CENTRO DE FORMAÇÃO
DE ALCÂNTARA

INTRODUÇÃO À PROGRAMAÇÃO
UFCD(s) 5117, 5091

GUIA DE LABORATÓRIO 3.1
CONTROLO DA EXECUÇÃO (ßeta)

OBJECTIVOS

- Aprofundar as noções sobre ciclos e decisões em Python e introduzir expressões condicionais
- Neste contexto, introduzir os mecanismos `BREAK`, `CONTINUE` e `ELSE` (ciclos)
- Alguns tipos de dados e funções muito utilizados em ciclos ou para evitar ciclos

INSTRUÇÕES

Ciclos e Decisões: Revisões e Aprofundamento

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

> *Os computadores seleccionam, guardam, acedem, percorrem e, de um modo geral, manipulam os dados que armazenam. Esta manipulação envolve tomadas de decisões e tarefas repetitivas que possuem um início e um fim. Um programa de software, escrito numa qualquer linguagem de programação, especifica essas actividades de manipulação e a ordem pela qual acontecem. Nesse sentido, um programa procura controlar a execução dessas operações. Controlar significa determinar que instruções devem ser executadas em cada instante, e qual a ordem pela qual devem ser executadas.*

Tipo	Membros / Exemplos
Decisão / Selecção	`if-elif-else` *instrução condicional (ou operador ternário)*
Ciclo	`while` `for`
Transferência de Controlo	`return` `break` `continue` `try-except-finally` `raise`
Invocação de funções/métodos	`len("Ola")` `math.pow(2, 3)` `"{:2}".format(x)` *etc.*
Iteradores e Geradores	`iter` `next` `yield` `yield from` gerador`.send/throw/close`
List/Dict/Set Comprehensions (*)	`[num for num in sequencia if num > 10]` `{k: v for k, v in dicionario if k[0] == 'A'}`
Generator Expressions (*)	`(x for x in sequencia if x % 2 == 0)`
Gestores de contexto	`with` funções/métodos em `contextlib`

Tabela não-exaustiva com mecanismos de controlo da execução em Python
(*) - Mecanismos declarativos com fluxo de execução implícito

2. Vamos solicitar ao utilizador um valor que é suposto ser preço sem IVA de um produto. Se o tipo de

produto for "Alimentação" o IVA a aplicar deverá ser 6%, se for "Higiene" deverá ser 13%, caso contrário deverá ser 23%. Depois calculamos e exibimos o montante de IVA e o preço final . Crie um script com o nome `iva_por_categoria.py` e acrescente o seguinte código:

```
from decimal import Decimal as dec

preco = dec(input("Preço do produto? "))
tipo_prod = input("Tipo do produto? ")

if tipo_prod.upper() == 'A':
   IVA = dec('0.06')
elif tipo_prod.upper() == 'H':
   IVA = dec('0.13')
else:
   IVA = dec('0.23')

print(
    "Montante de IVA: {0:,.2f} €  (IVA: {1:.2%})\n"
    "Preço Final: {2:,.2f} €"
    .format(preco*IVA, IVA, preco*(1 + IVA))
)
```

*Como vimos no laboratório anterior, a instrução **if** é uma instrução condicional designada por decisão. Em geral, se a condição de um **if** for verdadeira, ie, se a expressão lógica que sucede a palavra-reservada "**if**" produzir o valor booleano **True**, então a instrução ou bloco de instruções que sucede o **if** é executado. Se a condição for falsa, então duas coisas podem acontecer: se a instrução **if** possuir uma parte (designada por "cláusula") que se inicie com a palavra reservada **else**, então o bloco de instruções desse **else** é executado em vez do bloco de instruções do **if**. No fundo estamos a dizer algo como "se isto for verdade então faz aquilo, senão faz outra coisa qualquer". Se não utilizarmos a cláusula **else** (é opcional) então, o programa prossegue na instrução que sucede a toda instrução composta **if**.*

