Curso Programação Backend com Python

IO básico

Exceptions

Bom, se algo pode dar errado e você pode prever isso, então deveria faze-lo. Suponha o seguinte programa:

```
a=open('arquivo.txt')

print(a.read())
```

Se o arquivo chamado “arquivo.txt” existir, estiver na mesma pasta que o programa e o usuário que está executando o programa possuir permissões no sistema operacional, o conteúdo do arquivo será lido e exibido na console.

Dado o programa abaixo, o que aconteceria se o arquivo que ele tenta acessar (“arquivo.txt”) não existisse?

```
a=open('arquivo.txt')

print(a.read())
```

Eis o erro que aconteceria, seja na console interativa ou rodando o programa:

```
Python 3.8.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> a=open('arquivo.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory:
'arquivo.txt'
>>> print(a.read())
```

Há dois tipos de problemas:

- Known Unkowns: Riscos que podemos prever;
- Unkown Unknowns: Riscos que não podemos prever.

Sobre os riscos que podemos prever, podemos oferecer algum tipo de ajuda ao usuário. Por exemplo, é possível que um dia o “arquivo.txt” não exista, então, podemos fazer assim:

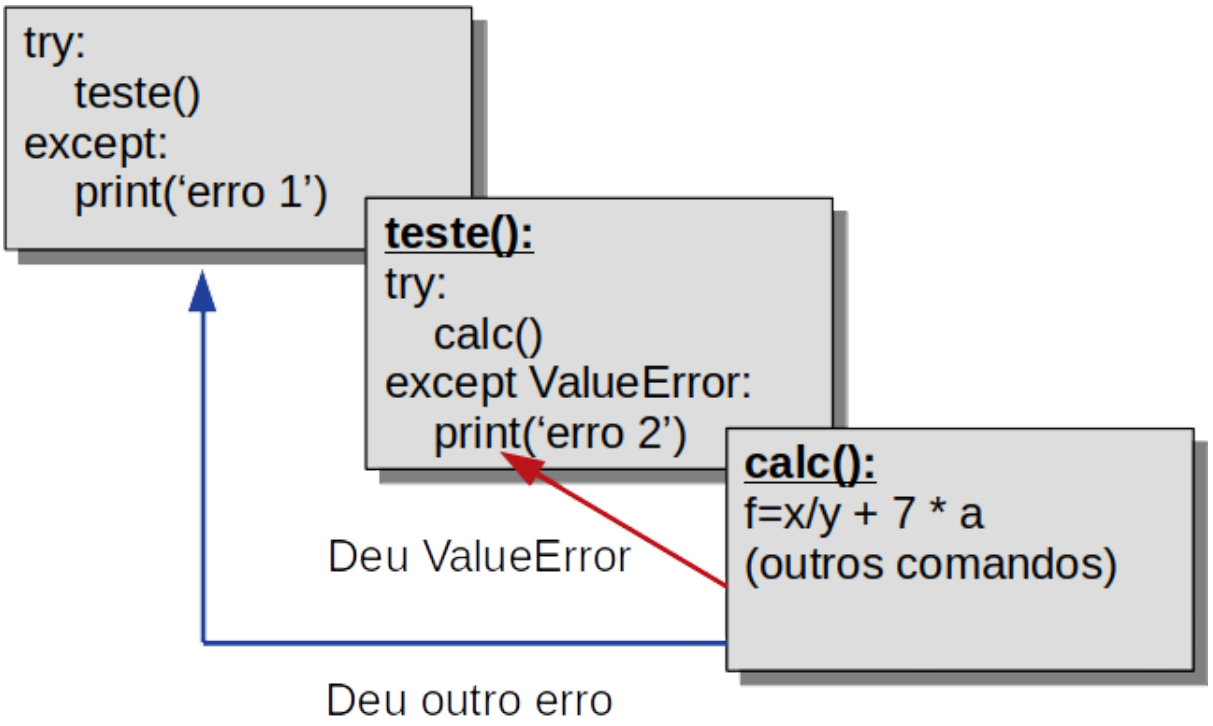
```
try:
     a=open('arquivo.txt')
     print(a.read())
except:
     print('Arquivo inexistente')
```

