CENTRO DE FORMAÇÃO
DE ALCÂNTARA

GUIA DE LABORATÓRIO 3.2
FUNÇÕES E ÂMBITOS (*SCOPES*) (ßeta)

OBJECTIVOS

- Aprender a utilizar e a definir **Funções**
- Noção de **Parâmetro**, **Argumento**, **Retorno**, **Parâmetros Opcionais**, parâmetros "***Keyword***"
- Perceber a noção de **Espaço de Nomes** ***(Namespace)*** e **Âmbito**/**Escopo** ***(Scope)***
- Introdução aos **Iteradores** e à programação funcional através de *lambdas* e funções **`map`** e **`filter`**

INSTRUÇÕES

Funções: Introdução

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

2. Crie o script `funcoes1.py` no editor de text/IDE.

> *Até agora temos utilizado diversas funções como* ***`len`***, ***`max`***, ***`min`***, ***`print`***, `shutil.disk_usage`, *etc. Queremos agora ter a possibilidade de definir as nossas funções de modo a podermos generalizar e reutilizar código. Uma função é o mecanismo mais importante que o Python disponibiliza para esse efeito.*

3. Num dos exercícios anteriores desenvolvemos código para calcular a potência de dois números. Como essa e uma necessidade comum em programação, vamos definir uma função designada por `potencia`. Introduza o seguinte código no início do *script*:

```
def potencia(base, exp):
    resultado = 1.0
    for i in range(abs(exp)):
        resultado *= base
    return resultado if exp >= 0 else 1/resultado
```

NOTA: Esta função encontra-se definida no módulo `math` com o nome `pow` (ou seja, podemos utilizá-la fazendo `math.pow`)

> *As funções e métodos permitem dividir um programa em sub-operações e permitem que os programadores utilizem o que outros produziram em vez de começarem do zero. Uma correcta decomposiçao do programa em funções permite organizá-lo em várias partes independentes que podem e devem ignorar os detalhes de implementaçao umas das outras. Uma funçao é, assim, uma espécie de caixa negra que recebe instruções para executar uma operaçao, executa-a e, no final, comunica o resultado. Tal como uma variável, que é um nome para um objecto, uma* ***função*** *consiste de um bloco de código com um nome. Esse bloco de código pode ser executado desde que outra parte do código, ou outra função, a* ***função invocadora****, refira o nome do tal bloco de código, i que vamos designar por* ***função invocada****.*

*Em Python, a **definição** de uma função é feita da seguinte forma:*

```
def nome da funcao(lista de parâmetros):
    bloco de instruções da função
```

Por exemplo, uma função para devolver o maior de dois valores poderia ser definida das seguintes duas maneiras (equivalentes):

```
def max(x, y):
    if x > y:
      return x
    else:
      return y
```

```
def max(x, y):
  return x if x > y else y
```

***def** é uma palavra-reservada que indica ao Python que uma função vai ser definida. A esta palavra segue-se o **nome da função**, que é apenas uma forma de identificar a função, tal como o nome de uma variável identifica um valor ou objecto. A sequência de nomes entre parênteses (`x` e `y` no caso de `max`), constitui a **lista de parâmetros** da função. Por vezes é necessário "passar" informação para a função processar. Essa informação vai parametrizar a função. Um **parâmetro** é uma variável, local à função, utilizada para uma outra função, dita **função invocadora**, "comunicar" com esta função, que é a **função invocada**. Ou seja, quando uma função é utilizada, os seus parâmetros ficam associados aos valores (designados por **argumentos**) indicados durante a **invocação da função**. O bloco de instruções da função pode ser qualquer sequência de instruções válidas em Python. No entanto, existem algumas instruções que só são válidas dentro de funções. Por exemplo, a instrução **return** só pode aparecer dentro uma função e serve para devolver um valor para a função invocadora. Assim que o fluxo de execução dentro da função invocada "atinge" uma instrução **return**, a função termina e o resultado da expressão "à direita" do **return**, se alguma existir, é devolvido para a função invocadora. Note-se que, em Python, não é obrigatório que uma função defina parâmetros nem que devolva qualquer valor explícito. Se uma função não possuir um **return**, então o Python garante que essa função devolve **None**. Por vezes designamos por **procedimento** uma função que não devolve qualquer resultado.*
*Cada parâmetro constitui um novo nome, ie, uma nova variável local (ver em baixo). Os valores dos argumentos são passados para os parâmetros da função invocada por **referência de objecto** (ou por **cópia de referência**), isto é, cada parâmetro recebe uma cópia da referência para o objecto referenciado pelo argumento. Se o valor do argumento for mutável (eg, uma lista) e a função invocada alterar esse valor através do parâmetro correspondente, essa alteração irá afectar a função invocadora.*
*As variáveis definidas dentro de uma função, parâmetros incluidos, são **locais** à função, isto é, elas "existem" apenas dentro da função e durante a invocação desta (são eliminadas quando a função terminar).*
*Opcionalmente, a primeira instrução de uma função pode ser uma docstring. Uma **docstring** é uma string definida com um* triplete *de aspas ou plicas no início de uma função, método, classe ou módulo e que permite introduzir um comentário a respeito dessa função, método, classe ou módulo. No caso de funções, a docstring pode indicar qual o seu propósito, que valor produz, que valores espera encontrar nos parâmetros, e outros comentários pertinentes.*