*Suponhamos que alguém projecta as seguintes alternativas para um dia de férias: se estiver sol vai à praia, se estiver encoberto vai para o campo e se estiver a chover vai ao mercado. Temos aqui um conjunto de três alternativas que dependem de três condições. Em Python, uma alternativa condicional indica-se com **elif**. Não existe limite para o número de alternativas condicionais. Note-se que a cláusula **else** é uma alternativa incondicional dado que é sempre executada caso a condição do **if** seja falsa.*
*Resumindo, 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 **elifs** é designada por alternativa condicional, ao passo que a cláusula **else** é designada por alternativa incondicional.*

3. Para este tipo de situação podemos também utilizar um dicionário como um tipo de estrutura de selecção baseada no valor do tipo de produto (que podemos encarar como uma chave deste dicionário). No exemplo anterior, comente todo o ***if-elif-else***.

*Determinadas linguagens possuem um mecanismo de selecção baseado no valor de uma expressão. Em linguagens derivadas de C, esse mecanismo é o **switch-case**. Exemplo em C#:*

```
string tipo;
// tipo é inicializado de alguma forma
decimal IVA;
switch(tipo) {
  case "A"
    IVA = 0.06;
    break;
  case "H":
    IVA = 0.13;
    break;
  default:
    IVA = 0.23;
    break;
}
```

4. Agora acrescente o seguinte código após a instrução `tipo = input(....` em substituição do ***if***:

```
IVA = {
    'A': dec('0.06'),
    'H': dec('0.13'),
}.get(tipo_prod.upper(), dec('0.23'))
```

5. Passemos para outro mecanismo de selecção. Introduza agora o seguinte código no REPL:

```
>>> x = 19
```

6. Agora pretendemos inicializar a variável `y` em função do valor de `x`, de acordo com a seguinte lógica: se `x` for superior a `10`, `y` deverá ter `'sup'`, caso contrário deverá ter o valor `'inf'`:

```
>>> if x > 10:
...     y = 'sup'
... else:
...     y = 'inf'
```

7. Alternativamente, pode utilizar uma expressão condicional:

```
>>> y = 'sup' if x > 10 else 'inf'
```

> *Uma expressão condicional possui a seguinte síntaxe:*
>
> *EXPR_CONSEQUÊNCIA* ***if*** *CONDIÇÃO* ***else*** *EXPR_ALTERNATIVA*
>
> *Uma expressão condicional é um mecanismo declarativo de avaliação de uma de duas expressões mediante o resultado de uma condição. Em determinadas linguagens este "mecanismo" é designado por "operador ternário".*

8. Vamos agora inicializar o valor da variável `z`, que deverá ser `'sup'` se `x > 15`, `'interm'` se `10 <= x <= 15` e `'inf'` caso contrário.

```
>>> z = 'sup' if x > 15 else 'interm' if x >= 10 else 'inf'
```

BREAK, CONTINUE e ELSE com ciclos

9. Uma das função que vamos abordar com maior detalhe é a função **`range`** . Introduza:

```
>>> range(10)
range(0, 10)
>>> r = range(10)
>>> r[0], r[-1]
(0, 9)
>>> list(range(10)) # list(r)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

> *A função* ***`range`*** *é, na verdade, uma classe/tipo de dados (mas isso fica para analisar depois). Chamada apenas com um argumento gera uma sequência imutável de valores. Entre outras coisas,* ***`range`*** *é útil para gerar gamas de valores para serem iterados através do ciclo* ***`for`***.

10. Podemos utilizar o ciclo for com um range para percorrer sequências de valores:

> *E é assim que temos um ciclo for mais semelhante ao ciclo* ***`for`*** *das linguagens derivadas de C. Nessas linguagens, ciclo* ***`for`*** *é um ciclo com quatro partes: inicialização, condição, actualização e bloco de instruções. O seu formato geral é:*
>
> ***`for`*** *(INICIALIZAÇÃO; CONDIÇÃO; ACTUALIZAÇÃO)*
> *BLOCO_DE_INSTRUÇÕES*