try / except

O bloco “try” encapsula código que pode dar algum tipo de problema, o qual queremos prever e fornecer um tratamento diferenciado. É como qualquer bloco de comando Python: Com identação. A sintaxe completa é:

```
try:
    (comandos protegidos)
except [tipo]:
    (comandos a serem executados caso o erro determinado ocorra)
else:
    (comandos a serem executados caso não ocorra erro algum)
finally:
    (comandos a serem sempre executados, ocorrendo ou não um erro)
```

Como funciona o try? É um mecanismo de **SEH**: Structured Exception Handling (um termo do MS Windows, que se aplica perfeitamente bem neste contexto). Se um erro ocorreu e houver algum tratamento para ele no bloco try, então este tratamento será executado. Se não houver tratamento definido, então é procurado um bloco try superior, que esteja protegendo a invocação do módulo onde ocorreu o erro. E vai assim até "explodir" na cara do usuário.



Nesta figura, o módulo principal invoca uma função "teste()" (que pode ou não estar dentro dele). Esta função, invoca outra função "calc()". Se der um erro dentro da função "calc()", não haverá tratamento para ele, logo, o SEH procurará se há tratamento em quem invocou a função, que é a função "teste()". Ela só trata erros do tipo "ValueError", portanto, se for um ValueError, a mensagem "erro 2" será exibida e o processamento continuará depois do bloco try.

E se for um outro tipo de erro? Note que o bloco try, dentro da função "teste()" só possui tratamento para ValueError. Neste caso, o SEH procurará no nível mais alto, onde há um except sem especificação de tipo de erro, e este será executado.

Veja este exemplo:

```
import sys
def fn1():
     raise ValueError('Ferrou')

def fn2():
     try:
     print('comando1')
```

```
        fn1()
        print('outro comando')
        except OSError as oserro:
        print("Erro do Sistema Operacional: {0}".format(oserro))

try:
        fn2()
except ValueError as erro:
        print('Value Error: {}'.format(erro))
except:
        print('ERRO:', sys.exc_info()[0])
        raise
```

O que pensa que aconteceria se executasse este programa? Pense um pouco…

O resultado seria este:

```
python3 teste.py
comando1
Value Error: Ferrou
```

O bloco principal oferece tratamento para o ValueError, utilizando a sintaxe **as** para especificar uma variável (erro) que receber o objeto da **exception**.

Temos coisas interessantes neste código. Por exemplo, dentro da "fn1()" foi lançado um **ValueError** de propósito, com o comando **rise**, e ainda foi atribuída uma mensagem a ele.

O **ValueError** lançado não foi tratado dentro de "fn1()" e nem dentro de "fn2()". A função "fn2()" apenas oferece tratamento para erros do tipo OSError. Portanto, o erro vai subir e chegar no bloco principal, onde são especificados dois tratamentos de erro: Um para ValueError e outro para qualquer tipo de erro (o **except** sem tipo).

Quando não sabemos que tipo de erro pode ocorrer, colocamos um comando **except** sem especificar nada. Mas isto pode ser um problema, pois podemos

"mascarar" erros graves desta forma. Uma boa prática é sempre colocar o **except** vazio como última cláusula do **try**, e sempre usar o comando **raise** puro, para relançar o erro, dando assim a chance de alguém tratá-lo.

A tupla **exc_info**, da biblioteca **sys**, tem sempre as informações sobre a **exception** que ocorreu (caso você tenha esquecido de usar o as ou esteja em um bloco **except** puro).

