INTRODUÇÃO À PROGRAMAÇÃO
UFCD(s) 5117, 5091

GUIA DE LABORATÓRIO 2.2

CONTROLO DA EXECUÇÃO, *SCRIPTING* E BIBLIOTECA PADRÃO (ßeta)

OBJECTIVOS

- Utilizar o interpretador em modo interactivo e não-interactivo
- Preparar os laboratório seguintes, introduzir brevemente alguns tópicos mais avançados
- Introdução à biblioteca padrão do Python

INSTRUÇÕES

Modo Interactivo e *Scripts*

1. 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.

2. Apenas para testes, acrescente as seguintes instruções no início do ficheiro:

```
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)
```

3. 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**. 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.

4. 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
```

5. 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.

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

```
$ ola.py
```

Controlo da Execução: Breve Introdução

7. 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

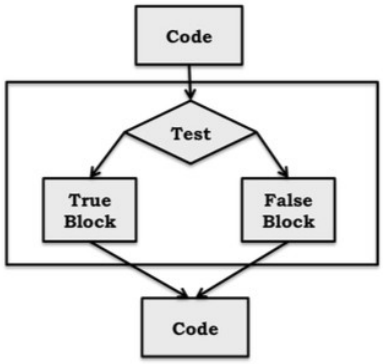
Crie o ficheiro `lista_pasta.py`:

```
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")
```

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`.

*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.
*Já agora, a instrução **`if`** é uma instrução composta pois necessita de mais do que uma (sub)instrução (os dois prints) para ficar completamente definida.*



Fluxograma a ilustrar o fluxo de execução de um IF

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.*

8. 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.).*

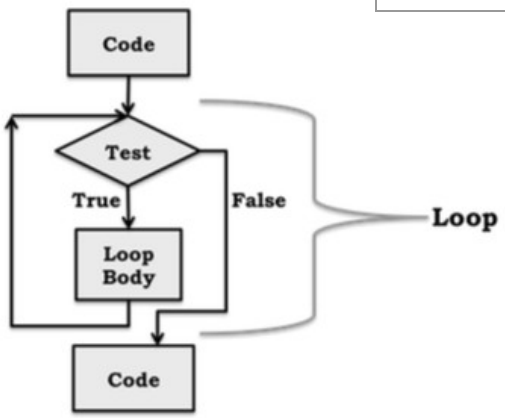
Podemos utilizar o ***if*** *para tomar mais do que duas decisões. Neste caso utilizamos o formato*
if `cond1:` … ***elif*** `cond2:` … ***elif*** `condN:` … ***else:*** …
Note-se que o bloco ***else*** *não é obrigatório. Cada uma das cláusulas* ***elif****s é designada de alternativa condicional, ao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

9. Agora vamos fazer um programa que solicita um número ao utilizador e indica o resultado da soma de todos os números até esse número (existe uma fórmula que permite obter directamente o resultado da soma, mas que, por agora, vamos ignorar). Crie o ficheiro `soma_ate.py` e acrescente:

```
num = int(input("Indique um numero: "))
soma = 0
i = 1
while i <= num:
  soma +=  i
  i += 1
print("Resultado:", soma)
```

***NOTA**: Nos laboratórios seguintes veremos outros modos (alguns melhores) de resolver este tipo de problemas.*

*O ciclo while é semelhante a um **if**, com a diferença que o bloco de instruções do ciclo é executado enquanto a condição for verdadeira.*
*Em Python, o ciclo **while** (e não só) pode também ter uma cláusula **else**, aspecto que iremos verficar no próximo laboratório. De resto este tipo de problemas resolve-se melhor com o ciclo **for**, mas este também é um tópico que vamos deixar para o próximo laboratório.*



1a forma de repetição (loop): O ciclo while

10. Vamos agora fazer um exemplo mais elaborado. Queremos um programa que indique a dimensão em bytes do conteúdo de uma determinada directoria. O nosso programa vai indicar a dimensão apenas do conteúdo directamente "por baixo" do caminho fornecido, não entrando recursivamente em sub-directorias. Crie o ficheiro `utilizacao_dir.py` e acrescente:

```
import sys
import os.path

if len(sys.argv) != 2:
  print("Utilização: python3", __file__, "caminho")
else:
  caminho = sys.argv[1]
  conteudo = os.listdir(caminho)

  if not conteudo:
      # terminamos logo se pasta esta vazia
      print("A pasta", conteudo, "está vazia!")
      sys.exit(0)

  i = dim_bytes = 0
  while i < len(conteudo):
      # cada ficheiro devolvido na lista apenas tem o nome
```

Novidades:
. `sys.exit`: função que termina imediatamente o programa; passamos 0 para indicar que o programa terminou normalmente
. `os.path`: módulo com utilitários relacionados com caminhos
. `os.path.getsize`: função que devolve a dimensão de um ficheiro.
`"...{:,}".format(...)` : a "," no código de formatação permite visualizar o separador de milhares.

```
        # (é assim que os.listdir "funciona"), logo necessitamos
        # de concatenar o caminho se não getsize não encontra
        # os ficheiros
        dim_bytes += os.path.getsize(caminho + "/" + conteudo[i])
        i += 1
    print("A utilização de {} é {:,} bytes".format(caminho, dim_bytes))
```

Funções: Breve Introdução

11. Temos utilizado várias funções e métodos (***len***, ***print***, ***list***.*append*, ***dict***.*update*, ...), mas agora queremos saber como definir uma nova função. Suponha que pretende uma função para calcular um valor com IVA. Crie um ficheiro iva.py e acrescente as seguintes definições:

```
def com_iva(valor, taxa_iva):
  return (valor * (100 + taxa_iva)) / 100

valor = float(input("Indique um montante: "))
print("{:.2F}".format(com_iva(valor, 23)))
```

*Uma **função** é um bloco de código com um nome. Este blogo de código pode depois ser reutilizado várias vezes através da invocação desse nome. Para definir uma função utilizamos a palavra-reservada def:*

```
def fun(param1, ... , paramN):
    inst1
    ...
    instN
```

*`fun` é o **nome** ou **identificador** da função. `param1`, ..., `paramN`, entre parêntesis, constitui a **lista de parâmetros** da função. **Parâmetros** são variáveis **locais** à função e servem para receber informação que vem de fora da função. Uma função pode não definir parâmetros. Para devolver valores para fora da função, a função utiliza a palavra-reservada **`return`**. A instrução **`return`** termina imediatamente a função e devolve o resultado das expressões à direita. Uma função é invocada fazendo*

```
fun(arg1, ..., argN)
```

***Argumentos** são os valores dos parâmetros, e devem ser indicados entre parênteses. No exemplo anterior, os argumentos de `com_iva` são a o valor da variável `valor` e o literal `23`. Podemos olhar para `input` e `float` como sendo também funções (`float` é especial, mas vamos pode ser encarado como uma função). No exemplo anterior, são **invocadas** recebendo como argumentos o texto `"Indique um montante"` e o texto efectivamente introduzido pelo utilizador, respectivamente.*
*Um **método** é uma função que é invocada "num contexto" especial, o contexto de um objecto. Por seu turno, um **objecto** é um "valor" de um determinado tipo de dados e que agrega atributos e operações. Esses atributos e essas operações são definidas na classe do objecto, sendo que a **classe** é também o tipo de dados do objecto. Cada uma dessas operações é, precisamente, um método. Ou seja, um método é uma função definida dentro de uma classe e que opera sobre um objecto, podendo também operar sobre outros argumentos. Por esta altura, é natural que estas noções pareçam demasiado abstractas e até confusas. O tempo e a prática vão ajudar a clarificá-las.*
De um ponto de vista prático, podemos olhar para um método como uma função que se invoca da seguinte forma:

```
arg1.met(arg2, ..., argN)
```

`arg1` é o tal objecto, que pertence a uma determinada classe (podemos saber qual através de `type(arg1)`), e na qual se encontra definido o método `met`. Quando dermos classes e objectos, tudo isto vai fazer mais sentido. Até lá, necessitamos apenas de saber que determinadas funções são invocadas (ie, utilizadas) indicando em primeiro lugar o nome da função e depois a lista de (...)

(...) argumentos. Existe uma outra categoria de funções, os métodos, que devem ser invocados escrevendo em primeiro lugar um objecto (o argumento principal da função), seguido do operador de acesso . (ponto), seguido do nome do método, seguido da lista de argumentos com os restantes argumentos.

Controlo da Execução: Ciclo For

12. Vamos fazer um programa que solicita ao utilizador o nome e exibe esse nome na vertical. Como não sabemos a dimensão do nome, vamos ter que utilizar um ciclo.

Crie o ficheiro `nome_vertical1.py` com o seguinte código:

```
nome = input("Como se chama? ")

i = 0
while i < len(nome):
  print(nome[i])
  i += 1
```

*Relembrando, uma **string** é uma sequência de caracteres numerada a partir de 0. A variável i é aqui utilizada para armazenar a posição (ie, o número de ordem) dos caracteres. Se o utilizador inserir `Armando`, então:*

```
i == 0 => nome[i] == 'A'
i == 1 => nome[i] == 'R'
...
```

*A função **print** automaticamente acrescenta uma nova linha e daí o efeito vertical.*

13. Teste o programa com o nome `Alberto`:

```
Como se chama? Alberto
A
L
B
E
R
T
O
```

*Em Python, o ciclo **for** permite aceder a todos os elementos de uma colecção (ou de uma estrutura de dados que possa ser "transformada" numa colecção). Noutras linguagens, este ciclo é designado de foreach ("para cada"), nome que é mais apropriado para a semântica da instrução.*
*O formato geral do **for** é:*

```
for var in sequência/colecção:
  instruções_que_utilizam var
```

*Ao contrário da linguagem C e derivadas, onde o ciclo for é um ciclo geral, semelhante ao **while**, e mais utilizado para percorrer gamas de valores em progressão, aqui o **for** percorre os itens da colecção pela ordem pela qual eles estão dispostos nessa colecção.*

14. Vamos agora utilizar o ciclo ***for***. Crie o ficheiro `nome_vertical2.py` e acrescente o código:

```
nome = input("Como se chama? ")
for car in nome:
  print(car)
```

15. Queremos agora desenvolver um programa para calcular a média de valores passados através da linha de comandos. O programa deve começar por ler os números para uma lista de valores e depois é que calcula a soma (esta lista de números não é necessária, mas vamos assumir que é importante guardar os valores introduzidos para posterior processamento).
Vamos desenvolver duas versões, uma com ***while***, e outra com ***for***. Noutros laboratórios veremos uma forma mais directa de resolver este problema.

Crie um ficheiro com o nome `media1.py` e comece por acrescentar o seguinte código:

```
import sys
```

```
if len(sys.argv) < 3:
    print("Utilização: python3", __file__, "num1 num2 [num3 ... numN]", file=sys.stderr)
    sys.exit(2)

nums = []
i = 1
while i < len(sys.argv):
    # Acrescenta o float na i-ésima posição de argv
    nums.append(float(sys.argv[i]))
    i += 1

print("Números lidos com sucesso.")
```

*`sys.argv[0]` é uma alternativa a **`__file__`**. Enquanto **`__file__`** indica sempre o nome do script (`media1.py` neste caso), `sys.argv[0]` indica como é que o script foi invocado. Se criarmos uma ligação simbólica (`ln -s` em Unixes) para o script, então este será invocado com um nome diferente do nome do ficheiro.*

Em caso de invocação indevida do script, a mensagem de utilização é enviada para a saída de erros padrão (STDERR - `sys.stderr`) e não para a saída padrão (STDOUT - `sys.stdout`), que fica assim reservada para o output "regular" do programa. Voltaremos a este assunto mais em baixo no laboratório.

Além de forçar o fim do programa, a função `sys.exit` permite devolver um código de erro para o SO. Um valor diferente de 0 significa que o script não teve sucesso. Fica ao critério de cada programa definir o significado de cada código de erro > 0. Porém, é comum utilizar o valor 2 para indicar erros de síntaxe na invocação do sript na linha de comandos.

16. Agora acrescente o ciclo que *itera* sobre a lista de números e efectua a soma dos elementos (atenção à indentação: este código deve estar alinhado com o ***`print`*** e o ***`while`*** anteriores).