```
>>> for i in range(10):
...  print(i)
...
0
1
2
3
4
5
6
7
8
9
```

Em linguagens tipo C e C++, a inicialização do ***for*** *é executada apenas uma vez antes do ciclo se iniciar. Depois, à semelhança do que sucede com o* ***while****, o bloco de instruções é executado enquanto que a condição for verdadeira. Após o bloco de instruções ter sido executado, a actualização é também executada e a condição volta a ser testada. Se for falsa, o fluxo de execução continua a seguir ao* ***for****, senão o bloco de instruções é repetido. Nessas linguagens, exibimos os números de 0 a 9 da seguinte forma (exemplo em C++):*

```
for (int i = 0; i < 10; ++i) {
  cout << i << endl;
}
```

11. Crie o ficheiro `primo_for.py`.

12. Vamos voltar ao problema do determinar se um número é primo, desta vez utilizando ciclo ***for*** e a instrução ***break***. Queremos que um programa indique todos os primos até um determinado número introduzido pelo utilizador. Introduza o seguinte código:

```
num = int(input("número> "))

for n in range(2, num+1):
    sem_factores = True
    for x in range(2, n):
        if n % x == 0:
            print(n, "é igual a", x, '*', n//x)
            sem_factores = False
            break

    if sem_factores:
        # ciclo terminou sem interrupção
        print(n, "é um número primo")
```

No âmbito dos ciclos, as palavras reservadas ***break*** *e* ***continue*** *permitem transferências de controlo directas para determinados "locais". Assim, a instrução* ***break*** *termina o ciclo que envolve directamente esta instrução. A próxima instrução a ser executada após u m* ***break*** *é a instrução que surge após este ciclo interrompido.*
O ***continue*** *passa o controlo para a próxima iteração de um ciclo, saltando por cima do código que resta executar nesta iteração.*

13. Em Python os ciclos ***for*** suportam uma cláusula ***else*** que pode simplificar este problema. Comente todo o ciclo de ***for*** exterior e acrescente as seguintes instruções:

```
for n in range(2, num+1):
    for x in range(2, n):
        if n % x == 0:
            print(n, "é igual a", x, '*', n//x)
            break
    else:
        # ciclo terminou sem interrupção
        print(n, "é um número primo")
```

No âmbito de um ciclo (***for*** *ou* ***while****), o bloco de instruções de um* ***else*** *é executado apenas se não ocorrer um* ***break****, isto é, se o ciclo terminar por via da condição do* ***while****, ou se todos os elementos de uma colecção forem "vistos" pelo* ***for****. Neste contexto, o termo "else" pode ser algo confuso. Na verdade,* `nobreak` *seria um nome mais apropriado para esta cláusula.*

14. Podemos utilizar ciclos ***for*** para percorrer a maioria das estruturas de dados Python. Experimente:

```
>>> nomes = ['Alberto', 'Armando', 'Arnaldo']
>>> for nome in nomes:
...   print(nome, end=' ')
... else:
...   print()  # assegura que o >>> não fica na mesma linha que o último nome
...
Alberto Armando Arnaldo

>>> pessoa = {'nome': 'Alberto', 'apelido': 'Antunes', 'idade': 27}
>>> for campo in pessoa:
...  print(campo)
...
idade
nome
apelido

>>> for campo in pessoa:   # o mesmo que "for campo in pessoa.keys():"
...  if campo == 'idade':
...    print('idade ->', pessoa[campo])
...
idade -> 27
>>> for campo in pessoa.keys():
...  print(campo, '=', pessoa[campo])
...
idade = 27
nome = Alberto
apelido = Antunes
>>> for campo, valor in pessoa.items():
...  print(campo, '=', valor)
...
idade = 27
nome = Alberto
apelido = Antunes
```

Recordando, o método `keys` devolve a sequência de chaves de um dicionário, o método `values` devolve uma sequência com os valores e o método `items` devolve uma sequência de pares chave-valor. Estas sequências podem ser iteradas através de um ***for****.*

15. Como foi possível observar, podemos atribuir mais do que uma variável num ciclo ***for***. Vejamos um exemplo com tuplos:

```
>>> coords_p1 = (20, 10, 30)
>>> coords_p2 = (-1, 5.2, 10)
>>> coords_p3 = (-1.7, -2, -3)
```

Podemos olhar para um tuplo como uma lista imutável de valores. Logo, o tuplo também é uma sequência. Por ser imutável, não podemos modificar os elementos individuais de um tuplo, nem acrescentar ou remover elementos. O tuplo é ideal para devolver múltiplos valores de uma função.
O Python permite "desempacotar" (unpacking) valores de sequências para variáveis, sendo que os tuplos são habitualmente utilizados para isto. É o que fazemos no exemplo com `coords_p1` e as variáveis `x`, `y`, `z`.

```
>>> x, y, z = coords_p1
>>> print(x, y, z)
20 10 30
>>> pontos = [coords_p1, coords_p2, coords_p3]
>>> for x, y, z in pontos:
...  print("  {:4} {:4} {:4}".format(x, y, z))
...
    20   10   30
    -1  5.2   10
  -1.7   -2   -3
```

Funções RANGE, ENUMERATE, REVERSED, SORTED, ZIP e SET

16. Suponhamos que pretendemos exibir os números pares de 10 a 100. A função **range** é ideal para isto:

```
>>> for n in range(10, 101, 2):
...  print(n)
...
10
12
…
100
```

As "funções" utilitárias mencionadas nesta parte do laboratório são muito importantes para percorrer sequências de valores, dicionários, gamas de valores, etc. Daí serem referidas aqui, neste laboratório que se dedica ao estudo dos mecanismos para controlo da execução em Python.
*Nem todas são realmente funções. Algumas são tipos de dados do módulo **builtin**. Mais à frente, após mencionarmos iteradores, classes e objectos, ficaremos a perceber melhor a diferença. Para já, atendendo ao facto de utilizarmos estes tipos de dados com o ciclo **for** como se estivéssemos a invocar uma função, continuaremos a utilizar o termo "função".*

17. Também podemos iterar ao contrário:

```
>>> for n in range(100, 9, -2):
...  print(n)
...
100
98
…
10
```

*Como vimos, a função **range** gera uma sequências de valores. Quando invocada com três argumentos, o primeiro indica um limite inferior, o segundo indica um limite superior fora da gama de valores, e o terceiro argumento especifica o espaçamento (ou passo, ou distância, ou intervalo) entre os números. Neste caso queremos uma sequência de 10 a 100, de dois em dois valores.*

18. Por vezes, quando pretendemos utilizar um ciclo ***for***, temos necessidade de aceder ao índice (ie, à posição) de um elemento de uma lista ou de um tuplo. É aqui que entra ***enumerate***. Por exemplo:

```
>>> nomes = ['Alberto', 'Armando', 'Arnaldo', 'António']
>>> for i, nome in enumerate(nomes):
...  print(i, '->', nome)
...
0 -> Alberto
1 -> Armando
2 -> Arnaldo
3 -> António
```

*A função **enumerate** retorna pares (indíce, valor) para uma sequência de valores. Opcionalmente, podemos indicar qual o valor inicial dos indíces.*
https://docs.python.org/3/library/functions.html#enumerate

```
>>> for i, nome in enumerate(nomes, 1):
...  print(i, '->', nome)
...
1 -> Alberto
2 -> Armando
3 -> Arnaldo
4 -> António
```

19. Dada uma sequência (*) de elementos , a função ***sorted*** devolve uma nova lista com todos os elementos ordenados.