Aplicando estas noções à função desenvolvida neste passo do laboratorio, temos:
*. **nome da função**: `potencia`*
*. **parâmetros**: `x` e `y`*
*. **tipo de dados do valores devolvidos pela função**: `float`*
*. **variáveis locais**: `x`, `y` e `resultado`*
*. **retorno**: o valor da variável `resultado`*
*. **bloco de instruções da função**:*

```
resultado = 1.0
for i in range(abs(exp)):
    resultado *= base
return resultado if exp >= 0 else 1/resultado
```

4. Defina a função `potencia` no REPL. Os procedimentos exactos dependem do ambiente de desenvolvimento que está a utilizar:

1. Pode abrir o REPL na pasta onde gravou o *script* e depois pode fazer `import funcoes1`. Se alterar o *script* e não pretender sair do REPL, pode utilizar a função `importlib.reload(funcoes1)` depois, é claro, de ter feito `import importlib`.

2. Em ambientes mais sofisticados, como o PyCharm ou o Sublime Text com a extensão SublimeREPL, pode simplesmente "enviar" a função para o REPL pressionando uma combinação de teclas.

3. Se estiver a utilizar o REPL IPython, pode fazer `%run funcoes1`.

4. Em última análise, pode copiar e colar a função no REPL, mas isto nem sempre funciona bem...

5. Insira agora no REPL:

```
>>> potencia(2, 3)
8.0
>>> potencia(4.2, 2)
17.64
>>> res = potencia(2, 0)
>>> print("{:.2f}".format(res))
1.00
>>> soma = potencia(2, 3) * potencia(3, 4)
>>> print("{}".format(soma))
648.0
>>> print("{:.2f}".format(potencia(3, 2)*3 + 10/potencia(2, 2)))
29.50
```

*Vamos assumir que uma função invoca (ou seja, utiliza) outra função. Como vimos, a primeira função é designada de **função invocadora**, ao passo que a segunda é designada de **função invocada**. A invocaçao de uma função envolve os seguintes passos:*
*1. Passar os valores - ie, os **argumentos** -da função invocadora para os **parâmetros** função invocada, se for o caso disso.*
2. Suspender a execução na função invocadora e executar a função invocada.
*3. Apos a execução da função invocada ter terminado, se houver necessidade, passar o **valor devolvido** para a função invocadora.*
4. Retomar a execuçao da função invocadora.
*Uma função termina assim que o fluxo da execução atingir uma instrução **return** ou, caso este fluxo não envolva nenhum **return** ou a função nem sequer inclua uma instrução deste tipo, após ter sido executada a última instrução.*

6. Funções são objectos, isto é, são "valores" com propriedades. Por isso podemos associar uma função a outros nomes e aceder às suas propriedades. Execute:

```
>>> potencia
<function potencia at 0x1021a5488>
>>> potencia.__name__
'potencia'
>>> pot = potencia
>>> pot(2, 3)
8.0
```

7. Vamos agora definir uma função para pedir um valor numérico inteiro ao utilizador compreendido entre dois limites (eg, pedir um valor entre 10 e 20). Se o utilizador inserir um "valor" que não é um número inteiro, a função deve indicar se trata de um número inválido; se inserir um valor fora da gama de valores estabelecida pelos dois limites, a função indica precisamente isso. Em ambos os casos, a função volta a solicitar um número ao utilizador, até que este insira de facto um número compreendido entre ambos os limites.

