



PYTHON – AULA 3

LÓGICA DE PROGRAMAÇÃO APLICADA



AGENDA

Aula 1

Instalando e conhecendo o Python e as ferramentas do curso.
Revisitando lógica de programação em Python.

Aula 2

Revisitando lógica de programação em Python –
continuação.

Aula 3

Trabalhando com listas, tuplas e dicionários.
Leitura e escrita de JSON.

Aula 4

Leitura e escrita de arquivos.
Introdução a Numpy e Pandas.

AGENDA

Aula 5

Orientação a objeto no Python.

Aula 6

Trabalhando com os algoritmos.
Ordenação e Recursão. – Pesquisa em largura.

Aula 7

Algoritmo de Dijkstra.

Aula 8

Algoritmos Genéticos.

AULA 3

TRABALHANDO COM ESTRUTURAS DE DADOS



PROGRAMANDO EM PYTHON

TRABALHANDO
COM LISTAS



Nesse momento do seu estudo da linguagem Python você já conhece as variáveis, os desvios condicionais e os loops.

Porém, até esse momento não é possível resolver um problema simples: **como permitir a manipulação de volumes maiores de dados?**

Vamos partir de um exemplo simples: você é um fanático por Star Wars e decidiu criar um programa onde seja possível listar todos os cavaleiros Jedi que aparecem em cada um dos filmes.

Não basta *printar* os nomes desses cavaleiros! Queremos poder incluir, verificar quantos deles existem, alterar o nome de alguns deles, enfim, queremos manipular esses dados!

Quem solucionará esse problema são as listas!

Crie um novo script chamado **listas.py**.

As *lists* são estruturas de dados na linguagem python que permitem armazenarmos diversos dados simultaneamente.

A grande vantagem desse tipo de estrutura é que, ao contrário de uma variável, diversos valores podem ser preservados durante a execução do nosso programa.

Vamos conhecer um pouco mais sobre elas.



```
nome_da_list = [valor1, valor2, valor3, valor4]
```

A sintaxe de criação de uma lista é feita através da indicação do nome da nova lista e do símbolo de [] (colchetes).

Quando listamos uma sequência de valores dentro do símbolo de colchetes, cada um desses valores ocupa uma posição dentro da lista que está sendo criada.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]
```

A lista acima, por exemplo, contém 4 valores de texto: Yoda, Luke, Obi-Wan e Anakin.

Cada um desses valores está armazenado em uma posição dentro da nossa lista.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
print(jedi)
```

Ao utilizarmos o comando de print e o nome da lista, ela é *printada* inteira e exatamente como é: com os sinais de colchetes, valores entre aspas e separada por vírgulas.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
print(jedi[1])
```

Se quisermos exibir um item específico da nossa lista, podemos utilizar o *índice*.

O índice é um valor numérico inteiro, iniciado em zero, que indica a posição dos elementos da lista.

No caso acima, estamos printando o nome “Luke”, que ocupa a posição 1 na nossa lista.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
for nome in jedi:  
    print(nome)
```

Podemos utilizar um loop for para conseguirmos iterar por toda a nossa lista.

Nesse caso ele compreenderá que os valores presentes na lista *jedi* representam o nosso *intervalo* e a variável contadora *nome* assumirá cada um dos valores da lista, um por volta.

Dessa forma, quando a linha do *print* for executada, a variável contadora conterá (a cada volta) um dos nomes da lista.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
for nome in jedi:  
    print(nome)  
  
jedi.append("Mace Windu")  
  
for nome in jedi:  
    print(nome)
```

Para incluirmos um valor no FINAL da lista, basta utilizarmos o método *append*, indicando entre parênteses qual valor será inserido.

Com o script acima podemos comprovar que o nome “Mace Windu” não pertencia originalmente à lista e foi inserido no final.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
for nome in jedi:  
    print(nome)  
  
jedi.insert(2, "Luminara")  
  
for nome in jedi:  
    print(nome)
```

Para incluirmos um valor em uma posição específica da lista, basta utilizarmos o método *insert*, indicando entre parênteses qual a posição e o valor que será inserido.