```
>>> nums1 = (1, 4, 2, 5, 3)
>>> nums2 = sorted(nums1)
>>> nums2
[1, 2, 3, 4, 5]
>>> nums1
(1, 4, 2, 5, 3)
>>> sorted("XPQAZ")
['A', 'P', 'Q', 'X', 'Z']
>>> ''.join(sorted("XPQAZ"))
'APQXZ'
>>> sorted(nums1, reverse=True)
[5, 4, 3, 2, 1]
```

*Dada uma sequência (*), a função **sorted** devolve uma nova lista ordenada de valores. Por omissão a ordenação é ascendente, mas podemos indicar outra ordem através do parâmetro opcional `reversed`. Existe outro parâmetro opcional, o parâmetro `key`, que recebe uma função que aplica a todos os valores da sequência a ordenar. A ordenação é feita com base nos valores que são devolvidos por estas funções (que designamos por 'key functions').*
https://docs.python.org/3/library/functions.html#sorted

```
>>> palavras = ['Dia', 'Coimbra', 'domingo', 'Mesa']
>>> sorted(palavras)
['Coimbra', 'Dia', 'Mesa', 'domingo']
>>> sorted(palavras, key=str.lower)
['Coimbra', 'Dia', 'domingo', 'Mesa']
>>> from operator import itemgetter
>>> sorted(palavras, key=itemgetter(1))
['Mesa', 'Dia', 'Coimbra', 'domingo']
```

(*) Na verdade, a função ***sorted*** aceita um iterável, conceito que estudaremos à frente. Todas as sequências são iteráveis.

20. Se pretendermos exibir os valores por ordem inversa utilizamos a função ***reversed***. Esta função devolve um objecto iterador (veremos mais à frente) que o ciclo ***for*** sabe "utilizar" para aceder a cada elemento da sequência. Vejamos:

```
>>> for nome in reversed(nomes):
...  print(nome)
...
António
Arnaldo
Armando
Alberto
```

```
>>> for i, nome in enumerate(reversed(nomes)):
...  print(i, '->', nome)
...
0 -> António
1 -> Arnaldo
2 -> Armando
3 -> Alberto
```

> *Em Python, um* ***set*** *(conjunto) é uma colecção de elementos distintos sem ordem. Não existem elementos duplicados num* ***set****. Mesmo que tentemos adicionar dois elementos iguais, um* ***set*** *apenas aceita um dos elementos. Habitualmente, os sets são utilizados para testes de pertença (para verificar se um elemento pertence a um conjunto), para remover duplicados de um sequência e para calcular intersecções, uniões, diferença entre conjuntos.*
> *Criamos um set com:*
> *.* ***set****(sequência) (*)*
> *. {elemento1, elemento2, …}*
>
> *(*) Na verdade, criamos um set a partir de um iterável, noção que vamos estudar mais à frente.*
> *https://docs.python.org/3/library/functions.html#func-set*

21. Os ***set***s (conjuntos) são úteis para filtrar elementos duplicados:

```
>>> nums = [19, 2, 90, 91, 19, 90, 14]
>>> nomes = ('ana', 'tiago', 'ana', 'bruno', 'tiago')
>>> set(nums)
{2, 19, 90, 91, 14}
>>> set(nomes)
{'ana', 'tiago', 'bruno'}
>>> set(nomes[0])
{'a', 'n'}
>>> set(str(998272))
{'2', '8', '9', '7'}
>>> 'a' in set(nomes[-1])
True
>>> 'b' in set(nomes[-1])
False
```

22. Utilizamos os operadores `&` e `|` para representar a intersecção e a união entre dois conjuntos:

```
>>> txt1 = {'x', 'y', 'z'}
>>> txt2 = 'xyzw'
>>> txt3 = 'aybx'

>>> # união
>>> txt1 | set(txt2)
{'z', 'y', 'x', 'w'}
>>> txt1.union(txt2)
{'z', 'y', 'x', 'w'}

>>> # intersecção
>>> set(txt3) & txt1
{'y', 'x'}
>>> set(txt3).intersection(txt1)
```

23. Finalmente, a função ***zip*** permite processar duas ou mais sequências, elemento a elemento, segundo a ordem pela qual estão na sequência. Vejamos um exemplo:

```
>>> letras = 'abc'
>>> codigos = [97, 98, 99]
>>> for letra, cod in zip(letras, codigos):
...  print(letra, '->', cod)
...
a -> 97
b -> 98
c -> 99
```

EXERCÍCIOS DE REVISÃO

1. Indique os mecanismos de controlo de execução introduzidos neste laboratório.

2. Para que serve a instrução ***continue*** ? Exemplifique.

3. Com que valores ficam as variáveis nas seguintes atribuições:

3.1 `b = 72 if 0 != 1 else 99`

3.2 `b = 44 if 0 == 1 else 33 if 'a' < 'A' else 22`

3.3 `nome = '/'.join("ALBERTO"[::2].lower())`

4. Quantos dígitos tem o maior número de 128 bits? Responda a esta questão com uma (e uma só) instrução de Python. Indique também que número é esse, formatando-o com o separador de milhares/milhões/....

5. Indique uma maneira de determinar se uma string contém pelo menos um dos caracteres de outra string.

6. Considerando que inicialmente `itens = [('a', 13), ('d', 11), ('b', 15), ('c', 10)]` , responda às seguintes questões.

6.1 `itens[::2]` `= ____`

6.2 `list(reversed(itens))[-4:-1]` `= ____`

6.3 `sorted(itens, reverse=True)` `= ____`

6.4 `sorted(itens, key=operator.itemgetter(1))` `= ____`

6.5
```
outros_itens = []
for i1, i2 in zip(itens, reversed(itens)):
    outros_itens.append((i1[0], i2[1]))

outros_itens = _______
```

EXERCÍCIOS DE PROGRAMAÇÃO

7. Faça um programa para calcular uma potência. O programa solicita ao utilizador a base e o expoente, após o que exibe o resultado. Desenvolva o algoritmo; não utilize o operador `**`, nem `math.pow`, nem quaisquer métodos equivalentes "já feitos". Assuma que o expoente é sempre um inteiro (negativo ou positivo). Após cada cálculo o programa pergunta ao utilizador se pretende repetir. Nesta parte o programa apenas deve aceitar `"s"`, `"sim"`, `"n"` ou `"não"` (maiúsculas ou minúsculas).

8. Pretende gerar uma lista com 10.000 tuplos com o seguinte conteúdo:

```
[(0, 0, 0, 0) , (0, 0, 0, 1) , ... , (9, 9, 9, 8) , (9, 9, 9, 9)]
```

Faça um programa para obter esta lista utilizando dois algoritmos diferentes.

9. O objectivo deste exercício consiste em desenvolver um programa para obter endereços MAC (endereços nível 2) a partir de uma lista de endereços IPv4 passada a partir da linha de comandos. O programa deverá ser invocado da seguinte forma:

```
$ python3 l2addr.py [-r] ipv4_addr1 [ipv4_addr2...]
```

O parâmetro opcional `-r` "refresca" a cache de IPs através do comando `ping` (ver descrição em baixo).

Em Linux e macOS, o comando `arp` permite obter a informação pretendida:

```
$ arp -n 192.168.56.1
```

Saída do comando `arp` em Linux:

```
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.56.1             ether   0a:00:27:00:00:00   C                     eth0
```

Saída em macOS:

```
? (192.168.56.101) at a:0:27:0:0:0 on vboxnet0 ifscope [ethernet]
```

De modo a garantir que o endereço IPv4 se encontra na cache ARP, é conveniente "pingar" primeiro o destino (mas limitando o número de pacotes ICMP a enviar).

Para invocar os comandos `ping` e `arp` utilize a função `popen` (*Process Open*) do módulo `os`. Esta função invoca um programa lançando um processo e criando uma *pipe* entre a saída padrão do processo invocado e o processo invocador (que é o *script* em Python que estamos a desenvolver). Acedemos à pipe como a um ficheiro de texto, uma vez que `os.popen` devolve um *file object*, isto é, um objecto que representa um ficheiro de texto.

Utilize o módulo `re` (*regular expressions*) para extrair os endereços de nível 2 através de uma expressão regular apropriada.

10. Faça um *script* que exibe um ficheiro de texto de forma paginada, exibindo em cada linha o número da mesma, seguido do conteúdo da dita. A exibição paginada deverá ser levada a cabo por uma função que recebe o caminho do ficheiro e quantas linhas por página. Após cada página, o programa deve aguardar que o utilizador pressione ENTER antes de avançar para a página seguinte.