Acrescente a seguinte definição ao ficheiro `funcoes1.py`:

Consultar na documentação do Python a função `str.isdigit`.

```
def pede_num(msg, lim_inf, lim_sup):
    while True:
        num_str = input(msg)  # .strip()-> remove espaços "à volta"
        if num_str.isdigit():
            num = int(num_str)
            if lim_inf <= num <= lim_sup:
                return num
            else:
                print("ERRO: Número fora do intervalo de "
                      "{0} a {1}.".format(lim_inf, lim_sup))
        else:
            print("Número inválido!")
```

*Note-se que a instrução **`return`**, além de permitir devolver um valor, também termina a função imediatamente. Neste sentido é como que um **`break`** para toda a função onde está envolvida.*

8. Esta função executa duas validações (se o utilizador inseriu um número e se este está entre dois valores) sobre o conteúdo que o utilizador introduziu. Só se passar estas validações é que o valor é devolvido. Uma outra forma, muito habitual em Python, de escrever este código é a seguinte:

```
def pede_num(msg, lim_inf, lim_sup):
    while True:
        num_str = input(msg).strip()
        if not num_str.isdigit():
            print("ERRO: Número inválido!")
            continue
        num = int(num_str)
        if not lim_inf <= num <= lim_sup:
            print("ERRO: Número fora do intervalo de "
                  "{0} a {1}.".format(lim_inf, lim_sup))
            continue
        return num
```

Questão: Como está, a função `pede_num` aceita números negativos?

9. Agora invoque a função com:

```
>>> pede_num("Indique um valor entre 10 e 20: ", 10, 20)
Indique um valor entre 10 e 20: não introduza nada e pressione ENTER
ERRO: Número inválido!
Indique um valor entre 10 e 20: 9
ERRO: Número fora do intervalo de 10 a 20.
```

```
Indique um valor entre 10 e 20: 21
ERRO: Número fora do intervalo de 10 a 20.
Indique um valor entre 10 e 20: 17
17
```

Argumentos com Nome (Keyword Arguments) e Parâmetros Opcionais

10. Note que a função `pede_num` também pode ser invocada das seguintes formas:

```
pede_num("Valor? ", 10, lim_sup=20)
pede_num("Valor? ", lim_inf=10, lim_sup=20)
pede_num(msg="Valor? ", lim_inf=20, lim_sup=20)
pede_num(lim_sup=20, msg="Valor? ", lim_inf=10)
```

> *Os argumentos de uma função podem ser passados por posição, ie, pela ordem pela qual os respectivos parâmetros foram especificados, ou por nome. Neste caso a ordem não tem que corresponder à ordem dos parâmetros na lista de parametros. Por exemplo, consideremos a função*
>
> ```
> def f(a, b):
>     …
> ```
>
> *Se pretedermos invocar a função `f` passando o valor 2 para o parâmetro `a` e o valor 3 para o parâmetro `b`, podemos fazê-lo das seguintes formas: `f(2, 3)`, `f(a=2, b=3)`, `f(b=3, a=2)` ou `f(2, b=3)`. Um argumento passado por nome é designado por "*keyword argument"*, que vamos traduzir livremente para argumento com nome. De notar que não é possível invocar a função assim: f(a=2, 3) ou f(b=3, 2). Porquê? Porque um argumento possicional não pode suceder a um argumento com nome.*

11. Vamos definir valores/argumentos por omissão para os parâmetros `lim_inf` e `lim_sup`. O limite inferior vai ter como argumento por omissão o menor valor de 64 bits - `-(2**63-1))` -, ao passo que o limite superior possui o maior valor de 64 bits - `2**63-1`.

```
def pede_num(msg, lim_inf=-(2**63-1), lim_sup=2**63-1):
    while True:
        num_str = input(msg).strip()
        if not num_str.isdigit():
            print("ERRO: Número inválido!")
            continue
        num = int(num_str)
        if not lim_inf <= num <= lim_sup:
            print("ERRO: Número fora do intervalo de "
                  "{0} a {1}.".format(lim_inf, lim_sup))
            continue
        return num
```

