pythondrops.com

Curso Básico de Python

Lição 5: I/O e outras “cositas”

Sumário

Antes de mais nada

Gostaria de relembrar a você o objetivo deste curso. E sim: É um curso! Há lições, haverá vídeos e também correções de exercícios. Mas o objetivo é ensinar o Python básico. Não apenas o "print('hello world')", mas o suficiente para você começar a criar aplicações.

Dentro do **http://pythondrops.com** há outros tutoriais, mais avançados, que complementarão o seu conhecimento. Pode parecer que estou mostrando muita coisa, muitos detalhes, mas acredite: Não estou! E é proposital. Eu quero que você aprenda Python para usar no seu dia a dia.

Meus conselhos são:

1. **Faça os exercícios.** Sem olhar! Tente fazer e só olhe quando estiver para pular da janela;
2. **Estude os tutoriais avançados.** Sim, mas só depois de concluir todo o curso;
3. **Faça a trilha Python do HackerRank.com**. Vá devagar e procure fazer um por dia, depois de concluir o curso.

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.

Notou a "plantação de pé de SE"? São muitas premissas, certo? Uma ou mais delas podem falhar e você pode prever isso facilmente. Se não o fizer, o que acontecerá? Veremos! Apague o arquivo "arquivo.txt", da pasta do curso (você clonou o repositório, certo?) e rode o programa "semexcept.py", que está na pasta da lição 5.

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ python semexcept.py
Traceback (most recent call last):
  File "semexcept.py", line 1, in <module>
    a=open('arquivo.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'arquivo.txt'
```

Você acha “legal” essa mensagem? Talvez… Mas esse é um programa muito pequeno… E se for um programa grande? E se este problema ocorrer em um módulo, que está sendo chamado de outro módulo? E se for o usuário a tomar esse erro? Certamente, ele xingará a sua progenitora!

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')
```

Veja o script ‘semexcept.1.py’.

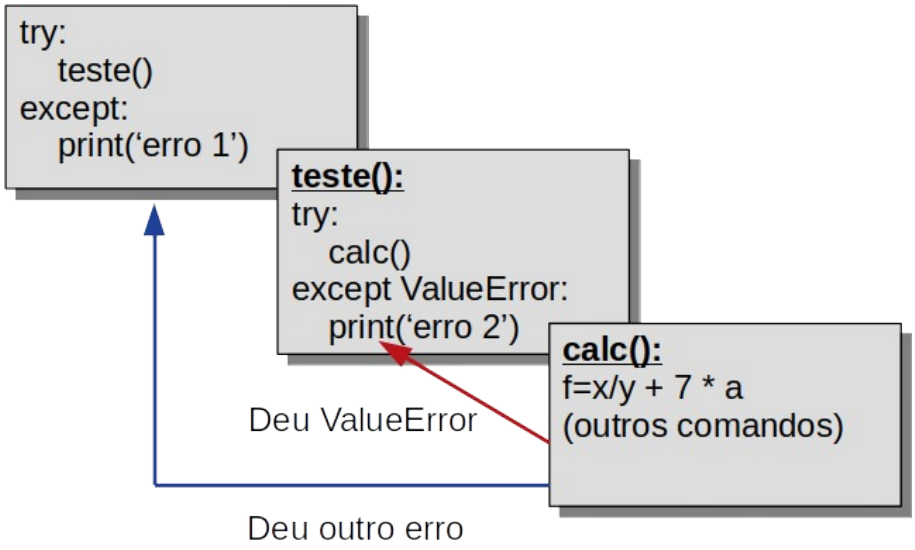
Agora, mostramos uma mensagem decente e não aquela porcaria toda. Ainda podemos melhorar este código, mas vamos examinar o comando ***try***.

try

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.

Abra o script 'except.py' e o execute:

```
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 resultado será:

```
cleuton@cleuton-pc:~/Documents/projetos/python/curso/licao5$ python except.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 muitas coisas legais nesse código. Por exemplo, dentro da "fn1()" eu levantei um ValueError de propósito, com o comando ***raise***, e ainda atribuí uma mensagem a ele.