Tenha em mente que esse método faz com que os índices dos outros valores sejam alterados para manter a sequência. Dessa forma o valor “Obi-Wan” passará a ter índice 3.



```
. Jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
for nome in Jedi:  
    print(nome)  
  
+ Jedi.pop()  
  
for nome in Jedi:  
    print(nome)
```

Podemos remover um valor da lista ao usarmos o método *pop*.

Quando esse método é utilizado sem nenhum valor entre parênteses, o último valor da lista será removido.



```
. Jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin"]  
for nome in Jedi:  
    print(nome)  
  
+ Jedi.pop(1)  
  
for nome in Jedi:  
    print(nome)
```

Podemos remover um valor da lista ao usarmos o método *pop*.

Quando esse método é utilizado com um valor entre parênteses, apenas o elemento referente àquela posição será removido.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin", "Yoda"]  
for nome in jedi:  
    print(nome)
```

```
contagem = jedi.count("Yoda")  
print("Nessa lista o nome Yoda aparece {} vezes".format(contagem))
```

É possível contar quantas vezes determinado elemento aparece em uma lista através do uso do método *count*.

Basta indicar, entre parênteses, o elemento que se deseja contar.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin", "Yoda"]  
for nome in jedi:  
    print(nome)
```

```
jedi.sort()  
for nome in jedi:  
    print(nome)
```

É possível ordenar uma lista em ordem crescente ou alfabética utilizando o método `sort()`.



```
jedi = ["Yoda", "Luke", "Obi-Wan", "Anakin", "Yoda"]  
for nome in jedi:  
    print(nome)
```

```
jedi.reverse()  
for nome in jedi:  
    print(nome)
```

É possível inverter uma lista usando o método *reverse()*.

Que tal aplicarmos esses recursos que as listas nos dão em um caso prático?

Crie um script chamado `listas_ex01.py`, onde criaremos um programa para que um professor seja capaz de digitar todas as notas de uma turma.

Ao final ele deve ser capaz de exibir as notas em ordem crescente e saber quantos alunos tiraram 10.



```
notas = []
encerrar = "NÃO"

while "N" in encerrar.upper() :
    notas.append(float(input("Por favor, digite uma nova nota")))

    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))

notas.sort()
notas.reverse()

print("NOTAS DA TURMA:")
for nota in notas:
    print(nota)
```

Iniciamos o programa criando uma lista vazia. O Python sabe que trata-se de uma lista pois utilizamos o símbolo de colchetes



```
notas = []
encerrar = "NÃO"

while "N" in encerrar.upper() :
    notas.append(float(input("Por favor, digite uma nova nota")))

    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))

notas.sort()
notas.reverse()

print("NOTAS DA TURMA:")
for nota in notas:
    print(nota)
```

Para que o professor possa controlar quantas notas deseja digitar, criamos um loop While que é executado enquanto ele digitar NÃO após digitar uma nota.



```
notas = []
encerrar = "NÃO"

while "N" in encerrar.upper() :
    notas.append(float(input("Por favor, digite uma nova nota")))

    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))

notas.sort()
notas.reverse()

print("NOTAS DA TURMA:")
for nota in notas:
    print(nota)
```

A cada nota digitada realizamos um *append*, para que ela seja incluída no final da nossa lista de notas.



```
notas = []  
encerrar = "NÃO"
```

```
while "N" in encerrar.upper() :  
    notas.append(float(input("Por favor, digite uma nova nota")))  
  
    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")
```

```
print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))
```

```
notas.sort()  
notas.reverse()
```

```
print("NOTAS DA TURMA:")  
for nota in notas:  
    print(nota)
```

O método *count* é usado para contar quantas notas 10 foram digitadas e printarmos essa informação na tela.



```
• • • • • +
• • • notas = []
• + • encerrar = "NÃO"

+ • while "N" in encerrar.upper() :
•   notas.append(float(input("Por favor, digite uma nova nota")))

   encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

| print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))
+

notas.sort()
notas.reverse()

print("NOTAS DA TURMA:")
for nota in notas:
    print(nota)
```

Para colocarmos a lista em ordem decrescente, podemos utilizar um sort e depois um reverse.
Uma solução ainda mais elegante é utilizarmos apenas o método sort e entre parênteses ativarmos a ordenação decrescente. Assim: `notas.sort(reverse=True)`



```
notas = []
encerrar = "NÃO"

while "N" in encerrar.upper() :
    notas.append(float(input("Por favor, digite uma nova nota")))

    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

print("Ao total, {} alunos tiraram nota 10!".format(notas.count(10)))

notas.sort()
notas.reverse()

print("NOTAS DA TURMA:")
for nota in notas:
    print(nota)
```

Por fim, printamos cada uma das notas presentes na nossa lista.