> *Em Python podemos definir **parâmetros com um argumento (ou valor) por omissão**. Quando um parâmetro tem um valor por omissão, então não é obrigatório passar um argumento para esse parâmetro. Deste modo, um parâmetro com valor por omissão é também designado de **parâmetro opcional**. Parâmetros com argumento por omissão devem ocorrer depois dos parâmetros obrigatórios na lista de parâmetros da função.*

12. O método `isdigit` não reconhece números inteiros prefixados com '-' ou '+'. Vamos definir a função booleana `e_inteiro` que recebe uma string e devolve `True` se essa string for um número inteiro, esteja ele prefixado ou não com um sinal.

```
def e_inteiro(num_str):
```

```
    return (num_str.isdigit() or
            (num_str and num_str[0] in ('-', '+') and num_str[1:].isdigit()))
```

13. Já agora, seria útil incluir um parâmetro opcional na definição de `e_inteiro` para indicar se espaços em branco a envolver o número devem ser ou não ignorados:

```
def e_inteiro(num_str, ignorar_espacos=True):
    num_str = num_str.strip() if ignorar_espacos else num_str
    return (num_str.isdigit() or
            (num_str and num_str[0] in ('-', '+') and num_str[1:].isdigit()))
```

14. Agora a função `pede_num` passa a recorrer à função `e_inteiro` para decidir se um número é inteiro ou não.

```
def pede_num(msg, lim_inf=-(2**63-1), lim_sup=2**63-1):
    while True:
        num_str = input(msg)  # já não é necessário .strip() aqui
        if not e_inteiro(num_str):
            print("ERRO: Número inválido!")
            continue
        num = int(num_str)
        if not lim_inf <= num <= lim_sup:
            print("ERRO: Número fora do intervalo de "
                  "{0} a {1}.".format(lim_inf, lim_sup))
            continue
        return num
```

15. Teste agora a função com:

```
>>> pede_num("Num> ", -10, 10)
Num> -15
ERRO: Número fora do intervalo de -10 a 10.
Num> 15
ERRO: Número fora do intervalo de -10 a 10.
Num> -4
-4
>>> pede_num("Num> ", 17)
Num> 10
ERRO: Número fora do intervalo de 17 a 9223372036854775807.
Num> 18
18
>>> pede_num("Num ", lim_sup=100)
Num> 150
ERRO: Número fora do intervalo de -9223372036854775807 a 100.
Num> 57
57
```

*Repare que pode passar um argumento para `lim_inf` "posicionalmente" e não especificar um para `lim_sup`. No entanto, se pretender dar um valor a `lim_sup` e deixar que `lim_inf` fique com o valor por omissão, então tem que utilizar argumentos com nome (*keyword arguments*).*

Número Variável de Parâmetros -*args e **kwargs

16. Uma função pode aceitar um número variado de argumentos , quer argumentos posicionais, quer argumentos com nome. A seguinte função junta numa string todos os seus argumento separados por espaço:

```
def concat(*args):
    return ' '.join(args)
```

*Ao ser precedido com * o parâmetro `args` vai "apanhar" todos os argumentos posicionais da função concat. Este parâmetro é um **tuplo** onde `args[0]` corresponde ao primeiro argumento, `args[1]` ao segundo, etc.*
*Após um parâmetro do tipo *args apenas podemos especificar parâmetros opcionais.*
Também podemos indicar que um parâmetro para apanhar todos os argumentos com nome.
*Para tal, basta definir um parâmetro precedido de **, normalmente designado de `kwargs` ou `kargs`. Este parâmetro, que tem que ser o último da lista de parâmetros da função, vai ser um **dicionário** com todos os* keyword arguments *não especificados na lista de parâmetros da função.*

17. Teste a função com `concat('Alberto', 'Armando')`

18. Podemos personalizar o separador. Modifique a função para:

```
def concat(*args, sep=' '):
    return sep.join(args)
```

19. Agora teste com `concat('Alberto', 'Armando', sep='/')`

20. Finalmente, defina a seguinte função:

```
def func1(arg1, arg2, *args, kwarg1=None, kwarg2=-1, **kwargs):
    print('arg1   ->', arg1)
    print('arg2   ->', arg2)
    print('args   ->', args)
    print('kwarg1 ->', kwarg1)
    print('kwarg2 ->', kwarg2)
    print('kwargs ->', kwargs)
```

