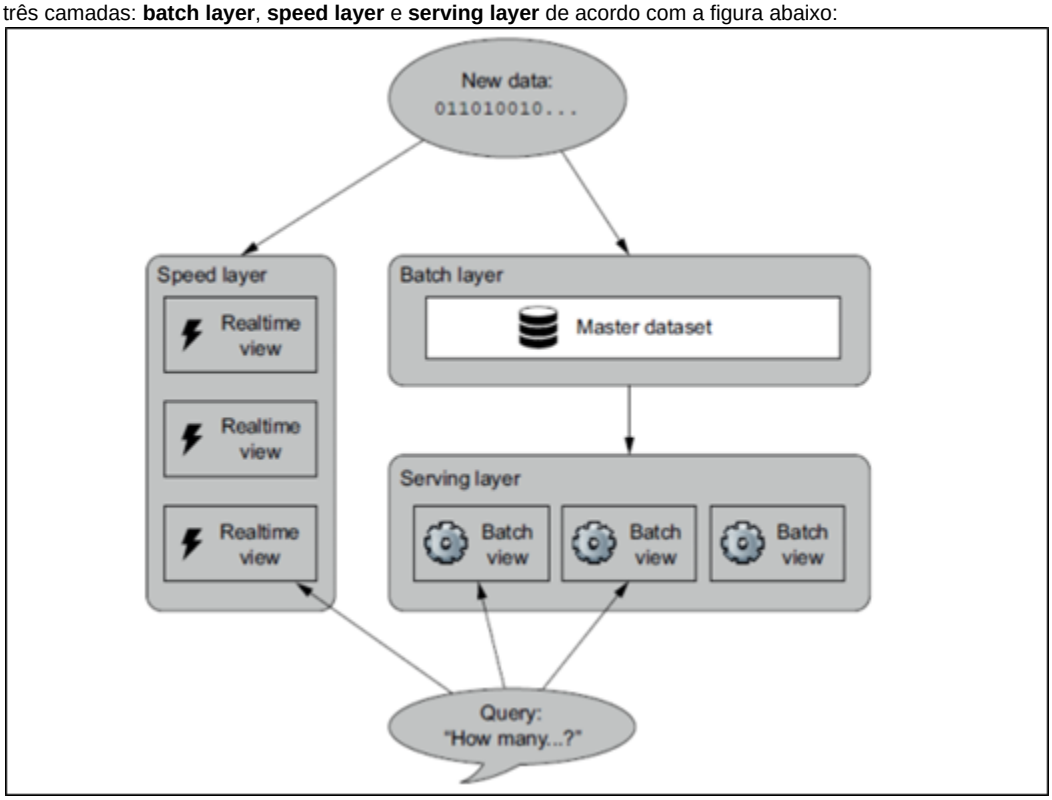
Arquitetura Lambda, ETL/ELT, Bibliotecas Python NumPy e Pandas

Arquitetura Lambda

Primeiramente, arquitetura Lambda não tem nada a ver com o serviço da AWS. Arquitetura Lambda é um modelo de arquitetura Big Data proposto por Nathan Marz [5]. Este modelo independe de soluções tecnológicas específicas para a ingestão, armazenamento e processamento dos dados, ou seja, é um modelo teórico. Marz reforça que não há uma única ferramenta que provem uma solução completa de Big Data, sendo necessário utilizar uma variedade de ferramentas e técnicas para tal. Desta forma, é de suma importância concentrar esforços no desenho de arquitetura de uma solução.

Para que os dados sejam processados e entregues atingindo uma expectativa de tempo dos stakeholders, a arquitetura Lambda é dividida em três camadas: **batch layer**, **speed layer** e **serving layer** de acordo com a figura abaixo:



O dado a ser processado é direcionado para duas camadas (batch e speed). Dentro da **batch layer** esse dado é armazenado de maneira atômica, ou seja, nada é atualizado ou sobrescrito. Caso exista a necessidade de uma mudança, uma nova versão do dado já alterada é armazenada e a anterior não é removida e continua sem mudanças. Isso permite que o dado em seu formato original sempre esteja disponível. Esses dados que estão na **batch layer** são então processados para gerar visualizações pré-calculadas com as informações ajustadas e organizadas de acordo com a necessidade de negócio.

Como a quantidade de dados armazenados a cada dia só cresce, e os dados da **serving layer** só são recebidos ao final do processamento da **batch layer**, o resultado é que cada vez o intervalo para a atualização no **serving layer** fica maior.

Visando compensar esse intervalo a **speed layer** foi criada. Essa camada, que recebe a mesma estrutura atômica de dados, irá processá-los em tempo real e disponibilizá-los para que os sistemas finais disponham dessas informações enquanto esperam pela **batch layer**.

A execução na **speed layer** é bem mais complexa uma vez que os dados precisam ser atualizados e agrupados de acordo com a necessidade do negócio, pois se fossem mantidos em sua forma original inviabilizaria o processamento. Porém essa camada somente precisa se preocupar com os dados que ainda *não foram entregues* pela **batch layer**, o que reduz imensamente a quantidade de dados a ser processada. Assim que o processamento da **batch layer** termina e uma nova versão é disponibilizada, esses dados na **speed layer** podem ser descartados.

Utilizando as três camadas dessa arquitetura é possível processar uma quantidade imensa de dados e mantê-los em sua estrutura original na **batch layer**, disponibilizar esses dados em visualizações pré-computadas (**serving layer**), compensar os intervalos da camada batch e continuar entregando as informações em tempo real (**speed layer**).

Como a arquitetura propõe que os dados sejam armazenados de maneira atômica, o sistema fica protegido até mesmo de erro humano. Também é possível garantir uma visão consistente sem a necessidade de esperar até o final do processamento batch, se beneficiando da **speed layer** e ainda tornando tudo isso transparente aos sistemas finais por meio da **serving layer**.

Data Lake

Um Data Lake é um local central para armazenar todos os seus dados, independentemente de sua origem ou formato [1]. Data Lakes são alimentados com informações em sua forma nativa com pouco ou nenhum processamento realizado para adaptar a estrutura a um esquema corporativo. A estrutura dos dados coletados, portanto, **não é conhecida quando é inserida** no Data Lake, mas é encontrada somente por meio da descoberta, quando lida. Por isso uma grande vantagem de um Data Lake é a **flexibilidade**.

