# GUIA DE LABORATÓRIO 1.2
# ELEMENTOS BÁSICOS DA LINGUAGEM - PARTE II (ßeta)

## OBJECTIVOS

- Estruturas de dados agregadas mais habituais em Python: strings, listas, tuplos e dicionários.
- Controlo da execução: decisões IF-ELIF-ELSE e MATCH-CASE
- Escrita e execução de *Scripts*

## INSTRUÇÕES

### Strings (continuação da Parte I)

**1.** As strings suportam imensas operações. Resumidamente, aqui ficam algumas das mais comuns

```
>>> nome = 'aRMANDO'
>>> nome = nome.capitalize()
>>> nome
'Armando'
>>> # ver também endswith
>>> nome.startswith('ar')
False
>>> nome.startswith('ra')
False
>>> nome.startswith('Ar')
True
>>> nome.startswith('ma', 2)
True
>>> nome.upper()
'ARMANDO'
>>> nome = nome.lower()
>>> nome
'armando'
>>> # find: devolve índice de 'ma'
>>> nome.find('ma')
2
>>> nome.find('a')
0
# procura a partir do índice 1
>>> nome.find('a', 1)
3
>>> nome.rfind('a')
3
```

*Consultar estas e outras operações relacionadas com **strings** em:*
*https://docs.python.org/3/library/stdtypes.html#string-methods*

*Consultar operações relacionadas com **sequências** (strings, listas, tuplos e outras estruturas de dados) em:*
*https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range*

*É interessante notar que, ao contrário de outras operações, como **len**, **print**, **dir**, e outras, cuja sintaxe de utilização é*

`operacao(argumentos)`

*com a strings estamos a utilizar a forma*

`uma_string.operacao(outros_argumentos)`

*Isto deve-se ao facto de **len** e "companhia" serem funções globais. Ou seja, são operações que estão definidas apenas no espaço de nomes global de um determinado módulo. Os seus operandos, também designados de argumentos, devem ser passados entre parênteses. As operações que manipulam strings são específicas de … strings e foram definidas no espaço de nomes da classe **str**. Falaremos de classes em maior detalhe noutros laboratórios, em todo o caso ficamos já a saber que uma **classe** é como que um modelo a partir do qual criamos objectos; nesse modelo indicamos as operações suportadas pelos objectos. Essas operações, que também são funções, são designadas de **métodos** e acedemos a elas fazendo `obj.operacao(...)`.*

*As operações, isto é, os métodos `startswith`, `endswith`, `upper`, etc, estão definidos na class **str**. Assim sendo, por vezes vamos nos referir a elas desta forma: **str**.`startswith`, **str**.`endswith`, **str**.`upper`, etc.*

```
>>> # split: divide nome "à volta" de
>>> # uma string (neste caso, de 'm')
>>> nome.split('m')
['ar', 'ando']
>>> nome.split('a')
['', 'rm', 'ndo']
```

**2.** Em Python, uma das operações mais importantes é **`str.format`** (também existe uma função *built-in* com nome `format` mas que não vamos abordar para já). Com o Python 3.6 apareceram as f-strings que são uma notação compacta para obter o mesmo efeito que se obteria com **`str.format`**. Vejamos alguns exemplos:

```
>>> "{0}, {1}, {2}".format('a', 'b', 'c')
a, b, c
>>> "{}, {}, {}".format('a', 'b', 'c')
a, b, c
>>> "{0}{1}{0}".format('abra', 'cad')
abracadabra
>>> "{0}, {1}, {0}".format('a', 'b', 'c')
a, b, a
```

*O método **`str`**`.format` permite controlar a formatação de forma muito precisa. Aqui apresentamos apenas alguns exemplos.*
*Este método define uma linguagem de formatação que poderá consultar em:*
*https://docs.python.org/3/library/string.html#format-string-syntax*

```
>>> print("|{}|".format('abc'))
|abc|
>>> print("|{:10}|".format('abc'))
|abc       |
>>> print("|{:>10}|".format('abc'))
|       abc|
>>> print("|{:^10}|".format('abc'))
|   abc    |
>>> print("{} uvas\n{} uvas\n{} uvas".format(2, 121, 71))
... propositadamente não exibido, comparar com saída em baixo ...
>>> print("{:>4} uvas\n{:>4} uvas\n{:>4} uvas".format(2, 121, 71))
... propositadamente não exibido ...
```

```
>>> i, k = 20, 2.77
>>> '{} {}'.format(i, k)
'20 2.77'
>>> '{:d} {:f}'.format(i, k)
'20 2.770000'
>>> '{:f} {:f}'.format(i, k)
'20.000000 2.770000'
>>> '{:.1f} {:.1f}'.format(i, k)
'20.0 2.8'
>>> '{:8.1f} {:8.1f}'.format(i, k)
'    20.0      2.8'
```

```
>>> i, k = 20, 2.77
>>> f'{i} {k}'
'20 2.77'
>>> f'{i:d} {k:f}'
'20 2.770000'
>>> f'{i:f} {k:f}'
'20.000000 2.770000'
>>> f'{i:.1f} {k:.1f}'
'20.0 2.8'
>>> f'{i:8.1f} {k:8.1f}'
'    20.0      2.8'
```