```
soma = 0
i = 0
while i < len(nums):
    soma += nums[i]
    i += 1
```

17. Finalmente, as instruções que apresentam o resultado:

```
print("Soma:  ", soma)
print("Média: ", soma/len(nums))
```

18. Teste o programa.

19. Vamos agora desenvolver a versão para o ciclo ***`for`***. Crie o ficheiro `media2.py` e substitua o primeiro ciclo pelo seguinte:

```
for num_txt in sys.argv[1:]:
    nums.append(float(num_txt))
```

Pergunta: porquê `sys.argv[1:]` e não apenas `sys.argv`?

20. E agora substitua o segundo ciclo por:

```
for num in nums:
    soma += num
```

*Este ciclo **`for`** é, na verdade, desnecessário porque podemos sempre utilizar a built-in **`sum`** que soma todos os números presentes num objecto iterável, como é o caso de uma lista. Ou seja, em vez do ciclo **`for`** bastava escrever o seguinte:*

```
soma = sum(nums)
```

Tópicos Variados...

21. Em Python, qualquer objecto pode ser avaliado num contexto booleano, isto é, qualquer objecto pode ser sujeito a um teste de verdadeiro ou falso para, por exemplo, ser utilizado numa condição de um *if* ou de um *while*. Os seguintes valores são avaliados a *False* em Python:

. *False*

. *None*

. Valor `0` de qualquer tipo numérico (`0`, `0.0`, `0j`)

. Uma sequência vazia (`''`, `[]`, `()`) , um dicionário ou um conjunto vazio (`{}`, `set()`)

. Objectos de classes que definem os métodos *__len__* ou *__bool__* e para os quais este métodos devolvem o inteiro `0` ou *False* (veremos isto quando falarmos de classes)

Alguns exemplos:

```
>>> txt = ''
>>> if txt:
      print("A string txt tem conteudo")
>>> if not txt:
      print("A string txt está vazia")
A string txt está vazia
>>> bool(0)
False
>>> bool(0.0)
False
```

```
>>> bool(19)
True
>>> bool([])
False
>>> bool([1, 2])
True
>>> bool(None)
False
>>> bool([None])
True
```

22. Duas funções *built-in* úteis para trabalhar como números: *abs* - devolve o valor absoluto do número - e *divmod* - devolve numa só operação o resultado e o resto da divisão inteira.

```
>>> x, y = -4, 4
>>> abs(x)
4
>>> abs(x) == y
True
>>> quociente, resto = divmod(11, 4)
>>> quociente, resto
(2, 3)
>>> quociente, resto = divmod(11.0, 4)
>>> quociente, resto
(2.0, 3.0)
```

23. Em Python 3 a função *built-in* ***input*** devolve apenas texto. Quando precisamos de obter um valor numérico a partir de informação introduzida pelo utilizador, então temos que converter o resultado da função input para o tipo de dados numérico pretendido. Alternativamente, podemos fazer:

1. `eval(input(...))`
2. `ast.literal_eval(input(...))`

Um dos aspectos mais poderosos da linguagem Python, e que torna a linguagem "muito" dinâmica, é o facto de termos sempre à nossa disposição o interpretador (através de funções como ***eval****,* ***exec****, etc.) que podemos utilizar para avaliar instruções e expressões construídas em tempo de execução. Isto permite uma técnica de programação designada de metaprogramação: programas que escrevem programas. Abstracto, complexo, mas poderoso! Para utilizar com cautela e sem nunca perder de vista a Lei Homem-Aranha: "Muito poder, traz muita responsabilidade!" ("With great power, comes great responsability!")*

Ambas as alternativas invocam o interpretador para avaliar o resultado de ***input***, mas a primeira pode ter sérios problemas de segurança, ao passo que a segunda avalia apenas valores literais (mas é necessário fazer import `ast`) e, como tal, não permite código arbitrário. Mas vamos explorar as alternativas:

```
>>> idade = int(input("Qual a sua idade? "))   # quem diz int, diz float, complex...
Qual a sua idade? 31
>>> idade, type(idade)
(31, <class 'int'>)

>>> idade = eval(input("Qual a sua idade? "))
Qual a sua idade? 31
>>> idade, type(idade)
(31, <class 'int'>)
>>> idade = eval(input("Qual a sua idade? "))
Qual a sua idade? 31.5
>>> idade, type(idade)
(31.5, <class 'float'>)