Schema	Schema-on-read (Descobre-se a estrutura dos dados em tempo de leitura)
Escala	Escala para grandes volumes a um custo baixo
Métodos de Acesso	Acessado através de sistemas como SQL, programas criados por desenvolvedores e outros métodos
Workload	Suporta processamento batch, além de um recurso aprimorado sobre EDWs para suportar consultas interativas de usuários
Dado	Bruto (Raw), Confiável (Trusted), Refinado(Refined)
Complexidade	Processamento Complexos
Custo/Eficiência	Uso eficiente das capacidades de armazenamento e processamento a um custo muito baixo
Benefícios	· Transforma a economia financeira do armazenamento de grandes quantidades de dados · Suporta HiveQL, Spark e entre outros frameworks de programação de alto nível · Escala para executar em dezenas de milhares de servidores · Permite o uso de qualquer ferramenta · Permite que a análise comece assim que os dados chegam · Permite o uso de conteúdo estruturado e não estruturado em um único armazenamento · Suporta modelagem ágil, permitindo que os usuários alterem modelos, aplicativos e consultas (queries)

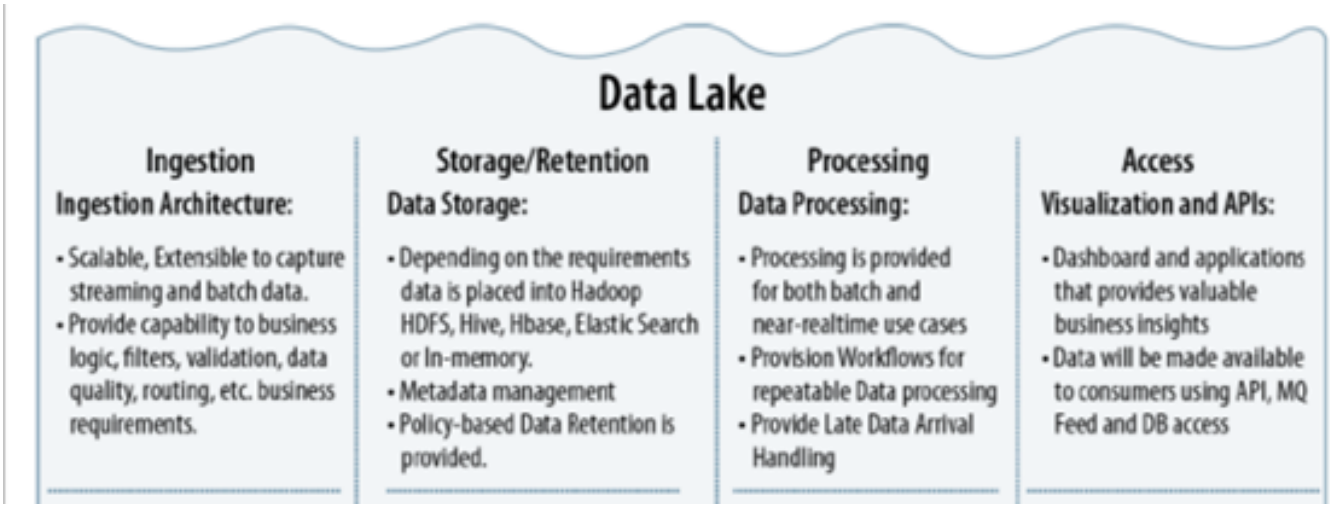
Atributos Chaves de um Data Lake

Um repositório para ser considerado um Data Lake deve ter pelo menos as sete (7) características abaixo:

- Deve ser um único repositório compartilhado de dados da organização.
- Aceita todos os tipos de dados estruturados, semi-estruturados e não-estruturados.
- Baixo custo de armazenamento
- Incluir capacidades de orquestração e agendamento de tarefas (jobs)
- Conter um conjunto de aplicativos ou de workflows para consumir, processar ou agir de acordo com os dados
- Suporta regras de segurança e proteção de dados.
- Desacopla o armazenamento do processamento (permitindo alta performance e alta escala).

Funções básicas de um Data Lake

Um Data Lake possui 4 funções básicas: ingestão, armazenamento, processamento e consumo (ingestion, storage/retention, processing, and access). A figura abaixo sintetiza essas funções.



Ingestão

Camada para capturar por *streaming* e *batch* os dados das diferentes origens. Importante dessa camada ser gerenciada, pois assim dá controle sobre como o dado foi ingerido, de onde veio, quando chegou e onde está armazenado no Data Lake.

Uma importante parte de uma arquitetura de Data Lake é primeiro colocar o dado em uma área de transição ou *stage* (*transitional area*) antes de movê-lo para o repositório de dados brutos (raw).

Processos de governança podem incidir na fase de ingestão como: criptografia, procedência ou linhagem, capturar metadados e limpar os dados.

Armazenamento

Por definição um Data Lake prove melhor eficiência de custos para o armazenamento que um EDW. Como Data Lakes se utilizam de técnicas como *schema on-read*, cada parte dos dados é armazenada com otimização, pois não existem linhas ou colunas nulas. Além disso, o Data Lake pode ser implementado por meio de hardware mais barato se comparado a um EDW.

Processamento

Processamento é a fase no qual os dados podem ser transformados em um formato padronizado por usuários de negócios ou cientistas de dados. Essa fase é necessária pois na ingestão não há processos para este fim.

Um dos maiores benefícios dessa metodologia é que diferentes usuários de negócios podem executar diferentes padronizações e transformações, dependendo de suas necessidades. Muito diferente de um EDW.

Com as ferramentas certas, você pode processar dados para casos de uso em *batch* e em *near real-time*. O processamento em *batch* é para cargas de trabalho ETL tradicionais ou grandes blocos de dados. Já *streaming* é para cenários em que relatórios precisam ser entregues em *real -time* ou *near real-time* e não podem esperar por uma atualização diária.

Consumo

Existem várias formas de acessar os dados: queries, extrações baseadas em ferramentas ou extrações que precisam acontecer por meio de uma API. A visualização é uma parte importante desta etapa, onde os dados são transformados em gráficos para facilitar a compreensão pelo usuário.

Para garantir o bom funcionamento destas etapas básicas, há uma importante função ainda não listada, que é o gerenciamento e monitoramento.

Gerenciamento e Monitoramento

A governança de dados está se tornando uma parte cada vez mais importante de um Data Lake corporativo. Soluções que apresentem a linhagem do dado, gestão e captura dos metadados, glossário de negócios, entre outras necessidades importantes na organização do Data Lake.

