: crie uma lista dos primeiros 100 inteiros não-negativos.

[5]:
```
os_100 = range(100) # range é uma função geradora
print(list(os_100)) # casting com 'list'
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

Exemplo: crie o conjunto $\{x \in \mathbb{Z}\,;\, -20 \leq x < 10\}$

[6]:
```
print(list(range(-20,10))) # print é usado para imprimir column-wise
```

```
[-20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4,
-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Exemplo: crie o conjunto $\{x \in \mathbb{Z}\,;\, -20 \leq x \leq 10\}$

[7]:
```
print(list(range(-20,11))) # para incluir 10, 11 deve ser o limite. Por quê?
```

```
[-20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4,
-3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.4 Adicionando e removendo elementos

Há vários métodos aplicáveis para adicionar e remover elementos em listas.

1.4.1 Adição por apensamento

Adiciona elementos por concatenação no final da lista.

[8]:
```
times.append('Botafogo')
times
```

[8]:
```
['Bahia', 'Sport', 'Fortaleza', 'Flamengo', 'Botafogo']
```

[9]:
```
times.append('Fluminense')
times
```

[9]:
```
['Bahia', 'Sport', 'Fortaleza', 'Flamengo', 'Botafogo', 'Fluminense']
```

1.4.2 Adição por extensão

Para incluir elementos através de um objeto iterável, sequenciável, usamos `extend`.

[10]:
```
times.extend(['Vasco, Atlético-MG']) # usa outra lista pra estender a lista
times
```

[10]:
```
['Bahia',
 'Sport',
 'Fortaleza',
 'Flamengo',
 'Botafogo',
 'Fluminense',
 'Vasco, Atlético-MG']
```

Iteração e indexação A *iteração* sobre uma lista é o processo de "passear" por seus elementos de modo sequenciado. Ao fornecermos o *índice* (posição) de seus elementos, podemos indexá-los.

Em Python, a indexação de listas começa a partir de `0` e termina em `n - 1`, onde `n` é o tamanho da lista.

Por exemplo, analise a seguinte correspondência:
posição : $\{p = 0, p = 1, \ldots, p = n - 1\}$
elementos na lista : $[x_1, x_2, \ldots, x_n]$

Quer dizer, o primeiro elemento, x_1 está na posição 0, $p = 0$, ao passo que o último elemento, x_n, está na posição $n - 1$, $p = n - 1$. Logo, se escolhermos uma variável chamada p que assume o valor $0, 1, \ldots, n - 1$, mediante a posição (ordenada) do elemento na lista, diremos que p é um *iterador*, os inteiros de 0 a $n - 1$ são os *índices* e n é o *tamanho da lista*.

Esta mesma idéia é aplicável a qualquer coleção, sequencia ou objeto iterável.

1.4.3 Remoção por índice

Suponha que tivéssemos criado a lista:

[11]:
```
pares = [0,2,5,6] # 5 não é par
pares
```

[11]:
```
[0, 2, 5, 6]
```

Como 5 não é par, não deveria estar na lista. Para excluírmos um elemento em uma posição específica, usamos `pop` passando o *índice* onde o elemento está.

[12]:
```
pares.pop(2) # o ímpar 5 está na posição 2 e NÃO 3!
pares
```

[12]:
```
[0, 2, 6]
```

1.4.4 Adição por índice

Nesta lista, podemos pensar em incluir 4 entre 2 e 6. Para isto, usamos `insert(posicao,valor)`, para `valor` na `posicao` desejada.

[13]:
```
pares.insert(2,4) # 4 é inserido na posição de 6, que é deslocado
pares
```

[13]:
```
[0, 2, 4, 6]
```

1.4.5 Apagar conteúdo da lista

Podemos apagar o conteúdo inteiro da lista com `clear`.

[14]:
```
times.clear()
times # lista está vazia
```

[14]:
```
[]
```

Podemos contar o número de elementos da lista com `len`.

[15]:
```
len(times) # verifica que a lista está vazia
```

[15]:
```
0
```

[16]:
```
type([]) # a lista é vazia, mas continua sendo lista
```

[16]:
```
list
```

1.4.6 Outros métodos de lista

Conte repetições de elementos na lista com `count`.

[17]:
```
numeros = [1,1,2,3,1,2,4,5,6,3,4,4,5,5]
print( numeros.count(1), numeros.count(3), numeros.count(7) )
```