21. E agora "medite" sobre ela, testando-a com:

```
func1()                  | func1(12, 13, 14, 15)
func1(12)                | func1(12, 13, 14, 15, kwarg1=16)
func1(12, 13)            | func1(12, 13, 14, 15, kwarg1=16, kwarg2=17)
func1(12, 13, 14)        | func1(12, 13, 14, 15, kwarg1=16, kwarg2=17, kwarg3=18)
func1(12, 13, 14, 15)    | func1(12, 13, 14, 15, kwarg1=16, kwarg2=17, x=18, y=19)
```

Escopos e Espaços de Nomes

22. "Escopos" (*scopes*) ou "Espaços de Nomes" (*namespaces*) estão relacionados com o contexto onde os nomes que criamos (para dar a variáveis, funções etc.) são válidos. Neste passo do laboratório, a sua tarefa consiste em ler a secção 4.1.3 do livro "Introduction to Computation and Programming Using Python" referido na secção de referências do programa destes módulos.

23. Estudar o que são variáveis locais e globais e verificar qual a utilidade de global e nonlocal. Consultar **https://docs.python.org/3/faq/programming.html#what-are-the-rules-for-local-and-global-variables-in-python**.

24. Consultar também 4.4 do mesmo livro.

Docstrings

25. Uma *docstring* é uma string definida com `"""` ou `'''` colocada logo no início da função com o intuito de a documentar. Por exemplo, a função pede_num poderia ser definida com o seguinte documentação inicial.

```
def pede_num(msg, lim_inf=-(2**63 - 1), lim_sup=2**63 - 1):
    """
    Solicita a introdução de um número inteiro ao utilizador. O número
    deverá ser superior ou igual a `lim_inf` e inferior ou igual a
    `lim_sup`. Se o utilizador não introduzir um valor nestas condições
    a função `pede_num` indica o erro e volta a tentar. O parâmetro
    `msg` corresponde à mensagem de solicitação.

    A função devolve o número introduzido.
    """
    while True:
        num_str = input(msg)

        ... e agora viria o resto da função, tal como definida em cima ...
```

26. Teste agora fazendo:

Ver na documentação do Python a definição da função `str.isalnum`.

```
>>> print(pede_num.__doc__)
... conteúdo propositadamente não exibido ...
>>> help(pede_num)
... conteúdo propositadamente não exibido ...
```

Funções Internas e *Closures*

27. Em Python podemos definir funções dentro de funções. A título de exemplo, vamos definir uma função para detectar se uma string é um palíndromo (texto que se lê de igual forma da esquerda para a direita ou da direita para a esquerda). Esta função apenas considera letras e números e ignora todos os restantes símbolos (eg, pontuação) e a capitalização das letras.

```
def e_palindromo(txt):
    """
    Detecta se a string `txt` é um palíndromo. Ignora a capitalização
    das letras e todos os caracteres que não sejam letras ou números.
    """
    def filtra_alfanum():
        chs = []
        for ch in txt.lower():
            if ch.isalnum():
                chs.append(ch)
        return chs
```

```
    def e_pal(seq_chs):
        if len(seq_chs) <= 1:
            return True
        else:
            return seq_chs[0] == seq_chs[-1] and e_pal(seq_chs[1:-1])

    return e_pal(filtra_alfanum())
```

A função `e_palindromo` utiliza duas funções auxiliares `filtra_alfanum`, que faz o que nome indica (ou seja, filtra os caracteres alfanuméricos), e `e_pal` que na verdade contém o algoritmo de reconhecimento do palíndromo. Ambas as funções poderiam ter sido definidas fora da classe, só que dado que não têm utilidade fora dela (bem, talvez `filtra_alfanum` tenha utilidade noutros contextos), ***foi decidido "arrumá-las" dentro da própria função e `palindromo`****. A função `filtra_num` devolve uma sequência de caracteres apenas com letras e números. Como uma* ***função interna tem acesso às variáveis definidas na função externa****, `filtra_alfanum` não necessita de parâmetros.*
De notar que a função `e_pal` invoca-se a si própria na sua própria definição. É o que se designa por ***função recursiva****. De facto, um palíndromo é um texto onde o primeiro e último caracteres são iguais e onde o restante texto é ele próprio um palíndromo. A função `e_pal` traduz directamente noção. Esta possibilidade, de uma função invocar-se a si própria, permite exprimir determinados algoritmos de forma mais simples e declarativa. Todavia, na maioria das linguagens, e em Python em particular, funções recursivas tendem a necessitar mais memória temporária (*stack memory*). Devem ser utilizadas com cautela.*