Essas soluções são mais completas em ferramentas de Catálogo de Dados e utilizam diferentes abordagens. Um método top-down usa as práticas recomendadas das experiências de EDW das organizações e tenta impor governança e gerenciamento a partir do momento em que os dados são ingeridos no lago de dados. Outras soluções adotam uma abordagem bottom-up que permite aos usuários explorar, descobrir e analisar os dados de maneira muito mais fluida e flexível. E algumas soluções combinam as duas abordagens.

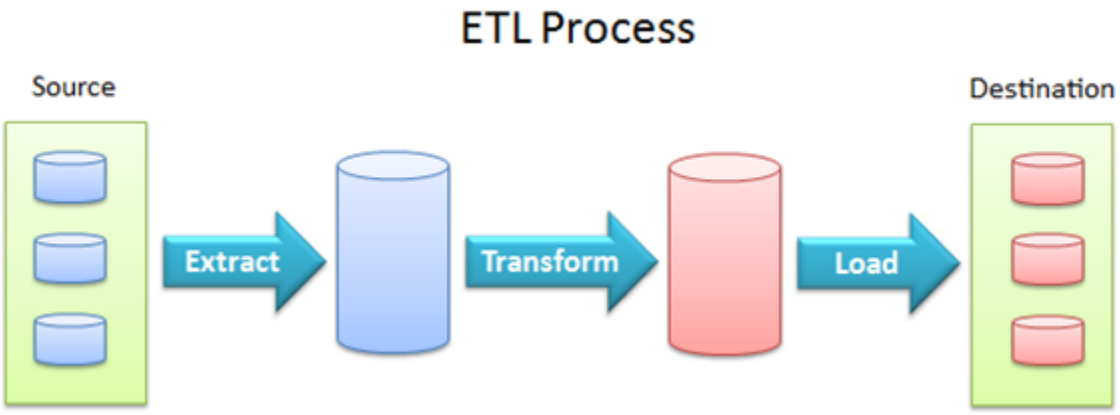
Metadados são extraordinariamente importantes para gerenciar um Data Lake. Existem três tipos distintos, mas igualmente importantes, de metadados para coletar: **dados técnicos**, **operacionais** e **de negócios**.

Técnico	Captura a forma e a estrutura de cada conjunto de dados	Tipo de dados (texto, JSON, Avro), estrutura dos dados (os campos e seus tipos)
Operacional	Captura a linhagem (lineage), qualidade, perfil e governança do dado	Localização de origem e destino dos dados, tamanho, número de registros e a linhagem
Negócio	Captura o que significa para o usuário	Nomes comerciais (business), descrições, tags, qualidade e regras de mascaramento

Todos esses tipos de metadados devem ser criados e ativamente curados (curated) - caso contrário, o Data Lake é simplesmente uma oportunidade desperdiçada. Esses recursos protegem os dados e reduzem os riscos, pois o data manager sempre saberá de onde os dados vieram, onde estão e como estão sendo usados.

ETL/ELT

Extração, Transformação, Carregamento, do inglês *Extract Transform Load* (ETL), é o processamento de blocos de dados em etapas de Extração, Transformação e Carga. ETLs são comumente utilizados para construir um *Data Warehouse* (DW) e *Business Intelligence* (BI) mas podem ser utilizados para outras finalidades, como por exemplo a ingestão de dados frios ou históricos num Data Lake. De acordo com a SAS, nesse processo os dados são retirados (extraídos) de um sistema-fonte, convertidos (transformados) em um formato que possa ser analisado, e armazenados (carregados) em um armazém ou outro sistema. Vale ressaltar que algumas ferramentas não fazem o ETL tradicional e sim uma variação, o ELT.



Na prática, por exemplo, se você precisa importar um arquivo CSV (arquivo texto separado por vírgulas) em um banco de dados, convertendo os valores das colunas para tipos de dados aceitos em um banco de dados, você provavelmente usará alguma ferramenta de ETL como o Oracle Data Integrator, Pentaho ou Talend.

Para resolver o exemplo acima, os passos de ETL são: um componente extrai os dados de um arquivo texto, componentes de derivação e conversão de colunas transformam os dados e um outro componente carrega os dados no banco de dados de destino [3].

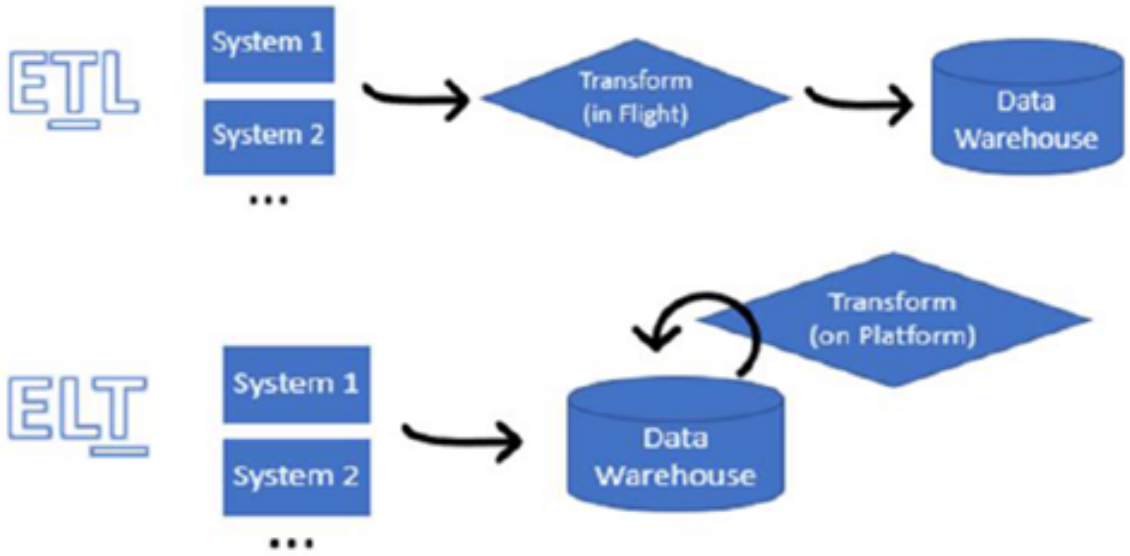
Perfeito! Seus dados são carregados em um banco de dados usando uma tecnologia simples e confiável e todos ficam felizes. Então por que precisamos de uma nova sigla misteriosa para supostamente melhorar esta técnica?