```
3 2 0
```

Localize a posição de um elemento com `index`.

[18]:
```
numeros.index(5) # retorna a posição da primeira aparição
```

[18]:
```
7
```

Remova a primeira aparição do elemento com `remove`.

[19]:
```
numeros.remove(1) # perde apenas o primeiro
numeros
```

[19]:
```
[1, 2, 3, 1, 2, 4, 5, 6, 3, 4, 4, 5, 5]
```

Faça uma reflexão ("flip") *in-place* (sem criar nova lista) da lista com `reverse`.

[20]:
```
numeros.reverse()
numeros
```

[20]:
```
[5, 5, 4, 4, 3, 6, 5, 4, 2, 1, 3, 2, 1]
```

Ordene a lista de maneira *in-place* (sem criar nova lista) com `sort`.

[21]:
```
numeros.sort()
numeros
```

[21]:
```
[1, 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6]
```

1.5 Concatenação de listas

Listas são concatenadas ("somadas") com +. Caso já possua listas definidas, use `extend`.

[22]:
```
['Flamengo', 'Botafogo'] + ['Fluminense']
```

[22]:
```
['Flamengo', 'Botafogo', 'Fluminense']
```

[23]:
```
['Flamengo', 'Botafogo'] + 'Fluminense' # erro: 'Fluminense' não é list
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-23-3b000c9317b9> in <module>
----> 1 ['Flamengo', 'Botafogo'] + 'Fluminense' # erro: 'Fluminense' não é list


TypeError: can only concatenate list (not "str") to list
```

[24]:
```
times_nordeste = ['Fortaleza','Sport']
times_sul = ['Coritiba','Atlético-PR']
times_nordeste + times_sul
```

[24]:
```
['Fortaleza', 'Sport', 'Coritiba', 'Atlético-PR']
```

[25]:
```
times_nordeste.extend(times_sul) # mesma coisa
times_nordeste
```

[25]:
```
['Fortaleza', 'Sport', 'Coritiba', 'Atlético-PR']
```

1.6 Fatiamento de listas

O fatiamento ("slicing") permite que selecionemos partes da lista através do modelo `start:stop`, em que `start` é um índice incluído na iteração, e `stop` não.

[26]:
```
letras = ['a','b','c','d','e','f','g']
letras[0:2]
```

[26]:
```
['a', 'b']
```

[27]:
```
letras[1:4]
```

[27]:
```
['b', 'c', 'd']
```

[28]:
```
letras[5:6]
```

[28]:
```
['f']
```

[29]:
```
letras[0:7] # toda a lista
```

[29]:
```
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

1.6.1 Omissão de `start` e `stop`

[30]:
```
letras[:3] # até 3, exclusive
```

[30]:
```
['a', 'b', 'c']
```

[31]:
```
letras[:5] # até 5, exclusive
```

[31]:
```
['a', 'b', 'c', 'd', 'e']
```

[32]:
```
letras[4:] # de 4 em diante
```

[32]:
```
['e', 'f', 'g']
```

[33]:
```
letras[6:] # de 6 em diante
```

[33]:
```
['g']
```

1.6.2 Modo reverso

[34]:
```
letras[-1] # último índice
```

[34]:
```
'g'
```

[35]:
```
letras[-2:-1] # do penúltimo ao último, exclusive
```

[35]:
```
['f']
```

[36]:
```
letras[-3:-1]
```

[36]:
```
['e', 'f']
```

[37]:
```
letras[-4:-2]
```

[37]:
```
['d', 'e']
```

[38]:
```
letras[-7:-1] # toda a lista
```

[38]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

[39]:
```
letras[-5:]
```

[39]:
```
['c', 'd', 'e', 'f', 'g']
```

[40]:
```
letras[:-3]
```

[40]:
```
['a', 'b', 'c', 'd']
```

1.7 Elementos alternados com `step`

Podemos usar um dois pontos duplo (`::`) para dar um "passo" de alternância.

[41]:
```
letras[::2] # salta 2-1 intermediários
```

[41]:
```
['a', 'c', 'e', 'g']
```

[42]:
```
letras[::3] # salta 3-1 intermediários
```

[42]:
```
['a', 'd', 'g']
```

[43]:
```
letras[::7] # salto de igual tamanho
```

[43]:
```
['a']
```

[44]:
```
letras[::8] # salto além do tamanho
```

[44]:
```
['a']
```

1.8 Mutabilidade de listas

Podemos alterar o conteúdo de elementos diretamente por indexação.

