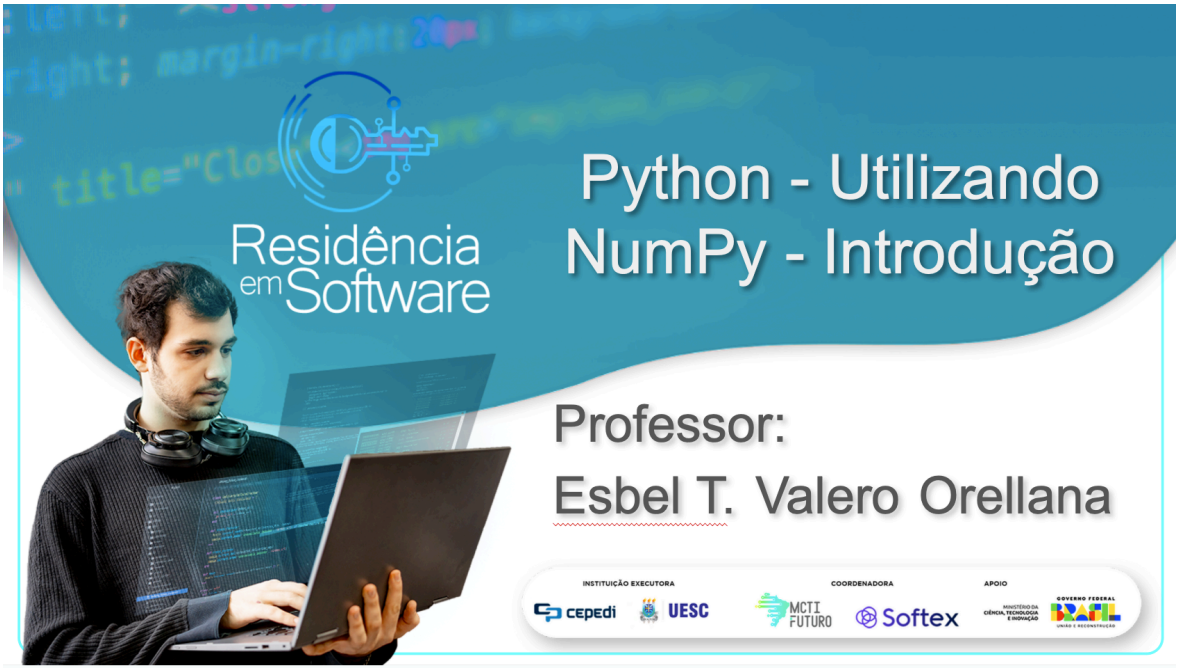
Módulo de Programação Python

Trilha Python - Aula 8: Utilizando NumPy - Introdução



Objetivo: Trabalhar com pacotes e módulos disponíveis em python: **Numpy**. Apresentar as limitações dos tipos de dados básicos em python e a alternativa oferecida pelo **NumPy**.

NumPy: Uma rápida introdução

A implementação de modelos computacionais eficientes passa pela escolha de estratégias apropriadas para representar e processar os dados de forma apropriada.

Por este motivo, técnicas eficientes para carregar, armazenar e manipular dados na memória são de grande importância em análise, processamento de dados e computação científica.

Levando em consideração que os dados podem vir em uma grande quantidade de tipos e formatos, o primeiro passo para transformar eles em uma estrutura que possa ser analisada, é construir arranjos de números de valores.

Por este motivo a manipulação e o armazenamento eficiente de arranjos numéricos é uma ferramenta indispensável em toda linguagem de programação.

Nesta aula começaremos a apresentar o módulo **NumPy**, um recurso muito poderoso disponível em **Python** para trabalhar com arranjos numéricos.

Mais documentação sobre o módulo no site do projeto NumPy (http://www.numpy.org)

Sobre tipos em Python

Já foi discutido antes o fato de **Python** ser uma linguagem dinamicamente tipada e suas implicações práticas.