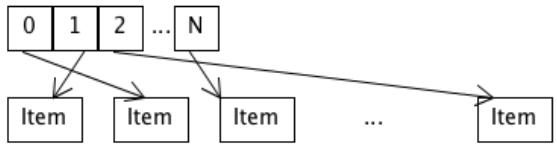
### Listas

**3.** Vamos agora trabalhar com listas. Comece por introduzir:

```
>>> nomes = ["Alberto", "António", "Armando", "Arnaldo"]
>>> nomes
```

```
['Alberto', 'António', 'Armando', 'Arnaldo']
>>> # lista podem ser indexadas e fatiadas
>>> nomes[0], nomes[len(nomes) - 1], nomes[-len(nomes)], nomes[-1]
('Alberto', 'Arnaldo', 'Alberto', 'Arnaldo')
>>> nomes[-1:]  # devolve uma lista com último
['Arnaldo']
>>> nomes[1:]
['António', 'Armando', 'Arnaldo']
>>> # 'António' está em nomes?
>>> 'António' in nomes
True
```

*Consultar estas e outras operações em:*
*https://docs.python.org/3/tutorial/datastructures.html#more-on-lists*




***Uma lista em memória***

*Uma lista é uma sequência de elementos de qualquer tipo de dados dispostos consecutivamente em memória. Geralmente os elementos são do mesmo tipo de dados, mas isso não é obrigatório. Note-se que ao contrário de estruturas de dados como* arrays *(que também existem em Python, mas são mais comuns em C ou C++), os elementos não estão alinhados contiguamente em memória; as referências para os elementos, essas sim, é que estão alinhadas contiguamente em memória.*

*Ao contrário de strings e tuplos, as listas são* ***mutáveis****. Quer isto dizer que, não só podemos alterar o conteúdo dos elementos da lista, como podemos modificar a sua estrutura (acrescentar ou remover elementos). Note-se que as fatias devolvem novas listas. No entanto, ao contrário das strings, podemos utilizar fatias para modificar parte das listas.*

**4.** Podemos modificar o conteúdo da lista:

```
>>> nomes[0] = 'Alberta'
>>> nomes
['Alberta', 'António', 'Armando', 'Arnaldo']
>>> nomes[1:3] = ['Antónia', 'Armanda']
>>> nomes
['Alberta', 'Antónia', 'Armanda', 'Arnaldo']

>>> # vamos substituir o último
>>> nomes[-1] = 'Arnalda'
>>> # [-1:] devolve uma lista com o ultimo
>>> nomes[-1:]
['Arnalda']
>>> # vamos acrescentar um nome
>>> nomes[-1:] = [nomes[-1], 'Andreia']
>>> nomes
['Alberta', 'Antónia', 'Armanda', 'Arnalda', 'Andreia']

>>> # também podemos usar + ou += para acrescentar
>>> nomes2 = nomes + ['Anabela', 'Arlete']
>>> nomes2 += ['Ana']
>>> nomes, nomes2
(['Alberta', … , 'Andreia'], ['Alberta', … , 'Andreia', 'Anabela', 'Arlete', 'Ana'])

>>> # mas a melhor maneira de acrescentar é com append e extend
>>> nomes.append("Anabela")
>>> nomes.extend(["Arlete", "Ana"])

>>> # remover parte(s) da lista ([] é a lista vazia)
>>> nomes[1:3] = []   # também dá -> del nomes[1:3]
>>> nomes
```

```
['Alberta', 'Arnalda', 'Andreia', 'Anabela', 'Arlete', 'Ana']
>>> # se pretendermos aceder e/ou remover um elemento
>>> # podemos usar pop
>>> nomes.pop(1)  # índice é opcional; por omissão é 0
'Arnalda'
>>> nomes
['Alberta', 'Andreia', 'Anabela', 'Arlete', 'Ana']
>>> # e agora vamos apagar a lista
>>> nomes[:] = []  # mesmo que nomes.clear() ou del nomes[:]
>>> nomes
[]

>>> # Algumas formas de criar listas...
>>> nums_repetidos = [4.2] * 10
>>> nums_repetidos
[4.2, 4.2, 4.2, 4.2, 4.2, 4.2, 4.2, 4.2, 4.2, 4.2]
>>> txt_repetido = ['A'*3] * 6
['AAA', 'AAA', 'AAA', 'AAA', 'AAA', 'AAA']
>>> type([1, 2, 3])
<class 'list'>
>>> list("ABC")
['A', 'B', 'C']
```