Consultar secção 4.3 de "Introduction to Computation and Programming Using Python"

28. Teste a função de forma apropriada.

29. Uma função externa pode devolver uma função interna podendo esta "aprisionar" o estado de uma variável definida na função externa. Por exemplo, vamos definir uma função que recebe um valor e devolve um somador de N unidades (ie, devolve uma outra função que recebe um argumento, soma-lhe N e devolve o resultado).

```
def somador(n):
    def soma_n(x):
        return n + x
    return soma_n
```

A função interna `soma_n` "aprisiona" o valor de `n` que é passado para `somador`. De facto, depois da função `somador` terminar, a sua variável local `n` não pode ser logo destruída porque existe código que ainda a irá referencia: a função interna `soma_n`. Uma função como `soma_n`, que "prende" parte do contexto que lhe deu origem, é designada por **closure** *(desconhecemos uma tradução, mas poderia ser algo como "fechamento")*
A função `somador`, para todos os efeitos, é uma função geradora de código. Ela gera uma outra função que soma `n` ao argumento.

30. Teste `somador` com:

```
>>> s1 = somador(10)
>>> s1(3)
13
>>> s1
<function somador.<locals>.soma_n at 0x1021a5730>
>>> s2 = somador(15)
>>> s2(3)
18
>>> s1
<function somador.<locals>.soma_n at 0x10244eea0>
```

Funções Anónimas (*Lambdas*)

31. Vamos redefinir a função `soma_n` de modo a utilizar uma **`lambda`** (ie, uma função anónima):

```
def somador(n):
    return lambda x: n + x
```

*A função `soma_n` também poderia ser definida com uma **`lambda`**. Uma lambda é uma função sem nome, ie, anónima. O termo lambda vem de uma área da matemática designada por cálculo lambda. Em Python as lambdas servem para definir pequenas funções, de uma instrução apenas, sendo que essa instrução tem que ser uma expressão, ie, tem que devolver um valor. Como tal, as lambda não devem incluir a palavra reservada **`return`** porque esse **`return`** é implícito. A sintaxe geral de uma **`lambda`** é:*

***`lambda`** zero ou mais parâmetros: expressão*

Uma lambda pode possuir vários parâmetros, obrigatórios ou não, e pode receber argumentos com nome.

32. Outro exemplo: dada uma lista de valores pretendemos filtrar todos os valores pares para uma outra lista. Primeiro começamos por definir a função `e_par` que devolve **`True`** se o seu argumento for um número par:

```
def e_par(x):
    return x % 2 == 0
```

33. Para efeitos de teste, defina a lista de números `nums`:

```
nums = [1, 7, 2, 8, 5, 171, 90, 17]
```

34. Agora vamos recorrer à função *built-in* **`filter`** passando-lhe a função `e_par` como primeiro argumento.

```
>>> list(filter(e_par, nums))
[2, 8, 90]
```

35. As funções anónimas são especialmente interessantes para serem utilizadas como argumentos de outras funções. De seguida vemos o exemplo anterior mas com uma lambda a substituir a função `e_par`:

```
>>> list(filter(lambda x: x % 2 == 0, nums))
[2, 8, 90]
```

36. Experimente agora o seguinte:

```
>>> list(map(lambda x: x * 2, nums))
[2, 14, 4, 16, 10, 342, 180, 34]
```

*A função **`filter`** recebe uma função e um conjunto de valores (*) e aplica essa função a cada um dos valores do conjunto, devolvendo um objecto designado por iterador. Noutro laboratório veremos o que são iteradores e geradores, mas, em geral, podemos pensar num **iterador** como um objecto que, dada uma colecção de elementos, sabe qual o próximo elemento a aceder (é uma espécie de apontador para o próximo elemento). Neste caso concreto, o iterador sabe que elementos é que foram filtrados e qual o próximo a aceder. Passando o iterador para uma lista, ou para um tuplo, ou para outro conjunto qualquer que saiba processar estes iteradores, obtemos então os elementos filtrados.*