Num ETL simples, como o exemplo mostrado acima, não seria um grande problema em termos de implementação, manutenção e complexidade. Porém, suponha que você precisa fazer isso com milhares de arquivos criados diariamente e cada arquivo tem muitos GB de dados e estes dados devem ser disponibilizados para um grande número de consumidores de dados, como Data Warehouses e/ou outras aplicações? Bem, neste contexto, o seu ETL pode se tornar algo bastante complexo, enfrentando problemas de desempenho e de disponibilidade de dados. Além disso, seus usuários só poderão acessar os dados extraídos depois que todo o processo terminou, o que pode levar um tempo considerável dependendo das tecnologias usadas e do contexto da sua organização [3].

Extrair / carregar / transformar (ELT) da mesma forma extrai dados de uma ou várias origens, mas os carrega para o destino sem nenhuma outra formatação. A transformação de dados, em um processo ELT, ocorre no banco de dados de destino. O ELT necessita de menos fontes remotas, exigindo apenas dados brutos e despreparados.

O ELT existe há algum tempo, mas ganhou um interesse maior com as novas ferramentas para movimentação de dados como o Apache Hadoop. Uma grande tarefa, como transformar petabytes de dados brutos, foi dividida em pequenos trabalhos, sendo processada remotamente e retornada para carregamento no banco de dados [4].

Cada método tem suas vantagens. Ao planejar a arquitetura de dados, os tomadores de decisão de TI devem considerar os recursos internos e o crescente impacto das tecnologias em nuvem ao escolher ETL ou ELT.



NumPy

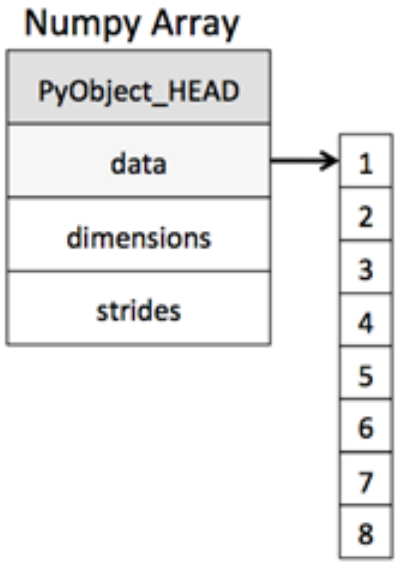
Introdução

- Numpy é uma biblioteca para cálculo vetorial e matricial disponibilizada em Python.
- Várias outras bibliotecas utilizam NumPy como base para seus cálculos
- Utilizar NumPy ao invés das estruturas básicas de Python (exemplo:. listas) apresenta melhorias de desempenho
- Para instalar via pip utilize o comando `pip install numpy`

```
C:\WINDOWS\system32>pip install numpy
Collecting numpy
  Using cached https://files.pythonhosted.org/packages/bd/51/7df1a3858ff0465f760b482514f1292836f8be08d84aba411b48dda72f
9/numpy-1.17.2-cp37-cp37m-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.17.2
WARNING: You are using pip version 19.2.1, however version 19.2.3 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
```

Arrays NumPy

- Funcionam como arrays em C++/Java
- Alocam espaços contíguos em memória
- São utilizados em várias funções numpy



- Utilizar arrays permite cálculos ainda mais rápidos que funções em C de Python
- Por que NumPy é tão rápido?
  - Além de usar dados contíguos em memória, as funções numpy usam a biblioteca BLAS (Basic Linear Algebra Subprograms):
    - São sub-rotinas para realizar cálculos matemáticos e matriciais, disponíveis para CPUs e GPUs
    - MATLAB e Octave também utilizam BLAS
- São estruturas homogêneas
- Possui um tipo, dentre eles:
  - **np.int8**
  - **np.int16**
  - **np.int32**
  - **np.int64**
  - **np.uint8**
  - **np.uint16**
  - **np.uint32**
  - **np.uint64**
  - **np.float32**
  - **np.float64**
  - **np.complex64**
  - **np.complext128**
  - **np.unicode** (Utilizado para string. Possui algum tamanho específico, exemplo: <U10)
  - Lista completa em: Numpy Data types
- Possuem os seguintes atributos:
  - **ndarray.ndim:** Número de dimensões do array (exemplo: 2 se for uma matriz)
  - **ndarray.shape:** Tupla que representa o formato do array (exemplo: (3,3))
  - **ndarray.size:** Número total de elementos do array, é equivalente a `np.prod(ndarray.shape)`
  - **ndarray.dtype:** Tipo de dado do array (exemplo: `np.float32`)
  - **ndarray.itemsize:** Tamanho em bytes que cada elemento do array ocupa (exemplo: 8 para um `np.float64`)

Construindo arrays

np.array(...)

- Constrói um *array* com base nos dados que são passados à estrutura
- Dados podem ser provenientes de outros contêineres (lista, tupla), conforme demonstramos no código abaixo

```
import numpy as np

tupla = ('a', 'b', 'c','d')
lista = [1,3,6,9,7,9,12]

array1 = np.array(tupla)
array2 = np.array(lista)


print("Conteúdo:{}, shape: {}, Tipo: {} ".format(array1, array1.shape,
array1.dtype))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(array2, array2.shape,
array2.dtype))


array3 = np.array(lista,dtype=np.float32)
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(array3, array3.shape,
array3.dtype))


"""
Saída:

Conteúdo:['a' 'b' 'c' 'd'], shape: (4,), Tipo: <U1
Conteúdo:[ 1  3  6  9  7  9 12], shape: (7,), Tipo: int32
Conteúdo:[ 1.  3.  6.  9.  7.  9. 12.], shape: (7,), Tipo: float32

"""
```

np.arange(...)