### Tuplos

**5.** Tuplos são listas imutáveis, ie, listas *"read-only"*. Tal como listas, são sequências de valores separados entre virgulas, só que agora utilizamos parênteses curvos - ( e ) - para delimitar a lista:

```
>>> vogais = ('a', 'e', 'i', 'o', 'u')
>>> vogais[0]
'a'
>>> vogais[-1]
'u'
>>> vogais[1:3]
('e', 'i')
```

*Estrturas de dados **imutáveis** são muito úteis em programação: não só elas indicam a quem estiver a ler o código que determinado conjunto de valores não vai ser alterado, como também transmitem essa informação ao ambiente de execução do Python, e este vai actuar como um "polícia" e não vai permitir que, por acidente, alguém tente alterar o tuplo. Além do mais, em determinadas circunstâncias, sabendo de antemão que uma estrtura de dados não vais ser alterada, o Python consegue optimizar o acesso a esses dados e com isso melhorar o desempenho do código.*

*Consultar estas e outras operações em:*
*https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences*

**6.** Em determinadas circustâncias os parênteses curvos são opcionais:

```
>>> vogais = 'a', 'e', 'i', 'o', 'u'
>>> vogais
('a', 'e', 'i', 'o', 'u)
```

**7.** Como indicado anteriormente, os tuplos, à semelhança das strings, são imutáveis. Quer isto dizer que não podemos alterar a estrutura do tuplo, quer removendo elementos, quer substituindo os elementos em determinadas posições:

```
>>> valores = (10, 20, 30)
>>> valores[0] = 100
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

'tuple' object does not support item assignment
>>> valores[0:2] = (100, 200)
Traceback (most recent call last):
etc...
```

**8.** Note que se pretendermos definir um tuplo apenas com um elemento (situação pouco habitual, diga-se), devemos colocar uma virgula após esse elemento.

```
>>> pessoas = ('alberto')
>>> pessoas       # pessoas é uma string
'alberto'
>>> pessoas = ('alberto',)
>>> pessoas       # pessoas é um tuplo
('alberto',)
>>> pessoas = ['alberto']
>>> pessoas       # pessoas é uma lista
['alberto']
```

*Um tuplo com um elemento necessita de uma virgula a seguir a único elemento do tuplo. Isto deve-se ao facto de o Python não perceber se os parênteses estão a ser utilizados como "agrupadores" de expressões ou como delimitador de tuplos. Tal não é necessário com outras estruturas de dados (como listas ou sets, por exemplo).*

**9.** Um tuplo pode conter elementos mutáveis como listas ou dicionários. Esses elementos, mesmo estando dentro do tuplo, continuam a poder ser alterados:

```
>>> listas = ([10, 20, 30], [100, 200, 300])
>>> listas[0] = [1, 2, 3]  # erro
Traceback (most recent call last):
etc.
>>> listas[0][0] = 700   # ok
>>> listas
([700, 20, 30], [100, 200, 300])
```

### Dicionários

**10.** O dicionário é outra estrutura de dados fundamental em Python:

```
>>> portos = {'ftp': 21, 'ssh': 22, 'smtp': 25, 'http': 80}
>>> portos
{'ssh': 22, 'ftp': 21, 'http': 80, 'smtp': 25}
>>> len(portos)
4
>>> portos['ftp']
21
>>> portos['sssh']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'sssh'

# próxima operação não devolve nada
>>> portos.get('sssh')
```