[45]:
```
from sympy.abc import x,y

ops = [x+y,x-y,x*y,x/y]
ops2 = ops.copy() # cópia de ops
ops
```

[45]:
```
[x + y, x - y, x*y, x/y]
```

[46]:
```
ops[0] = x-y
ops
```

[46]:
```
[x - y, x - y, x*y, x/y]
```

[47]:
```
ops[2] = x/y
ops
```

[47]:
```
[x - y, x - y, x/y, x/y]
```

[48]:
```
ops[1], ops[3] = x + y, x*y # mutação por desempacotamento
ops
```

[48]:
```
[x - y, x + y, x/y, x*y]
```

[49]:
```
ops[1:3] = [False, False, True] # mutação por fatiamento
ops
```

[49]:
```
[x - y, False, False, True, x*y]
```

[50]:
```
ops = ops2 # recuperando ops
ops
```

[50]:
```
[x + y, x - y, x*y, x/y]
```

[51]:
```
ops2 is ops
```

[51]:
```
True
```

[52]:
```
ops3 = [] # lista vazia
ops3
```

[52]:
```
[]
```

[53]:
```
ops2 = ops + ops3 # concatenação cria uma lista nova
ops2
```

[53]:
```
[x + y, x - y, x*y, x/y]
```

[54]:
```
ops2 is ops # agora, ops2 não é ops
```

[54]:
```
False
```

[55]:
```
print(id(ops), id(ops2)) # imprime local na memória de ambas
```

```
140530916025224 140530917641608
```

[56]:
```
ops2 == ops # todos os elementos são iguais
```

[56]:
```
True
```

O teste de identidade é `False`, mas o teste de igualdade é `True`.

Exemplo: Escreva uma função que calcule a área, perímetro, comprimento da diagonal, raio, perímetro e área do círculo inscrito, e armazene os resultados em uma lista.

[57]:
```
# usaremos matemática simbólica
from sympy import symbols
from math import pi

# símbolos
B, H = symbols('B H',positive=True)

def propriedades_retangulo(B,H):
    '''
        A função assume que a base B
        é maior do que a altura H. Senão,
        as propriedades do círculo inscrito
        não serão determinadas.
    '''
    d = (B**2 + H**2)**(1/2) # comprimento da diagonal
    r = d/2 # raio do círculo inscrito
    return [B*H, 2*(B+H), d, d/2, 2*pi*r, pi*(r)**2]

# lista de objetos símbolos
propriedades_retangulo(B,H)
```

[57]:
```
[B*H,
 2*B + 2*H,
 (B**2 + H**2)**0.5,
 (B**2 + H**2)**0.5/2,
 3.14159265358979*(B**2 + H**2)**0.5,
 0.785398163397448*(B**2 + H**2)**1.0]
```

[58]:
```
# substituindo valores
B, H = 4.0, 2.5

# desempacotando
propriedades_retangulo(B,H)
```

[58]:
```
[10.0,
 13.0,
 4.716990566028302,
 2.358495283014151,
 14.818862909286873,
 17.47510913559322]
```

1.8.1 Formatação de strings

Os valores na lista acima poderiam ser impressos de uma maneira mais legível. Até o momento, estivemos habituados em imprimir valores passando-s à função `print`. Entretanto, a Python nos oferece uma ampla gama de recursos para formatar strings. Veremos mais detalhes sobre *templating* e formatação de strings mais à frente no curso. Por enquanto, vamos ver como podemos imprimir melhor os `float` anteriores.

O *template* a seguir usa a função `format` para substituição de valores indexados.

```
templ = '{0} {1} ... {n}'.format(arg0,arg1,...,argn)
```

Nota: Para ajuda plena sobre formatação, consultar:

```
help('FORMATTING')
```

[59]:
```
# considere R: retângulo; C: círculo inscrito

res = propriedades_retangulo(B,H) # resultado

props = ['Área de R',
         'Perímetro de R',
         'Diagonal de R',
         'Raio de C',
         'Perímetro de C',
         'Área de C'
        ] # propriedades

# template
templ = '{0:s} = {1:.2f}\n\
{2:s} = {3:.3f}\n\
{4:s} = {5:.4f}\n\
{6:s} = {7:.5f}\n\
{8:s} = {9:.6f}\n\
{10:s} = {11:.7f}'.format(props[0],res[0],\
                         props[1],res[1],\
```

```
                          props[2],res[2],\
                          props[3],res[3],\
                          props[4],res[4],\
                          props[5],res[5])

# impressão formatada
print(templ)
```

```
Área de R = 10.00
Perímetro de R = 13.000
Diagonal de R = 4.7170
Raio de C = 2.35850
Perímetro de C = 14.818863
Área de C = 17.4751091
```

1.8.2 Como interpretar o que fizemos?

- `{0:s}` formata o primeiro argumento de `format`, o qual é `props[0]`, como `str` (`s`).
- `{1:.2f}` formata o segundo argumento de `format`, o qual é `res[0]`, como `float` (`f`) com duas casas decimais (`.2`).
- `{3:.3f}` formata o quarto argumento de `format`, o qual é `res[1]`, como `float` (`f`) com três casas decimais (`.3`).