PROGRAMANDO EM PYTHON

TRABALHANDO
COM TUPLAS



- • • • • +
 - • • • • •
 - + •
 - + •
- As *lists* são incríveis, mas foram criadas para um cenário bem específico: trabalhar com listas de dados que serão modificados ao longo do programa.

Para fazer isso, elas acabam ocupando bastante espaço em memória.

Vamos fazer um teste?

Abra o terminal do seu sistema operacional:

```
C:\Users\Emerson Abraham>python
Python 3.10.6 (tags/v3.10.6:9c7b4bd, Aug  1 2022, 21:53:49) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> lista = []
>>> sys.getsizeof(lista)
56
>>>
```

Dentro do interpretador vamos criar uma lista vazia e descobrir seu tamanho.

Primeiro digitaremos o comando *import sys* e depois teclamos ENTER.

Para criar a lista, digitamos *lista = []* e depois teclamos ENTER.

Para saber seu tamanho, digitamos *sys.getsizeof(lista)* e teclamos ENTER

Essa pequena experiência nos revelou que uma lista VAZIA ocupa 56bytes da nossa memória.

Pode parecer pouco, mas é importante lembrar que placas de prototipação capazes de rodarem python, como a ESP-8266 (muito usada em projetos de IoT), possuem memória limitadíssima.

É importante, portanto, sermos inteligentes no uso dos nossos recursos.

Uma lista só é *gastona* porque ela pode ser modificada de diferentes formas durante a execução do programa (adição ou remoção de elementos, ordenação, etc).

Para casos de dados que não serão modificados durante a execução do script temos uma estrutura mais adequada: **a tupla (*tuple*)**.

A tupla pode ser entendida como uma lista que não pode sofrer alterações.

Vamos conhece-la melhor? Crie um script chamado **tuplas.py**



```
nome_da_tupla = (valor1, valor2, valor3, valor4)
```

A sintaxe de criação de uma tupla é feita através da indicação do nome da nova tupla e do símbolo de () (parênteses).

Quando listamos uma sequência de valores dentro do símbolo de parênteses, cada um desses valores ocupa uma posição dentro da tupla que está sendo criada.



```
categorias_jedi = ("Youngling", "Padawan", "Knight", "Master")  
print(categorias_jedi)
```

A criação de uma tupla para as categorias dos cavaleiros jedi (que são imutáveis) se assemelha muito a de uma lista: a única diferença é o símbolo de parênteses.



```
. categorias_jedi = ("Youngling", "Padawan", "Knight", "Master")  
print(categorias_jedi[2])
```

Também podemos exibir um item específico de uma tupla através do seu índice.



```
. categorias_jedi = ("Youngling", "Padawan", "Knight", "Master")  
  
print(categorias_jedi[-1])
```

Um índice negativo nos dá o último elemento da dupla.



```
categorias_jedi = ("Youngling", "Padawan", "Knight", "Master")
```

```
for categoria in categorias_jedi:  
    print(categoria)
```

É possível também criarmos um loop for que passe por todos os elementos de uma tupla.

Lembre-se: se um conjunto de valores puder ser alterado durante a execução de um script, as LISTAS são uma saída interessante.

Por outro lado, se esses valores não puderem ser alterados, as TUPLAS são um caminho melhor.

O uso das tuplas fará mais sentido à medida que avançarmos nosso estudo.



PROGRAMANDO EM PYTHON

TRABALHANDO
COM DICIONÁRIOS



- Agora que você já conhece as listas e as tuplas, possui duas novas ferramentas incríveis para trabalhar com dados.

|
+ Imagine que seu problema agora é permitir o cadastro de vários cavaleiros Jedi e as categorias a que pertencem.