Para saber quais exceptions o Python oferece, verifique na documentação: https://docs.python.org/3/library/exceptions.html

Voltando ao exemplo do início:

```
try:
     a=open('arquivo.txt')
     print(a.read())
except:
     print('Arquivo inexistente')
```

Note que há mais comandos dentro do bloco try. E se o erro for outro? E se for em uma subrotina? Vamos supor que o arquivo esteja em um pendrive, e ele tenha sido removido... Este tratamento de erro é insatisfatório. Uma versão melhor seria esta:

```
import sys
try:
     a=open('arquivo.txt')
     print(a.read())
except FileNotFoundError:
     print('Arquivo inexistente')
except:
     print('Erro inesperado: {}'.format(sys.exc_info()[0]))
     raise
else:
     print('não deu erro')
finally:
```

```
    print('com erro ou sem erro, eu sempre executarei!')
```

Este código oferece um tratamento de erro bem melhor. Ele prevê o erro **FileNotFoundError**, que ocorre quando o arquivo não existe, e também prevê qualquer outro tipo de erro, avisando que ocorreu e re-lançando. Supondo que tenhamos um arquivo chamado “arquivo.txt”, e retirarmos a sua permissão de leitura, este seria o resultado:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ sudo chmod -r arquivo.txt
[sudo] password for cleuton:
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ ls -la
total 116
drwxrwxr-x 2 cleuton cleuton  4096 jan 26 08:29 .
drwxr-xr-x 8 cleuton cleuton  4096 jan 26 07:02 ..
-rw-rw-r-- 1 cleuton cleuton   273 jan 26 08:25 arqexception.py
--w------- 1 root   root      7 jan 26 08:29 arquivo.txt
-rw-rw-r-- 1 cleuton cleuton 84616 jan 26 08:22 curso-python-cleuton-licao5.odt
-rw-rw-r-- 1 cleuton cleuton   380 jan 26 08:10 except.py
-rw-r--r-- 1 cleuton cleuton  80 jan 26 08:22 .~lock.curso-python-cleuton-licao5.odt#
-rw-rw-r-- 1 cleuton cleuton  91 jan 26 07:27 semexcept.1.py
-rw-rw-r-- 1 cleuton cleuton  37 jan 26 07:27 semexcept.py
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ python arqexception.py
Erro inesperado: <class 'PermissionError'>
com erro ou sem erro, eu sempre executarei!
Traceback (most recent call last):
  File "arqexception.py", line 3, in <module>
    a=open('arquivo.txt')
PermissionError: [Errno 13] Permission denied: 'arquivo.txt'
```

Este comando está rodando em **Linux**. Para fazer a mesma coisa em **MS Windows**, basta usar o Explorer.EXE e retirar a permissão de leitura do arquivo.

Como pode ver, as mensagens de erro aparecem e demos a chance de algum módulo superior tratar o problema. Note que o bloco **else** só será executado se der algum erro, e o **finally** será executado sempre.

Note que a ordem dos **excepts** é fundamental! Primeiro, colocamos os **excepts** que prevemos e só por último o **except** puro. Aliás, melhor seria não termos **excepts** puros.

I/O básico

Baixe o arquivo “exemplos.zip” e use-os nos exercícios e demonstrações.

Vamos criar um arquivo texto. É simples! Rode o programa "cria.py" (todos os programas estão no arquivo “exemplos.zip”):

```
arq=open('novo.txt','w')
arq.write('Minha terra tem palmeiras')
arq.write('Onde canta o sabiá')
arq.write('As aves que aqui gorjeiam')
arq.write('Não gorjeiam como lá')
arq.close()
```

O arquivo foi criado. Vamos executar e verificar o conteúdo:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ python cria.py
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ ls
arqexception.py  arquivo.txt  cria.py  curso-python-cleuton-licao5.odt  except.py  novo.txt  semexcept.1.py  semexcept.py
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ cat novo.txt
Minha terra tem palmeirasOnde canta o sabiáAs aves que aqui gorjeiamNão gorjeiam como lácleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$
```

No **Linux**, usamos o comando cat para listar um arquivo no terminal. No **Windows**, pode usar type.