O ValueError que eu levantei 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 eu ofereço 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 re-lançar 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

Ok, voltando ao exemplo do início, vejamos o programa "semexcept.1.py":

```
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 está insatisfatório. Uma versão melhor seria o "arqexception.py":

```
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. Eu retirei a permissão de leitura do "arquivo.txt" e rodei para ver 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'
```

Como pode ver, as minhas mensagens de erro apareceram e eu dei 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

Vamos criar um arquivo texto. É simples! Rode o programa "cria.py":

```
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 quê 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 leríamos 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 eu mostro 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 que eu forneci 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 caracter ***linefeed*** ao final do texto;
2. Usando o ***print*** para o arquivo.

O caracter 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, eu o acrescentei ao texto:

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

E quando usei o ***print***, o ***linefeed*** já foi acrescentado para mim. Foi só usar o parâmetro ***file***:

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

Porém, o mais importante foi o tratamento de erros. Neste caso, estou 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 quero deixar recursos do sistema operacional presos, caso ocorra um erro, então eu criei 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!')
```

Eu quero criar o arquivo apenas se ele não existir. Estou fazendo isso para te mostrar como é 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 dadas. 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 varíaveis 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 %H:%M:%S")
>>> print(type(sdata))
<class 'str'>
>>> novadata = datetime.datetime.strptime(sdata,"%d/%m/%Y %H:%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 vou mostrar 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 eu queira 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)
```

Eu criei uma lista e fui anexando objetos ***dictionary***, cada um contendo os dados de uma venda. Eu joguei a data da segunda venda para 10 minutos depois. Note que usei o ***with*** e passei 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, eu quero 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['quantidade'])
        print('Cliente: {}, Produto: {}, Data: {}, Quantidade: {}'.format(
            venda['cliente'],nome_produto,sdata,quantidade
        )
        )
except JSONDecodeError:
    print('O arquivo de movimento está inválido!')
```

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!
```

Banco de dados PostgreSQL

Cara, SQL é muito avançado para um curso básico, mas eu tenho que te ensinar alguma coisa. Se você já programou em linguagens modernas, como Java ou C#, então vai entender perfeitamente. Caso contrário, relaxe e estude SQL em primeiro lugar.

O PostgreSQL (https://www.postgresql.org/) é um SGBD relacional open source e muito utilizado em empresas. Provavelmente, você o utilizará com Python, portanto, vou mostrar os passos iniciais para se fazer isto.

Não se preocupe em instalar o PostgreSQL na sua máquina! Minha filosofia é: Aprenda o essencial! Não é o momento de aprender a instalar e usar PostgreSQL! Vamos usar o ElephantSQL, que é um "postgresql as a service", na nuvem, que tem um nível gratuito para usarmos.