Uma variável em **Python** não é apenas um bloco de memória para guardar um valor de tipo predeterminado, mas um ponteiro para um objeto utilizado para armazenar um tipo de dado específico. Desta forma o armazenamento de informação em variáveis acarreta uma série de informações extras que podem sobrecarregar o processo, quando comparado a linguagens estaticamente tipadas como **C/C++**.

Esta sobrecarga fica mais evidente quando se trata de armazenar e processar um conjunto ou arranjo de dados. Vejamos, por exemplo, as listas em **Python**.

Para os que já trabalharam com arrays em outras linguagens fica evidente que as listas, com toda sua flexibilidade e dinamismo, são estruturas pouco eficientes desde o ponto de vista de uso da memória.

A pesar de que podem ser acessadas utilizando índice, da mesma forma que os arrays, as listas não seriam uma escolha apropriada para se tratar de problemas que envolvem estruturas como vetores e matrizes.

Vejamos o seguinte exemplo:

Multiplicação de Matrizes

Um problema comum na área de IA e redes neurais é a multiplicação de matrizes. Esta operação pode aparecer, por exemplo, no processo de treinamento de modelos baseados no uso de Redes Neurais. Veja uma explicação de uma implementação avançada em **Python** do algoritmos de multiplicação geral de matrizes (GEMM - General amatrix Multiplicaction) aqui (https://spatial-lang.org/gemm).

Como representar uma matriz valores de ponto flutuante em **Python**? A abordagem mais simples e direta é utilizando listas de listas.

In [1]:
```
matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matC = [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0., 0., 0.0]]

def GEMM(alpha, A, B, beta, C):
    for i in range(len(A)):
        for j in range(len(B[0])):
            C[i][j] *= beta
            for k in range(len(A[0])):
                C[i][j] += alpha * A[i][k] * B[k][j]
    return C

matC = GEMM(1.0, matA, matB, 0.0, matC)
print(matC)
```

```
[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
```

Vamos testar esta implementação utilizando `%timeit` e matrizes de maior porte.

In [2]:
```
from random import random
matSize = 512
matA = [[random() for i in range(matSize)] for j in range(matSize)]
matB = [[random() for i in range(matSize)] for j in range(matSize)]
matC = [[random() for i in range(matSize)] for j in range(matSize)]
print(len(matA), len(matA[0]))
```

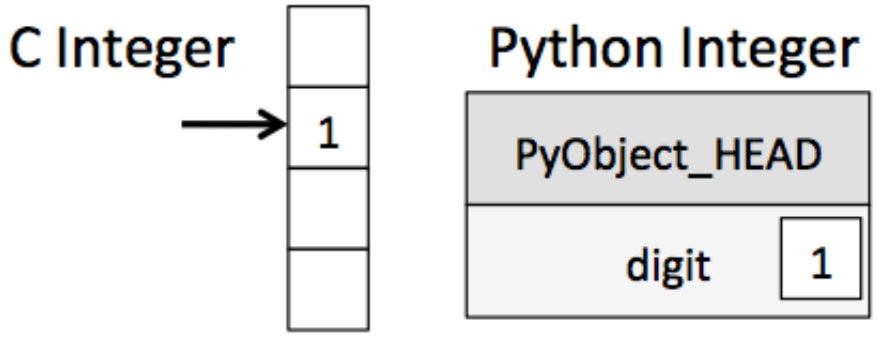
```
512 512
```

In [3]:
```
%timeit GEMM(2.0, matA, matB, 0.5, matC)
```

```
18.9 s ± 81.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop eac
h)
```

O que acontece quando usamos uma estrutura de dados **Python** que contém diversos objetos Python. O contêiner, com suporte a múltiplos tipos, mutável padrão em Python é a lista. Podemos criar uma lista de `float` da como vimos no exemplo anterior. Devido à tipagem dinâmica do Python, as listas podem ser heterogêneas, incluindo objetos de tipos diferentes.

