CENTRO DE FORMAÇÃO
DE ALCÂNTARA

INTRODUÇÃO À PROGRAMAÇÃO

UFCD(s) 5118, 5119

GUIA DE LABORATÓRIO 3.3
ITERÁVEIS, EXCEPÇÕES E GESTORES DE CONTEXTO (ßeta)

OBJECTIVOS

- Aprender o que são **Iteráveis** , **Iteradores** e **Geradores**
- Aprender o que são **Excepções** e introduzir **Gestores de Contexto**
- Utilizar geradores para definir iteradores e gestores de contexto (estes com **@contextmanager**)

INSTRUÇÕES

Iteráveis, Iteradores: Uma Introdução

1. Inicie o REPL do Python e abra o editor de texto/IDE que costuma utilizar.

2. Adicione o seguinte código ao REPL:

```
>>> nums = [10, 9, 1, 7]
>>> it = iter(nums)
>>> next(it)
10
>>> next(it)
9
>>> next(it)
1
>>> next(it)
7
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

3. E agora tente:

```
>>> it = iter(nums)
>>> for v in it:
...   print(v)
...
10
9
1
7
```

Iteração *é um processo repetitivo com vista a "chegar" a todos os valores obtidos a partir de uma fonte, isto é, a partir de um* ***iterável****. Um* ***iterador*** *é um objecto que representa um "fluxo" de dados criado a partir dessa fonte. Em particular, esse objecto, o iterador, sabe como obter o próximo valor presente nesse fluxo de dados. Podemos definir um* ***iterável*** *como sendo um objecto para o qual se consegue obter um* ***iterador****.*

Em termos da linguagem Python, um iterável é um objecto que permite aceder a fluxo de valores através de um ciclo ***for****. Verifica uma ou mais das seguintes propriedades:*

1. Implementa a operação (ie, o método) ***__iter__****, normalmente invocada pela instrução* ***for****, mas à qual podemos aceder através da função* ***iter****. Esta operação devolve um iterador para o objecto iterável.*

2. É uma sequência, ou seja, implementa a operação ***__getitem__*** *e* ***__len__****, como tal, pode ser indexado numericamente a partir de 0 e eleva a excepção (ver à frente)* ***IndexError*** *caso o índice seja inválido. O Python sabe como obter um iterador para uma sequência.*

Um iterador é um objecto que implementa a operação ***__next__*** *(acessível indirectamente através da função built-in* ***next****). Esta operação devolve o próximo elemento e mantém um "ponteiro" para o elemento seguinte. Se não for possível devolver mais um valor, porque, por exemplo, o fluxo de dados terminou, a operação* ***__next__*** *lança uma excepção do tipo* ***StopIteration****. Por vezes, um iterável pode ser também um iterador, mas um iterador é sempre um iterável. Por outro lado, um iterador poder permitir aceder a um fluxo infinito de valores…*

A maioria dos tipos de dados de Python suportam iteradores e daí poderem ser utilizados em ciclos ***for****. É o caso das listas, strings, tuplos, dicionários, conjuntos (sets), etc.*

Quais as vantagens de utilizar iteradores? Várias. Eis duas delas:

*1. Um iterador permite **"poupar" memória** porque devolve apenas o próximo valor, ao invés de devolver toda a estrutura que está a iterar. Por exemplo, suponhamos que precisamos de consultar uma tabela de uma BD externa com informação sobre 40 milhões de cidadãos. Queremos fazer algum processamento aos 25 milhões de cidadãos nascidos após 1975. Sem iteradores, a consulta teria que "carregar" para memória (eg, para uma lista) os 25 milhões de registos. Com iteradores, podemos estruturar o nosso código de modo que apenas um registo seja necessário em memória: o próximo registo a ser processado (é claro que internamente, para acelerar o processamento, o iterador pode utilizar um*
buffer *com os próximos N registos, mas isso é transparente para o código que utiliza o iterador).*

*2. Um iterador permite **separar o acesso a uma estrutura de dados da sua implementação interna**. Por exemplo, numa determinada fase podemos utilizar um **dict** para guardar informação e mais à frente alterar para uma lista de tuplos (ou outra "coisa qualquer"), desde que continuemos a obedecer ao protocolo dos iteradores (ou seja, a função **iter** deve devolver um iterador; a função **next** aplicada a um iterador permite aceder ao elemento seguinte).*

4. Pode constatar no REPL que as seguintes instruções são equivalentes:

```
for v in iter(nums):
    print(v)

for v in nums:
    print(v)
```

5. E tente agora:

```
>>> it1 = iter(nums)
>>> it2 = iter(it1)
>>> it1 is it2
True
>>> it1, it2
(<list_iterator object at 0x10203aba8>,
 <list_iterator object at 0x10203aba8>)
```

6. Note que podemos (mas não devemos) fazer avançar um iterador com **next** ou com **.__next__** :

```
>>> next(it1)
10
>>> it1.__next__()
9
```

*O seguinte ciclo **for***

```
for x in ITERAVEL:
    print(x)
```

é equivalente a:

```
it = iter(ITERAVEL)
while True:
    try:
        x = next(it)
    except StopIteration:
        break
    else:
        print(x)
```

*Ou seja, internamente a instrução **for** utiliza iteradores e excepções para controlar a iteração. Mais à frente neste laboratório vamos estudar excepções. Por agora interessa saber que, quando um iterador não consegue/pode devolver mais valores, ele "lança" uma excepção do tipo **StopIteration** que é apanhada pela cláusula except **StopIteration** da instrução **try-except**.*

*Apesar de conceptualmente diferentes, por razões práticas um **iterador** é também um **iterável**, ou seja, é possível aplicar a função **iter** a um iterador. O que obtemos? O próprio iterador de volta …*

*Outro aspecto relevante que convém ter em mente é que as funções **next** e **iter** definem o que se chama o **protocolo dos iteradores**. Ora, este protocolo assume que os iteradores avançam apenas num sentido e que não é possível voltar a trás.*

7. Também podemos obter um iterador para outras sequências e, por seu turno, materializar esse iterador num outro tipo de sequência:

```
>>> it1 = iter('Alberto')
>>> chars1 = tuple(it1)
```

```
>>> it2 = iter(chars1)
>>> chars1
('A', 'l', 'b', 'e', 'r', 't', 'o')
>>> chars2 = list(it2)
>>> chars2
['A', 'l', 'b', 'e', 'r', 't', 'o']

>>> idades = {'Armanda': 19, 'Alberta': 15}
>>> nomes = list(iter(idades))
>>> nomes
['Alberta', 'Armanda']
>>> nomes = list(idades)
>>> nomes
['Alberta', 'Armanda']
```

*O Python sabe lidar com iteradores em situações diferentes, sendo a mais importante quando está "à direita" da instrução **for**. Além do **for**, o operador **in** também sabe verificar se um elemento pertence ao fluxo de dados representado pelo iterador. Para tal, o **in** vai avançando o iterador até verificar se o elemento "aparece" antes que o fluxo de dados se esgote.*
*A maioria das funções do Python está preparada para lidar com iteradores e com iteráveis. Aliás, os iteradores são a "cola" que une as várias partes da linguagem/biblioteca Python que lidam com tipos de conjuntos diferentes. É o caso das funções **max**, **min**, **sum**, **all**, **any**, **map**, **filter**, **sorted**, **zip**, etc.*
Note-se que os "objectos ficheiro" também são iteráveis. Daí podermos fazer:

```
for linha in open('bla.txt'):
    # faz qq coisa com linha
```

8. Podemos utilizar o operador **in** com iteradores, mas temos que ter em atenção que o iterador avança até encontrar ao elemento que queremos verificar se pertence ao fluxo de dados:

```
>>> letras = iter('Alberto')
>>> 'e' in letras
True
>>> next(letras)
'r'
>>> 'A' in letras
False
>>> next(letras)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Como podemos criar os nosso próprios iteradores? Como definir, por exemplo, um iterador que devolva os números pares presentes numa lista? É isso que vamos aprender a seguir. De um modo geral, podemos definir um iterador de duas formas:
*1. Definimos uma classe para o nosso iterador. Nesta classe devemos implementar os métodos **__iter__** e o método **__next__** (o método realmente importante é o **__next__**; necessitamos de implementar **__iter__** porque um iterador também deve ser um iterável). Veremos como fazer isto quando aprendermos a trabalhar com classes.*
*2. Alternativamente podemos criar um **gerador**, que é um tipo de iterador. Esta é a forma mais comum e mais simples e aquela que vamos utilizar neste laboratório.*

Geradores

9. Vamos agora aprender a trabalhar com geradores. Crie o ficheiro `geradores1.py` e acrescente o seguinte código

```
def olaMundo():
    yield 'Olá'
    yield 'Mundo!'
```

10. Agora, no REPL, execute o seguinte:

```
for p in olaMundo():
    print(p)
```

***Geradores** são funções que permitem aceder a um fluxo de dados, tal com um iterador. Aliás, um **gerador** é um iterador. Uma função "normal" devolve um valor através da instrução **return** e termina. Um gerador é uma função que devolve um valor através da instrução **yield**, mas que não termina se ainda existir outro **yield** mais à frente. No fundo, um gerador é uma função que consegue produzir um valor, e outro, e outro … até que o iterador subjacente ao gerador se esgote.*
*Qualquer função que possua a instrução **yield** é reconhecida pelo Python como sendo um gerador. Da primeira vez que chamamos a função não obtemos um valor, mas sim um gerador. Para obtermos o primeiro valor (e os seguintes) devemos utilizar a função next ou então passá-lo para um contexto que aceite iteradores (porque um gerador também é um iterador).*

11. Também podemos obter as palavras uma-

a-uma com **next**:

```
>>> g = olaMundo()
>>> g
<generator object olaMundo at 0x1020ba288>
>>> next(g)
'Olá'
>>> next(g)
'Mundo!'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

*Um dos aspectos mais interessantes dos geradores é a possibilidade de manterem o estado das suas variáveis locais entre invocações. Quando uma função normal termina, as variáveis criadas no seu espaço de nomes privado são eliminadas. Com um gerador isso não acontece se este ainda não tiver esgotado o seu fluxo de dados. Aliás, a função next faz com que o gerador retome o fluxo de execução logo a seguir ao último yield. As variáveis que existiam antes do último yield continuam a existir e com os valores que tinham. Os geradores são como que "funções retomáveis" (*resumable functions*).*
Um gerador pode, no entanto, recorrer à instrução ***return*** *para devolver um valor e terminar completamente. O* ***return*** *faz com que o fim do gerador seja assinalado também por via da excepção* ***StopIteration*** *e o valor devolvido faz parte da excepção.*

12. Defina agora a seguinte função:

```
def paresAte(N):
    for i in range(0, N+1, 2):
       yield i
```

13. E agora teste-a com:

```
>>> g = paresAte(4)
>>> next(g)
0
>>> next(g)
2
>>> next(g)
4
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> for p in paresAte(4):
...   print('Número par ->', p)
...
Número par -> 0
Número par -> 2
Número par -> 4
```

14. Para observamos melhor o fluxo de execução do gerador, defina este novo gerador:

```
def paresAteV2(N):
    print("Início do gerador")
    for i in range(0, N+1, 2):
        print("Mais um valor")
        yield i
    print("Fim do gerador")
```

15. Teste esta função tal como testou a primeira versão em cima.

16. Repare que a função **filter**, utilizada numa outra parte do laboratório, poderia ser definida com um gerador. Defina a função `filtra`, o parente luso de **filter**.