- Mesmo funcionamento da função `range`
- Várias assinaturas da função

```
import numpy as np

a = np.arange(10)
b = np.arange(2,5)
c = np.arange(0,10,2)
d = np.arange(5,dtype=np.float32)



print("Conteúdo:{}, shape: {}, Tipo: {} ".format(a, a.shape, a.dtype))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(b, b.shape, b.dtype))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(c, c.shape, c.dtype))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(d, d.shape, d.dtype))

"""
Saída:

Conteúdo:[0 1 2 3 4 5 6 7 8 9], shape: (10,), Tipo: int32
Conteúdo:[2 3 4], shape: (3,), Tipo: int32
Conteúdo:[0 2 4 6 8], shape: (5,), Tipo: float32
Conteúdo:[0. 1. 2. 3. 4.], shape: (5,), Tipo: float32
"""
```

Alterando as dimensões de um array (reshape)

O método `reshape` permite alterar a estrutura dimensional do array. Observe a ilustração e o código exemplificando o funcionamento da operação.

```
import numpy as np

vector = np.arange(stop=20, dtype=np.int16)
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(vector, vector.shape,
vector.dtype))

matrix = vector.reshape((5,4))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(matrix, matrix.shape,
matrix.dtype))

cube = vector.reshape((5,2,2))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(cube, cube.shape, cube.
dtype))


"""
Conteúdo:[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19],
shape: (20,), Tipo: int16
Conteúdo:[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]], shape: (5, 4), Tipo: int16
Conteúdo:[[[ 0  1]
  [ 2  3]]

 [[ 4  5]
  [ 6  7]]

 [[ 8  9]
  [10 11]]

 [[12 13]
  [14 15]]

 [[16 17]
  [18 19]]], shape: (5, 2, 2), Tipo: int16
"""
```

np.zeros(...) e np.ones(...)

- Constrói *arrays* onde todos os valores são `0` ou `1`

```
import numpy as np

zeros = np.zeros((3,3),dtype=np.int16)
ones  = np.ones((3,2),dtype=np.int16)
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(zeros, zeros.shape,
zeros.dtype))
print("Conteúdo:{}, shape: {}, Tipo: {} ".format(ones, ones.shape, ones.
dtype)),


"""
Saída:

Conteúdo:[[0 0 0]
 [0 0 0]
 [0 0 0]], shape: (3, 3), Tipo: int16
Conteúdo:[[1 1]
 [1 1]
 [1 1]], shape: (3, 2), Tipo: int16
"""
```

Aplicando Slicing

- Assim como em listas de Python, NumPy arrays também podem ser fatiados (*slicing*)
  - *Slicing* é a técnica de “fatiar” um contêiner que suporta indexação linear
  - O fatiamento se dá adicionando um par de colchetes ao fim da variável
  - Possui 3 parâmetros: início (incluso), fim (excluso) e passo

Em nosso primeiro exemplo, estamos utilizando um array unidimensional (vetor), aplicando sobre ele operações de fatiamento.

```
import numpy as np

array = np.arange(20)
print("Array completo - ", array)
print("A partir do índice 2 (incluso) - ", array[2:])
print("Até o índice 2 (excluso) - ", array[:2])
print("Do índice 4 até 8 (excluso) - ", array[4:8])
print("Desconsiderar os 2 últimos - ", array[:-2])
print("Considerar apenas os 2 últimos - ", array[-2:])
array[18:20] = [-1,-2]
print("Array modificado após atribuição - ", array)

"""
Saída:

Array completo -  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19]
A partir do índice 2 (incluso) -  [ 2  3  4  5  6  7  8  9 10 11 12 13
14 15 16 17 18 19]
Até o índice 2 (excluso) -  [0 1]
Do índice 4 até 8 (excluso) -  [4 5 6 7]
Desconsiderar os 2 últimos -  [ 0  1  2  3  4  5  6  7  8  9 10 11 12
13 14 15 16 17]
Considerar apenas os 2 últimos -  [18 19]
Array modificado após atribuição -  [ 0  1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 -1 -2]
"""
```

E o mesmo princípio de aplica para *arrays* com duas ou mais dimensões.

```
import numpy as np

matrix = np.arange(10).reshape((2,5))

print("Array completo - ", matrix)
print("Primeira linha - ", matrix[0,:])
print("Primeira coluna - ", matrix[:,0])
print("Última linha - ", matrix[-1,:])
print("Última coluna - ", matrix[:,-1])
print(" Colunas 2 e 3 - ", matrix[ :,2:4])


"""
Array completo -  [[0 1 2 3 4]
 [5 6 7 8 9]]
Primeira linha -  [0 1 2 3 4]
Primeira coluna -  [0 5]
Última linha -  [5 6 7 8 9]
Última coluna -  [4 9]
 Colunas 2 e 3 -  [[2 3]
 [7 8]]
 """
```

Operações com arrays

Algumas operações com *arrays* incluem:

- divisão, multiplicação, soma, subtração
- junção
- transposição
- atribuição
- operações binárias e lógicas

Algumas operações com *arrays* incluem:

- **divisão, multiplicação, soma, subtração**

```
import numpy as np

a = np.arange(0,5,dtype=np.int32)
b = np.arange(6,11,dtype=np.int32)

print("A:",a)
print("B:",b)
print ("A + B:",a+b)
print ("A + 2:",a+2)
print ("B - 5.6:",b-5.6)
print ("A * 4:",a*5)
print ("B / 2", b / 2)
print ("A / B: ", a / b)

"""
A: [0 1 2 3 4]
B: [ 6  7  8  9 10]
A + B: [ 6  8 10 12 14]
A + 2: [2 3 4 5 6]
B - 5.6: [0.4 1.4 2.4 3.4 4.4]
A * 4: [ 0  5 10 15 20]
B / 2 [3.  3.5 4.  4.5 5. ]
A / B:  [0.         0.14285714 0.25       0.33333333 0.4       ]
"""
```

- **Somas e médias**

```
a = np.arange(1,10).reshape(3,3)
print("Elementos: ",a)
print("Soma dos elementos: ",np.sum(a))
print("Soma as linhas: ", np.sum(a,axis=0))
print("Média as linhas: ", np.average(a,axis=0))
print("Soma das colunas: ", np.sum(a,axis=1))
print("Média das colunas: ", np.average(a,axis=1))

"""
Elementos:  [[1 2 3]
 [4 5 6]
 [7 8 9]]
Soma dos elementos:  45
Soma as linhas:  [12 15 18]
Média as linhas:  [4. 5. 6.]
Soma das colunas:  [ 6 15 24]
Média das colunas:  [2. 5. 8.]
"""
```