Mas esta flexibilidade tem um custo: para permitir estes tipos flexíveis, cada item da lista deve conter o seu próprio tipo, contagem de referências e outras informações. Ou seja, cada item é um objeto Python completo. No caso especial de todas as variáveis serem do mesmo tipo, muitas dessas informações são redundantes, portanto pode ser muito mais eficiente armazenar os dados em um array de tipo fixo. A diferença entre uma lista de tipo dinâmico e uma matriz de tipo fixo, como os arrays em **C/C++** é ilustrada na figura a seguir:



No nível de implementação, o array contém essencialmente um único ponteiro para um bloco contíguo de dados.

A lista Python, por outro lado, contém um ponteiro para um bloco de ponteiros, cada um dos quais, por sua vez, aponta para um objeto Python completo, como o inteiro Python que vimos anteriormente.

A vantagem da lista é a flexibilidade: como cada elemento da lista é uma estrutura completa contendo dados e informações de tipo, a lista pode ser preenchida com dados de qualquer tipo desejado.

Matrizes de estilo **C/C++**, de tipo fixo, não possuem essa flexibilidade, mas são muito mais eficientes para armazenar e manipular dados.

Arrays de tipo fixo

Para resolver as limitações das listas, no que se refere a armazenamento de grandes volumes de informação de um tipo específico, **Python** oferece algumas alternativas. Veja por exemplo o módulo array utilizado no seguinte exemplo:

In [4]:
```
import array
L = list(range(10))
x_1 = array.array('i', L)
print(x_1)
y_1 = array.array('d', L)
print(type(x_1))
#help(array.array)
```

```
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
<class 'array.array'>
```

Entretanto o recurso mais utilizado com esta finalidade são os objetos de tipo *ndarray* (n-dimensional array), implementados no módulo **NumPy**. Este módulo disponibiliza não apenas um estrutura para armazenamento eficiente mas um conjunto de operações e funções para manipular as mesmas. Vamos fazer uma revisão dos recursos disponíveis começando por como criar um *ndarray* a partir de uma lista.

In [5]:
```
import numpy as np
print("NumPy: ", np.__version__)
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
NumPy:  1.26.2
```

In [6]:
```
x_2 = np.array(L)
print(x_2)
print(type(x_2))
```

```
[0 1 2 3 4 5 6 7 8 9]
<class 'numpy.ndarray'>
```

Ao contrário das listas onde cada item pode ter um tipo específico diferente dos outros itens ...

In [7]:
```
print("type(L): ", type(L))
print("type(L[0]): ", type(L[0]))
```

```
type(L):  <class 'list'>
type(L[0]):  <class 'int'>
```

... os arrays são criados utilizando um único tipo.

In [8]:
```
print("x_2.dtype: ", x_2.dtype)
```

```
x_2.dtype:  int64
```

Quando encontrados tipos diversos na lista utilizada para criar a array, o construtor faz um casting automático para o tipo mais abrangente.

In [9]:
```
L = [1, 2.0, 3, 4.0, 5, 6.0, 7]
x_3 = np.array(L)
print("x_3.dtype: ", x_3.dtype)
```

```
x_3.dtype:  float64
```

O construtor da classe *ndarray* cria então o objeto a partir dos objetos que encontra na lista. Entretanto o tipo do pode ser especificado no construtor.

In [10]:
```
print("x_3.dtype: ", x_3.dtype)
x_4 = np.array(L, 'float32')
print("x_4.dtype: ", x_4.dtype)
```

```
x_3.dtype:  float64
x_4.dtype:  float32
```

Dado que as matrizes **NumPy** armazenam valores de um único tipo, é importante conhecer os tipos que estão disponíveis. Como **NumPy** foi desenvolvido em **C/C++**, estes tipos de dados podem parecer familiares para desenvolvedores de linguagens imperativas tradicionais.