*Dicionários pertencem a uma categoria de tipos abstractos de dados designados por **mapas**. Mapas, como o nome indica, mapeiam valores, que designamos por **chaves** (**keys**), noutros valores. Tal como listas, e ao contrário de strings, dicionários são tipos de dados **mutáveis**.*
*Consultar estas e outras operações em:*
*https://docs.python.org/3/library/stdtypes.html#mapping-types-dict*

```
>>> # caso chave não exista, devolve -1
>>> portos.get('sssh', -1)
-1
>>> portos.get('http')
80
>>> portos.keys()
dict_keys(['ssh', 'ftp', 'http', 'smtp'])
>>> portos.values()
dict_values([22, 21, 80, 25]
>>> portos.items()
dict_items([('ssh', 22), ('ftp', 21), ('http', 80), ('smtp', 25)])
>>> 'ftp' in portos, 25 in portos.values()
(True, True)

>>> portos.update({'ftp': 19, 'https': 443})
>>> portos
{'ssh': 22, 'ftp': 19, 'https': 443, 'smtp': 25, 'http': 80}
>>> portos['ftp'] = 21
>>> portos['pop3']
... obtemos KeyError tal como em cima pq a chave 'pop3' não existe ...
>>> portos['pop3'] = 110  # mas podemos aceder p/ acrescentar
>>> portos
{'ssh': 22, 'ftp': 21, 'https': 443, 'smtp': 25, 'http': 80, 'pop3': 110}
>>> del portos['smtp']
>>> portos
{'ssh': 22, 'ftp': 21, 'https': 443, 'http': 80, 'pop3': 110}

>>> idades1 = dict(alberto=20, armando=27, António=19)
>>> idades2 = dict([['alberto', 20], ['armando', 27], ['António', 19]])
>>> idades1, idades2
({'alberto': 20, 'armando': 27, 'António': 19}, {'alberto': 20, 'armando': 27, 'António': 19})
>>> idades1 == idades2, idades1 is idades2
(True, False)
```

*Todos os tipos de dados podem ser utilizados como chaves desde que sejam imutáveis. Dicionários são objectos do tipo* ***dict*** *e podemos criá-los com a função* ***dict*** *ou fazendo:*

```
>>> meu_dic = {chave1: valor1,…, chaveN: valorN}
>>> meu_dic2 = dict(chave1=valor1, …,
                    chaveN=valorN)
```

*O dicionário vazio é dado por {}.*

### Modo Interactivo e *Scripts*

**11.** Nem sempre é conveniente introduzir comandos directamente no REPL. Especialmente agora que vamos introduzir instruções mais complexas. Numa localização apropriada, utilizando um editor da sua preferência, crie o ficheiro `ola.py`.

**NOTA**: Este laboratório não irá cobrir aspectos relacionados com a utilização de editores de texto ou IDEs. Se necessitar de ajuda, peça ao formador para o apoiar nesta tarefa.

**12.** Apenas para testes, acrescente as seguintes instruções:

```
print("Olá, aqui deste script de Python!")
print("Olá", input("Como se chama? "))
```

*A segunda instrução deste programa também poderia ser separada em duas partes:*

```
nome = input("Como se chama? ")
print("Olá,", nome)
```

**13.** Após ter gravado o ficheiro, pode executá-lo na linha de comandos do sistema operativo com:

```
$ python3 ola.py
```

*A o **import**armos um módulo, o código é lido e interpretado imediatamente. Ora, todo o código que está "encostado" à esquerda, fora de definições de funções e classes (lá iremos...), também é executado. Deste modo, é possível executar um script de acções via **import.** (...)*

**14.** Um ficheiro .py é um módulo (ou seja, em cima criámos o módulo `ola`). Deste modo, pode importar o módulo `ola` no interpretador de Python fazendo:

```
$ python3
>>> import  ola
Olá, aqui deste script de Python!
Como se chama? Alberto
```

*(...) Problema? O módulo só é lido uma vez. Se alterarmos o código e voltarmos a fazer **import** sem sair do interpretador, nem por isso o módulo é lido de novo.*
*Já agora, noutras linguagens, como C ou C++, não é possível ter código fora de definições. É um erro sintático.*

**15.** Outra alternativa no mundo Unix, consiste em tornar o programa executável ao nível do sistema operativo (tipicamente via comando `chmod`) e acrescentar uma linha com indicação do interpretador (*shebang line*).

```
#!/usr/bin/python3
print("Olá, aqui deste script de Python!")
print("Olá", input("Como se chama? "))
```

*Na linha `#!` deve colocar o caminho correcto para a sua implementação de Python 3. Pode utilizar os comandos `which` ou `whereis` (o primeiro é melhor) para tentar descobrir.*

**16.** Tendo feito (e tornado o *script* executável), pode invocar o programa com:

```
$ ola.py
```

**Controlo da Execução: Decisão IF-ELIF-ELSE**

**17.** Por vezes interessa-nos tomar decisões mediante determinadas condições. Por exemplo, queremos "esboçar" um programa que:

1. Pergunta ao utilizador pelo nome de uma pasta e
2. Se a pasta tiver conteúdo, lista o seu conteúdo
3. Senão exibe uma mensagem apropriada a dizer que a pasta está vazia

*A função `os.listdir` devolve uma lista com o conteúdo do caminho indicado. Por conteúdo, entenda-se os nomes de todas as sub-pastas e ficheiros contidos do caminho passado como argumento de `os.listdir`.*