*A função **`map`** recebe uma função e um ou mais conjuntos de valores (*) e devolve um iterador que permite aceder a um outro conjunto de elementos que resulta de aplicar a função a cada um dos elementos do conjunto original. No exemplo do laboratório, através de **`map`** a lambda multiplica cada um dos elementos de `nums` por dois. No final obtemos uma lista em que cada elemento é o dobro do elemento correspondente em `nums`.*

*(*) Na verdade, **`filter`** e **`map`** não recebem conjuntos mas iteráveis. O que são iteráveis? Aguarde por um dos próximos laboratórios...*

37. Como tarefa final, consulte, estude e medite sobre os exemplos presentes em `funcoes2.py`, ficheiro que será fornecido juntamente com este guia de laboratório.

EXERCÍCIOS DE REVISÃO

1. Defina *função*, *parâmetro*, *argumento*, *argumento com nome* e *parâmetro opcional*.

2. Suponha que pretende utilizar a função `math.pow` mas pretende invocá-la com o nome elevado_a. Como é que poderia proceder para atingir esse objectivo?

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

`y = 10`
`def f(x):`
`  print(x + y)`
`f(3)`
`def g():`
`  y = 20`
`  f(3)`
`g()`	

`y = 5`
`def func1():`
`  def f(x):`
`  print(x + y)`
`  f(10)`
`def func2():`
`  def f(x):`
`  print(x + y)`
`  y = 50`
`  f(10)`
`func1()`
`func2()`	

`y = 10`
`def func3():`
`  def f1():`
`  y = 1`
`  def f2():`
`  nonlocal y`
`  y = 2`
`  def f3():`
`  global y`
`  y = 3`
`  y = 0`
`  f1()`
`  print(y)`
`  f2()`
`  print(y)`
`  f3()`
`  print(y)`
`func3()`
`print(y)`	

4. O que é uma função anónima e qual a finalidade da palavra-reservada `lambda`? Quantas instruções ou comandos podem pertencer a uma *lambda*?

5. As *lambdas* introduzem alguma funcionalidade na linguagem que, de outro modo, não poderia ser obtida?

6. O que fazem as *built-in* `globals` e `locals`?

7. No laboratório, definimos a função interna `filtra_alfanum` para filtrar os caracteres alfanuméricos de uma string. Como poderia definir esta função em termos de `filter`?

8. Dada o tuplo `vals = (2, 0, 1, 3, 2, 0, 1, 5)` e a string `txt = 'Dinamarca'`, com que valores ficam as variáveis nas atribuições seguintes:

8.1 `a = list(filter(lambda x: x > 2, vals))`

8.2 `b = list(map(lambda x: x > 2, vals))`

8.3 `c = tuple(filter(lambda ch: ch in 'aeiou', txt))`

8.4
```
d = '/'.join(filter(lambda ch: ch in 'aeiouAEIOU',
                    map(lambda ch: chr(ord(ch)+1), txt)))
```

EXERCÍCIOS DE PROGRAMAÇÃO

9. Desenvolva a função `dobro` que recebe um valor e devolve o dobro desse valor. De seguida, desenvolva as funções `triplo`, `quadrado` e `cubo`.

10. Desenvolva uma função para calcular a fórmula resolvente. A função deve chamar-se `resolvente`, deve possuir três parâmetros de entrada, `a`, `b` e `c`, e deve devolver um tuplo com os dois resultados.

NOTA: Dada uma equação do segundo grau

$$ax^2+bx+c=0$$

podemos determinar o valor de x que verifica a igualdade fazendo

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

11. Desenvolva a função `confirma` que solicita uma confirmação ao utilizador. Enquanto o utilizador não introduzir `'sim'`, `'s'`, `'não'`, `'nao'` ou `'n'`, a função repete a mensagem de confirmação. Ao fim de um número de tentativas (quatro, por omissão) a função desiste e assume que o utilizador não confirmou. A função devolve `True`, se o utilizador confirmar, e `False`, caso contrário. Além da mensagem de confirmação e do número de tentativas, a função recebe uma mensagem de erro que deverá ser exibida

quando o utilizador não introduzir o pedido. Este parâmetro deve possuir o valor por omissão: `"Introduza (S)im ou (N)ao"`.

12. Desenvolva a função `inverte` que recebe uma sequência de elementos e devolve uma lista com os elementos por ordem inversa. Não utilize `reversed` ou qualquer outra função já desenvolvida para o efeito.