Tipo	Descrição
`bool_`	Booleano (True ou False) armazenado como um byte
`int_`	Tipo inteiro padrão (igual a C `long` ; normalmente `int64` ou `int32`)
`intc`	Idêntico a C `int` (normalmente `int32` ou `int64`)
`intp`	Inteiro usado para indexação (igual a C `ssize_t` ; normalmente `int32` ou `int64`)
`int8`	Bytes (–128 a 127)
`int16`	Inteiro (–32768 to 32767)
`int32`	Inteiro (–2147483648 to 2147483647)
`int64`	Inteiro (–9223372036854775808 to 9223372036854775807)
`uint8`	Inteiro sem sinal (0 to 255)
`uint16`	Inteiro sem sinal (0 to 65535)
`uint32`	Inteiro sem sinal (0 to 4294967295)
`uint64`	Inteiro sem sinal (0 to 18446744073709551615)
`float_`	Abreviação para `float64`
`float16`	Float de meia precisão: bit de sinal, expoente de 5 bits, mantissa de 10 bits
`float32`	Float de precisão simples: bit de sinal, expoente de 8 bits, mantissa de 23 bits
`float64`	Float de precisão dupla: bit de sinal, expoente de 11 bits, mantissa de 52 bits
`complex_`	Abreviação para `complex128`
`complex64`	Número complexo, representado por dois floats de 32-bit
`complex128`	Número complexo, representado por dois floats de 64-bit

Observe que ao construir um array, eles podem ser especificados usando uma string como mostrado nos exemplos anteriores.

Por outro lado, quando o construtor encontra uma lista de listas, cria um *ndarray* multidimensional.

In [11]:
```
matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matA = np.array(matA)
matB = np.array(matB, 'float64')
print(matA)
print(matB)
print("matA.dtype: ", matA.dtype)
print("matB.dtype: ", matB.dtype)
```

```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
matA.dtype:  float64
matB.dtype:  float64
```

In [12]:
```
matA = [[random() for i in range(matSize)] for j in range(matSize)]
matB = [[random() for i in range(matSize)] for j in range(matSize)]
matA = np.array(matA)
matB = np.array(matB, 'float64')
matC = np.array([[random() for i in range(1024)] for j in range(102
print(matA.dtype, matB.dtype, matC.dtype)
```

```
float64 float64 float64
```

Criando um ndarray de zero

Até aqui utilizamos listas para criar arrays. Entretanto, **NumPy** permite também criar *ndarrays* de diversas outras formas. Veja os exemplos a seguir.

Em alguns casos, por exemplo, se faz necessário criar um arrays de contadores. Os contadores precisam ser inicializados com zero. Neste caso podemos utilizar o construtor `zeros` .

In [13]:
```
cont = np.zeros(10, dtype='int32')
print(cont)
print(type(cont ))
print(cont.dtype)
```

```
[0 0 0 0 0 0 0 0 0 0]
<class 'numpy.ndarray'>
int32
```

Na realidade o construtor espera, como parâmetro de entrada, uma tupla que especifique as dimensões do *ndarray*. No exemplo anterior foi passado apenas um inteiro, mas podemos criar então uma matriz de contadores de inteiros de 16 bits.

In [14]:
```
# Criando um ndarray de 10 inteiros preenchido com zero
cont2D = np.zeros((2,5), dtype='int16')
print(cont2D)
```

```
[[0 0 0 0 0]
 [0 0 0 0 0]]
```

Se não for especificado o tipo da, por padrão, o *ndarray* é criado como de `float64` .

In [15]:
```
matC = np.zeros((3, 3))
print(matC)
print(type(matC))
print(matC.dtype)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
<class 'numpy.ndarray'>
float64
```

Agora, se já temos um *ndarray* e se faz necessário criar outro com as mesmas dimensões mas todo preenchido com zeros? Neste caso é posível utilizar a função `zeros_like` .

In [16]:
```
matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matA = np.array(matA)
matB = np.array(matB, 'float64')
matC = np.zeros_like(matA)
print("matA: \n", matA)
print("matB: \n", matB)
print("matC: \n", matC)
print("matA.dtype: ", matA.dtype)
print("matB.dtype: ", matB.dtype)
print("matC.dtype: ", matC.dtype)
```

```
matA:
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
matB:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
matC:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
matA.dtype:  float64
matB.dtype:  float64
matC.dtype:  float64
```

Da mesma forma que `zeros` , o `ones` permite criar *ndarrays* mas agora preenchidos com uns. Se precisarmos criar uma arrys de multiplicadores, que precisam ser inicializados com uns teremos então:

In [17]:
```
mult = np.ones((10), dtype='int32')
print(mult)
print(type(mult))
```

```
[1 1 1 1 1 1 1 1 1 1]
<class 'numpy.ndarray'>
```

Novamente a dimensão do *ndarray* é passada na forma de uma tuplae o tipo padrão é `float64__` .

