LINGUAGEM DE PROGRAMAÇÃO

AULA 4

Prof. Wellington Rodrigo Monteiro

CONVERSA INICIAL

Olá! Quando trabalhamos com algoritmos, é importante destacar **para que** eles servem. Afinal, por que alguém ficaria horas e horas produzindo várias linhas em uma linguagem que não aprendemos no berço? Qual seria a **utilidade** disso? Dependendo de suas experiências e de seus contatos, você pode ter se deparado com frases como “desenvolvemos uma **solução** de vendas”, “temos um aplicativo (ou aplicação) de internet *banking*”, “entrei em uma plataforma de jogos”.

Veja que esses termos descrevem coisas nas quais trabalhamos e/ou nas quais vemos sentido. São **coisas que resolvem problemas**. Você dificilmente instalará um aplicativo de internet *banking* sem nem ao menos ter uma conta no banco: você terá o aplicativo para poder **resolver seus problemas**: pagamento de contas, transferências bancárias, entre outros. Você também não entrará em uma plataforma de jogos se ela não tiver algo de seu interesse, nem se você não quiser resolver o seu problema de (falta de) diversão por meio dela. Você também não entrará em uma loja virtual se ela não suprir sua necessidade de comprar um determinado produto.

Desse modo, não fazemos algoritmos apenas para rodar em nosso computador: eles servem para nos comunicarmos com pessoas e com outros sistemas. Como você deve imaginar, vivemos em um emaranhado de algoritmos, sistemas, soluções: uma compra naquela plataforma de jogos precisa “conversar” (guarde este termo nada técnico) com um sistema de autenticação para saber que você é você. Ela também precisa conversar com o sistema do banco para poder receber pagamentos; precisa conversar com o seu e-mail para enviar notificações e recibos de compra, e assim por diante.

É nesse sentido que chegamos a esta aula: vários destes sistemas podem ser escritos em linguagens diferentes e com algoritmos diversos. Isto é: um deles pode ter sido desenvolvido por alguém em Python, no Brasil, em 2022. Outro, por alguém que fez em C#, em outra empresa, na Romênia, em 2020. Outro, em Swift, nos EUA, em 2019. Independentemente de quem ou de qual empresa fez, é importante que esses sistemas tenham uma forma comum de comunicação – algum tipo de arquivo no qual os sistemas possam intermediar uma conversa. É como se um brasileiro conversasse com um russo: é provável que ambos acabem utilizando o inglês para que os dois se entendam. No mundo dos

algoritmos, há alguns tipos de arquivo que podemos utilizar para trafegar (e armazenar) dados. Trabalharemos alguns dos mais utilizados atualmente.

TEMA 1 – XML E JSON

Você já deve ter visto alguma coisa escrita em HTML, certo? O HTML é usado para mostrar páginas de sites em um navegador, e possui uma sintaxe parecida com esta:

```
<html>
  <head>
    <title>Título da página</title>
  </head>
  <body>
    <div>
        <p>Texto.</p>
    </div>
  </body>
</html>
```

O HTML foi desenvolvido há cerca de três décadas, e é um padrão usado até hoje. Alguns anos depois, o XML foi desenvolvido e possui uma sintaxe bem parecida (ao menos visualmente) em relação ao HTML. Por outro lado, enquanto o HTML é utilizado para visualizar sites, o XML é utilizado atualmente para transportar dados entre diferentes sistemas, como documentos, entre outros.

Mas nem tudo são flores: se estamos comunicando um volume alto de dados, o XML pode se tornar uma opção pesada. Abordaremos esse tema a seguir, mas é para isso que uma segunda alternativa também é utilizada nas integrações entre sistemas: o JSON, um outro tipo de arquivo.

Vamos fazer uma analogia: imagine que o XML é um trabalho que você **precisa** entregar nas normas da ABNT, possui várias seções além do corpo do trabalho produzido e, como resultado, um alto volume de páginas. Já o JSON é o mesmo arquivo, mas sem capa, contracapa, folha de rosto e demais seções, com exceção do corpo do trabalho.

Vamos a um exemplo prático: queremos, antes de mais nada, entender as diferenças entre um JSON e um XML. Em seguida, veremos como trabalhar com esses dados em Python.

Imaginemos dois exemplos. No primeiro, temos uma tabela contendo vários detalhes de um grupo de estudantes de uma universidade. Vamos supor que a instituição tenha dois sistemas diferentes, um deles é responsável pela

aplicação das provas aos estudantes e o outro é responsável pela gestão das notas e do cadastro dos estudantes. Como o primeiro sistema **calcula** as notas das provas, ele naturalmente precisa também enviar essas notas ao segundo sistema para o devido registro e armazenamento.

Considerando esses pontos, observe a Tabela 1. Ela possui quatro colunas: a primeira possui o código do estudante; a segunda possui o código da disciplina; a terceira contém a data da última atualização da nota; e, finalmente, a quarta coluna possui a nota da prova.

Tabela 1 – Informações dos estudantes

Código do estudante	Disciplina	Data de atualização	Nota
10000144	25000	2021-12-01	5.4
10000145	54000	2021-06-20	5.7
10000146	54050	2021-06-25	7.9
10000147	25020	2021-09-30	9.5
10000148	25020	2021-04-15	10.0
10000149	25020	2021-05-01	10.0
10000149	54000	2021-12-15	5.0
10000149	54050	2021-12-15	3.2

Agora, pensemos em um segundo exemplo: um site de apostas precisa conectar-se automaticamente ao sistema da Federação de Futebol da Mongólia para obter a tabela do campeonato da primeira divisão de futebol daquele país. Algumas informações compõem essa tabela, como a colocação da equipe na tabela, o nome da equipe, a quantidade de partidas jogadas, entre outros.

Tabela 2 – Informações dos times da Federação de Futebol da Mongólia

Posição	Equipe	Partidas	Vitórias	Empates	Derrotas	Gols pró	Gols contra	Saldo de gols	Pontos
1	Athletic 220	9	7	0	2	24	8	16	21
2	Falcons	9	6	1	2	21	10	11	19
3	Ulaanbaatar	9	5	3	1	17	5	12	18
4	Khaan Khuns – Erchim	9	5	2	2	16	6	10	17
5	Deren	9	5	1	3	17	10	7	16
6	Ulaanbaatar City	9	4	2	3	16	13	3	14
7	Khangarid	9	3	1	5	17	27	-10	10
8	Khoromkhon	9	2	1	6	11	25	-14	7
9	BCH Lions	9	2	0	7	14	29	-15	6
10	UB Mazaalaynuud	9	0	1	8	2	22	-20	1

Para trabalharmos com esses dados entre diferentes sistemas, é importante termos um *padrão* para trafegar. Em tese, até poderíamos criar um padrão nosso totalmente customizado para trafegar, mas isso não é muito bem-visto no mercado. Como trafegamos dados entre sistemas de diferentes equipes em uma empresa, e de diferentes empresas, é preferível ter um padrão que esses diferentes locais conheçam e saibam implementar. É melhor trabalharmos em padrões que já tenham bibliotecas (tanto em Python como em outras linguagens) do que reinventarmos algo do zero.