A partir daí, percebe-se que um template `{X:.Yf}` diz para formatar o argumento `X` como `float` com `Y` casas decimais, ao passo que o template `{X:s}` diz para formatar o argumento `X` como `str`.

Além disso, temos:

- `\n`, que significa "newline", isto é, uma quebra da linha.
- `\`, que é um *caracter de escape* para continuidade da instrução na linha seguinte. No exemplo em tela, o *template* criado é do tipo *multi-line*.

Nota: a contrabarra em `\n` também é um caracter de escape e não um caracter *literal*. Isto é, para imprimir uma contrabarra literalmente, é necessário fazer `\\`. Vejamos exemplos de literais a seguir.

Exemplos de impressão de caracteres literais

[60]:
```
print('\\') # imprime contrabarra literal
print('\\\\') # imprime duas contrabarras literais
print('\'') # imprime plica
print('\"') # imprime aspas
```

```
\
\\
'
"
```

f-strings Temos uma maneira bastante interessante de criar templates usando f-strings, que foi introduzida a partir da versão Python 3.6. Com f-strings a substituição é imediata.

[61]:
```
print(f'{props[0]} = {res[0]}') # estilo f-string
```

```
Área de R = 10.0
```

Estilos de formatação Veja um comparativo de estilos:

[62]:
```
print('%s = %f ' % (props[0], res[0])) # Python 2
print('{} = {}'.format(props[0], res[0])) # Python 3
print('{0:s} = {1:.4f}'.format(props[0], res[0])) # Python 3 formatado
```

```
Área de R = 10.000000
Área de R = 10.0
Área de R = 10.0000
```

Exemplo: Considere o conjunto: $\mathrm{V} = \{c \in \mathbb{A}\,;\, c \text{ é vogal}\}$. Crie a concatenação de todos os elementos com f-string.

[63]:
```
V = ['a','e','i','o','u']
V
```

[63]:
```
['a', 'e', 'i', 'o', 'u']
```

[64]:
```
f'{V[0]}{V[1]}{V[2]}{V[3]}{V[4]}' # pouco Pythônico
```

[64]:
```
'aeiou'
```

Veremos à frente meios mais elegantes de fazer coisas similares.

1.9 Controle de fluxo: laço `for`

Em Python, podemos realizar iterar por uma coleção ou iterador usando *laços*. Introduziremos aqui o laço `for`. Em Python, o bloco padrão para este laço é dado por:

```
for valor in sequencia:
    # faça algo com valor
```

Acima, `valor` é um iterador.

[65]:
```
for v in vogais: # itera sobre lista inteira
    print(v)
```

```
a
e
i
o
u
```

[66]:
```
for v in vogais[0:3]: # itera parcialmente
    print(v + 'a')
```

```
aa
ea
ia
```

[67]:
```
for v in vogais[-2:]:
    print(f'{v*10}')
```

```
oooooooooo
uuuuuuuuuu
```

1.10 Compreensão de lista

Usando `for`, a criação de listas torna-se bastante facilitada.

Exemplo: crie a lista dos primeiros 10 quadrados perfeitos.

[68]:
```
Q = [q*q for q in range(1,11)]
Q
```

[68]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A operação acima equivale a:

[69]:
```
Q2 = []
for q in range(1,11):
    Q2.append(q*q)
Q2
```

[69]:
```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Exemplo: crie a PA: $a_n = 3 + 6(n-1)$, $1 \leq n \leq 10$

[70]:
```
PA = [3 + 6*(n-1) for n in range(1,11) ]
PA
```

[70]:
```
[3, 9, 15, 21, 27, 33, 39, 45, 51, 57]
```

Exemplo: se $X = \{1,2,3\}$ e $Y = \{4,5,6\}$, cria a "soma" $X + Y$ elemento a elemento.

[71]:
```
X = [1,2,3]
Y = [4,5,6]