- **Operações lógicas**

```
a = np.array([0,0,1,1],dtype=np.bool8)
b = np.array([0,1,0,1], dtype=np.bool8)

print("A:", a)
print("B:", b)
print("A && B: ", a & b)
print("A || B: ", a | b)
print("~B: ", np.logical_not(b))
print("A^B: ", np.logical_xor(a,b))


"""
A: [False False  True  True]
B: [False  True False  True]
A && B:  [False False False  True]
A || B:  [False  True  True  True]
~B:  [ True False  True False]
A^B:  [False  True  True False]

"""
```

Álgebra linear

- NumPy é uma das bibliotecas preferidas para cálculo matricial
- Apesar de existir um tipo matrix, na prática ele não é muito utilizado

Multiplicação de matrizes

- Realizada por meio do operador a ou método `dot`

```
a = np.arange(12).reshape((3,4))
b = np.arange(12).reshape((4,3))
print("A:", a)
print("B:", b)
print("A * B: ", a@b)
print("A * B: ", np.dot(a,b))

"""
Saída:
A: [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
B: [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
A * B:  [[ 42  48  54]
 [114 136 158]
 [186 224 262]]
A * B:  [[ 42  48  54]
 [114 136 158]
 [186 224 262]]

"""
```

Ordenação

Existem diversos algoritmos disponíveis para ordenação na biblioteca NumPy, dentre eles:

- Quicksort
- Mergesort
- Heapsort

Cada um desses algoritmos possui características que os tornam mais atrativos, dependendo do caso de uso.

Algoritmo	Estável	Inplace
Quicksort	não	sim
Mergesort	sim	não
Heapsort	não	sim

```
a = np.random.choice(30,size=10, replace=False)
print("A: ",a)
print("A ordenado com quicksort",np.sort(a))
print("A ordenado com mergesort",np.sort(a,kind="mergesort"))

"""
Saída:

A:  [29 11  4 14  2 10 28 20  7  6]
A ordenado com quicksort [ 2  4  6  7 10 11 14 20 28 29]
A ordenado com mergesort [ 2  4  6  7 10 11 14 20 28 29]

"""
```

Ordenação baseada em eixos

```
a = np.random.choice(30,size=(5,5), replace=False)
print("A: ",a)
print("A ordenado por linha",np.sort(a,axis=1))
print("A ordenado por coluna",np.sort(a,axis=0))


"""
A:  [[27 12  2 16 20]
 [13 26 17 14 25]
 [15  7  6 22  8]
 [ 4  0 23 18 29]
 [10  1 21 24  5]]
A ordenado por linha [[ 2 12 16 20 27]
 [13 14 17 25 26]
 [ 6  7  8 15 22]
 [ 0  4 18 23 29]
 [ 1  5 10 21 24]]
A ordenado por coluna [[ 4  0  2 14  5]
 [10  1  6 16  8]
 [13  7 17 18 20]
 [15 12 21 22 25]
 [27 26 23 24 29]]

 """
```

Busca por valores

Permite a busca por valores máximos e mínimos dentro do array.

- `argmax`: retorna o índice do maior valor
- `argmin:` retorna o índice do menor valor
- `max`: retorna o maior valor
- `min`: retorno o menor valor

```
a = np.random.choice(30,size=10, replace=False)
print("A: ",a)
print("Maior valor",np.max(a))
print("Menor valor",np.min(a))
print("Índice do maior valor",np.argmax(a))
print("Índice do menor valor",np.argmin(a))

"""
A:  [ 7 23 19 16 20  1  4 21  9 14]
Maior valor 23
Menor valor 1
Índice do maior valor 1
Índice do menor valor 5
"""
```

Números Randômicos

NumPy oferece uma biblioteca, `numpy.random`, para geração de números randômicos. Os principais métodos são:

- `random`
  - Gera amostra valores no intervalo [0, 1)
  - Aceita como parâmetro o número de dimensões do array

```
a = np.random.random(4)
b = np.random.random((5,3))
print("A:", a)
print("B:", b)

"""
A: [0.90422177 0.3854884  0.72581636 0.43670599]
B: [[0.58173397 0.61339149 0.47915415]
 [0.59212647 0.13451955 0.62767973]
 [0.3276273  0.22364636 0.9041094 ]
 [0.73682705 0.68886214 0.62716047]
 [0.22860333 0.82501864 0.9640303 ]]

 """
```

- `randint`
  - Gera amostra de valores inteiros
  - Aceita como parâmetros:
    - limite inferior
    - limite superior
    - dimensões do array

Para gerar amostra de apenas um valor, no intervalo [0, 5), utilizamos:

```
a = np.random.randint(5)
print(a)
```

Para gerar amostra de apenas um valor, no intervalo [6, 10), utilizamos:

```
a = np.random.randint(6,10)
print(a)
```

Para gerar amostra de 3 valores no intervalo [6, 10), utilizamos:

```
a = np.random.randint(low=6,high=10,size=3)
print(a)
```

Para gerar amostra de uma matrix (2,2) com valores no intervalo [6, 10), utilizamos:

```
a = np.random.randint(low=6,high=10,size=(2,2))
print(a)
```

- `choice`
  - Gera amostra valores a partir de uma distribuição de probabilidades
  - Aceita como parâmetros:
    - a - Valores
    - *size* - Número de amostras
    - *replace* - Se a amostragem é com reposição
    - *p* - Distribuição de probabilidades

Para gerar amostra de apenas um valor, com distribuição uniforme:

```
caracteres=['a','b','c','d','e','f']
a = np.random.choice(caracteres)
print(a)
```

Para gerar 10 amostras de valores, com reposição:

```
caracteres=['a','b','c','d','e','f']
a = np.random.choice(caracteres,size=10)
print(a)
```

Para gerar 3 amostras de valores, sem reposição:

```
caracteres=['a','b','c','d','e','f']
a = np.random.choice(caracteres,size=3,replace=False)
print(a)
```

Ao indicar para a função não repetir valores, você deve garantir que o universo de valores, em nosso caso, a lista de caracteres, contém a quantidade de valores distintos exigidos pelo parâmetro `size`.