É também melhor trabalharmos com padrões de dados que diferentes pessoas possam trabalhar. Imagine que você acabou de entrar em uma nova empresa e, logo no primeiro dia, existem problemas de integração entre os sistemas. Ao analisar o código, você descobre que outra pessoa criou um padrão totalmente customizado, sem usar nenhuma documentação no código. Logo, você se vê no escuro, com uma bela pressão no ambiente de trabalho, para entender rapidamente o que as pessoas que vieram antes de você fizeram, para tentar resolver a situação. Por outro lado, se você já usasse um padrão como um XML ou um JSON, poderia rapidamente encontrar informações sobre eles na internet.

1.1 Arquivos XML

Um arquivo XML possui uma estrutura hierárquica em árvore. Logo, sempre possui um elemento principal (raiz, ou *root*) e, dentro dela, existem elementos filhos (*child*). Estes elementos filhos também podem ter mais filhos (*subchild*), e assim por diante. Observe o exemplo:

```
<?xml version='1.0' encoding='utf-8'?>
<root>
  <child>
    <subchild>5</subchild>
    <subchild>2</subchild>
  </child>
</root>
```

Se fôssemos colocar em estrutura de tópicos, teríamos algo assim:

1. Root
    - Child
        - Subchild: valor 5

- Subchild: valor 2

Note que **dentro** do *root* temos um outro elemento (chamado aqui de *child*). Dentro do *child* temos mais dois elementos: ambos chamados *subchild*, com valores diferentes entre si: um armazena o valor 5 e, o outro, o valor 2. Ou seja, dentro de um elemento podemos ter outros elementos ou valores.

Além disso, a primeira linha do XML sempre define a versão do arquivo (aqui, 1.0) e o *encoding* (no caso, UTF-8). Cremos que vale a pena uma pequena pausa para falarmos de *encoding*. É um assunto bem importante para o tratamento de dados.

1.1.1 *Encoding*

As informações trocadas entre softwares são armazenadas em *bits* e *bytes*. Partindo dessa premissa, dados como sons, textos, imagens e vídeos entram todos nesta regra. Naturalmente, isto se aplica também a letras, números e caracteres, como espaços, hífens, asteriscos e quebras de linha.

Nas primeiras gerações de computadores, cada empresa acabava um padrão diferente para “traduzir” os caracteres em *bytes* e, desses padrões de *encoding*, o ASCII se sobressaía. Observe a Tabela 3.

Tabela 3 – Tabela ASCII

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Fonte: ZZT32, 2007.

Note que o espaço é, “por baixo dos panos”, armazenado com o valor 32 (20, em hexadecimal). O “k”, em maiúsculo, é armazenado com o valor 775 (4B, em hexadecimal), o “n”, em minúsculo, é armazenado com o valor 110 (6E, em hexadecimal), e assim por diante. Por outro lado, essa tabela comporta até 128 caracteres, o que é um problema. Pensemos juntos: onde estão os acentos? O “ç”? O “á”, ou o “à”? E se precisássemos de caracteres que o francês ou o espanhol possuem? Ou a língua romena? Ou a russa? Ou a árabe? É por isso que, com o avançar dos anos, outros *encodings* foram implementados e são usados por padrão. Desses, os mais usados hoje são:

1. UTF-8;
2. ISO-8859-1; e
3. Windows-1252.

Todos esses três utilizam o ASCII como base, mas vão além: lembra-se de que a tabela ASCII vai até o número 127? O ISO-8859-1 vai até 255, e inclui vários caracteres com acentos dos idiomas italiano, espanhol e português, por exemplo. O Windows-1252 modifica algumas coisas do ISO-8859-1 e inclui alguns caracteres, como as aspas curvas (“) e (”). Por outro lado, há alguns anos, outro padrão se sobressaiu: o UTF-8, que comporta até 1112064 caracteres, logo, alfabetos como o grego, árabe, cirílico, japonês, coreano, chinês e até os *emojis* estão dentro desse padrão. Logo, se você conseguiu ler corretamente o *emoji* 🙇, você está lendo um texto corretamente processado em UTF-8.

É importante você saber desses padrões porque *nem todos* trabalham com UTF-8. Quando lemos um texto em um *encoding* diferente daquele projetado originalmente, teremos alguns erros. Por exemplo: o texto “*Vô Zé*” em UTF-8 pode aparecer como “*VÃµ ZÃ©*” se você tentar ler este texto como se ele estivesse em Windows-1252.

Observe que essa primeira linha não existe só pelas aparências: saber o *encoding* correto é **muito** importante para trabalharmos com esses arquivos – independentemente se ele é um XML ou qualquer outro tipo de arquivo. Voltemos ao XML.

1.1.2 Exemplos de XML

Veja como ficaria a tabela das notas dos estudantes no formato XML. Para fins ilustrativos, somente as cinco primeiras linhas são mostradas:

```
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <codigoestudante>10000144</codigoestudante>
    <disciplina>25000</disciplina>
    <dataatualizacao>2021-12-01 00:00:00</dataatualizacao>
    <nota>5.4</nota>
  </row>
  <row>
    <codigoestudante>10000145</codigoestudante>
    <disciplina>54000</disciplina>
    <dataatualizacao>2021-06-20 00:00:00</dataatualizacao>
    <nota>5.7</nota>
  </row>
  <row>
    <codigoestudante>10000146</codigoestudante>
    <disciplina>54050</disciplina>
    <dataatualizacao>2021-06-25 00:00:00</dataatualizacao>
    <nota>7.9</nota>
  </row>
  <row>
    <codigoestudante>10000147</codigoestudante>
    <disciplina>25020</disciplina>
    <dataatualizacao>2021-09-30 00:00:00</dataatualizacao>
    <nota>9.5</nota>
  </row>
  <row>
    <codigoestudante>10000148</codigoestudante>
    <disciplina>25020</disciplina>
    <dataatualizacao>2021-04-15 00:00:00</dataatualizacao>
    <nota>10.0</nota>
  </row>
</data>
```

Agora, veja como ficaria a tabela do campeonato da Mongólia em XML. Somente os dois primeiros colocados são mostrados (para fins ilustrativos):

```
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <posicao>1</posicao>
    <equipe>Athletic 220</equipe>
    <partidas>9</partidas>
    <vitorias>7</vitorias>
    <empates>0</empates>
    <derrotas>2</derrotas>
    <golspro>24</golspro>
    <golscontra>8</golscontra>
    <saldogols>16</saldogols>
    <pontos>21</pontos>
  </row>
  <row>
    <posicao>2</posicao>
    <equipe>Falcons</equipe>
    <partidas>9</partidas>
    <vitorias>6</vitorias>
    <empates>1</empates>
    <derrotas>2</derrotas>
    <golspro>21</golspro>
    <golscontra>10</golscontra>
    <saldogols>11</saldogols>
    <pontos>19</pontos>
  </row>
</data>
```

Vamos observar algumas semelhanças entre estes exemplos.