XsY = [ X[i] + Y[i] for i in range(len(X)) ]
XsY
```

[71]:
```
[5, 7, 9]
```

Exemplo: se $X = \{1,2,3\}$ e $Y = \{4,5,6\}$, cria o "produto" $X * Y$ elemento a elemento.

[72]:
```
XpY = [ X[i]*Y[i] for i in range(len(X)) ]
XpY
```

[72]:
```
[4, 10, 18]
```

1.11 Tuplas

Tuplas são são sequencias imutáveis de tamanho fixo. Em Matemática, uma tupla é uma sequência ordenada de elementos. Em geral, o termo $n-$upla ("ênupla") é usado para se referir a uma tupla com n elementos.

Por exemplo, tuplas de um único elemento são chamadas de “singleton” ou “mônada”. Tuplas de dois elementos são os conhecidos “pares ordenados”. Com três elementos, chamamos de “trio” ou “tripleta”, e assim por diante.

Em Python, tuplas são criadas naturalmente sequenciando elementos.

[73]:
```
par = 1,2; par
```

[73]:
```
(1, 2)
```

[74]:
```
trio = (1,2,3); trio
```

[74]:
```
(1, 2, 3)
```

[75]:
```
quad = (1,2,3,4); quad
```

[75]:
```
(1, 2, 3, 4)
```

[76]:
```
nome = 'Nome'; tuple(nome) # casting
```

[76]:
```
('N', 'o', 'm', 'e')
```

Tuplas são acessíveis por indexação.

[77]:
```
quad[1]
```

[77]:
```
2
```

[78]:
```
quad[1:4]
```

[78]:
```
(2, 3, 4)
```

[79]:
```
quad[3] = 5 # tuplas não são mutáveis
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-79-59bcf52d23f3> in <module>
----> 1 quad[3] = 5 # tuplas não são mutáveis


TypeError: 'tuple' object does not support item assignment
```

Se na tupla houver uma lista, a lista é modificável.

[80]:
```
super_trio = tuple([1,[2,3],4]) # casting
super_trio
```

[80]:
```
(1, [2, 3], 4)
```

[81]:
```
super_trio[1].extend([4,5])
super_trio
```

[81]:
```
(1, [2, 3, 4, 5], 4)
```

Tuplas também são concatenáveis com +.

[82]:
```
(2,3) + (4,3)
```

[82]:
```
(2, 3, 4, 3)
```

[83]:
```
('a',[1,2],(1,1))*2 # repetição
```

[83]:
```
('a', [1, 2], (1, 1), 'a', [1, 2], (1, 1))
```

1.11.1 Desempacotamento de tuplas

[84]:
```
a,b,c,d = (1,2,3,4)
```

[85]:
```
for i in [a,b,c,d]:
    print(i) # valor das variáveis
```

```
1
2
3
4
```

[86]:
```
a,b = (1,2)
a,b = b,a # troca de valores
a,b
```

[86]:
```
(2, 1)
```

1.11.2 `enumerate`

Podemos controlar índice e valor ao iterar em uma sequencia.

[87]:
```
for i,x in enumerate(X): # (i,x) é uma tupla (índice,valor)
    print(f'{i} : {x}')
```

```
0 : 1
1 : 2
2 : 3
```

Exemplo: Construa o produto cartesiano

$$A \times B = \{(a,b) \in \mathbb{Z} \times \mathbb{Z};\ -4 \leq a \leq 4 \wedge 3 \leq a \leq 7\}$$

[88]:
```
AB = ([(a,b) for a in range(-4,4) for b in range(3,7)])
print(AB)
```

```
[(-4, 3), (-4, 4), (-4, 5), (-4, 6), (-3, 3), (-3, 4), (-3, 5), (-3, 6), (-2,
3), (-2, 4), (-2, 5), (-2, 6), (-1, 3), (-1, 4), (-1, 5), (-1, 6), (0, 3), (0,
4), (0, 5), (0, 6), (1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (2, 4), (2, 5), (2,
6), (3, 3), (3, 4), (3, 5), (3, 6)]
```

1.12 Dicionários

Dicionários, ou especificamente, objetos `dict`, possuem extrema versatilidade e são muito poderosos. Criamos um `dict` por diversas formas. A mais simples é usar chaves e pares explícitos.

[89]:
```
d = {} # dict vazio
d
```

[89]:
```
{}
```

[90]:
```
type(d)
```

[90]:
```
dict
```

Os pares chave-valor incorporam quaisquer tipos de dados.

[91]:
```
d = {'par': [0,2,4,6,8], 'ímpar': [1,3,5,7,9], 'nome':'Meu dict', 'teste': True}
d
```

[91]:
```
{'par': [0, 2, 4, 6, 8],
 'ímpar': [1, 3, 5, 7, 9],
 'nome': 'Meu dict',
 'teste': True}
```

1.12.1 Acesso a conteúdo

Para acessar o conteúdo de uma chave, indexamos pelo seu nome.

[92]:
```
d['par']
```

[92]:
```
[0, 2, 4, 6, 8]
```

[93]:
```
d['nome']
```

[93]:
```
'Meu dict'
```

Exemplo: construindo soma e multiplicação especial.

[94]:
```
# dict
op = {'X' :[1,2,3], 'delta' : 0.1}

