INSTITUTO DO EMPREGO
E FORMAÇÃO PROFISSIONAL

Introdução à Programação

UFCD(s) 5117, 5091, 5118, 5119

GUIA DE LABORATÓRIO 2
ELEMENTOS BÁSICOS DA LINGUAGEM (ßeta)

OBJECTIVOS

- Introduzir e/ou aprofundar as noções de objecto, identificador, variável, tipo de dados, expressão, módulo
- Percorrer os tipos de dados e operadores principais da linguagem Python

INSTRUÇÕES

Objectos, Valores, Expressões, Operadores, Literais e Tipos de Dados

1. Aceda à linha de comandos do seu sistema operativo e inicie o REPL/*shell* do Python:

```
$ python3
```

O ambiente interactivo (ie, o REPL, ou a shell, *ou a linha de comandos) do Python possui dois símbolos de linha de comandos:*
>>> linha de comandos principal
. . . linha de comandos secundária
Conforme veremos, a segunda linha de comandos indica a continuação de um comando iniciado numa das linhas anteriores. Nestes exemplos de laboratório, linhas que não começem com um destes símbolos, são a saída (output) dos comandos que estamos a introduzir. Tudo o que está à direita do símbolo # é um comentário e é ignorado pelo interpretador

2. Comece por introduzir o seguinte:

```
>>> 10
10
>>> 3*3
9
>>> 30 - 5*6
0
>>> 2*3.14159265*4
25.1327412
>>> (50 - 5*6) / 4
5.0
>>> 6 / 4
1.5
```

*Um programa em Python consiste numa sequência de **definições** e de **comandos**. São estes os dois tipos básicos de instruções. Ambos os tipos de instruções alteram o estado do ambiente, mas as instruções do primeiro tipo servem para acrescentar novos conceitos à linguagem, ao passo que as outras são executadas e produzem imediatamente algo.*
Estas instruções, o script (ie, o programa), podem ser introduzidas uma a uma no REPL, ou carregadas a partir de um ficheiro.
*Um **comando** (**command** ou **statement**) instrui o intrepretador a fazer algo. Vamos começar por ver alguns exemplos de comandos e depois passamos às definições.*
*Podemos ainda dividir as instruções em **simples** e **compostas**. Uma instrução simples (que é sempre um comando) vale por si só, isto é, não faz parte de um conjunto de instruções. Uma instrução composta "necessita " de outra instruções para ficar completamente definida. As definições são sempre instruções compostas, mas não são as únicas.*

3. Introduza agora:

```
>>> "Olá, Mundo!"
'Olá, Mundo'
```

Começámos por utilizar o REPL como uma calculadora. Introduzimos expressões, o interpretador calculou o seu valor e o REPL exibiu esse valor.

```
>>> print("Olá, Mundo!")
Olá, Mundo!
>>> print("Bom dia,", "Alberto")
Bom dia, Alberto
>>> print("Bom dia," + "Alberto")
Bom dia,Alberto
```

*O Python organiza a informação em **objectos**. "Objectos" são "valores" com um determinado tipo de dados, **tipo de dados** esse que define as operações que estão disponíveis para esses objectos. Por exemplo, para objectos do tipo "número inteiro", como `3` ou `10`, podemos utilizar as habituais operações aritméticas. Para objectos do tipo **`str`** (texto), como `"Bom dia,"` ou `"Alberto"`, podemos fazer pesquisas, converter as letras maiscúlas/minúsculas em minúsculas/maiscúlas, entre muitas outras operações.*