1. Todos possuem aquela linha inicial declarando o *encoding* e que, de fato, é um arquivo XML.
2. Veja que todos possuem *tags*. Por coincidência, ambos possuem um elemento principal (*data*) e, dentro desse elemento, existem outros elementos (*row*), contendo os dados de uma das linhas das tabelas que vimos anteriormente.
3. Sabemos quando começa e quando termina um elemento ao olharmos as *tags*: por exemplo, é possível saber quando um determinado *row* começa ao procurarmos por um “<row>”, e sabemos quando ele termina ao procurarmos pelo **próximo** “</row>”.
4. Dentro de cada *row* temos os dados de cada um dos seus atributos. No caso do campeonato da Mongólia, todo *row* possui os dados específicos de uma equipe, como o seu nome (elemento *equipe*); posição (elemento *posicao*); saldo de gols (elemento *saldogols*), entre outros. Esses

elementos se repetem a cada *row* – ou seja, dentro de um único arquivo XML podemos ter vários elementos com o nome *saldogols*, mas cada um desses elementos refere-se a uma equipe em específico.

5. Veja que os nomes das *tags* **não possuem espaços**: em vez de “saldo de gols”, tivemos que usar “saldogols”, sem espaços.
6. Veja que os valores ficam **entre** as *tags* de início e de fim de um elemento, e não precisamos de aspas: em <*equipe*>*Falcons*</*equipe*>, sabemos que o valor para o elemento “equipe” é “Falcons”, sem aspas.

Temos algumas opções para manipularmos arquivos XML em Python. Vamos pensar em três delas.

1. A biblioteca padrão do Python (lembra-se de que já conversamos sobre ela?) também possui um módulo específico para XML[1]. Observe que é possível ler tanto arquivos em XML como *strings* que possuem conteúdo em XML.
2. Outra alternativa é usar a biblioteca lxml[2]. Com ela, é possível acessar (ou criar) diferentes elementos como se o arquivo XML fosse uma lista. Dessa maneira, você consegue utilizar tudo o que aprendeu em oportunidades anteriores (como o *for, print* e *append*) também nessa biblioteca.
3. Além disso, o pandas também é capaz de ler[3] arquivos em XML e convertê-los em DataFrames, e o contrário também[4] (isto é, converter DataFrames em arquivos XML). Isso é muito útil quando necessitamos fazer modelagem de dados de forma rápida e integrar com APIs quando necessário.

1.2 Arquivos JSON

O JSON é utilizado na prática como uma alternativa mais “leve” do que o XML (lembra-se daquela analogia do trabalho nas normas da ABNT?). Naturalmente, ele apresenta vantagens e desvantagens. Anteriormente, trouxemos alguns exemplos básicos do XML. Por outro lado, ele possui suporte

[1] Disponível em: <https://docs.python.org/3/library/xml.etree.elementtree.html#module-xml.etree.ElementTree>. Acesso em: 21 dez. 2021.
2 Disponível em: <https://lxml.de/installation.html>. Acesso em: 21 dez. 2021.
3 Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.read_xml.html>. Acesso em: 21 dez. 2021.
4 Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_xml.html>. Acesso em: 21 dez. 2021.

para mais coisas, como atributos, comentários e metadados de forma separada dos dados, e vários desses itens não existem no JSON.

Um arquivo JSON é iniciado e finalizado por chaves (“{” e “}”), e sempre possui uma sequência de nomes e valores (você pode encontrar muitas referências sobre isso ao pesquisar, em inglês, por *key-value pair*). Os valores podem ser uma *string*, um número, uma lista, um booleano (true ou false), um objeto ou nulo (null). Além disso, separamos os *key-value pairs* com vírgulas.

Observemos, inicialmente, o caso a seguir. É exatamente o mesmo XML de exemplo que utilizamos anteriormente, mas agora convertido em JSON. Para relembrar, o XML era assim:

```
<?xml version='1.0' encoding='utf-8'?>
<root>
  <child>
    <subchild>5</subchild>
    <subchild>2</subchild>
  </child>
</root>
```

Já o JSON fica assim:

```
{
   "child": [
      5,
      2
   ]
}
```

Agora, relembremos dos XMLs que geramos para representar as tabelas dos estudantes e do campeonato da Mongólia: observe como **exatamente** ficariam aqueles dados, mas agora em JSON. Primeiro, os cinco estudantes que vimos, anteriormente, em XML.

```
{
	"data": [
		{
			"Código do Estudante": 10000144,
			"Disciplina": 25000,
			"Data de Atualização": "2021-12-01T00:00:00.000Z",
			"Nota": 5.4
		},
		{
			"Código do Estudante": 10000145,
			"Disciplina": 54000,
			"Data de Atualização": "2021-06-20T00:00:00.000Z",
			"Nota": 5.7
		},
		{
			"Código do Estudante": 10000146,
			"Disciplina": 54050,
			"Data de Atualização": "2021-06-25T00:00:00.000Z",
			"Nota": 7.9
		},
		{
			"Código do Estudante": 10000147,
			"Disciplina": 25020,
			"Data de Atualização": "2021-09-30T00:00:00.000Z",
			"Nota": 9.5
		},
		{
			"Código do Estudante": 10000148,
			"Disciplina": 25020,
			"Data de Atualização": "2021-04-15T00:00:00.000Z",
			"Nota": 10.0
		}
	]
}
```


Agora, os dois primeiros colocados do campeonato da Mongólia:

```
{
	"data": [
		{
			"Posição": 1,
			"Equipe": "Athletic 220",
			"Partidas": 9,
			"Vitórias": 7,
			"Empates": 0,
			"Derrotas": 2,
			"Gols Pró": 24,
			"Gols Contra": 8,
			"Saldo de Gols": 16,
			"Pontos": 21
		},
		{
			"Posição": 2,
			"Equipe": "Falcons",
			"Partidas": 9,
			"Vitórias": 6,
			"Empates": 1,
			"Derrotas": 2,
			"Gols Pró": 21,
			"Gols Contra": 10,
			"Saldo de Gols": 11,
			"Pontos": 19
		}
	]
}
```


Pensemos agora sobre a sintaxe desses arquivos.