# função
def sp(op):
    s = [x + op['delta'] for x in op['X']]
    p = [x * op['delta'] for x in op['X']]

    return (s,p) # retorna tupla

soma, prod = sp(op) # desempacota

for i,s in enumerate(soma):
    print(f'pos({i}) | Soma = {s} | Prod = {prod[i]}')
```

```
pos(0) | Soma = 1.1 | Prod = 0.1
pos(1) | Soma = 2.1 | Prod = 0.2
```

```
pos(2) | Soma = 3.1 | Prod = 0.30000000000000004
```

1.12.2 Inserção de conteúdo

[95]:
```
# apensa variáveis
op[1] = 3
op['novo'] = (3,4,1)
op
```

[95]:
```
{'X': [1, 2, 3], 'delta': 0.1, 1: 3, 'novo': (3, 4, 1)}
```

1.12.3 Alteração de conteúdo

[96]:
```
op['novo'] = [2,1,4] # sobrescreve
op
```

[96]:
```
{'X': [1, 2, 3], 'delta': 0.1, 1: 3, 'novo': [2, 1, 4]}
```

1.12.4 Deleção de conteúdo com `del` e `pop`

[97]:
```
del op[1] # deleta chave
op
```

[97]:
```
{'X': [1, 2, 3], 'delta': 0.1, 'novo': [2, 1, 4]}
```

[98]:
```
novo = op.pop('novo') # retorna e simultaneamente deleta
novo
```

[98]:
```
[2, 1, 4]
```

[99]:
```
op
```

[99]:
```
{'X': [1, 2, 3], 'delta': 0.1}
```

1.12.5 Listagem de chaves e valores

Usamos os métodos `keys()` e `values()` para listar chaves e valores.

[100]:
```
arit = {'soma': '+', 'subtr': '-', 'mult': '*', 'div': '/'} # dict

k = list(arit.keys())
print(k)
val = list(arit.values())
print(val)
for v in range(len(arit)):
    print(f'A operação \'{k[v]}\' de "arit" usa o símbolo \'{val[v]}\'.')
```

```
['soma', 'subtr', 'mult', 'div']
['+', '-', '*', '/']
A operação 'soma' de "arit" usa o símbolo '+'.
```

```
A operação 'subtr' de "arit" usa o símbolo '-'.
A operação 'mult' de "arit" usa o símbolo '*'.
A operação 'div' de "arit" usa o símbolo '/'.
```

1.12.6 Combinando dicionários

Usamos `update` para combinar dicionários. Este método possui um resultado similar a `extend`, usado em listas.

[101]:
```
pot = {'pot': '**'}
arit.update(pot)
arit
```

[101]:
```
{'soma': '+', 'subtr': '-', 'mult': '*', 'div': '/', 'pot': '**'}
```

1.12.7 Dicionários a partir de sequencias

Podemos criar dicionários a partir de sequencias existentes usando `zip`.

[102]:
```
arit = {'soma', 'subtr', 'mult', 'div', 'pot'}
ops = {'+', '-', '*', '/', '**'}

dict_novo = {}

for chave,valor in zip(arit,ops):
    dict_novo[chave] = valor

dict_novo
```

[102]:
```
{'pot': '-', 'mult': '+', 'div': '/', 'soma': '*', 'subtr': '**'}
```

Visto que um `dict` é composto de várias tuplas de 2, podemos criar um de maneira ainda mais simples.

[103]:
```
dict_novo = dict(zip(arit,ops)) # visto que dicts
dict_novo
```

[103]:
```
{'pot': '-', 'mult': '+', 'div': '/', 'soma': '*', 'subtr': '**'}
```