```
def filtra(select, iteravel):
    for i in iteravel:
        if select(i):
            yield i
```

17. E agora teste-a com:

```
>>> nums = (15, 1, 2, 0, 19)
>>> g = filtra(lambda x: x % 2 == 1, nums)
>>> g
<generator object filtra at 0x1023c51f8>
>>> list(g)
[15, 1, 19]
```

18. Podemos passar valores para dentro de um gerador através do método **.send**. Estes valores serão "apanhados" e devolvidos pela própria instrução **yield**. Crie a seguinte função:

```
def somador(iteravel):
    for i in iteravel:
        x = (yield)  # este yield recebe um valor passado por .send
        yield x + i  # este apenas devolve o resultado da soma
```

19. Teste a função com:

```
>>> g = somador(nums)
>>> next(g)
>>> g.send(1)
16
>>> next(g)
>>> g.send(1)
2
    … etc …
>>> next(g)
>>> g.send(1)
20
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> g = somador(nums)
>>> for v in g:
...   print(v, g.send(1))
...
None 16
None 2
None 3
None 1
None 20
```

*A primeira operação possível de ser executado sobre um gerador é sempre um **next**. No caso de somador, o primeiro **next** avança o gerador até ao primeiro **yield**. Como este **yield** está à direita de um '=' , o gerador para, devolve o valor à direita do **yield** (neste caso, como não existe nenhum, é devolvido **None**) e aguarda por um **.send**. No exemplo fazemos **g.send(1)** o que significa que estamos a enviar o valor 1 para a primeira ocorrência do **yield** no gerador. O gerador retoma a sua execução, guarda o valor 1 na variável x e avança até ao **yield** seguinte. Este **yield** devolve x + i sendo que i corresponde ao primeiro valor de nums - 15. Assim, na segunda ocorrência do **yield**, o gerador devolve 16. Um novo **next** retoma a execução do gerador e este continua a iterar a sequência nums.*

20. E para finalizar a temática dos iteradores e geradores, recomenda-se, qual cereja em cima do bolo, o visionamento do seguinte vídeo: https://www.youtube.com/watch?v=EnSu9hHGq5o .

Consulte também o exemplo `robot.py` fornecido juntamente com este laboratório.

*Iteradores e geradores são um dos mecanismos mais importantes da linguagem Python e que a tornam uma linguagem poderosa e flexível. Geradores, em particular, pelo facto de serem funções que guardam o estado entre invocações possibilitam estruturar o código em funções colaborativas - funções essas que são designadas de **corotinas**. Este mecanismo é relativamente recente e ainda estão por identificar muitas das técnicas de programação que os geradores permitem.*
Para já, vamos utilizar geradores para criar iteradores de forma mais simples, para definir gestores de contexto (ver à frente) e para definirmos funções que se mantêm "vivas" entre invocações. poderíamos obter tudo isto sem geradores, utilizando classes, mas para os casos mais comuns é mais simples utilizar geradores.

Consultar:
https://docs.python.org/3/tutorial/classes.html#iterators
https://docs.python.org/3/howto/functional.html
https://docs.python.org/3/howto/functional.html#iterators

Excepções e TRY-EXCEPT-FINALLY

21. Crie o script `excepcoes1.py` no editor de text/IDE.

***Excepções** são eventos inesperados na vida de um programa. Normalmente, são situações de erros (eg, dados errados introduzidos pelo utilizador, ligações de rede que "vão abaixo" etc.) que uma determinada parte do programa não consegue dar conta. Por exemplo, a maioria das excepções* built-in *de Python assinalam situações para os quais não está definida uma semântica (eg, a excepção **IndexError** assinala um acesso a um elemento de uma sequência que não existe). Noutras ocasiões, são situações raras mas que não estão relacionados com erros. O Python possui um mecanismo para lidar com essas situações inesperadas que, não só permite a um programa assinalar um **evento** fora do comum (ie, uma excepção), como também possibilita que outra parte do programa decida como resolver o problema.*
*As excepções podem ser **assinaladas** - "**elevadas**", na terminologia do Python, "lançadas", na terminologia de linguagens de programação derivadas de C++, como Java e C# - por código nalguma parte do programa ou pelo próprio interpretador do Python. Quando assinalada através da instrução **raise**, uma excepção é propagada até ser apanhada por uma cláusula **except** de um **try-except**. A palavra-reservada **raise** provoca a interupção imediata do bloco de código que está executada, permitindo a devolução de um objecto, a excepção, que poderá ser apanhada num bloco **try-except**. A sintaxe do **raise** é a seguinte:*