>>> idade = eval(input("Qual a sua idade? "))
Qual a sua idade? print('Vai ser executado código arbitrário. Podia ser malicioso!')
Vai ser executado código arbitrário. Podia ser malicioso!
>>> import ast
>>> idade = ast.literal_eval(input("Qual a sua idade? "))
Qual a sua idade? print('Vai ser executado código arbitrário. Podia ser malicioso!')
Kaboom....!! ValueError! Código potencialmente malicioso não executado!
>>> idade = ast.literal_eval(input("Qual a sua idade? "))
Qual a sua idade? 31
>>> idade, type(idade)     # literal_eval só aceita literais
(31, <class 'int'>)
```

24. Em Python 3, todos os valores são um objecto de uma determinada classe. Essa classe deriva da classe ***object***. Além disto, cada objecto tem um identificador único (que, em geral, é o seu endereço de

memória). Para compararmos se dois objectos são o mesmo utilizamos o operador *is*; para verificarmos que não são utilizamos *is not*. Eis alguns aspectos mais relacionados com esta temática:

```
>>> id(1), id(2)
(4297366496, 4297366528)
>>> x = y = 1
>>> id(x), id(y)
(4297366496, 4297366496)
>>> x is y
True
>>> x is not y
False

>>> str1 = 'abc'
>>> str2 = 'abc'
>>> str1 is str2, str1 == str2
(True, True)
>>> id(str1), id(str2)
(4300399536, 4300399536)
>>> str2 = '12abc'[2:]
>>> str1, str2
('abc', 'abc')
>>> str1 is str2, str1 == str2
(False, True)
>>> id(str1), id(str2)
(4300399536, 4302751648)
```

```
>>> type(19)
<class 'int'>
>>> type(False)
<class 'bool'>
>>> isinstance(19, int)
True
>>> isinstance(19, bool)
False
>>> isinstance(False, bool), \
    isinstance(False, int)
(True, True)
>>> isinstance(False, object), \
   isinstance(19, object)
(True, True)
>>> type(19) is object, \
   type(False) is object
(False, False)