1. Ambos começam e terminam com chaves.
2. Ambos possuem, primeiramente, um único *key-value pair*. O nome da chave é “data” e o seu valor é uma lista de itens. No caso dos estudantes, é uma lista com 5 elementos. Já no caso do campeonato, é uma lista com 2 elementos. Em ambos os casos, cada um dos elementos é composto por um conjunto de vários *key-value pairs*.
3. No caso dos estudantes, cada um desses elementos possui 4 *key-value pairs*, correspondentes aos nomes das colunas da tabela que vimos anteriormente.
4. No caso do futebol, cada um desses elementos possui 10 *key-value pairs*, também correspondentes aos nomes das colunas.
5. Os nomes das chaves aceitam acentos e espaços, ao contrário do XML. Por exemplo:

- em *"Equipe" – "Falcons"* o valor está em aspas duplas porque se trata de uma *string*; e
- em *"Gols Pró" – 21* o valor está sem aspas duplas porque se trata de um número.

6. Cada *key-value pair* é separado por vírgulas, ou seja, após uma vírgula, sabemos que um novo *key-value pair* começa.

Em Python também é possível trabalhar com arquivos em JSON. Há algumas opções.

1. A biblioteca padrão do Python também possui um módulo específico para JSON[5]. Dessa biblioteca, destacamos principalmente duas funções: a dumps[6], que converte uma variável (como um dicionário) em Python para um JSON; e a loads[7], que faz o contrário: converte um JSON que está em uma *string* para uma variável em Python.

2. O pandas também pode manipular arquivos JSON. Da mesma forma que opera com XML, ele também consegue ler[8] arquivos em JSON e convertê-los em DataFrames, bem como salvar[9] DataFrames em JSON.

TEMA 2 – CSV E PARQUET

Discutimos sobre *encoding* e sobre a transação de dados entre sistemas de diferentes locais (por *local*, entendemos organizações, empresas ou órgãos governamentais). Por outro lado, o XML e o JSON são úteis quando esse tipo de comunicação entre sistemas existe em primeiro lugar, isto é, quando existem integrações (chamadas de APIs) entre dois sistemas que possibilitam o tráfego desse tipo de arquivo.

Contudo, há casos nos quais não existem APIs construídas; logo, você não consegue acessar os dados em XML e JSON. Existem vários sites

[5] Disponível em: <https://docs.python.org/3/library/json.html>. Acesso em: 21 dez. 2021.
[6] Disponível em: <https://docs.python.org/3/library/json.html#json.dumps>. Acesso em: 21 dez. 2021.
[7] Disponível em: <https://docs.python.org/3/library/json.html#json.loads>. Acesso em: 21 dez. 2021.
[8] Disponível em: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.io.json.read_json.html#pandas.io.json.read_json>. Acesso em: 21 dez. 2021.
[9] Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_json.html>. Acesso em: 21 dez. 2021.

(principalmente governamentais) nos quais só conseguimos acessar os dados a partir do *download* de uma planilha em Excel. E o que fazemos nesses casos?

Ou, ainda, existem casos nos quais temos um banco de dados na mesma empresa para a qual estamos construindo uma solução, mas não temos acesso **direto** a este banco de dados. Nessas situações, não é raro uma pessoa baixar uma massa de dados e disponibilizar para nossa manipulação.

Para ambos os casos entram dois atores: o CSV e o parquet.

2.1 Arquivos CSV

CSV significa *comma-separated values*, ou seja, "valores separados por vírgulas". Lembra-se da tabela das notas dos estudantes? Ela foi criada no Excel. Logo, uma pessoa sem experiência em TI só iria conseguir *abrir* aquele arquivo em seu computador se você tivesse o Excel (ou algum software similar) instalado em seu computador. Observe como fica aquela mesma tabela em CSV:

```
Código do Estudante,Disciplina,Data de Atualização,Nota
10000144,25000,2021-12-01,5.4
10000145,54000,2021-06-20,5.7
10000146,54050,2021-06-25,7.9
10000147,25020,2021-09-30,9.5
10000148,25020,2021-04-15,10
10000149,25020,2021-05-01,10
10000149,54000,2021-12-15,5
10000149,54050,2021-12-15,3.2
```

Um arquivo CSV nada mais é do que um conteúdo armazenado em um arquivo de texto (desses que podemos abrir no bloco de notas) cujo ***separador*** dos dados é a vírgula. Ou seja: qualquer software que for ler este arquivo (como um algoritmo feito em Python ou o Excel) saberá que uma coluna começa e a outra termina quando encontra uma vírgula. Por outro lado, temos um problema: o português brasileiro não usa o ponto como separador de decimais, mas, sim, vírgulas. Logo, para representar "5 inteiros e 4 décimos" não usamos "5.4", mas sim "5,4". E aí? Como vamos usar a vírgula como separador sem confundir tudo? É para esses casos que existem variações, e o separador pode ser o ponto-e-vírgula, como mostrado a seguir:

```
Código do Estudante;Disciplina;Data de Atualização;Nota
10000144;25000;2021-12-01;5,4
10000145;54000;2021-06-20;5,7
10000146;54050;2021-06-25;7,9
10000147;25020;2021-09-30;9,5
10000148;25020;2021-04-15;10
10000149;25020;2021-05-01;10
10000149;54000;2021-12-15;5
10000149;54050;2021-12-15;3,2
```

Ainda podem ser definidos outros separadores, como a tabulação (Tab, ou "\t").

Por outro lado, como leríamos arquivos assim em Python?

1. Como não poderia deixar de ser, a biblioteca padrão do Python também possui um módulo[10] para CSV. Veja que é possível informarmos o delimitador[11] entre outros parâmetros.
2. Outra alternativa é – adivinhe – o pandas. Temos também opções para ler[12] e escrever[13] arquivos CSV. Veja que existem parâmetros para especificar o delimitador, o separador de decimal, o *encoding*, entre outros.

2.2 Arquivos parquet

É muito simples trabalharmos em CSV quando o volume de dados é baixo: afinal, é fácil de ler e escrever, e é um arquivo único, possibilitando enviá-lo de um lugar para outro. Por outro lado, o CSV não é uma opção tão interessante quando o volume de dados é alto. Ler um único arquivo CSV de 10 GB seria bem pesado e, dependendo do computador, poderia travar tudo. Pensemos ainda no dia a dia – será que precisaríamos de **todas** as colunas **todas** as vezes que formos trabalhar com este arquivo? Provavelmente a resposta seria **não**.

Para endereçar esse tipo de material existe o formato parquet. Pensado em *big data*, é um formato orientado a colunas, e não a linhas. O que isso

[10] Disponível em: <https://docs.python.org/3/library/csv.html>. Acesso em: 21 dez. 2021.
[11] Disponível em: <https://docs.python.org/3/library/csv.html#csv-fmt-params>. Acesso em: 21 dez. 2021.
[12] Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html>. Acesso em: 21 dez. 2021.
[13] Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html>. Acesso em: 21 dez. 2021.

significa? Imagine que cada uma dessas colunas (isto é, cada cor), seja armazenada em **um arquivo individual e separado dos demais**.