Talvez o código a seguir resolvesse o problema (mas não da forma mais adequada)



```
#Criando as duas listas
```

```
personagens=[]
```

```
categorias=[]
```

```
#Executando um loop 10 vezes
```

```
for x in range(10):
```

```
    #pedindo que o usuário informe um nome e colocando na lista de personagens
```

```
    personagens.append(input("Informe o nome do personagem: "))
```

```
    #pedindo que o usuário informe a categoria e colocando na lista de categorias
```

```
    categorias.append(input("Informe a categoria do personagem: "))
```

```
#Executando um loop 10 vezes
```

```
for indice in range(10):
```

```
    #exibindo a cada volta o que está contido em um índice de personagens e categorias
```

```
    print("O personagem {} é um(a) {}".format(personagens[indice], categorias[indice]))
```

O código acima cria duas listas: uma para os personagens e outra para as categorias (categoria foi criada como lista para que possam ser adicionadas novas ao longo do programa).

O usuário digita um personagem e uma categoria e cada dado vai para uma lista.

Apesar de parecer uma boa solução, o código faz com que o PROGRAMADOR seja responsável por associar qual personagem pertence a qual categoria.

Semanticamente não há nada que relacione de fato as duas estruturas criadas.

É aí que o **dicionário** vem mostrar sua força.

- • • • • +
- • • • •
- + •
- + •
- **Dicionário** é uma estrutura do Python que funciona seguindo a lógica *chave-valor*.

|

+ Essa lógica permite que indiquemos uma *chave identificadora* e o *valor que corresponde a ela*, na mesma estrutura.

Ela é IMPORTANTÍSSIMA para o trabalho com dados e está presente em diversas estruturas que você encontrará.

•

• +

■ □ •

• •

•

• + • •

• • •



```
nome_do_dicionario = {chave1:valor1, chave2:valor2}
```

A criação de um novo dicionário é feita indicando o símbolo de {} (chaves).

Dentro do símbolo de abrir e fechar chaves, devemos indicar uma *key* e um *value*, separados por : (dois pontos).



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
               "Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
               "Dex": "Balconista"}
```

O dicionário acima contém a chave “Yoda” associada ao valor “Mestre Jedi”, a chave “Mace Windu” associada ao valor “Mestre Jedi” e assim sucessivamente.



```
. personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}  
  
+ print(jedi["R2-D2"])
```

Podemos imprimir o conteúdo de um valor presente em um dicionário indicando a chave desse valor específico.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for valor in personagens.values():  
    print(valor)
```

É possível exibir todos os valores presentes em um dicionário ao utilizarmos o método *values()* – que nos retorna todo o intervalo de valores – dentro de um *for*.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for chave in personagens.keys():  
    print(chave)
```

É possível exibir todas as chaves presentes em um dicionário ao utilizarmos o método `keys()` – que nos retorna todo o intervalo de chaves – dentro de um `for`.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

É possível exibir o dicionário completo utilizando o método *items()* – que nos retorna todo o intervalo de chaves e valores – dentro de um for.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
personagens["Darth Vader"] = "Sith"
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

É possível inserir um novo item no dicionário ao indicar o nome do dicionário, a chave entre colchetes e o novo valor após o sinal de igual.

É importante ressaltar que essa mesma estrutura SUBSTITUIRÁ os valores existentes caso a chave já esteja no dicionário



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

```
personagens.pop("R2-D2")
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

O método `pop` com a indicação de uma chave irá remover o item do dicionário.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

```
personagens.popitem()
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

O método *popitem* remove do dicionário o último item inserido.



```
personagens = {"Yoda": "Mestre Jedi", "Mace Windu": "Mestre Jedi",  
"Anakin Skywalker": "Cavaleiro Jedi", "R2-D2": "Dróide",  
"Dex": "Balconista"}
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

```
personagens.clear()
```

```
for chave, valor in personagens.items():  
    print("O personagem {} é da categoria {}".format(chave,  
valor))
```

O método *clear* apaga todos os itens de um dicionário.

Lembra do nosso script de notas?

Vamos tentar resolver o mesmo problema, mas pedindo que o professor insira o NOME do aluno e a nota que ele tirou, armazenando essas informações no nosso dicionário.