>>> issubclass(int, object), \
   issubclass(bool, object), \
   issubclass(bool, int)
(True, True, True)
```

25. E aqui ficam todas as palavras-reservadas da linguagem Python até à versão 3.4:

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

EXERCÍCIOS DE REVISÃO

1. Quantas palavras-reservadas foram abordadas até agora?

2. Crie duas strings, atribua a ambas o valor `"Alberto"` mas garanta que ficam com ids diferentes.

3. Quais as diferenças entre os operadores `==` e `is` ?

4. O que significa "imutabilidade"? Dos tipos de dados até agora vistos, indique três tipos imutáveis.

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

5.1 `b, c, d = not [], not [[]], not 'Alberto'`

5.2 `x , y, z = 2 is 2, [2] is [2], 2 == 2`

5.3 `c = '.'.join("ABCDEFGHIJKLMNOPQRSTUVWXYZ")[10]`

5.4 `m = 5/2 + 5//2 + 5%2 + 5**2`

5.5 `p = [1, 2, 3, 4][:3][-3] + [1, 2, 3, 4][:3][2]`

5.6 `x = [{'a': 1, 'b': 2}, {'a': 10, 'b': 20}][:-1][0]['b']`

5.7 `g = ord('g') - ord('f')`

5.8 `r = 4 * 3 ** 2`

5.9 `s = set("armanda") & set(['igreja', 'cidade'][{'a': 0, 'b': 1}['b']])`

6. Considerando que inicialmente `vals = [22, 23, 24, 25, 26]` , responda às seguintes questões.

6.1 `vals[2:-2]` = _____

6.2 `vals[:-len(vals)]` = _____

6.3 `vals[:-len(vals) + 1]` = _____

6.4 `vals[1:], vals[0] = vals[:-1], vals[-1]` `vals` = _____

6.5 `vals[:-1], vals[-1] = vals[1:], vals[0]` `vals` = _____

7. Indique como juntar duas listas através de *slicing*. Este processo pode ser aplicado a tuplos ou *strings*?

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

`x=3,5/2` `print(x)`	
`codigo = {'A': 19, 'B': 20}` `print(list(codigo))`	

`codigo = {'A': 19, 'B': 20}` `print(codigo['B'], codigo.get('C'), codigo.get('C', 21))`	

`codigo = {'A': 19, 'B': 20}`
`d = dict(a=list(codigo.values())[0],`
`b=list(codigo.values())[-1])`
`print(d)`	

`processos = {'ls': 192, 'grep': 321, 'init': 1}`
`print('ls' in processos, 321 in processos)`
`print((192 in processos)*2)`
`processos.update(ls=292, mkfs=19)`
`print(list(processos.items()))`	

`pessoa = {'nome': 'alberto', 'idade': 23, 'altura': 190}`
`print(pessoa['nome'].upper())`
`print(pessoa.get(pessoa['idade'], 'altura'))`	

`x = 'ABC-DEF-GHI--JKL'.split('-')` `print("|{:^5}|".format(x[2:5][2]).replace(chr(32), '.'))`	

`dados = [1993, "MANEL[1, 7, 8, 2]I", {'m': 12, 'n': 14}]`
`dados[0] += 17`
`print(str(dados[0])[2])`
`print(eval(dados[1][5:-1])[-3:-1])`	

`formulas = [[1, -1, 2], [3, 2, 2]]`
`m = formulas[:]`
`m[0] = [9, 9, 9]`
`print(m, '--', formulas[0])`
`m = formulas[:]`
`m[0][0] = 9`
`print(m, '--', formulas[0])`	

`vals = [1, 2, 3]`
`print(vals[-1:])`
`vals[-1:] = [4, 5]`
`print(vals)`	

`t = (1, 2, 3)`
`print(t[-1:])`
`vals = [10, 20, 30]`
`vals[-2:] = t`
`print(vals)`	

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

10. Quantas partes possui um ciclo FOR em linguagens derivadas de C?

11. Escreva um ciclo *for* para exibir os números pares de 0 a 100.

12. Converta os seguintes ciclos *while*/*for* em ciclos *for*/*while*:

`i = 0`
`while i < 10:`
`  print(i)`
`  i += 2`	

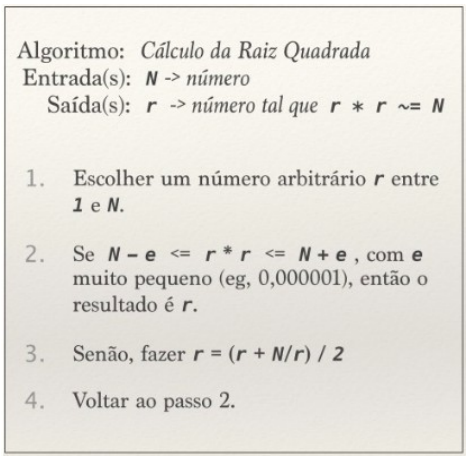
`for k in range(10, 2, -2):` `  print(k)`	

`nums = (10, 2, 7, 9, 6, 5, 8)`
`i = 0`
`while i < len(nums):`
`  print(nums[i])`
`  i += 2`	

EXERCÍCIOS DE PROGRAMAÇÃO

13. Faca um programa para exibir os códigos numéricos () das letras de `'a'` a `'z'` e de `'A'` a `'Z'`. Poderá necessitar de utilizar as funções built-in **`ord`** e **`chr`**.

14. Faça um programa para calcular a raiz quadrada de um número de acordo com o seguinte algoritmo



Algoritmo: *Cálculo da Raiz Quadrada*
Entrada(s): `N` -> *número*
Saída(s): `r` -> *número tal que* `r * r ~= N`

1. Escolher um número arbitrário `r` entre `1` e `N`.
2. Se `N - e <= r * r <= N + e`, com `e` muito pequeno (eg, 0,000001), então o resultado é `r`.
3. Senão, fazer `r = (r + N/r) / 2`
4. Voltar ao passo 2.

Utilize a função `uniform` do módulo `random` para obter um número `float` entre 1 e N.

15. Vamos fazer um programa de adivinha. Este programa começa por solicitar um número ao utilizador e, caso este número seja igual a um número pré-definido (o *número mágico*), o programa felicita o utlizador por ter acertado. Caso contrário, indica que o utilizador falhou.

16. Investigue o módulo `random` e acrescente possibilidade de o número mágico ser um número (pseudo) aleatório entre 1 e 20.

→ **17.** Acrescente a possibilidade de repetição ao programa anterior enquanto o utilizador não acertar. O programa deve dar pistas ao utilizador. Se este estiver a três valores de distância, então o programa indica que "está próximo", se estiver a um valor, o programa diz que "está muito próximo".

18. Acrescente a conversão inversa ao programa de conversão de câmbios. O programa deverá por começar por perguntar qual o sentido da conversão, apresentando depois a conversão. Exemplo:

```
Escolha o sentido da conversao
1. Euros   -> Dolares
2. Dolares -> Euros
> 2

Montante em dólares: 2000
Euros -> 1438.85
```

19. Acrescente a possibilidade de o utilizador poder repetir a conversão enquanto desejar.

20. Faça um programa para validar um NIF (Número de Identificação Fiscal). O programa deve aceitar o número através da linha de comandos. Se o utilizador não inserir nenhum número a partir da linha de comandos, então deve solicitar a introdução de um número ao utilizador.

De acordo com a Wikipedia (procurar por 'Número de identificação fiscal') as regras são as seguintes:

"*O NIF tem 9 dígitos, sendo o último o digito de controlo. Para ser calculado o digito de controlo:*

1. *Multiplique o 8.º dígito por 2, o 7.º dígito por 3, o 6.º dígito por 4, o 5.º dígito por 5, o 4.º dígito por 6,o 3.º dígito por 7, o 2.º dígito por 8, e o 1.º digito por 9*
2. *Adicione os resultados*
3. *Calcule o Módulo 11 do resultado, isto é, o resto da divisão do número por 11.*
4. *Se o resto for 0 ou 1, o dígito de controle será 0*
5. *Se for outro algarismo x, o dígito de controle será o resultado de 11 - x"*

21. Agora faça um programa que utiliza o anterior e devolve todos os números primos até ao número introduzido pelo utilizador na linha de comandos.

22. Investigue o módulo `datetime` e faça um programa que quando chamado sem argumentos indica a data/hora actual. Alternativamente, pode receber uma ou duas datas, indicando o número de dias entre estas datas. Se apenas receber uma data, utiliza como segunda data a data actual.

23. Desenvolva um programa que recebe texto através da entrada padrão e formata-o a X colunas. Leia todo o texto da entrada padrão através do método `sys.stdin.read` (que veremos na próxima parte do laboratório). Utilize o módulo `textwrap` para formatar o texto. Exemplo:

Conteúdo de "diario.txt"	**$ python3 formata_txt.py 20 < diario.txt**
10/10/10: Fui ao mercado comprar peixe para o jantar. Encontrei o Alberto e o Armando. Convidei-os para jantar. Conversámos sobre o António, que eles encontraram no casamento do Arnaldo.	10/10/10: Fui ao mercado comprar peixe para o jantar. Encontrei o Alberto e o Armando. Convidei-os para jantar. Conversámos sobre o António, que eles encontraram no casamento do Arnaldo.

24. Faça um programa para traduzir as coordenadas "simbólicas" do Excel para coordenadas lineares. Por exemplo, em Excel, internamente, a célula A1 corresponde à célula na linha 0 e coluna 0. Exemplos:

```
Indique as coordenadas: Z 2
Linha: 1  Coluna: 25
---
Indique as coordenadas: AA 3
Linha: 2  Coluna: 26
--
```

```
Indique as coordenadas: AB 17
Linha: 16  Coluna: 27
--
Indique as coordenadas: CD 17
Linha: 16  Coluna: 81
--
Indique as coordenadas: sair
Fim do programa
```

25. Investigue o módulo `smtplib` e depois utilize-o para desenvolver um programa para enviar um email. O email deve ser apenas composto por texto, não sendo necessário suportar outro tipo de conteúdos (eg, HTML, imagens, etc.) ou anexos. Recorra a um servidor de SMTP. Pode, por exemplo, utilizar o seu fornecedor de email, desde que este suporte SMTP.

O programa deve solicitar interactivamente a introdução do endereço de email do emissor, do destinatário e o assunto. Depois lê o conteúdo de email até o utilizador terminar com uma linha contendo apenas os caracteres `#$%` , após o que envia o email. Utilize o módulo `re` e uma expressão regular apropriada para validar o endereço de email.