Tabela 3 – Colunas (arquivo parquet)

Código do estudante	Disciplina	Data de atualização	Nota
10000144	25000	2021-12-01	5.4
10000145	54000	2021-06-20	5.7
10000146	54050	2021-06-25	7.9
10000147	25020	2021-09-30	9.5
10000148	25020	2021-04-15	10.0
10000149	25020	2021-05-01	10.0
10000149	54000	2021-12-15	5.0
10000149	54050	2021-12-15	3.2

3. Há algumas opções para manipular o parquet. Dessas, uma das principais é o pandas. De forma análoga aos outros arquivos que vimos anteriormente, existem duas funções principais: uma de leitura[14] e outra de escrita[15]. É importante destacar aqui o armazenamento. Ao contrário do CSV (e, quem sabe, do XML e do JSON), não conseguimos visualizar o resultado ao abrir o arquivo gerado no bloco de notas, pois temos como resultado um conjunto de arquivos que são compactados dentro de um arquivo só.

TEMA 3 – SQLITE

Sistemas de bancos de dados relacionais são pensados geralmente como soluções que rodam de forma separada de um site ou de outros servidores que compõem uma infraestrutura mais complexa. Em outras palavras, em grandes aplicações temos o banco de dados sendo executado de forma segregada do *back-end* e este, por sua vez, rodando de forma segregada em relação ao *front-end*.

Esses sistemas de bancos de dados relacionais geralmente possuem como foco a segurança dos dados, escalabilidade e controle. Por outro lado, nem

[14] Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.read_parquet.html>. Acesso em: 21 dez. 2021.
[15] Disponível em: <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_parquet.html>. Acesso em: 21 dez. 2021.

sempre queremos (ou podemos) nos conectar a sistemas de bancos de dados assim, pois existem casos em que queremos (ou estamos limitados a) rodar um banco de dados de forma **integrada** ao nosso algoritmo. Exemplos disso incluem:

1. casos em que não temos garantia de que teremos uma conexão com um servidor contendo um banco de dados, como eletrodomésticos, robôs, *videogames* e outros equipamentos que fazem parte da internet das coisas (IoT);
2. sites com baixo movimento;
3. cache para dados corporativos;
4. bancos de dados temporários ou de uso interno;
5. treinar ou aprender sobre bancos de dados relacionais; e

4. para essas situações, existe o SQLite.

O próprio site do SQLite[16] argumenta que ele não deve ser entendido como um **substituto** dos sistemas de banco de dados que usam SQL, mas, sim, como uma alternativa para casos como os exemplos citados: situações nas quais nem sempre esses sistemas seriam uma boa opção.

3.1 SQLite

O SQLite usa a linguagem SQL como base: logo, comandos como *CREATE TABLE, SELECT, JOIN, WHERE, DELETE, INSERT* e outros funcionam aqui.

A biblioteca padrão do Python também possui um módulo[17] para trabalharmos com o SQLite. Ao trabalharmos com o SQLite, salvaremos os dados em um arquivo, o qual armazenará as tabelas e dados nela contidos.

Saiba mais

É importante que você saiba que o SQLite não faz parte do Python: várias outras linguagens como C#, Java, Swift, PHP, Kotlin, entre outros, também possuem suporte ao SQLite. Isso se aplica também aos outros formatos de dados que vimos, como o parquet e o XML.

[16] Disponível em: <https://www.sqlite.org/whentouse.html>. Acesso em: 21 dez. 2021.
[17] Disponível em: <https://docs.python.org/3/library/sqlite3.html>. Acesso em: 21 dez. 2021.

O exemplo a seguir cria uma nova base de dados em um arquivo chamado “aula4.db”. Depois, criamos duas tabelas: uma com os dados dos estudantes, e outra, do campeonato. Utilizaremos as duas tabelas para inserirmos os dados que trabalhamos durante essa aula. Observe que algumas colunas possuem aspas; quando as colunas possuem espaços nos nomes, é essencial que as delimitemos com aspas para que o SQLite não tenha erros de sintaxe.

```
import sqlite3
con = sqlite3.connect(path + 'aula4.db')

cur = con.cursor()

# criando a tabela dos estudantes
# tipos de dados obtidos em https://www.sqlite.org/datatype3.html
cur.execute('''CREATE TABLE estudante
               ("Código do Estudante" integer, Disciplina text,
               "Data de Atualização" text, Nota real)''')

# criando a tabela do campeonato
# tipos de dados obtidos em https://www.sqlite.org/datatype3.html
# espaços não são permitidos para os nomes das colunas
cur.execute('''CREATE TABLE campeonato
               (Posição integer, Equipe text, Partidas integer, Vitórias integer,
               Empates integer, Derrotas integer, "Gols Pró" integer,
               "Gols Contra" integer, "Saldo de Gols" integer, Pontos integer)''')
```

Agora, vamos fazer as inserções. Um exemplo de como fazer isso é pelo INSERT INTO que conhecemos no SQL. Utilizando a primeira linha das duas tabelas que utilizamos nessa aula teríamos:

```
# inserindo uma linha por tabela
# (no caso, a primeira linha dos dataframes que vimos até agora)
cur.execute("INSERT INTO estudante VALUES (10000144, 25000, '2021-12-01', 5.4)")
cur.execute("INSERT INTO campeonato VALUES (1, 'Athletic 220', 9, 7, 0, 2, 24, 8,
16, 21)")
```

Outra possibilidade é inserir diretamente via pandas. Como já inserimos a primeira linha dos dois DataFrames, pularemos essa primeira linha para ambos os casos com o *iloc*. No caso, *iloc[1:]* significa “vamos pegar todas as linhas do DataFrame a partir da posição 1”. Como a contagem sempre começa em 0, a posição 1 é a segunda linha.

```
# inserindo a partir da segunda linha (via iloc[1:]) de ambas as tabelas
df_aulas.iloc[1:].to_sql('estudante', if_exists='append', index=False, con=con)
df_fut.iloc[1:].to_sql('campeonato', if_exists='append', index=False, con=con)
```

Finalmente, salvamos todas essas operações (criação das tabelas e sua inserção) com o *commit*:

```
# salvando os inserts
con.commit()
```