Crie o arquivo `dicionario_ex01.py`.



```
notas = {}
encerrar = "NÃO"

while "N" in encerrar.upper() :
    aluno = input("Por favor, digite o nome do aluno")
    nota = float(input("Por favor, digite a nota do aluno"))
    notas[aluno] = nota

    encerrar = input("Deseja FINALIZAR a digitação? S - SIM ou N - NÃO")

total = 0
for nota in notas.values() :
    if nota == 10.00:
        total = total + 1

print("Ao total, {} alunos tiraram nota 10!".format(total))

print("NOTAS DA TURMA:")
for aluno, nota in notas.items():
    print("O aluno {} tirou nota {}".format(aluno, nota))
```




TRABALHANDO COM JSON



Nos anos 2000 Douglas Crockford propôs um padrão legível a seres humanos que se baseie no formato atributo-valor.

Após algumas revisões ao longo das décadas, o formato JSON caiu no gosto dos desenvolvedores e hoje já se tornou obrigatório em qualquer aplicação de grande porte.

Para provar o quão legível é o formato JSON, vamos ver um conjunto de dados nesse padrão:



```
{
  "Clark Kent":{
    "Celular":"123456",
    "Email":"super@krypton.com"
  },
  "Bruce Wayne":{
    "Celular":"654321",
    "Email":"bat@caverna.com.br"
  }
}
```

Mesmo sem qualquer explicação, você é capaz de visualizar que o atributo Clark Kent possui como valor um objeto, que possui como valor o atributo celular e como valor o conteúdo 123456, além do atributo email que possui como conteúdo o valor o conteúdo super@krypton.com


Essa é a beleza do formato atributo-valor, e aí está uma das grandes forças do JSON.



PROGRAMANDO EM PYTHON



DICIONÁRIO PARA
JSON



Como nós já conhecemos uma estrutura do Python baseada no conceito de chave-valor, é fácil decidirmos qual estrutura usaremos para gerar nossos arquivos JSON: os dicionários!

Crie dentro do seu projeto um novo script, chamado *dicionario_para_json.py*



```
contatos = {  
    "Clark Kent":  
        {"Celular": "123456",  
         "Email": "super@krypton.com"},  
    "Bruce Wayne":  
        {"Celular": "654321",  
         "Email": "bat@caverna.com.br"}  
}
```

Para começarmos nosso código, vamos criar um dicionário de dicionários.

Perceba que o objeto *contatos* armazena um dicionário e que a chave *Clark Kent* está vinculado ao valor `{"Celular": "123456", "Email": "super@krypton.com"}` (que por sua vez é um dicionário, com as chaves *Celular* e *Email* e valores *123456* e *super@krypton.com*).

A chave “*Bruce Wayne*”, por sua vez, também tem um dicionário como valor.

Agora que temos um dicionário, precisamos de uma forma de convertê-lo para o formato json.

Quem nos ajudará com essa missão é um módulo nativo do Python, chamado... json!

Um módulo nada mais é do que um conjunto de métodos e funções que cumprem alguma tarefa em comum e para importa-los pra dentro do nosso script, usamos o comando *import*.



```
import json

contatos = {
    "Clark Kent":
        {"Celular": "123456",
         "Email": "super@krypton.com"},
    "Bruce Wayne":
        {"Celular": "654321",
         "Email": "bat@caverna.com.br"}
}

dados_json = json.dumps(contatos)
```

Para gravarmos nosso arquivo de texto com dados em formato JSON precisamos converter nosso dicionário para uma String.

Isso pode ser feito através da função *dumps()* que está presente no módulo json.

Dessa forma, nossa variável *dados_json* conterá uma String formatada como um JSON.



```
import json

contatos = {
    "Clark Kent":
        {"Celular": "123456",
         "Email": "super@krypton.com"},
    "Bruce Wayne":
        {"Celular": "654321",
         "Email": "bat@caverna.com.br"}
}

dados_json = json.dumps(contatos, indent=4)

print(dados_json)
```

Para garantirmos que nossa string contém o JSON com os espaçamentos que facilitam a leitura, podemos incluir o argumento *indent*, passando o valor 4 para que cada indentação tenha 4 espaços.