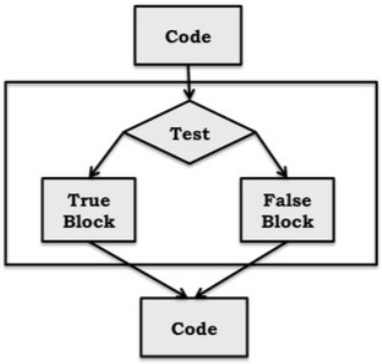
Crie o ficheiro `lista_pasta.py`:

```
import os   # funções para trabalhar com o sistema operativo

nome_pasta = input("Indique o caminho da pasta: ")
conteudo = os.listdir(nome_pasta)
if conteudo:
    print(conteudo)
else:
    print("A pasta", nome_pasta, "está vazia!")
print("Fim")
```

*Os exemplos que vimos até agora foram bastante lineares na sua execução. Cada instrução é executada pela ordem em que é introduzida. A instrução **if** permite "ramificar" o fluxo de execução em duas partes mediante o resultado de uma expressão booleana que, no contexto do **if**, designamos por **condição**.*

*Se a condição for verdadeira, a variável `conteudo` é exibida. Senão (**`else`**) é exibida a mensagem `"A pasta ..."`. Poderíamos ter escrito a condição das seguintes formas: `conteudo != []` ou `len(conteudo) != 0` ou simplesmente `conteudo`. Porquê? Porque sequências, como listas, strings, tuplos, etc., e outros conjuntos, avaliam a verdadeiro se não estiverem vazias. Avaliam a falso se estiverem vazias.*




***Fluxograma a ilustrar o fluxo de execução de um IF***

*Já agora, a instrução **`if`** é uma instrução **composta** pois necessita de mais do que uma (sub)instrução (os dois **print**s) para ficar completamente definida.*
*O formato geral do **`if`** é o seguinte:*

```
if expressão_booleana:
    bloco_código_expressao_verdadeira
else:
    bloco_código_expressão_falsa
```

*A indentação é importante de um ponto de vista da semântica do programa. No exemplo em cima, se o **`print`** não estivesse indentado faria parte do bloco de código do **else** e, portanto, só veríamos a mensagem `Fim` quando o conteúdo fosse vazio. Noutras linguagens, a indentação serve apenas para facilitar a leitura, e não tem qualquer significado em termos da compilação/interpretação. Nessas linguagens blocos de código são delimitados com um determinado símbolo (eg, em linguagens derivadas de C são as chavetas { e }) ou palavra-reservada (eg, **`begin`** e **`end`**). Em Python, a aparência visual do código tem uma correspondência directa com a estrutura semântica do programa.*

**18.** Vamos fazer um programa que recebe pela linha de comandos o nome de um caminho e depois classifica a utilização do dispositivo armazenamento (eg, disco ou pen) em "cheio", "ok", "vazio" consoante essa utilização for superior a 80%, a 40% ou a 0%.

Crie o ficheiro `utilizacao_disco.py` e acrescente as seguintes instruções:

```
import sys
import shutil

if len(sys.argv) != 2:
  print("Utilização: python3", __file__, "caminho")
else:
  total, usado, livre = shutil.disk_usage(sys.argv[1])
  utilizacao = 100 * (usado/total)

  if utilizacao >= 80:
     print("Cheio")
  elif utilizacao >= 40:
     print("OK");
  else:
     print("Vazio")

print("Fim")
```

*O módulo `shutil` fornece um conjunto de utilitários para manipular ficheiros e caminhos.*
*O módulo `sys` possui informação sobre o sistema (versão do Python, do sistema operativo, variáveis de ambiente, parâmetros de invocação da linha de comandos, etc.).*

*Podemos utilizar o **`if`** para tomar mais do que duas decisões. Neste caso utilizamos o formato*
***`if`** `cond1:` … **`elif`** `cond2:` … **`elif`** `condN:` … **`else:`** …*
*Note-se que o bloco **`else`** não é obrigatório. Cada uma das **cláusulas** **`elif`**s é designada de **alternativa condicional**, ao passo que a cláusula **`else`** é designada de **alternativa incondicional**.*