Depois disso, vamos selecionar todo o conteúdo das tabelas para ver se realmente foram salvas ou não.

```
# mostrando as linhas das tabelas via sqlite3
print('Tabela do campeonato:')
for row in cur.execute('SELECT * FROM campeonato'):
    print(row)

print('\r\nTabela dos estudantes:')
for row in cur.execute('SELECT * FROM estudante'):
    print(row)
```

```
Tabela do campeonato:
(1, 'Athletic 220', 9, 7, 0, 2, 24, 8, 16, 21)
(2, 'Falcons', 9, 6, 1, 2, 21, 10, 11, 19)
(3, 'Ulaanbaatar', 9, 5, 3, 1, 17, 5, 12, 18)
(4, 'Khaan Khuns - Erchim', 9, 5, 2, 2, 16, 6, 10, 17)
(5, 'Deren', 9, 5, 1, 3, 17, 10, 7, 16)
(6, 'Ulaanbaatar City', 9, 4, 2, 3, 16, 13, 3, 14)
(7, 'Khangarid', 9, 3, 1, 5, 17, 27, -10, 10)
(8, 'Khoromkhon', 9, 2, 1, 6, 11, 25, -14, 7)
(9, 'BCH Lions', 9, 2, 0, 7, 14, 29, -15, 6)
(10, 'UB Mazaalaynuud', 9, 0, 1, 8, 2, 22, -20, 1)

Tabela dos estudantes:
(10000144, '25000', '2021-12-01', 5.4)
(10000145, '54000', '2021-06-20 00:00:00', 5.7)
(10000146, '54050', '2021-06-25 00:00:00', 7.9)
(10000147, '25020', '2021-09-30 00:00:00', 9.5)
(10000148, '25020', '2021-04-15 00:00:00', 10.0)
(10000149, '25020', '2021-05-01 00:00:00', 10.0)
(10000149, '54000', '2021-12-15 00:00:00', 5.0)
(10000149, '54050', '2021-12-15 00:00:00', 3.2)
```

Por fim, fechamos a conexão com o arquivo para que não fique aberta. Como boa prática, é interessante fecharmos a conexão se não estivermos mais utilizando-a. Se precisarmos reutilizá-la, basta reabri-la.

```
## fechando a conexão com o arquivo
# se fizermos o con.close() sem o commit que fizemos antes
# perderemos todos os inserts que fizemos até agora
con.close()
```

TEMA 4 – PERSISTINDO AQUI

Você pode ter percebido que, até esse momento, mostramos algumas formas de trabalhar com arquivos. Isso é bem importante pela necessidade de termos algo para aproveitar depois, pois são poucos os casos nos quais visualizamos algo na tela para nunca mais consultar novamente. Em algum momento no passado, já digitamos essas palavras e clicamos em *Salvar* para que você pudesse ler depois. Em algum momento, seu saldo no banco foi atualizado em algum sistema para que você pudesse consultá-lo com segurança a qualquer momento do dia. Em algum momento, suas notas foram salvas no sistema da universidade para a instituição possa consultá-las no futuro e assim poder elaborar o seu diploma e, assim, sucessivamente.

Há duas formas de salvarmos (isto é, *persistirmos*) as informações: uma delas é no nosso próprio computador, o que chamamos de *persistência local*. Imagine uma foto que você acabou de tirar com seu celular, e que você não sincronizou com nenhum serviço da nuvem, como o Dropbox, Google Drive ou OneDrive. A foto está salva em seu celular e só existe nele. Se algo de ruim acontecer com o seu aparelho, você perderá essa foto para sempre; esse, é um caso de persistência local.

Existem algumas formas de armazenarmos os dados localmente. Os exemplos mais comuns incluem:

1. armazenamento em arquivos, como XML e JSON;
2. armazenamento em arquivos, como planilhas do Excel ou arquivos CSV e parquet;
3. armazenamento em bases de dados locais, como o SQLite;
4. armazenamento em arquivos de texto;
5. armazenamento em *pickle*.

4.1 Salvando dados localmente

Podemos também ler e escrever outros tipos de arquivo que não sejam somente tabelas, pois nem todos os dados que precisamos manipular podem (ou precisam) ser representados como DataFrames. Vamos supor que somente

queiramos salvar uma frase em um arquivo. Como faríamos? Felizmente, o Python também possui suporte para isso em sua biblioteca padrão[18].

Note que existem algumas funções que podem ser relevantes para as nossas aplicações.

1. open[19]: a partir da open() definimos qual é o arquivo que queremos utilizar – seja para ler seu conteúdo, seja para escrever dados nele. O segundo parâmetro dessa função é responsável por informar se desejamos ler o conteúdo do arquivo [mode='r', de "*read*" (leitura)], ou escrever nele [mode='w', de "*write*" (escrita, substituindo o conteúdo anterior)], ou incluir novos dados (mode='a', de *"append"* – escrita a partir do final do arquivo atual).
2. read[20]: com essa função, é possível ler qualquer conteúdo no arquivo. Por padrão, lemos *todo* o conteúdo do arquivo (use com sabedoria: às vezes o arquivo pode ser muito grande e essa função pode travar algumas coisas em seu computador, caso você não mude o parâmetro informando quantos caracteres deseja ler).
3. readline[21]: com essa função, lemos somente uma linha do conteúdo do arquivo. O readlines[22] é similar, e possibilita seu uso dentro de um *loop*, como um "for".
4. write[23]: escrevemos uma *string* no arquivo.

Saiba mais

Outras linguagens de programação oferecem também possibilidades similares; logo, não é somente no Python que podemos fazer essas operações com os arquivos.

[18] Disponível em: <https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>. Acesso em: 21 dez. 2021.
[19] Disponível em: <https://docs.python.org/3/library/functions.html#open>. Acesso em: 21 dez. 2021.
[20] Disponível em: <https://docs.python.org/3/library/io.html#io.TextIOBase.read>. Acesso em: 21 dez. 2021.
[21] Disponível em: <https://docs.python.org/3/library/io.html#io.TextIOBase.readline>. Acesso em: 21 dez. 2021.
[22] Disponível em: <https://docs.python.org/3/library/io.html#io.IOBase.readlines>. Acesso em: 21 dez. 2021.
[23] Disponível em: <https://docs.python.org/3/library/io.html#io.TextIOBase.write>. Acesso em: 21 dez. 2021.

Para ilustrar, consideraremos uma lista contendo somente os nomes das equipes do campeonato da Mongólia, sem duplicatas. O resultado é o *array* do NumPy a seguir:

```
lista_equipes = df_fut['Equipe'].unique()
lista_equipes
```

```
array(['Athletic 220', 'Falcons', 'Ulaanbaatar', 'Khaan Khuns - Erchim',
       'Deren', 'Ulaanbaatar City', 'Khangarid', 'Khoromkhon',
       'BCH Lions', 'UB Mazaalaynuud'], dtype=object)
```

Vamos salvar esse conteúdo em um arquivo chamado “lista_equipes.txt”. Veja que estamos no modo de escrita (“w”), e salvaremos cada um dos itens em uma linha desse arquivo do bloco de notas. O “\r\n” indica uma quebra de linha, no caso. Logo, é como se apertássemos “Enter” no teclado após digitar o nome de cada equipe no arquivo. Observe também o uso das funções “open” e “write”, que comentamos anteriormente:

```
# abrindo o arquivo para escrita (note o 'w')
file_equipes = open(path + 'lista_equipes.txt', 'w')

# escrevendo uma equipe por vez com uma quebra de linha no final
for equipe in lista_equipes:
    file_equipes.write(equipe + '\r\n')

# fechando a conexão com o arquivo e salvando as alterações
file_equipes.close()
```