Bom, ele gravou o arquivo, mas tem um problema: Cadê a separação das linhas? Por que eu criaria um arquivo de texto (com o poema "Canção do exílio", de Gonçalves Dias) sem separar as linhas? E tem mais: E o tratamento de erros? Calma! Vamos analisar primeiramente a maneira como criamos o objeto file:

```
arq=open('novo.txt','w')
```

A função open cria um objeto do tipo file representando o arquivo em disco. O primeiro argumento é o caminho do arquivo e o segundo é o modo de abertura:

• 'r': Somente leitura (default). O programa pode apenas ler o arquivo;

• 'w': Somente gravação. O programa pode criar ou sobrescrever um arquivo existente;

• 'a': Anexação (append). O programa pode anexar dados ao final do arquivo existente;

• 'x': Criação. O programa pode criar o arquivo e resultará em erro, caso o arquivo já exista;

A partir deste comando open, você pode lidar com a variável que representa o objeto file, por exemplo, eu gravei dados com o método write:

```
arq.write('Minha terra tem palmeiras')
```

Mas é o método close que libera os recursos do sistema operacional e grava o buffer em disco:

```
arq.close()
```

Simples, não? E como podemos ler esse arquivo? Uma forma simples seria essa:

```
a=open('novo.txt')
print(a.read())
```

E o resultado seria este:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$
python le.py
Minha terra tem palmeirasOnde canta o sabiáAs aves que aqui
gorjeiamNão gorjeiam como lá
```

Faltou o close e faltou o tratamento de erros... Mas calma! O objetivo é você entender como ler e gravar arquivos, daqui a pouco veremos uma versão melhor. Notou que usei o método read sem argumentos? Desta forma, ele lê todo o

conteúdo do arquivo, disponibilizando-o através da variável criada para receber a saída. Neste caso, foi diretamente para o print.

Estamos lidando aqui apenas com arquivos textuais, que contém caracteres **unicode** (notou o "à" no conteúdo do arquivo?). Há também o modo **binário**, onde gravamos outros tipos de dados, mas este tipo de I/O é complexo e está em desuso, portanto, não vou mostrá-lo. Se você quiser mesmo gravar dados binários e formatados, então recomendo usar **JSON** ou então um banco de dados.

Agora, vejamos uma maneira bem melhor de criar um arquivo texto, separando seu conteúdo em linhas (cria2.py):

```
try:
     arq=open('novo2.txt','x')
     try:
          arq.write('Minha terra tem palmeiras\n')
          arq.write('Onde canta o sabiá\n')
          arq.write('As aves que aqui gorjeiam\n')
          print('Não gorjeiam como lá',file=arq)
          arq.close()
     finally:
          arq.close()
except FileExistsError:
     print('O arquivo "novo2.txt" já existe!')
```

Agora, vamos executar e mostrar o arquivo:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ python cria2.py
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ cat novo2.txt
Minha terra tem palmeiras
Onde canta o sabiá
As aves que aqui gorjeiam
Não gorjeiam como lá
```

Bem melhor, não? As linhas estão separadas! Como eu fiz isso? De duas maneiras diferentes:

1. Acrescentando um caractere **linefeed** ao final do texto;
2. Usando o print para o arquivo.

O caractere contra-barra ("\") é um escape. Temos vários escape que podemos usar:

- \n : linefeed;
- \t : tab;
- \\ : contra-barra;
- \' : aspas;
- \" : aspas duplas;

https://www.quackit.com/python/reference/python_3_escape_sequences.cfm

O método **write** não inclui um **linefeed**, logo, o acrescentamos ao texto:

```
arq.write('Minha terra tem palmeiras\n')
```

E quando usamos o print, o linefeed já foi acrescentado. Foi só usar o parâmetro file:

```
print('Não gorjeiam como lá',file=arq)
```