Más notícias

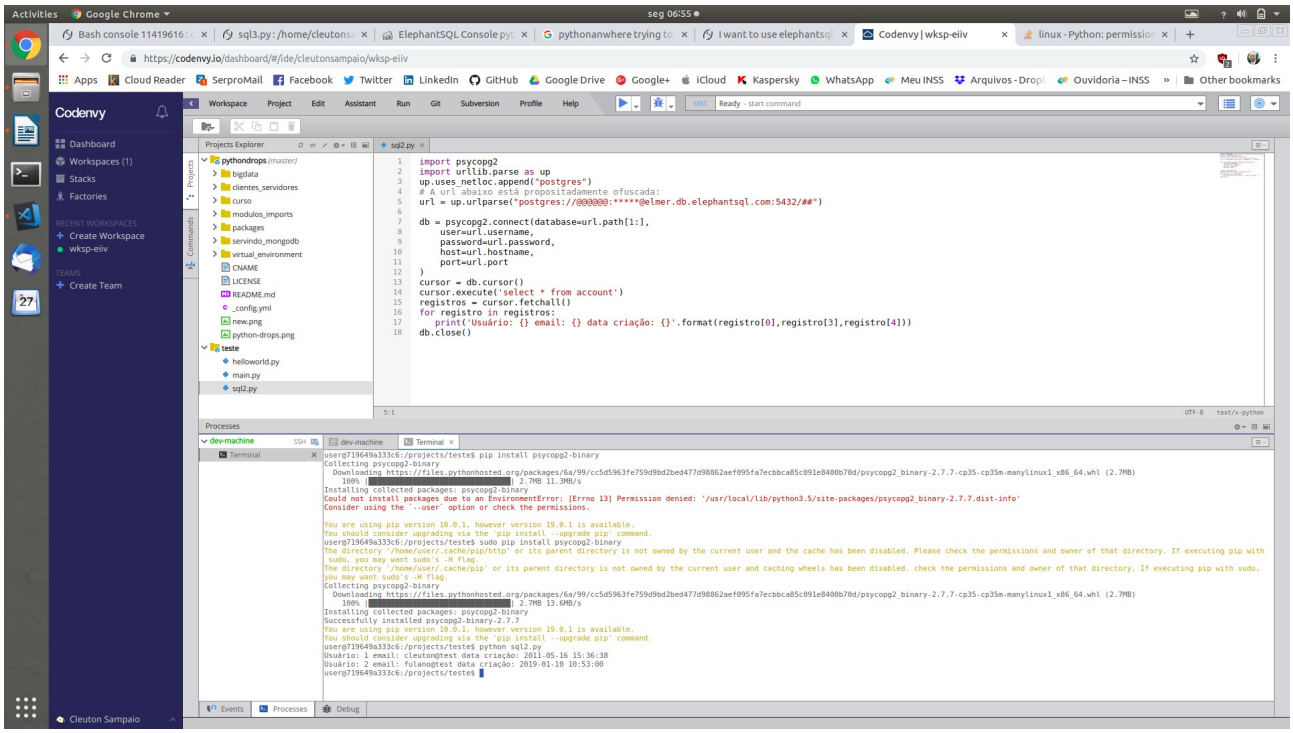
Tenho uma má notícia para você, que está usando o ***pythonanywhere***: No nível gratuito, o firewall dele bloqueia acessos para fora, ou seja, você não conseguirá acessar o banco de dados que está no ElephantSQL. Há duas opções:

- Instalar o PostgreSQL localmente, em sua máquina (https://www.devmedia.com.br/como-instalar-o-postgresql/17473);
- Usar o Codenvy.

Se você tiver o Docker (https://www.docker.com/) instalado em sua máquina, é só rodar um container com a imagem do PostgreSQL:

```
docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -d postgres
```

Mas o Codenvy suporta perfeitamente, mesmo com a conta gratuita:



Lembre-se que o uso do PIP no Codenvy exige o sudo:

`sudo pip install psycopg2-binary.`

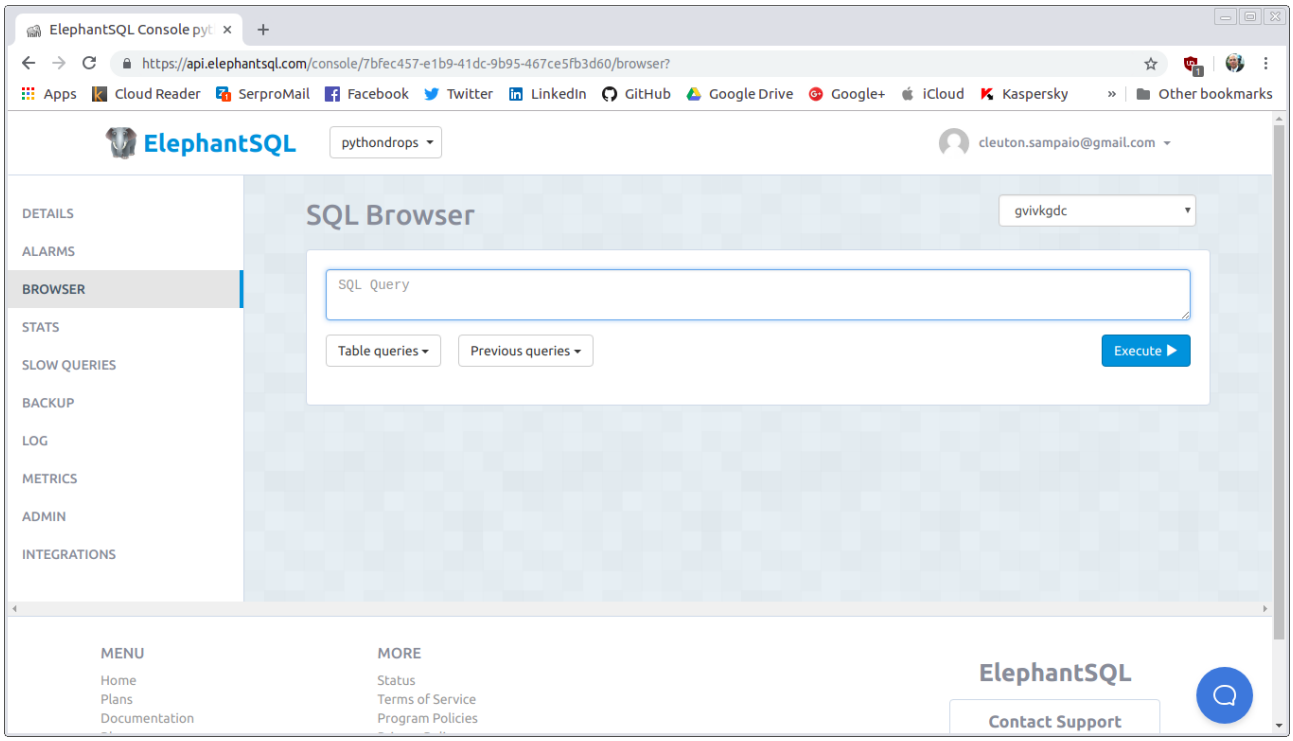
Dito isto, vamos ao ElephantSQL!

https://www.elephantsql.com/

Entre no site e crie uma conta. Você pode usar sua conta no Github, caso possua uma. Os passos são simples:

1. Crie uma instância (use a conta gratuita do plano "Tiny turtle");
2. Selecione sua instância e selecione "Browser" no menu esquerdo;
3. Digite sua consulta SQL e execute.

Você deve criar pelo menos uma tabela.



Para criar uma tabela simples, execute um CREATE TABLE:

```
CREATE TABLE account(
 user_id serial PRIMARY KEY,
 username VARCHAR (50) UNIQUE NOT NULL,
 password VARCHAR (50) NOT NULL,
 email VARCHAR (355) UNIQUE NOT NULL,
 created_on TIMESTAMP NOT NULL,
 last_login TIMESTAMP
);
```

A chave primária é "user_id".

Se você não conhece SQL e nem PostgreSQL, então é melhor dar uma olhada neste tutorial:

http://www.postgresqltutorial.com/postgresql-create-table/

Depois, insira alguns dados, por exemplo:

```
insert into account (user_id,username,password,email,created_on,last_login)
          values(2, 'FULANO', 'testfulano', 'fulano@test',
          '2019-01-10 10:53:00', NULL)
```

Ele converte o string para timestamp automaticamente.

Agora, vou mostrar um exemplo de código que lê os registros (script "sql1.py"). Eu utilizo o pacote psycopg2:

```
import psycopg2
import urllib.parse as up
up.uses_netloc.append("postgres")
# A url abaixo está propositadamente ofuscada:
url = up.urlparse("postgres://@@@@@@:*****@elmer.db.elephantsql.com:5432/##")
db = psycopg2.connect(database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)
cursor = db.cursor()
cursor.execute('select * from account')
registros = cursor.fetchall()
for registro in registros:
   print (registro)
db.close()
```

Ao rodar este programa, temos os resultados:

```
(1, 'cleuton', 'test', 'cleuton@test', datetime.datetime(2011, 5, 16, 15, 36,
38), None)
(2, 'FULANO', 'testfulano', 'fulano@test', datetime.datetime(2019, 1, 10, 10,
53), None)
```

Ok, muita calma nessa hora!