In [18]:
```
matA = np.ones((3, 3))
print(matA)
print(type(matA))
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
<class 'numpy.ndarray'>
```

Da mesma forma temos a versão `ones_like` que cria um *ndarray* a partir das dimensões o o tipo de um *ndarray* que já existe.

In [19]:
```
matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matA = np.array(matA)
matB = np.array(matB, 'float64')
matC = np.zeros_like(matA)
matU = np.ones_like(matA)
print("matA: \n", matA)
print("matB: \n", matB)
print("matC: \n", matC)
print("matU: \n", matU)
print("matA.dtype: ", matA.dtype)
print("matB.dtype: ", matB.dtype)
print("matC.dtype: ", matC.dtype)
print("matU.dtype: ", matU.dtype)
```

```
matA:
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
matB:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
matC:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
matU:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
matA.dtype:  float64
matB.dtype:  float64
matC.dtype:  float64
matU.dtype:  float64
```

Outra função importante é o que permite criar um *ndarray* preenchida com o mesmo valor. A função `full` e `full_like` podem ser utilizados com esta finalidade.

In [20]:
```
#outra forma de criar um contador
cont = np.full((10), 0, dtype='int32')
#outra forma de criar um multiplicador
mult = np.full_like(cont, 1)
print(cont)
print(mult)
```

```
[0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 1 1 1 1 1 1]
```

In [21]:
```
matA = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]
matB = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
matA = np.array(matA)
matB = np.array(matB, 'float64')
matC = np.zeros_like(matA)
matU = np.ones_like(matA)
matPix = np.full((3,3), 3.1415)
matImg = np.full_like(matA, 255, dtype='uint8')
print("matA: \n", matA)
print("matB: \n", matB)
print("matC: \n", matC)
print("matU: \n", matU)
print("matPix: \n", matPix)
print("matImg: \n", matImg)
print("matA.dtype: ", matA.dtype)
print("matB.dtype: ", matB.dtype)
print("matC.dtype: ", matC.dtype)
print("matU.dtype: ", matU.dtype)
print("matPix.dtype: ", matPix.dtype)
print("matImg.dtype: ", matImg.dtype)
```

```
matA:
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
matB:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
matC:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
matU:
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
matPix:
 [[3.1415 3.1415 3.1415]
 [3.1415 3.1415 3.1415]
 [3.1415 3.1415 3.1415]]
matImg:
 [[255 255 255]
 [255 255 255]
 [255 255 255]]
matA.dtype:  float64
matB.dtype:  float64
matC.dtype:  float64
matU.dtype:  float64
matPix.dtype:  float64
matImg.dtype:  uint8
```

Uma matriz particularmente importante é a matriz identidade que pode ser gerada em python com as funções `eye`.

In [22]:
```
#Criando um ndarray na forma de uma matriz identidade de 3x3
matI = np.eye(3) #identity
print(matI)
print(type(matI))
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
<class 'numpy.ndarray'>
```

Para os programadores **C/C++** temos ainda as funções `empty` e `empty_like` que cria *ndarrays* vazios, sem valores de inicialização.

In [23]:
```
contE = np.empty((10), dtype='int32')
print(contE)
print(type(contE))
matE = np.empty_like(matB)
print(matE)
print(type(matE))
```

```
[1 1 1 1 1 1 1 1 1 1]
<class 'numpy.ndarray'>
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
<class 'numpy.ndarray'>
```

Para criarmos *ndarrays* utilizando um iterador tipo `range` podemos utilizar a função `arange`. Esta função retorna valores espaçados uniformemente dentro de um determinado intervalo.