Agora, vamos gerar uma amostra de 100 valores, com reposição, usando a distribuição de probabilidades que desejamos. E, para validar o resultado, faremos o cálculo de frequência da amostra.

```
caracteres=['a','b','c','d','e','f']
probabilidades = [0.2,.1,.3,.1,.2,.1]

a = np.random.choice(caracteres,size=100,p=probabilidades)

from collections import Counter


for caractere, contagem in Counter(a).items():
    print("{} - Ocorrências: {} - Frequência: {}".format(caractere,
contagem,contagem/100))
```

Ao executar o código você perceberá que a frequência de ocorrência dos valores amostrados tende às probabilidades especificadas no método `choice`

Estatísticas

A biblioteca NumPy oferece diversas funções estatísticas aplicadas à distribuição de valores. Na sequência apresentamos as funções `mean`, `median`, `std` e `percentile`:

```
valores = np.random.randint(0,20, size=10)
valores.sort()
print("Valores:", valores)
print("Média aritmética: ",np.mean(valores))
print("Mediana: ",np.median(valores))
print("Desvio padrão: ",np.std (valores))
print("80º percentil", np.percentile(valores,80,method="nearest") )
```

Pandas

- É uma biblioteca de fácil uso para Python, open-source e de uso gratuito (sob uma licença BSD), que fornece ferramentas para análise e manipulação de dados.
- Possui duas estruturas básicas: **Series** e **DataFrame**

Series

- **Series** é utilizado para representar séries unidimensionais de informações, como um atributo (i.e. coluna) ou uma instância (i.e. linha)
- **DataFrame** é utilizado para representar informações bidimensionais, como datasets
- Para instalar o Pandas, utilize o comando `pip install pandas`

Series

- Representa um conjunto de dados unidimensionais (array, atributo)
- Pode ser resultado de uma seleção sobre um DataFrame ou poder ser criada diretamente

```
import pandas as pd

serie = pd.Series(['Amarelo','Verde','Azul'])
print(serie)
print("Valor para o índice 0", serie[0])

serie_2 = pd.Series( dict(amarelo=10,verde=3,pedro=60))
print(serie_2)
print("Valor para o índice 'amarelo'", serie_2['amarelo'])
print("Índices da série", serie_2.index)

"""
Saída:

0    Amarelo
1      Verde
2       Azul
dtype: object
Valor para o índice 0 Amarelo
amarelo    10
verde       3
pedro      60
dtype: int64
Valor para o índice 'amarelo' 10
Índices da série Index(['amarelo', 'verde', 'pedro'], dtype='object')

"""
```

DataFrame

- É a principal estrutura de dados do Pandas
- Representa uma **tabela** similar às tabelas de bancos de dados relacionais
- Permite dados heterogêneos (uma coluna do tipo float, outra do tipo str, por exemplo)
- Permite operações semelhantes a operações de banco de dados (Seleção, agregação, etc...)

```
import pandas as pd

df = pd.DataFrame(
    [
        ['a','b','c'],
        ['d',4,5],
        [6,7,8]
    ],
    index=['linha 0','linha 1','linha 2'],
    columns = ['coluna 0','coluna 1','coluna 2'],
    copy = True
)
print(df)

"""
Saída:

        coluna 0 coluna 1 coluna 2
linha 0        a        b        c
linha 1        d        4        5
linha 2        6        7        8

"""
```

- Existem diversas formas de iniciar dataframes. Uma das mais comuns é a partir de um arquivo., como CSV, por exemplo, utilizando `pandas.read_csv(...)`. Vamos utilizar o dataset *iris*, disponível para download no GitHub.

```
import pandas as pd

iris = pd.read_csv("./iris.csv")
print(iris)
print(iris[4:10]['sepal_length'])
print("Formato: ", iris.shape)

"""

Saída:
     sepal_length  sepal_width  petal_length  petal_width    species
0             5.1          3.5           1.4          0.2     setosa
1             4.9          3.0           1.4          0.2     setosa
2             4.7          3.2           1.3          0.2     setosa
3             4.6          3.1           1.5          0.2     setosa
4             5.0          3.6           1.4          0.2     setosa
..            ...          ...           ...          ...        ...
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica

[150 rows x 5 columns]
4    5.0
5    5.4
6    4.6
7    5.0
8    4.4
9    4.9
Name: sepal_length, dtype: float64
Formato:  (150, 5)
"""
```

- Da mesma forma que é possível criar dataframes a partir do conteúdo de arquivos, também podemos escrever arquivos a partir de dataframes. É o caso do exemplo a seguir, no qual escrevemos um arquivo parquet a partir de um subconjunto do dataframe *iris*.

```
import pandas as pd

iris = pd.read_csv("./iris.csv")
output = iris.loc[4:10,['sepal_length','sepal_width' ]]
print(output)
output.to_parquet("./iris.parquet")
```

Indexação simples

- Dataframes são indexados de duas formas: linhas ou colunas
- **Índices** sao rótulos para as linhas ou colunas
- As linhas podem ser indexadas de acordo com o índice do Dataframe
- O mesmo raciocínio é aplicado às colunas

Indexando colunas

- Pode-se indexar colunas diretamente: `df['NomeColuna']` ou múltiplas colunas `df[['Col1', 'Col2']]`

```
iris = pd.read_csv("./iris.csv")

print("Índices de linhas: ", iris.index)
print("Colunas: ", iris.columns)
print(iris['sepal_length'])
print(iris[['petal_length', 'petal_width']])
```

Indexando linhas

Podemos indexar linhas de duas maneiras:

- utilizando `df.loc[indice]` caso o índice do dataframe não seja composto por inteiros
- utilizando `df.iloc[indice]` caso o índice do dataframe seja composto por inteiros

```
import pandas as pd

iris = pd.read_csv("./iris.csv")

print("Linhas 3, 9 e 36:", iris.iloc[[3,9,36]])
print("Linha 86:", iris.loc[86])
```

Indexação composta

- A indexação padrão dos dataframes assume apenas um valor por índice
- Todavia, é possível utilizar índices compostos
- Útil quando estamos visualizando informações agregadas

```
import pandas as pd

index = pd.MultiIndex.from_tuples([("2018","Ka"),("1980","Fusca"),
("2014","Ka")],names=["ano","modelo"])

df = pd.DataFrame(
    data=[[4,50000],[2,5000],[2,15000]],
    index=index,
    columns=['Nr portas','Valor']

)

print(df.loc[[('1980','Fusca'),('2014','Ka') ]])

"""
             Nr portas  Valor
ano  modelo
1980 Fusca           2   5000
2014 Ka              2  15000

"""
```