**Controlo da Execução: Decisão MATCH-CASE**

**19.** A instrução MATCH-CASE (muitas vezes referida apenas como MATCH) é uma alternativa ao IF para situações em que pretendemos analisar o valor de uma variável ou expressão. Vamos supor que pretendemos implementar um menu para gestão de uma BD de registos de ocorrências. A nossa aplicação suporta 4 opções. Eis alguns exemplos de utilização desta "aplicação" :

```
1 - Pesquisar ocorrência
2 - Adicionar ocorrência
3 - Actualizar BD de ocorrências
0 - Terminar
>> 1
Introduza os dados para pesquisa...
```

```
1 - Pesquisar ocorrência
2 - Adicionar ocorrência
3 - Actualizar BD de ocorrências
0 - Terminar
>> 3
Vamos agora actualizar a BD...
```

```
1 - Pesquisar ocorrência
2 - Adicionar ocorrência
3 - Actualizar BD de ocorrências
0 - Terminar
>> alberto
ERRO: Opção alberto inválida
```

***NOTA:*** *Neste exemplo vamos apenas fazer "eco" da opção escolhida pelo utilizador e, caso a opção não seja reconhecida, indicamos opção inválida. O utilizador dispõe apenas de uma tentativa.*

**20.** Vamos primeiro implementar este programa utilizando a instrução IF. No ficheiro `match_exemplo1.py` introduza o seguinte código:

```
print("1 - Pesquisar ocorrência")
print("2 - Adicionar ocorrência")
print("3 - Actualizar BD de ocorrências")
print("0 - Terminar")

opcao = input("Opção: ");

if opcao == '1':
    print("Introduza os dados para pesquisa...")
elif opcao == '2':
    print("Introduza os dados do novo registo...")
elif opcao == '3':
    print("Vamos agora actualizar a BD...")
elif opcao == '0':
    print("Programa vai terminar")
else:
    print(f"ERRO: Opção {opcao} inválida")
```

**21.** Comente todo código IF-ELSE e acrescente o seguinte código equivalente:

```
match opcao:
    case '1':
        print("Introduza os dados para pesquisa...")
    case '2':
        print("Introduza os dados do novo registo...")
    case '3':
        print("Vamos agora actualizar a BD...")
    case '0':
        print("Programa vai terminar")
    case _:
        print(f"ERRO: Opção {opcao} inválida")
```

**22.** Agora pretendemos suportar opções com sinónimos. Por exemplo, a pesquisa passa a aceitar `'1'` e `'P'` como opções de selecção. Crie o ficheiro `match_exemplo2.py` com o código seguinte:

```
print("1/P  - Pesquisar ocorrência")
print("2/N  - Adicionar ocorrência")
print("3/A  - Actualizar BD de ocorrências")
print("0/T  - Terminar")

opcao = input("Opção: ");

if opcao == '1' or opcao == 'P':
    print("Introduza os dados para pesquisa...")
elif opcao == '2' or opcao == 'N':
    print("Introduza os dados do novo registo...")
elif opcao == '3' or opcao == 'A':
    print("Vamos agora actualizar a BD...")
elif opcao == '0' or opcao == 'T':
    print("Programa vai terminar")
else:
    print(f"ERRO: Opção {opcao} inválida")
```

**23.** Comente todo código IF-ELIF-ELSE e acrescente o seguinte código equivalente:

```
match opcao:
    case '1' | 'P':
        print("Introduza os dados para pesquisa...")
    case '2' | 'N':
        print("Introduza os dados do novo registo...")
    case '3' | 'A':
        print("Vamos agora actualizar a BD...")
    case '0' | 'T':
        print("Programa vai terminar")
    case _:
        print(f"ERRO: Opção {opcao} inválida")
```

**24.** Para terminar, aqui fica um primeiro exemplo de um programa para gestão de ficheiros. Este programa vai permitir apagar, duplicar ou mover ficheiros através da introdução de comandos. Aqui ficam alguns exemplos:

```
COMANDOS PARA GESTÃO DE FICHEIROS
>> apaga abc.txt
Pretende apagar o ficheiro abc.txt  (S/N)? S
Ficheiro abc.txt removido
```

```
COMANDOS PARA GESTÃO DE FICHEIROS
>> dup abc.txt
Criada cópia de abc.txt em abc.txt.dup
```

```
COMANDOS PARA GESTÃO DE FICHEIROS
>> mov abc.txt ../abc.txt
ERRO: Sintaxe inválida
    mov abc.txt ../abc.txt
```