Da mesma forma que `range`, `arange` pode ser chamado com um número variável de argumentos posicionais:

- `arange(stop)`: Os valores são gerados dentro do intervalo semiaberto [0, stop) (em outras palavras, o intervalo incluindo início, mas excluindo parada).
- `arange(start, stop)`: Os valores são gerados dentro do intervalo semiaberto [start, stop).
- `arange(start, stop, step)`: Os valores são gerados dentro do intervalo semiaberto [start, stop), com espaçamento entre os valores dado por passo.

Para argumentos inteiros, a função é aproximadamente equivalente ao `range` do Python, mas retorna um *ndarray* em vez de uma instância de `range`.

In [24]:
```
 forma semelhande ao range:
np.arange(10, dtype='int32') # Gerando 10 elementos de 0 a 9, 10 nã
t(x)
np.arange(-5, 5) # Gerando 10 elementos de -5 a 4, 5 não será inclu
t(x)
t(type(x))
t(x.dtype) # se o tipo não for especificado o tipo será int64
np.arange(0, 10, 2, dtype='float_') # Gerando 10 elementos de 0 a 9,
t(x)
t(type(x))
t(x.dtype)
np.arange(0, 10, like=cont)
t(x)
t(type(x))
t(x.dtype)
```

```
[0 1 2 3 4 5 6 7 8 9]
[-5 -4 -3 -2 -1  0  1  2  3  4]
<class 'numpy.ndarray'>
int64
[0. 2. 4. 6. 8.]
<class 'numpy.ndarray'>
float64
[0 1 2 3 4 5 6 7 8 9]
<class 'numpy.ndarray'>
int64
```

Importante ressaltar que, ao contrario de `range` que gera inteiros **Python** de tamanho ilimitado, a `arange` gera inteiros de tipo específico podendo gerar erros para valores muito grandes. Ou seja, teremos limitações de representação de inteiros como em **C/C++**.

Uma alternativa muito útil para o `arange` é o `linspace` . Esta função retorna números com espaçamento uniforme em um intervalo especificado. A quantidade de valores gerados pode ser especificada no parâmetro `num` mas, por padrão, é 50. Retorna `num` amostras com espaçamento uniforme, calculadas no intervalo fechado [start, stop]. O ponto final do intervalo pode ser opcionalmente excluído (`endpoint` por padrão é `True`).

In [25]:
```
# Criando um ndarray de 5 elementos igualmente espaçados entre 0 e
x_linspace = np.linspace(0,1,5)
print(x_linspace)
```

```
[0.   0.25 0.5  0.75 1.  ]
```

Se se deseja gerar um intervalo com escala logarítmica podemos utilizar `logspace`

In [26]:
```
x_logspace = np.logspace(0,1,5) # de 10^0 = 1 até 10^1 = 10
print(x_logspace)
```

```
[ 1.          1.77827941  3.16227766  5.62341325 10.        ]
```

O pacote **Numpy** possui um módulo **random** que permite gerar *ndarrays* com distribuições aleatórias. Veja os exemplos a seguir.

In [27]:
```
# Criando um ndarray de elementos aleatórios uniformemente
#  distribuídos no intervalo entre 0 e 1
A_random = np.random.random((3,3))
A_random
```

Out[27]:
```
array([[0.56746563, 0.16630811, 0.96265655],
       [0.63468884, 0.73767716, 0.13350821],
       [0.49314305, 0.67591875, 0.82638736]])
```

In [28]:
```
# Criando um ndarray de elementos aleatórios com distribuição
#  normal, media 0 e desvio padrão 1
A_normal = np.random.normal(0, 1, (3,3))
A_normal
```

Out[28]:
```
array([[-0.29095941,  1.19714098, -1.035797  ],
       [ 0.69118164, -0.15711836,  1.20153413],
       [ 1.38466794, -0.16143945,  0.76140274]])
```

In [29]:
```
# Criando um ndarray que simule 10 lançamentos de um dado
x_lan = np.random.randint(1,7,10)
x_lan
```

Out[29]:
```
array([6, 1, 5, 4, 2, 5, 4, 4, 4, 1])
```

Atividade sugerida: Explore na documentação outros recursos para geração de *ndarrys*, particularmente aqueles disponíveis no módulo **random**.