O print final é apenas para verificarmos como o conteúdo ficou depois de convertido.

Agora que aprendemos a converter nosso dicionário para o formato JSON, que tal se gravarmos dentro de um arquivo de texto?

Para fazermos isso basta usarmos a função *open* no modo de gravação de arquivos (veremos em mais detalhes na próxima aula), escrever o conteúdo da variável que contém nosso JSON e fechar o arquivo.



```
import json

contatos = {
    "Clark Kent":
        {"Celular": "123456",
         "Email": "super@krypton.com"},
    "Bruce Wayne":
        {"Celular": "654321",
         "Email": "bat@caverna.com.br"}
}
dados_json = json.dumps(contatos, indent=4)

arquivo = open("c:\\arquivos\\agenda.json", "w")
arquivo.write(dados_json)
arquivo.close()
```


Usando a função *open()* juntamente com o argumento *w* somos capazes de criar um arquivo de texto novo, agora com a extensão *json*.

Com o método *write()* passando como argumento a variável *dados_json*, podemos gravar nosso conteúdo e, por fim, usar o *close()* para encerrar o arquivo.



PROGRAMANDO EM PYTHON

JSON PARA
DICIONÁRIO



É chegada a hora de realizarmos o processo inverso ao que acabamos de fazer:

Vamos abrir um arquivo de texto contendo uma estrutura JSON, ler seu conteúdo e converter para um dicionário.

Para fazer isso, vamos utilizar novamente o módulo *json*.

Crie o arquivo *json_para_dicionario.py*



```
import json
```

```
arquivo = open("c:\\arquivos\\agenda.json")  
conteudo_do_arquivo = arquivo.read()  
arquivo.close()
```

Já conhecemos as linhas acima e suas funções dentro do nosso script: importar o módulo json, abrir um arquivo de texto em modo de leitura e colocar dentro do objeto *arquivo*, utilizar o método *read()* para recuperar todo o conteúdo do arquivo e jogar na variável *string conteudo_do_arquivo* e, por fim, fechar o arquivo que foi aberto.



```
import json
```

```
arquivo = open("c:\\arquivos\\agenda.json")  
conteudo_do_arquivo = arquivo.read()  
arquivo.close()
```

```
agenda = json.loads(conteudo_do_arquivo)
```

Para converter uma string em formato JSON para um dicionário, devemos utilizar o método *loads()* do módulo *String*.

No script acima estamos convertendo o conteúdo da variável *conteudo_do_arquivo* (que está no formato JSON) para o formato dicionário, e armazenando em *agenda*.

BIBLIOGRAFIA BÁSICA

- BEAZLEY, David. *Python Essential Reference*. 2009.
- BHARGAVA, ADITYA Y. *Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos*. São Paulo: Ed. Novatec, 2017
- CORMEN, THOMAS H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002
- COSTA, Sérgio Souza. *Recursividade*. Professor Adjunto da Universidade Federal do Maranhão.
- DOWNEY, ALLEN B. *Pense em Python. Pense como um cientista da computação*. São Paulo: Ed. Novatec, 2016
- GRANATYR, Jones; PACHOLOK, Edson. *IA Expert Academy*. Disponível em: <https://iaexpert.academy/>
- KOPEC, DAVID. *Problemas clássicos de ciência da computação com Python*. São Paulo: Ed. Novatec, 2019
- LINDEN, Ricardo. *Algoritmos Genéticos*. 3 edição. Rio de Janeiro: Editora Moderna, 2012
- MCKINNEY, WILLIAM WESLEY. *Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython*. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. **Estrutura de Dados Usando C**. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. **Estruturas de Dados**. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. **Estruturas de Dados: Conceitos e Tecnicas de implementacao**. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. **Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java**. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. **Lógica de programação e estruturas de dados**. 2016.
- SILVA, Osmar Quirino. **Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações**. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. **Projeto de algoritmos com implementações em pascal e C**. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP

Copyright © 2023 | Professor Dr. Emerson R. Abraham

Todos os direitos reservados. A reprodução ou divulgação total ou parcial deste documento é expressamente proibida sem o consentimento formal, por escrito, do professor/autor.