```
raise objecto-que-deriva-de-builtins.Exception
```

*Ou seja, o **raise** "lança" um objecto de uma classe que deriva da classe **Exception** ou de uma outra que derive desta. Ainda não estudámos classes e herança/derivação, todavia para efeitos ilustrativos, podemos criar directamente objectos desta classe.*

22. Vamos definir a função `fun1` com o seguinte código

```
def fun1():
    print("Antes do RAISE")
    raise Exception("Daqui não passa")
    print("Após o RAISE")
```

23. Excecute agora a `fun2` no REPL:

*Ao lançarmos uma excepção com **raise**, a função que o envolve termina imediatamente. Se não existir uma instrução **try-except** "algures" (a seguir vamos perceber o que significa este "algures"), então o programa também termina imediatamente.*
*Neste caso, o segundo **print** não chega a ser executado. A excepção é apanhada pelo REPL que mostra o seu conteúdo (que neste caso é a string "Daqui não passa").*

```
>>> fun1()
Antes do RAISE
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 3, in fun1
Exception: Daqui não passa
```

24. Crie agora a função `fun2` e de seguida execute-a:

```
def fun2():
    try:
        print("Antes do RAISE")
        raise Exception("Daqui não passa")
        print("Após o RAISE")
    except Exception:
        print("No EXCEPT/CATCH")
```

*A mensagem "Após o RAISE" continua a não ser exibida (ie, o **print** respectivo não é executado), mas agora ocorre uma alteração do fluxo do programa que salta da instrução **raise** directamente para a primeira instrução do bloco **except**.*

```
>>> fun2()
Antes do RAISE
No EXCEPT/CATCH
```

*Uma instrução **try**-**except** permite "apanhar" os objectos que constituem as excepções. A sintaxe do **try**-**except** é a seguinte:*

```
try:
   bloco_de_instruções
except [(excepção1, excepcao2, …) | excepcaoX] as var1]:
   bloco_de_instruções_deste_except
# ...
except [(excepçãoA, excepçãoB, …) | excepcaoY] as var2]:
   bloco_de_instruções_deste_except
finally: bloco_de_instruções_final
```

*Ou seja, uma instrução **try**-**except** pode possuir vários **except**s, um por cada tipo de excepção que pretendamos apanhar ou por cada tuplo de excepções. Por outro lado, podemos utilizar o **finally** com código que é executado sempre no final do bloco **try-except** (tendo ou não ocorrido uma excepção, tendo a excepção, caso tenha ocorrido, sido apanhada ou não). Não é obrigatória a definição de qualquer cláusula **except** se o try possuir um **finally**.*

25. No código anterior estamos a "detectar" a excepção, mas não estamos a obter o seu "valor". Defina `fun3` que é uma versão modificada de `fun2`:

```
def fun3():
    try:
        print("Antes do RAISE")
        raise Exception("Daqui não passa")
        print("Após o RAISE")

    except Exception as ex:
        print("No EXCEPT/CATCH. Mensagem -> ", ex)
```

26. É possível "passar" mais do que um valor na criação da excepção. Altere `fun3` e teste-a devidamente:

```
def fun3():
    try:
        print("Antes do RAISE")
        raise Exception("Daqui não passa", 10)
        print("Após o RAISE")
    except Exception as ex:
        print("No EXCEPT/CATCH. Mensagem ->", ex)
        print("E agora os argumentos da excepção ->\n",
              "\targ 0:", ex.args[0], "\targ 1:", ex.args[1])
```

27. Acrescente agora o seguinte código, onde definimos dois tipos de excepção que serão "elevadas" pela função `fun4`:

```
from random import randint

class MaiorQue6(Exception):
    pass

class Entre4e6(Exception):
    pass

def fun4():
    n = randint(1, 10)
    if n > 6:
        raise MaiorQue6(n)
    elif 4 <= n <= 6:
        raise Entre4e6(n)
    else:
        raise Exception("EXCEPTION")
```

*Até agora "lançámos" excepções do tipo **`Exception`**. Mas também é possível definirmos o nosso tipo de excepções. Uma excepção é um objecto de uma classe que deriva de **`Exception`**. Vamos estudar classes em maior detalhe num dos próximos laboratórios, mas ficam aqui alguns dos conceitos básicos:*

*. Uma **classe** é um conjunto de definições relacionadas. A esse conjunto de definições damos um nome, o nome da classe.*

*. Uma **classe** é um tipo de dados. Isto significa que podemos definir variáveis associadas a objectos de um tipo de dados definido com uma classe.*

*. Uma **classe** define-se com a palavra-reservada **`class`** , ao que se segue o nome da classe e, opcionalmente, uma lista de classes das quais esta deriva. De seguida segue-se um conjunto de definições que pode envolver variáveis, funções (que neste contexto se designam por métodos) e outras classes. A palavra-reservada **`class`** é similar a **`def`**, mas para classes.*

. Dada a classe `Xpto`, criamos um objecto desta classe fazendo `Xpto(...zero ou mais argumentos...)`.

*. Aqui ao lado definimos duas classes, `MaiorQue6` e `Entre4e6`, e indicamos que são classes próprias para serem utilizadas como excepções pelo facto de ambas estarem "relacionadas" com **`Exception`**. Como não necessitamos de definir nada dentro de cada uma delas, a sua definição apenas inclui a palavra-reservada **`pass`**.*

28. E agora defina a função `fun5` que invoca a função `fun4`:

```
def fun5():
    try:
        print("Antes de FUN4")
        fun4()
        print("Depois de FUN4")
    except MaiorQue6 as ex:
        print("MAIOR QUE 6 ->", ex)
    except Entre4e6 as ex:
        print("ENTRE 4 E 6 ->", ex)
    finally:
        print("Fim do TRY-EXCEPT-FINALLY")
```

*A cláusula **`finally`** permite especificar um bloco de instruções que será sempre executado após o **`try-except-finally`**, tenha ocorrido ou não uma expceção. O bloco **`finally`** é útil para libertar recursos que tenham sido adquiridos antes ou durante o **`try-except-finally`** e que devem ser libertados, quer ocorra uma excepção, quer o código entre o **`try`** e o **`except`** termine com sucesso (ie, sem o lançamento de excepçoes).*

29. Uma vez que o tipo de excepção lançada é determinado aleatoriamente, teste a função invocando-a várias vezes no REPL.

30. Vamos agora fazer um pequeno *script* que solicita ao utilizador a introdução de dois números e indica se o primeiro é múltiplo do segundo. Crie o ficheiro multiplo.py e acrescente:

```
while True:
    try:
        num1 = float(input("Número 1: "))
        num2 = float(input("Número 2: "))
        print(num1, "é" if num1 % num2 == 0 else "não é", "múltiplo de", num2)
        break
    except ValueError:
        print("ERRO: Número inválido")
    except ZeroDivisionError:
        print("ERRO: Número dois não pode ser 0")
    else:
        opcao = input("Deseja repetir? (N/s) ")
        if opcao.lower() not in ('sim', 's'):
            break
```

*A cláusula **`else`** permite especificar um bloco de instruções que apenas é executado se **não** tiver ocorrido qualquer excepção. Podemos ler este **`else`** como **`noexception`**, nome que também seria apropriado.*

31. A cláusula **`finally`** é útil para libertar recursos. Vamos fazer um script que lê um ficheiro CSV que representa as vendas de uma loja. Cada linha do ficheiro tem três campos por linha, todos separados por vírgula: descrição da despesa, data no formato DD-MM-YYYY e montante. O *script* ignora linhas em branco, mas termina se alguma linha contiver algum erro (um montante que não é um número, uma data/hora inválida, etc.).