13. Desenvolva a função `conta_palavras` que indica quantas palavras estão presentes numa string. Uma palavra é uma sequência de caracteres não brancos. Caracteres brancos podem ser o espaço, mudança de linha, tabulação, etc. Não utilize funcões de string com `split`, `partition`, `find` e similares.

14. Desenvolva a função `todas_palavras` que devolve uma lista com todas as palavras presentes numa string. Não utilize funcões de string com `split`, `partition`, `find` e similares.

15. Desenvolva a função `substitui` que recebe três strings e devolve uma nova string onde todas as ocorrências da segunda string na primeira são substituídas pela terceira string.

```
>>> substitui('aXYZbXYZc', 'XYZ', '1')  -> 'a1b1c'
>>> substitui('aXYZbXYZc', 'XYZ', '')   -> 'abc'
>>> substitui('abab', 'b', 'XYZ')       -> 'aXYZaXYZ'
```

Desenvolva esta versão sem utilizar funções/métodos para trabalhar com strings (como `str.replace`, `str.split`, `str.partition`, etc.). Pode utilizar listas, `str.join` e `str.find`. Se utilizar `str.find`, tente desenvolver uma segunda versão semelhante mas sem `str.find`.

16. Agora desenvolva uma nova versão da função anterior, desta vez sem restrições (mas não pode utilizar `str.replace`).

17. Desenvolva a função `remove_white` que remove todos os caracteres "brancos" (espaços, nova linha, tabs, etc.) de uma string.

18. Desenvolva a função `month_name` que pode receber um número de mês - de 1 a 12 - ou uma data - ie, um objecto do tipo `date` ou `datetime` - e devolve o nome do mês. Se a função não receber nenhum argumento, devolve o nome do mês actual. Ignore os dados de localização.

19. Desenvolva a função `rand_letter` que selecciona aleatoriamente uma letra e devolve-a. Esta função deverá possuir um parâmetro opcional `type_` que indica a `rand_letter` que tipo de letra deve seleccionar: se for `'U'` escolhe uma letra maiúscula; se for `'L'`, escolhe uma minúscula; para qualquer

outro valor de `type_`, `rand_letter` escolhe uma letra tendo em conta as duas categorias de valores. Por omissão `type_` tem o valor `''` (string vazia) o que indica que pode devolver uma letra maiúscula ou minúscula.

20. Desenvolva a função `rand_letters` que devolve uma string com letras seleccionadas aleatoriamente. Possui dois parâmetros: `n`, que indica quantas letras terá a string, e `type_` (ver `rand_letter`). Por omissão, `n` tem o valor `2` e `type_` tem o valor `''` (string vazia).

21. Consulte alguns tutoriais sobre *sockets* (ver ligações em baixo) e desenvolva as aplicações seguintes:

21.1 Uma aplicação de *chat* cliente/servidor para enviar mensagens de texto entre dois programas, possivelmente localizados em máquinas diferentes. Uma mensagem pode ter várias linhas mas não pode exceder os 140 caracteres. O utilizador deve terminar a última linha com `!SEND` para enviar a mensagem.

Não se preocupe com detecção e correcção de erros, nem aspectos mais avançados como *threads*, multiprocessamento ou mesmo formas mais avançadas de comunicação entre processos.

21.2 Adapte a aplicação anterior para transferir um ficheiro especificado na linha de comandos. A aplicação cliente deve enviar o ficheiro e o nome do ficheiro a guardar no servidor. O servidor guarda o ficheiro na directoria de arranque.

21.3 Utilizando o módulo `hashlib`, acrescente uma forma rudimentar de verificação da integridade das mensagens enviadas pela aplicação de *chat*. Gere um resumo (*hash*) da mensagem a enviar e anexe-a à mensagem. O receptor recebe a mensagem e o *hash* enviado, após o que verifica se a mensagem está consistente com o *hash*, gerando ele próprio um *hash* a partir da mensagem e comparando-o com o enviado.

Utilize em todos os casos *sockets stream* (TCPv4).
https://docs.python.org/3/howto/sockets.html
http://www.tutorialspoint.com/python/python_networking.htm (Python 2)
http://www.binarytides.com/programming-udp-sockets-in-python/ (Python 2)