Porém, o mais importante foi o tratamento de erros. Neste caso, estamos abrindo o arquivo com o modo de criação ("x"), portanto, se ele já existir é um erro e eu deveria prever isto, por esta razão eu estou interceptando o **FileExistsError**. Se ocorrer qualquer outro erro, eu quero que seja propagado para cima. E não queremos deixar recursos do sistema operacional presos, caso ocorra um erro,

então criamos um segundo try interno, que fecha o arquivo caso aconteça algum erro:

```
try:
     arq=open('novo2.txt','x')
     try:
          ...
     finally:
          arq.close()
```

with

Quando usamos I/O de arquivos, a maneira mais pythonica de fazer isto é com o comando **with** que garante o fechamento do mesmo ao final, mesmo em caso de erro. Vejamos o script cria3.py:

```
try:
     with open('novo3.txt','x') as arq:
          arq.write('Minha terra tem palmeiras\n')
          arq.write('Onde canta o sabiá\n')
          arq.write('As aves que aqui gorjeiam\n')
          print('Não gorjeiam como lá',file=arq)
except FileExistsError:
     print('O arquivo já existe!')
```

Queremos criar o arquivo apenas se ele não existir. Isto é para mostrar a maneira apropriada de tratar erros. Mas isto não é necessário, pois se especificarmos "w" ele vai sobrescrever o arquivo, caso já exista. Isso depende da sua regra de negócio.

O **with** executa o comando principal criando uma variável (através do **as** arq) e guarda o bloco de comandos contra **exceptions**, garantindo que o arquivo seja sempre fechado.

Bom, agora, podemos ler o arquivo de forma apropriada, tratando cada linha individualmente. Vejamos o script "ler3.py":

```
try:
     with open('novo3.txt') as texto:
          for linha in texto:
                    print(linha.strip())
except FileNotFoundError:
     print('O arquivo não existe!')
```

Como separamos os dados em linhas, eu as posso ler individualmente, utilizando o comando "**for**". Eu usei o strip para retirar **whitespaces** (espaços e caracteres de controle) de cada linha, caso contrário, ao mostrar na console, haveria duas quebras de linha, pois cada linha já termina em **linefeed**.

Se você precisar processar linha a linha, além de imprimir, então é melhor fazer um loop de leitura. Veja o script "ler4.py":

```
try:
     with open('novo3.txt') as texto:
          linha = texto.readline()
          i = 0
          while linha:
               i+=1
               print('Linha: {}: {}'.format(i,linha.strip()))
               linha = texto.readline()
     print('Eu li {} linhas'.format(i))
except FileNotFoundError:
     print('O arquivo não existe!')
```

O método readline, do objeto **file**, lê uma linha de cada vez.

Formatando números e datas

Em certas situações, você terá que gravar dados especiais, como: Números e datas. Neste caso, podemos usar os recursos de formatação. Por exemplo, vejamos como lidar com moedas e números:

```
Python 3.7.0 (default, Jun 28 2018, 13:15:42)
[GCC 7.2.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')
'pt_BR.UTF-8'
>>> valor = 105.98
>>> print(locale.currency(valor))
R$ 105,98
```

Podemos especificar qual é o idioma (locale) que estamos utilizando com o método setlocale, e podemos usar o método currency para transformar variáveis numéricas em strings monetários.

Serve para números comuns também, incluindo os separadores de classes:

```
>>> numero = 1305432.77
>>> print(locale.format('%.2f',numero,grouping=True))
1.305.432,77
```

Como eu já configurei o **locale**, a opção **grouping=True** formatará os separadores de classes no idioma do Brasil. O método format tem a opção de formato:

```
%<inteiros>.<decimais><tipo>
```

E o tipo:

- d : Inteiro sinalizado;
- f : Real.

Veja mais em:

https://www.python-course.eu/python3_formatted_output.php

Datas e horas

Podemos utilizar o módulo datetime para obter uma data e podemos formatá-la de acordo com o idioma:

```
>>> import datetime
>>> print(datetime.datetime.now())
2019-01-27 07:47:27.547680
>>> data = datetime.datetime.now()
>>> print(data.strftime(locale.nl_langinfo(locale.D_T_FMT)))
dom 27 jan 2019 07:48:34
```