```
from datetime import datetime
from decimal import Decimal, DecimalException

def mostraDespesas1(nome_fich):
    fich = None
    try:
        fich = open(nome_fich, 'r')
        for linha in fich:
            if linha.isspace():
                continue
            if linha.strip()[0] == '#':
                continue
            despesa = linha.split(',')
            desc = despesa[0].strip()
            data = datetime.strptime(despesa[1].strip(), '%d-%m-%Y').date()
            montante = Decimal(despesa[2])
            print("{:30} {} {:15.2f}".format(desc, data, montante))
    except (ValueError, DecimalException, IndexError) as ex:
        print("ATENÇÃO: Linha inválida\n->", linha)
    except Exception as ex:
        print("ATENÇÃO: Ocorreu um erro!\n", ex)
    finally:
```

Além de exibir o conteúdo das despesas, esta função detecta linhas inválidas. Uma linha inválida é uma linha que "manifesta" pelo menos um dos seguintes problemas:
*1. Possui menos do que três campos. Vamos tentar indexar um campo que não existe e será lançada a excepção built-in **`IndexError`** pelo operador de indexação.*
*2. A data é inválida. É lançada a excepção built-in **`ValueError`** pela função `datetime.strptime`.*
3. O montante é inválido. É lançada uma excepção do tipo `DecimalException`.

```
        if fich:
            fich.close()

if __name__ == '__main__':
    mostraDespesas1('ficheiro_despesas.txt')
```

A manipulação de um ficheiro envolve várias estruturas de dados do lado do S.O e da própria linguagem. Essas estruturas de dados, possuem atributos que, por exemplo, indicam que operações - leitura, escrita - são permitidas sobre o ficheiro, memorizam a zona do ficheiro a partir da qual a próxima operação deve proceder, representam blocos de memória RAM temporária - buffers - associados ao ficheiro para tornar mais eficiente o acesso ao mesmo, etc.. Todos estes ***recursos****, que são criados e associados ao ficheiro após a função* ***open****, devem ser libertados assim que não necessitarmos dele, mesmo que tenha ocorrido um erro durante a manipulação do ficheiro. Porquê? Para que outros programas possam aceder a este, para que a pataforma sincronize o conteúdo do ficheiro em memória (em* buffers*) com o conteúdo do ficheiro em disco, etc.*
Neste exemplo, o ficheiro é fechado no ***finally*** *porque o bloco de instruções desta cláusula é sempre executado, mesmo que tenha ocorrido uma excepção no bloco do* ***try****, tenha esta excepção sido ou não "capturada" por um* ***except****. Porquê o* ***if*** `fich` *? Porque, caso tenha ocorrido uma excepção na instrução* ***open****, o ficheiro não será aberto e, se for este o caso, não necessita (nem pode, porque o valor de* `fich` *é* ***None****) de ser fechado.*

32. Agora crie um ficheiro de texto com campos os referidos campos separados por vírgulas e definidos pela ordem indicada. Teste a função `mostraDespesas1`.

33. E agora, como complemento ao estudo de excepções, leia o capítulo 7 de *"Introduction to Computation and Programming Using Python"* de Jonh Guttag e o Capítulo 8 do "*The Python Tutorial*", presente na documentação oficial do Python.

Gestores de Contexto e WITH

34. Vamos agora criar a função `mostraDespesas2` que faz o mesmo que a anterior mas que utiliza um gestor de contexto para gerir o acesso ao ficheiro.

```
def mostraDespesas2(nome_fich):
    with open(nome_fich, 'r') as fich:
        try:
            for linha in fich:
                if linha.isspace():
                    continue
                if linha.strip()[0] == '#':
                    continue
                despesa = linha.split(',')
                desc = despesa[0].strip()
                data = datetime.strptime(despesa[1].strip(), '%d-%m-%Y').date()
                montante = Decimal(despesa[2])
                print("{:30} {} {:15.2f}".format(desc, data, montante))
        except (ValueError, DecimalException, IndexError) as ex:
            print("ATENÇÃO: Linha inválida\n->", linha)
            print(ex)
        except Exception as ex:
            print("ATENÇÃO: Ocorreu um erro!\n", ex)
```

*A palavra-reservada **`with`** permite criar e utilizar um gestor de contexto. Genericamente, um **gestor de contexto** é um objecto que sabe alojar e libertar os recursos que são necessários para determinadas operações. De um ponto de vista da linguagem Python, um gestor de contexto é um objecto que implementa as operações `__enter__` e `__exit__`. Quando o gestor de contexto é criado com a palavra-reservada **`with`**, a operação `__enter__` é invocada. Os recursos são alocados/criados. Quando o bloco de instruções do **`with`** termina, seja de forma "normal", seja porque ocorreu uma excepção, a operação `__exit__` é invocada. Um* file object*, os objectos criados e devolvidos pelo **`open`** e que representam ficheiros abertos, são eles próprios gestores de contexto.*

A sintaxe da palavra-reservada é a seguinte:

```
with EXPRESSÃO [as DESTINO] (, EXPRESSÃO as DESTINO)*:
    INSTRUÇÕES
```

*Um gestor de contexto é um objecto de uma determinada classe. Podemos, todavia, definir gestores de contexto sem utilizar (directamente) classes, recorrendo a geradores e à função **`contextmanager`** definida na biblioteca `contextlib`. Note que, no exemplo anterior, o **`with`** dispensa completamente a clásula **`finally`** do **`try-except`**.*

35. O exemplo anterior pode ser simplificado com geradores:

```
def linhasRelevantes(iteravel):
    for linha in iteravel:
        if linha.isspace():
            continue
        if linha.strip()[0] == '#':
            continue
        yield linha

def mostraDespesas3(nome_fich):
    with open(nome_fich, 'r') as fich:
        try:
            for linha in linhasRelevantes(fich):
                despesa = linha.split(',')
                desc = despesa[0].strip()
                data = datetime.strptime(despesa[1].strip(), '%d-%m-%Y').date()
                montante = Decimal(despesa[2])
                print("{:30} {} {:15.2f}".format(desc, data, montante))
        except (ValueError, DecimalException, IndexError) as ex:
            print("ATENÇÃO: Linha inválida\n->", linha)
            print(ex)
        except Exception as ex:
            print("ATENÇÃO: Ocorreu um erro!\n", ex)
```

EXERCÍCIOS DE REVISÃO

1. Um gerador é também um iterável? Justifique a resposta.

2. O que é exibido pelas seguinte instruções (se executadas através de um script):

`print("Bom dia, ", end='')`
`try:`
`  print("Manuel e ", end='')`
`  raise ValueError()`
`  print("Armando.")`
`except ValueError:`
`  print("Mario.")`
`except:`
`  print("Tiago.")`	

`idade = -1`
`while True:`
`  try:`
`  idade = int(input("> "))`
`  print("Dobro da sua idade: {0}".format(idade*2))`
`  break`
`  except Exception:`
`  print("Dados invalidos.")`
`print("Arigato!")`
 	
NOTA: Assuma que o utilizador introduz primeiro `A21` e depois `21`.	

3. O que faz a seguinte instrução: `sorted(set(open(ficheiro)))` ?

EXERCÍCIOS DE PROGRAMAÇÃO

4. Altere os primeiros dois exemplos `mostraDespesas` (1 e 2) de modo a não terminar logo assim que encontra uma linha errada. Ou seja, apenas alerta para o erro e depois prossegue para a próxima linha.

5. Modifique o exemplo `mostraDespesas3` de modo a indicar no final quantas linhas estavam erradas e que linhas foram essas.

6. Defina a função `seleccionaDespesas`, função que recebe o nome de um ficheiro de despesas, um predicado e devolve uma lista de "objectos" (implementados com dicionários) com todas as despesas filtradas por esse predicado. Deve possuir um parâmetro a indicar se linhas com erro devem ser ignoradas ou se deve assinalar ocorrência de uma destas linhas com uma excepção apropriada (como no exemplo do laboratório). Depois deve fazer duas funções `mostraDespesas` para exibir essa lista de objectos de duas formas: 1) de forma similar à da laboratório e 2) em HTML.

7. Defina um gerador para datas semelhante em "espírito" ao `range`. Ou seja, pode receber um, dois ou três argumentos. Se receber apenas um argumento, gera todas as datas da data actual até essa data. Se receber dois argumentos, assume que esses argumentos são datas e gera todas as datas entre as datas passadas como argumento. O terceiro argumento, caso seja utilizado, é um número inteiro que indica o

número de dias de intervalo entre as datas a gerar.

Dê o nome `dateRange` ao gerador.