**25.** Crie o ficheiro `gere_ficheiros1.py` e introduza aí o seguinte código:

```
print("COMANDOS PARA GESTÃO DE FICHEIROS")
cmd_args  = input(">> ")
```

```
match cmd_args.split():
    case ['ajuda']:
        print("\nComandos disponíveis:")
        print("  apaga fich")
        print("  dup fich")
        print("  move fich destino\n")

    case ['apaga' | 'APAGA', fich]:
        resp = input(f"Pretende apagar o ficheiro {fich} (S/N)? ")
        if resp in ('S', 's'):
            os.remove(fich)
            print(f"Ficheiro {fich} removido")

    case ['dup' | 'DUP', fich]:
        dup_fich = f'{fich}.dup'
        shutil.copy(fich, dup_fich)
        print(f"Criada cópia de {fich} em {dup_fich}")

    case ['move' | 'MOVE', fich, destino]:
        resp = input(f"Pretende mover o ficheiro {fich} para {destino} (S/N)? ")
        if resp in ('S', 's'):
            shutil.move(fich, destino)
            print(f"Ficheiro {fich} movido para {destino}")

    case _:
        print(f"ERRO: Sintaxe inválida:\n\t{cmd_args}")
```

*Não se esqueça de importar os módulos necessários para este programa.*
*Que módulos são esses? Atendendo ao código, como os deve importar: com* ***import*** *apenas ou com* ***from ... import ...*** *?*

**26.** A instrução MATCH-CASE suporta muito mais casos do que aqueles que foram aqui demonstrados. Sugere-se a leitura do tutorial do Python e do tutorial presente no documento de especificação desta instrução (PEP 636). Consulte:
https://docs.python.org/3/tutorial/controlflow.html#match-statements
https://peps.python.org/pep-0636/#tutorial

## EXERCÍCIOS DE REVISÃO

**1.** Considere strings, listas, tuplos e dicionários. Qual ou quais destas estruturas de dados são sequências (ie, cada objecto possui um número de ordem ou um índice)?

**2.** Quais os valores obtidos pelas expressões seguintes:

**2.1** `(20, 10, 1, 30, 5).index(1)`
**2.2** `'Alberto.Antunes.Almeida'.split('.')`
**2.3** `'//'.join(['abc', 'def', 'ghi'])`
**2.4** `'+'.join('a_l_b_e_r_t_o'.split('_'))`
**2.5** `'abcdef'.upper()`
**2.6** `'abc#def'.upper()`
**2.7** `'+'.join('a_l_b_e_r_t_o'.split('_')).upper()`
**2.8** `'Fernando'.startswith('f')`
**2.9** `'ABC,DEF,GHI,'.split(',')`

**3.** Considerando que inicialmente `vals = [12, 13, 14, 15, 16]` , responda às seguintes questões.

**3.1** `vals[2]` = ____
**3.2** `vals[2:]` = ____
**3.3** `vals[-len(vals)]` = ____
**3.4** `vals[-len(vals) + 1]` = ____
**3.5** `vals[2:-1]` = ____
**3.6**
```
nums = vals[1:4]
nums[1:3] = [4, 5]
nums = _____    vals = ____
```

**4.** Os seguintes programas ou fragmentos de programas apresentam alguns erros. Corrija-os:

| | |
|---|---|
| `x = [10, 2, 3`<br>`y = x{0}` | |
| `x, y = 3, 5, 6`<br>`print("X: ", X, "Y: ", Y)` | |
| `import pow from math`<br>`input("Valor da base : ") = base`<br>`input("Valor do expoente : ") = exp`<br>`_resultado = pow base, exp`<br>`print("Resultado":, _resultado)` | |

**5.** Considere o seguinte fragmento de código que atribui valores (**`int`** e **`float`**) às variáveis `x` e `y`:

```
x = 100
y = 7.49
```

Para cada instrução (**`str.`**) `format`/f-string na coluna da esquerda, preencha a grelha correspondente na coluna da direita. Uma quadrícula vazia entre caracteres indica a presença de um espaço. Assuma que todas as instruções estão envolvidas num **`print`**.

| | |
|---|---|
| `f"{y} "`<br>`"{0:f} {0:.1f}".format(y, y+1)` | |
| `"{:5}\n".format(x)`<br>`f"{x*2:^5}{x:<5}"` | |
| `f"""->({x:4.2f}+{y:.2f})/3=`<br>`{(x+y)/3:.2f}"""` | |
| `"X:{0:<5}\nY:{1:<5.2f}\n".`<br>`format(x, y)` | |

**6.** Considere o seguinte fragmento de código que declara e atribui valores a duas variáveis do tipo ***int*** e ***float***:

```
x = 10.18
y = 41
```

Para cada instrução (`str.`) `format`/ f-string na coluna da esquerda, preencha a grelha correspondente na coluna da direita. Uma quadrícula vazia entre caracteres indica a presença de um espaço. Assuma que todas as instruções estão envolvidas num ***print***.