O método **now**, do objeto **datetime**, do módulo **datetime**, nos dá a data e hora atuais. Se usarmos com print, teremos um formato diferente. É possível transformar este formato para nosso idioma, usando o método nl_langinfo, passando a constante **D_T_FMT**. Com isso, temos um formato de data e hora compatíveis com o Português Brasileiro. O objeto datetime tem o método **strftime**, que formata data e hora. Podemos formatar a data e/ou hora sem utilizar o **locale**:

```
>>> print(data.strftime(locale.nl_langinfo(locale.D_T_FMT)))
dom 27 jan 2019 07:48:34
>>> print(data.strftime("%d/%m/%Y %H:%M:%S"))
27/01/2019 07:48:34
>>> print(data.strftime("%d/%m/%Y"))
27/01/2019
>>> print(data.strftime("%H:%M:%S"))
07:48:34
>>> print(data.strftime("%Y-%m-%d"))
2019-01-27
```

Os códigos de formatação para data e hora são:

- %Y : Ano com 4 dígitos;
- %m : Mês com 2 dígitos;
- %d : Dia com 2 dígitos;
- %H : Hora (de 00 a 23);
- %M : Minuto (de 00 a 59);
- %S : Segundo (de 00 a 61*);

(*) Não pergunte... É assim mesmo.

Para mais informações: https://docs.python.org/3.2/library/time.html#time.strftime

E para converter strings em datas e horas? Primeiramente, você tem que saber o formato em que estes strings se encontram. Por exemplo:

```
>>> sdata = data.strftime("%d/%m/%Y ¾H:%M:%S")
>>> print(type(sdata))
<class 'str'>
>>> novadata = datetime.datetime.strptime(sdata,"%d/%m/%Y ¾H:%M:%S")
>>> print(novadata)
2019-01-27 00:48:34
>>> print(type(novadata))
<class 'datetime.datetime'>
```

Se o método **strftime** transforma data e hora em string, o método **strptime** (da classe **datetime**, do módulo **datetime**) transforma **string** em objeto **datetime**, desde que o formato seja compatível.

Cálculos com data e hora

Não é objetivo deste curso entrar nestes detalhes, mas mostraremos rapidamente como podemos fazer cálculos com data e hora (diferença e data futura).

Para saber uma data futura ou passada, podemos usar a função **timedelta**, do módulo **datetime**, que permite especificar **weeks** (semanas), **days** (dias), **hours** (horas), **minutes** (minutos) e **seconds** (segundos), inclusive negativos:

```
>>> import datetime
>>> data = datetime.datetime.now()
>>> print(data)
2019-01-27 08:13:10.514841
>>> semana = datetime.timedelta(weeks=1)
>>> uma_semana_depois=data + semana
>>> print(uma_semana_depois)
2019-02-03 08:13:10.514841
>>> dois_meses = datetime.timedelta(days=60)
>>> print('daqui a dois meses {}'.format(data+dois_meses))
daqui a dois meses 2019-03-28 08:13:10.514841
>>> anteontem = datetime.timedelta(days=-2)
>>> print('anteontem {}'.format(data + anteontem))
anteontem 2019-01-25 08:13:10.514841
```

Para saber o intervalo decorrido entre dois momentos no tempo:

```
>>> anteontem = datetime.timedelta(days=-2)
>>> print('anteontem {}'.format(data + anteontem))
anteontem 2019-01-25 08:13:10.514841
>>> print(data,anteontem)
2019-01-27 08:13:10.514841 -2 days, 0:00:00
>>> data_anteontem = data + anteontem
>>> diferenca = data - data_anteontem
>>> print(diferenca)
2 days, 0:00:00
```

```
>>> print(diferenca.days)
2
>>> print(diferenca.seconds)
0
```

A diferença entre dois objetos **datetime** será um **timedelta** e podemos acessar as propriedades days, seconds e microseconds, para saber quanto tempo, nessas unidades, transcorreu.

Sobre ler dados formatados

A tentação de gravar strings formatados em arquivos, para depois processá-los é grande. Eu recomendo fortemente que você utilize um banco de dados. Mas, se for absolutamente necessário, use o formato **JSON** (vou mostrar adiante).

É possível ler e traduzir strings em números e datas, conforme já mostrei, mas isto é muito ruim e sujeito a erros. Um banco de dados seria melhor (mostrarei ao final).