O código para ler os dados desse mesmo arquivo que acabou de ser salvo é bem similar: em vez do modo de escrita (“w”), usamos o modo de leitura (“r”). Após abrir o arquivo, passamos linha-a-linha dele até seu final, com o uso do “readlines”. É por isso que adicionamos uma quebra de linha ao escrever as equipes no arquivo. O resultado desse código é o aparecimento de um nome de equipe por vez, como você deve imaginar.

```
# abrindo o arquivo para leitura (note o 'r')
file_equipes = open(path + 'lista_equipes.txt', 'r')

# lendo uma linha por vez
for linha in file_equipes.readlines():
    print(linha)

# fechando a conexão com o arquivo e salvando as alterações
file_equipes.close()
```

```
Athletic 220
Falcons
Ulaanbaatar
Khaan Khuns - Erchim
Deren
Ulaanbaatar City
Khangarid
Khoromkhon
BCH Lions
UB Mazaalaynuud
```

4.2 *Pickle*

5. Por outro lado, existe uma coisa que é **específica** do Python: o *pickle*[24], que é uma forma própria do Python salvar as variáveis em um arquivo, e de podermos recuperar depois. Imagine que você criou um modelo preditivo ou, ainda, criou um dicionário em Python, contendo vários valores. O *pickle* serve para salvar dados em que você confia e cuja origem você conhece. Ele pode ser útil para quando você precisa salvar uma variável diretamente em arquivo, sem conversões. Isso pode se aplicar a listas, dicionários, tuplas, modelos de IA já treinados e demais variáveis em Python.

   Existem duas funções principais no *pickle*:

   a. dump[25]: armazena uma ou mais variáveis em um arquivo.
   b. load[26]: carrega uma ou mais variáveis a partir de um arquivo.

[24] Disponível em: <https://docs.python.org/3/library/pickle.html>. Acesso em: 21 dez. 2021.
[25] Disponível em: <https://docs.python.org/3/library/pickle.html#pickle.dump>. Acesso em: 21 dez. 2021.
[26] Disponível em: <https://docs.python.org/3/library/pickle.html#pickle.load>. Acesso em: 21 dez. 2021.

Saiba mais

Outras linguagens podem oferecer funcionalidades similares ao *pickle*. É uma operação que se chama serialização/desserialização.

Vamos usar o mesmo *array* do NumPy (no caso, o "lista_equipes") para salvar em *pickle*. Veja que não estamos mais salvando um **texto**, mas, sim, **a variável**, que pode ser um NumPy *array*, um DataFrame, imagens, músicas, duas variáveis juntas ou quaisquer outros tipos que você imaginar, como comentamos previamente.

Perceba que no código a seguir usamos novamente o "open", mas o modo de escrita é um pouco diferente. Em vez de "w" (escrita de **texto**) usamos o "wb" (escrita em modo binário, em que não olhamos *encoding* e salvamos o dado tal como é). Aqui, usamos o "*dump*" para salvar a variável no arquivo com o *Pickle*:

```
import pickle

# salvando o array em um arquivo pickle
# note que não estamos mais salvando o texto, mas uma variável
pickle.dump(lista_equipes, open(path + 'lista_equipes.pkl', 'wb'))
```

Para ler o arquivo do *pickle* utilizamos o "*load*". Note que o modo de escrita também é binário ("rb", em vez de "r"):

```
# lendo o pickle e colocando o seu conteúdo em uma variável
lista_pickle = pickle.load(open(path + 'lista_equipes.pkl', 'rb'))
lista_pickle
```

```
array(['Athletic 220', 'Falcons', 'Ulaanbaatar', 'Khaan Khuns - Erchim',
       'Deren', 'Ulaanbaatar City', 'Khangarid', 'Khoromkhon',
       'BCH Lions', 'UB Mazaalaynuud'], dtype=object)
```

TEMA 5 – PERSISTINDO LÁ

Em um mundo conectado, porém, é importante persistirmos os dados *fora* de nosso computador. Logo, como persistimos os dados em outros serviços, servidores e soluções? Como enviamos os dados para essas diferentes aplicações?

Quando estamos enviando alguma informação para algum software, é comum que utilizemos uma interface criada especialmente para seres humanos,

isto é, um site bonito, alguns botões coloridos para clicar, alguns instrumentos visuais para que possamos nos localizar pelo site, e assim por diante.

Entretanto, quando falamos entre uma comunicação entre softwares, ou seja, entre um algoritmo e outro, não é necessário ter uma interface visual; na realidade, não é desejável. É muito mais fácil um algoritmo ter uma "porta dos fundos" seguindo um padrão conhecido para que outros algoritmos possam se conectar, do que um algoritmo saber em qual botão clicar em um site, pois os botões podem mudar de posição dependendo da tela, do navegador ou do idioma usado na navegação.

É para isso que existem as APIs (*application programming interfaces*), que atendem exatamente a esse objetivo de facilitar a comunicação entre softwares e, assim, possibilitar a troca de informações entre os mesmos e a persistência dos dados de uma forma não local.

Há algumas formas de se trabalhar com APIs, como SOAP, REST, RPC e GraphQL. Dessas, a mais utilizada hoje no mundo corporativo é a REST (a SOAP encontra-se em desuso). As conexões que realizamos em REST utilizam os métodos HTTP para realizar suas comunicações. No caso, os mais comuns são o GET e o POST. O GET é utilizado para **obter** informações e o POST é utilizado para **criar** informações. Sites e serviços como o Github, Salesforce, Facebook, Twitter, Amazon, Google e serviços do governo utilizam o REST. Como é de se imaginar, a maioria dos serviços requerem um cadastro e existem algumas limitações sobre o que se pode acessar. Por outro lado, para que possamos testar comunicações em REST, usaremos dois serviços: o HTTPBin[27] e o POST Test Server[28]. Ambos funcionam como serviços em *echo*: o que você envia é o que você recebe, o que é útil para saber se o que você enviou funcionou de fato.

5.1 REST em Python

A comunicação com serviços externos utilizando Python pode ser realizada pelo módulo http.client[29] ou pela biblioteca requests[30], entre outras opções. Sua utilização é bem simples: informamos o tipo da *request* (GET ou POST ou, ainda, outros tipos de *requests* existentes como PUT, DELETE, HEAD

[27] Disponível em: <https://httpbin.org/>. Acesso em: 21 dez. 2021.
[28] Disponível em: <https://ptsv2.com/>. Acesso em: 21 dez. 2021.
[29] Disponível em: <https://docs.python.org/3/library/http.client.html>. Acesso em: 21 dez. 2021.
[30] Disponível em: <https://docs.python-requests.org/en/latest/>. Acesso em: 21 dez. 2021.

e OPTIONS); o destino (ou seja, a URL); dados de autenticação, e parâmetros adicionais necessários para que a *request* funcione com sucesso.