Antes de mais nada, você deve instalar o pacote do psycopg2:

```
pip install psycopg2-binary
```

Depois disso, você pode importar o pacote ***psycopg2***, conforme viu no código-fonte. Mas temos que especificar os dados para conexão com o servidor PostgreSQL que, neste caso, são do ElephantSQL. Volte à página do ElephantSQL e selecione a opção "DETAILS", no menu esquerdo. Copie o conteúdo do campo ***URL*** e cole dentro da chamada do método urlparse:

```
url = up.urlparse("cole aqui!!!!!")
```

A partir daí, abrimos uma conexão com o banco, utilizando os dados da url:

```
db = psycopg2.connect(database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)
```

Criamos um cursor para executar comandos e obter resultados, e rodamos nossa consulta SQL:

```
cursor = db.cursor()
cursor.execute('select * from account')
```

E vamos obter todos os registros, navegando por eles:

```
registros = cursor.fetchall()
for registro in registros:
   print (registro)
```

Finalmente, fechamos a conexão com o banco:

```
db.close()
```

Neste exemplo, estou passando para o print o registro inteiro, mas seria legal vermos campo a campo, não? Abra o script "sql2.py":

```
import psycopg2
import urllib.parse as up
up.uses_netloc.append("postgres")
# A url abaixo está propositadamente ofuscada:
url = up.urlparse("postgres://@@@@@@:*****@elmer.db.elephantsql.com:5432/##")
db = psycopg2.connect(database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)
cursor = db.cursor()
cursor.execute('select * from account')
registros = cursor.fetchall()
for registro in registros:
   print('Usuário: {} email: {} data criação:
{}'.format(registro[0],registro[3],registro[4]))
db.close()
```

O resultado é esse:

```
Usuário: 1 email: cleuton@test data criação: 2011-05-16 15:36:38
Usuário: 2 email: fulano@test data criação: 2019-01-10 10:53:00
```

Cada registro é uma tupla, com os campos, portanto podemos usar a sintaxe de vetor para pegar cada campo individualmente. Os campos são convertidos para os tipos de dados Python. Por exemplo, o ***timestamp*** é convertido para ***datetime.datetime***.

Também podemos inserir, atualizar ou deletar registros. O programa "sql3.py" mostra um pequeno exemplo:

```
import psycopg2
import urllib.parse as up
import datetime
data = datetime.datetime.now()
up.uses_netloc.append("postgres")
# A url abaixo está propositadamente ofuscada:
url = up.urlparse("postgres://@@@@@@:*****@elmer.db.elephantsql.com:5432/##")
db = psycopg2.connect(database=url.path[1:],
    user=url.username,
    password=url.password,
    host=url.hostname,
    port=url.port
)
cursor = db.cursor()
cursor.execute("INSERT INTO account
(user_id,username,password,email,created_on,last_login) VALUES (%s, %s, %s, %s,
%s, %s)",
    (3, 'Beltrano', 'bbb', 'beltrano@test', data.strftime('%Y-%m-%d %H:%M:%S'),
None))
db.commit()
db.close()
```

Passamos um comando SQL para o método ***execute***, e uma tupla de valores, que serão substituídos posicionalmente nos strings que utilizamos.

Assim como fiz insert, poderia fazer update ou delete. O importante é lembrar de invocar o método ***commit*** antes de encerrar o programa.

Exercício

Para este, não há correção! Crie uma tabela sua no ElephantSQL e faça uma aplicação CRUD (Create, Read, Update e Delete) para inserir dados nela.