| | |
|---|---|
| `f"{y}{x+10}"` | |

| | |
|---|---|
| `f"{y:5}\n"`<br>`"{1:>6}{0:<6}".format(x, x+2)` | |

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

| | |
|---|---|
| `p = 2.3`<br>`print("{:f}".format(p))`<br>`print(p*2,'\n')`<br>`print("{0}{1}".format(p*2, "\n"[0]))` | |
| `v1 = [0]*4`<br>`i = 0`<br>`v1[i] = 7; i+=1; v1[i] = 14; v1[len(v1) - 1] = 15`<br>`print(v1[i]*v1[i+1] + v1[1])` | |
| `x=3,5/2`<br>`print(x)` | |
| `codigo = {'A': 19, 'B': 20}`<br>`print(list(codigo))` | |
| `codigo = {'A': 19, 'B': 20}`<br>`print(codigo['B'], codigo.get('C'), codigo.get('C', 21))` | |
| `codigo = {'A': 19, 'B': 20}`<br>`d = dict(a=list(codigo.values())[0],`<br>`b=list(codigo.values())[-1])`<br>`print(d)` | |
| `processos = {'ls': 192, 'grep': 321, 'init': 1}`<br>`print('ls' in processos, 321 in processos)`<br>`print((192 in processos)*2)`<br>`processos.update(ls=292, mkfs=19)`<br>`print(list(processos.items()))` | |

| | |
|---|---|
| `txt = ''`<br>`nums = [10, 11]`<br>`if nums:`<br>`  print("Um")`<br>`  if not txt:`<br>`  print("Dois")`<br>`  txt = 'abc'`<br>`  nums = []`<br>`  else:`<br>`  print("Três")`<br>`  txt = 'xey'`<br>`  nums[-1:] = [12]`<br>`txt = txt.replace('a', '').replace('e', '')`<br>`print("Quatro" if len(nums) else "Cinco")`<br>`print(txt if len(nums) == 0 else nums)` | |

**8.** Considere o tuplo `t = ([1, 2], "abc")`. Indique quais das seguinte instruções são inválidas e qual o respectivo erro:

**8.1** `t[0] = [3]`

**8.2** `t[1] = "3"`

**8.3** `t[0][0] = 3`

**8.4** `t[1][0] = "3"`

**8.5** `x, y = t`

**9.** Converta as seguintes decisões *if-else*/*match-case* em *match-case*/*if-else*:

| | |
|---|---|
| `i = int(input("Valor de i: "))`<br>`if i == 10:`<br>`  print("Preparar operação")`<br>`elif i == 5:`<br>`  print("Começar operação")`<br>`elif i == 0:`<br>`  print("Missão crítica")`<br>`else:`<br>`  print("Operação abortada")` | |
| `nome = input("Nome? ")`<br>`match nome:`<br>`  case 'Alberto' \| 'Armando':`<br>`  print("Bom rapaz")`<br>`  case 'Arnaldo' \| 'Augusto':`<br>`  print("Gente estranha...")`<br>`  case _:`<br>`  print("Desconheço...")` | |

```
nums = ... # deve ser um tuplo com três valores
           # mas pode haver um erro...
if not isinstance(nums, tuple) or len(nums) != 3:
    print("Conjunto inválido")
elif nums[0] == 10:
    print(nums[1] + nums[2])
elif nums[0] == 1:
    print(nums[1] * nums[2])
else:
    print("Conjunto inválido")
```

**EXERCÍCIOS DE PROGRAMAÇÃO**

**10.** Faça um programa para calcular o preço de venda final de um produto. Para tal solicita, através da linha de comandos (*shell*), o preço do produto, o valor da taxa de IVA a aplicar e (opcionalmente) o valor de um desconto a aplicar ao valor final do produto. O programa deverá dar instruções ao utilizador de como deve ser invocado. O valor do IVA e do desconto deve ser dado em percentagem.

**11.** Faça um programa para indicar se um determinado ano introduzido pelo utilizador é bissexto ou não. Um ano é bissexto se for múltiplo de 4. No caso dos anos centenários, apenas são bissextos os anos múltiplos de 400. Não aceite anos negativos. Pode utilizar o operador % (resto ou módulo) para determinar se um número é múltiplo de outro (ou, dito de outra forma, se um número é divisível por outro).

**12.** Pretende-se calcular a idade em anos em função do dia, mês e ano de nascimento e dia, mês e ano atual. Tenha em atenção o seguinte: em condições normais a idade é a diferença entre o ano atual e ano de nascimento, porém, se o mês actual for inferior ao mês de nascimento ou o mês atual igual ao mês de nascimento e o dia atual inferior ao dia de nascimento a idade é o ano atual menos o ano de nascimento menos um.

**13.** Acrescente ao programa para gestão de ficheiros mais duas opções: REN para renomear ficheiros e APAGADIR para remover uma directoria.