É importante que você saiba que o resultado de toda e qualquer *request* possui um código, e que esse código é responsável por dizer se a operação foi finalizada com sucesso, ou não. Esses códigos seguem o seguinte padrão:

- 2xx (ex.: 200, 201, 202) – operação finalizada com sucesso;
- 3xx (ex.: 301, 302, 308) – redirecionamento;
- 4xx (ex.: 400, 401, 404, 415) – erro no lado do cliente (ou seja: provavelmente informamos algo errado); e
- 5xx (ex.: 500, 502, 504) – erro no lado do servidor (ou seja: provavelmente algo que não deveria acontecer no lado do servidor acabou acontecendo).

Pensemos em um exemplo utilizando o HTTPBin: nesse caso, vamos supor que o servidor esteja esperando por um parâmetro chamado "*mensagem*" e haja um texto dentro desse parâmetro. No caso, poderia ser "*Olá! Esperamos receber exatamente isto de volta.*". Vamos ver como fica isto em Python, utilizando tanto a biblioteca *Requestsi,* como o http.client:

```
import json
import requests
import http.client
from urllib.parse import urlencode

# definindo a mensagem a ser enviada
payload = {'mensagem': 'Olá! Esperamos receber exatamente isto de volta.'}
```

```
# fazendo o GET via Requests
response_get = requests.get('https://httpbin.org/get', params=payload)
display(response_get.json())
```

Observe a resposta do servidor: primeiro, que é um arquivo JSON (lembra-se)? Segundo, que, para esse servidor específico, temos alguns metadados sobre a resposta e, ainda, o conteúdo que queremos consumir. Em *args/mensagem* temos exatamente a mesma mensagem que enviamos. Esse servidor funciona para testes de *echo* e, se tudo deu certo, retornará exatamente o que enviamos a ele. Existem outros servidores que terão respostas diferentes e em campos diferentes – logo, nem todos terão uma estrutura com *args/mensagem*, mas poderão ter um *layout* similar.

```
{'args': {'mensagem': 'Olá! Esperamos receber exatamente isto de volta.'},
 'headers': {'Accept': '*/*',
  'Accept-Encoding': 'gzip, deflate',
  'Host': 'httpbin.org',
  'User-Agent': 'python-requests/2.23.0',
  'X-Amzn-Trace-Id': 'Root=1-617f0080'},
 'origin': '',
 'url':
'https://httpbin.org/get?mensagem=Olá!+Esperamos+receber+exatamente+isto+de+volta.
'}
```

Veja agora como ficaria a mesma *request*, mas utilizando o http.client em vez do *Requests*. Observe que o código muda um pouco porque foi escrito com paradigmas de programação diferentes, por desenvolvedores diferentes, em bibliotecas diferentes (e com objetivos diferentes), mas conseguimos produzir a mesma resposta em *args/mensagem*.

```
# fazendo o GET via http.client
connection = http.client.HTTPSConnection('httpbin.org')
connection.request("GET", f"/get?{urlencode(payload)}")
response = connection.getresponse()
display(json.loads(response.read()))
connection.close()
```

```
{'args': {'mensagem': 'Olá! Esperamos receber exatamente isto de volta.'},
 'headers': {'Accept-Encoding': 'identity',
  'Host': 'httpbin.org',
  'X-Amzn-Trace-Id': 'Root=1-617f04a4'},
 'origin': '',
 'url':
'https://httpbin.org/get?mensagem=Olá!+Esperamos+receber+exatamente+isto+de+volta.
'}
```

Note como fica o código para o POST no *Requests*. A lógica é similar ao que vimos anteriormente. Enquanto é comum termos parâmetros de busca e seleção na URL do GET (veja o realce em azul nos exemplos anteriores), no POST é comum termos dados da nossa *request* no corpo dessa.

```
# fazendo o POST via Requests
response_post = requests.post('https://httpbin.org/post', data=payload)
display(response_post.json())
```

```
{'args': {},
 'data': '',
 'files': {},
 'form': {'mensagem': 'Olá! Esperamos receber exatamente isto de volta.'},
 'headers': {'Accept': '*/*',
  'Accept-Encoding': 'gzip, deflate',
  'Content-Length': '64',
  'Content-Type': 'application/x-www-form-urlencoded',
  'Host': 'httpbin.org',
  'User-Agent': 'python-requests/2.23.0',
  'X-Amzn-Trace-Id': 'Root=1-617f003c'},
 'json': None,
 'origin': '',
 'url': 'https://httpbin.org/post'}
```

Observe que a *request* agora ficou em um *form* (realçado em amarelo – como é um serviço de *echo*, o que mandamos é exatamente o que recebemos). Logo, poderíamos enviar mais dados além da "mensagem", exatamente dessa mesma forma. No http.client, a lógica é parecida. Note que tivemos de informar um dado de cabeçalho da *request* (*headers*) para que o servidor soubesse que tipo de informação estamos lhe enviando. Isso não foi necessário no exemplo anterior.

```
connection = http.client.HTTPSConnection('httpbin.org')
headers = {"Content-type": "application/x-www-form-urlencoded"}
connection.request("POST", "/post", urlencode(payload), headers=headers)
response = connection.getresponse()
display(json.loads(response.read()))
connection.close()
```

```
{'args': {},
 'data': '',
 'files': {},
 'form': {'mensagem': 'Olá! Esperamos receber exatamente isto de volta.'},
 'headers': {'Accept-Encoding': 'identity',
  'Content-Length': '64',
  'Content-Type': 'application/x-www-form-urlencoded',
  'Host': 'httpbin.org',
  'X-Amzn-Trace-Id': 'Root=1-617f05d4'},
 'json': None,
 'origin': '',
 'url': 'https://httpbin.org/post'}
```

FINALIZANDO

Nessa aula, tivemos como objetivo mostrar como saímos de variáveis sendo manipuladas tão somente dentro de nosso algoritmo, para armazená-las e comunicá-las com outros serviços. Saber **o que é** um arquivo XML, um parquet, um JSON, ou como trabalhamos com o envio e recebimento desses

dados é parte essencial de nossa formação. Não é tão somente uma questão de **saber que existe**, mas também **saber como trabalhar** com isso.

É provável que, em sua trajetória profissional, você se depare com várias implementações que não estão de acordo com o esperado, ou que utilizam soluções antigas, ou, ainda, códigos que estão fora do padrão. Logo, jamais você terá contato **somente** com JSON, ou **somente** com o *encoding* UTF-8, por exemplo. É para evitar possíveis surpresas no trabalho que mostramos algumas das possibilidades mais comuns que você poderá encontrar.

REFERÊNCIAS

ZZT32. **ASCII Table**. Disponível em: <https://commons.wikimedia.org/wiki/File:Ascii-proper-color.svg>. Acesso em: 20 dez. 2021.