Seleção

- A seleção em pandas é similar ao que ocorre em bancos de dados relacionais
- É possível selecionar valores com base no valor de diversas colunas, simultaneamente. Vamos considerar o seguinte dataframe em nosso exemplo:

```
import pandas as pd

df = pd.DataFrame(data=np.random.random((5,5)),index=
['a','b','c','d','e'],columns=['k','l','m','n','o'])

print(df)

"""
Saída:

          k         l         m         n         o
a  0.495434  0.055880  0.399815  0.824124  0.633334
b  0.462256  0.763234  0.358540  0.357030  0.837309
c  0.098857  0.176630  0.207488  0.421153  0.106348
d  0.696148  0.857368  0.184816  0.664843  0.561403
e  0.370180  0.787413  0.956699  0.748910  0.960571

"""
```

- Linhas em que o valor da coluna `k` é maior que `0.5` E coluna o possui valor maior que `0.6`

```
df[(df['k'] > 0.5) & (df['o']>0.6)]
```

- Linhas em que o valor da coluna `k` é maior que `0.5` OU coluna o possui valor maior que `0.8`

```
df[(df['k'] > 0.5) | (df['o']>0.8)]
```

Seleção

A estrutura `loc` também aceita indexação por linhas e colunas. Primeiro informa-se a linha desejada, e depois o nome da coluna. Observe:

- Selecionar a linha **a**

```
df.loc['a']
```

- Seleciona as linhas **a** e **b** e as colunas **x** e **y**

```
df.loc[['a','b'],['k','l']]
```

Pandas e Numpy

A qualquer momento podemos converter os dados para Numpy utilizando a propriedade de DataFrames e Series utilizando o método `.values`.

```
df = pd.DataFrame(data=np.random.random((5,5)),index=
['a','b','c','d','e'],columns=['k','l','m','n','o'])

np_array = df.values

print(np_array)
print(type(np_array),np_array.shape,np_array.dtype)
```

Ordenação, Agregação, Junção e Split

Assim como NumPy, Pandas também suporta ordenação de linhas/colunas. A ordenação mantém a integridade das linhas/colunas.

- Ordena as linhas

```
df = pd.DataFrame(data=np.random.random((5,5)),index=
['a','b','c','d','e'],columns=['k','l','m','n','o'])
df.sort_values(by="o",axis=0)
```

- Ordena as colunas

```
df = pd.DataFrame(data=np.random.random((5,5)),index=
['a','b','c','d','e'],columns=['k','l','m','n','o'])
df.sort_values(by="b",axis=1)
```

Função map

- `pandas.DataFrame.apply`
- Semelhante a função nativa de Python
- Aplica uma função a cada uma das linhas ou colunas de um DataFrame

```
df = pd.DataFrame(data=np.random.randint(1,100,(5,5)),index=
['a','b','c','d','e'],columns=['k','l','m','n','o'])

def negativo_se_par(row):
    for c in ['k','l','m','n','o']:
        if row[c]%2==0:
            row[c]=-row[c]


df.apply(negativo_se_par,axis=1)

print(df)
```

Operações de banco de dados relacional

- União
- Junção
- Agrupamento

União

- `pandas.DataFrame.merge`
- Permite unir os registros de duas tabelas sem que seja necessário fazer um `join`
- Insere NaN nas colunas que não são compartilhadas entre as duas tabelas

Vamos considerar os seguintes dataframes:

```
cidades = pd.DataFrame(
    {
        'cidade_id':[1,2,3,4,5],
        'cidade_nome':['Cidade A', 'Cidade B','Cidade C','Cidade
D','Cidade E']
    }

)

estudantes = pd.DataFrame(
    {
        'estudante_id': [1,2,3,4,5],
        'estudante_nome':['Estudante A','Estudante B','Estudante
C','Estudante D','Estudande E'],
        'cidade_id':[2,3,3,5,6]

    }
)
```

Cidades

	cidade_id	cidade_nome
0	1	Cidade A
1	2	Cidade B
2	3	Cidade C
3	4	Cidade D
4	5	Cidade E

Estudante

	estudante_id	estudante_nome	cidade_id
0	1	Estudante A	2
1	2	Estudante B	3
2	3	Estudante C	3
3	4	Estudante D	5
4	5	Estudande E	6

Para realizarmos merge com estratégia `inner` podemos utilizar:

```
pd.merge(cidades, estudantes,how="inner")
```

E, para estratégia de `right outer join` podemos utilizar:

```
pd.merge(cidades, estudantes,how="right")
```

Por fim, para informar o nome das colunas que formam o critério de junção, utilizamos o parâmetro on, `left_on` e `right_on`

```
pd.merge(cidades, estudantes,how="right", left_on='cidade_id',
right_on='estudante_id')
```

Agrupamento

- `pandas.DataFrame.groupby`
- Opera sob o mesmo princípio de um *group by* de um banco de dados relacional: agrupa dados utilizando um atributo como pivô

```
estudantes = pd.DataFrame(
    {
        'estudante_id': [1,2,3,4,5],
        'estudante_nome':['Estudante A','Estudante B','Estudante
C','Estudante D','Estudande E'],
        'cidade_id':[2,3,3,5,6],
        'cor_pele':['branco','preto','preto','pardo','branco']

    }
)
estudantes.groupby(by='cor_pele').count()
```

Método agg

Permite agregar dados usando uma ou mais operações sobre algum eixo. No exemplo abaixo usando uma medida de média aritmética do NumPy

```
estudantes = pd.DataFrame(
    {
        'estudante_id': [1,2,3,4,5],
        'estudante_nome':['Estudante A','Estudante B','Estudante
C','Estudante D','Estudande E'],
        'cidade_id':[2,3,3,5,6],
        'cor_pele':['branco','preto','preto','pardo','branco'],
        'nota_redacao':[9,8,8.6,7,6]

    }
)

estudantes.agg({'nota_redacao':[np.mean,np.median,np.std]})
```

Referências

[1] https://data-flair.training/blogs/apache-sqoop-tutorial/

[2] https://www.tutorialspoint.com/sqoop/index.htm

[3]https://www.linkedin.com/pulse/etl-vs-elt-quando-duas-letras-fazem-muita-diferen%C3%A7a-rodrigo-r-g-/

[4] https://www.talend.com/resources/elt-vs-etl/

[5] https://pt.slideshare.net/Celio12/nosql-base-vs-acid-e-teorema-cap

[6] https://numpy.org/doc/stable/

[7] https://pandas.pydata.org/docs/user_guide/index.html