JSON

JavaScript Object Notation é o formato de serialização de objetos nativo do Javascript, e se tornou um padrão de verdade. O formato JSON nos permite criar arquivos estruturados que nos darão mais segurança para lermos e gravarmos dados. Conhece JSON? Não?

https://www.json.org/json-pt.html

Vamos supor que desejamos gravar o arquivo de movimento do dia:

```
{
    "cliente":<id do cliente, inteiro>,
    "mercadoria":<código do produto, inteiro>,
    "quantidade":<quantidade adquirida, real>,
    "data":<data e hora da compra, datetime>
}
```

Temos o módulo **json**, que possui o método **dump**, para gravar estrutura json em um arquivo, e o método load, para ler json de um arquivo. Vamos ver um exemplo (script "jsongrava.py"):

```
from datetime import datetime
from datetime import timedelta
import json
movimento=[]
agora=datetime.now()
depois=timedelta(minutes=10)
nova=agora+depois
formato="%d/%m/%Y %H:%M:%S"
movimento.append({"cliente":1,
    "mercadoria":10,
    "quantidade":5.5,
    "data":agora.strftime(formato)})
```

```
movimento.append({"cliente":2,
    "mercadoria":15,
    "quantidade":23.2,
    "data":nova.strftime(formato)})
with open('saida.json', 'w') as saida:
     json.dump(movimento,saida)
```

Criamos uma lista e anexamos objetos **dictionary**, cada um contendo os dados de uma venda. Colocamos a data da segunda venda para 10 minutos depois. Note que usamos o **with** e passamos o nome do arquivo para o método **dump**, do objeto **json**. Veja a saída disso:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$
python jsongrava.py
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$
cat saida.json
[{"cliente": 1, "mercadoria": 10, "quantidade": 5.5, "data":
"27/01/2019 08:45:33"}, {"cliente": 2, "mercadoria": 15,
"quantidade": 23.2, "data": "27/01/2019 08:55:33"}]
```

Parece que ele gravou direitinho. Agora, vamos ler e formatar esse arquivo. Vamos ver como fazer isso com o script "jsonler.py":

```
from datetime import datetime
import json
from json.decoder import JSONDecodeError
import locale
locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')
lista=[]
produtos={10:"Leite",15:"Iogurte",20:"Manteiga"}
formato="%d/%m/%Y %H:%M:%S"
try:
     with open('saida.json','r') as arq:
          lista=json.load(arq)
          for venda in lista:
               nome_produto=produtos[venda['mercadoria']]
```

```
                data=datetime.strptime(venda['data'],formato)

                sdata=data.strftime(locale.nl_langinfo(locale.D
                _T_FMT))

                quantidade=locale.format_string('%.2f',venda['q
                uantidade'])
                print('Cliente: {}, Produto: {}, Data: {},
Quantidade: {}'.format(
          venda['cliente'],nome_produto,sdata,quantidade
                    )
                )
except JSONDecodeError:
     print('O arquivo de movimento está inválido!')
```

Melhor ver o arquivo com um editor de código, como o Visual Studio Code, pois aqui pode desposicionar as linhas e a indentação é fundamental no python!

E o resultado:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$
python jsonler.py
Cliente: 1, Produto: Leite, Data: dom 27 jan 2019 08:45:33 ,
Quantidade: 5,50
Cliente: 2, Produto: Iogurte, Data: dom 27 jan 2019 08:55:33 ,
Quantidade: 23,20
```

Se você executou e olhou o programa com atenção, deve ter notado que eu substitui o método format, do **locale**, pelo **format_string**. É porque o **format** está se tornando depreciado (**deprecated**) e deve ser retirado em futuras versões do Python.

Outra coisa interessante é que estou prevendo o **JSONDecodeError**, pois o arquivo JSON pode estar mal formatado e eu quero prever essa situação. Eis o que acontece se eu bagunçar o arquivo "saida.json":

```
python jsonler.py
```

```
O arquivo de movimento está inválido!
```