Existem vários tipos de dados em Python, e podemos agrupá-los em duas categorias genéricas:
*.**Simples** ou **Primitivos** ou **Escalares**: objectos deste tipo são indivisíveis e possuem suporte especial ao nível do interpretador*
*.**Compostos** ou **Não-Escalares**: objectos deste tipo possuem uma estrutura interna complexa, foram, possivelmente, definidos em bibliotecas, e são constituidos por outros objectos de tipos escalares ou não-escalares*
Nos exemplos anteriores, os objectos numéricos `10`, `3`, `3.14159265`, etc., são escalares, ao passo que os objectos entre aspas são não-escalares. Em Python encontramos quatro tipos de objectos escalares:
*. **`int`** : é utilizado para representar números inteiros conforme nós os conhecemos da matemática*
*. **`float`** : é utilizado para representar números reais. Literais deste tipo incluêm um ponto decimal (eg, 3.0 ou -17.1). Também podem ser escritos em notação científica: 4.2E2 é igual a $4.2*10^2 = 4200$). A designação **`float`** deriva da representação binária: estes números são representados num formato designado por "binário com virgula flutuante" e definido pelo Institute of Electrical e Electronics Engineers (IEEE). Este formato é muito eficiente, mas conduz a várias imprecisões de representação. Para uma representação mais "correcta", ainda que menos eficiente e menos compacta, o Python disponibiliza na biblioteca padrão o formato `Decimal`. O formato **`float`** é utilizado em muitas linguagens modernas além de Python. Nalgumas linguagens, como JavaScript, não existe uma separação entre números inteiros e números reais. Em Python, o tipo de dados **`float`** corresponde ao tipo de dados **`double`** da linguagem C.*
*. **`bool`** : tipo com os dois valores, os valores lógicos **`True`** e **`False`** (primeira letra tem mesmo que ser uma maiúscula). O tipo **`bool`** é um subtipo do tipo **`int`**, ou seja, podemos fazer **`True + 3`** e obter **`4`***
*. **`NoneType`** : tipo que representa _nada_. Apenas tem um valor: **`None`** . Veremos a sua (imensa) utilidade mais à frente.*

*Em terminologia de linguagens de programação, designamos cada um dos valores "soltos" no código por **literais**. A cada um destes literais o compilador atribui um tipo de dados. Literais são objectos que não necessitam estar associados a uma zona de memória (a uma variável, conforme veremos a seguir). Assim, **`3`** é um literal do tipo **`int`**, e **`"Alberto"`** é um literal do tipo **`str`**.*
*Um conjunto de caracteres entre aspas, ou entre apóstrofos (plicas), constitui um literal do tipo **`str`** (abreviatura de string). O tipo de dados **`str`** consiste numa sequencia de zero ou mais caracteres e é utilizado para representar texto. Como o Python não interpreta o seu conteúdo, podemos escrever o que pretendermos dentro de uma string. Para além disto, podemos utilizar o operador + para concatenar uma string com outra string. Veremos à frente que podemos representar os literais do tipo **`str`** de outras duas formas.*

4. Como seria de esperar, o interpretador faz um verificação sintática das expressões introduzidas:

```
>>> 3 +
File "<stdin>", line 1
    3 +
      ^
SyntaxError: invalid syntax
```

5. Todos os valores são objectos com um tipo de dados. Vamos inspeccionar os tipos dos objectos:

```
>>> type(1)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type("abc")
<class 'str'>
```

*A função **type** devolve o tipo de dados de qualquer objecto da linguagem. Note-se que até os tipos de dados têm um tipo de dados que é o tipo de dados **type**. Esta função consegue ainda outras proezas que iremos estudar quando falarmos de classes.*

No código anterior, trabalhámos com os seguintes tipos de dados:
*. **int**: 10, 3, 4, etc.*
*. **float**: 3.14159265, 1.5 , etc.*
*. **str**: "Alberto",*
*. **builtin_function_or_method**: print*

*Em Python3, **print** é uma **função** que serve para (entre outras coisas) exibir uma mensagem no REPL. Veremos mais à frente o que são funções, mas, para já, importa reter que funções são objectos que "fazem algo" e, tal como os restantes objectos da linguagem, uma função possui um tipo de dados. No caso de **print** esse tipo é builtin_function_or_method. A função **print** pertence à categoria de funções pré-definidas, e que o Python designa por **built-ins**. Durante este laboratório vamos utilizar outras funções built-in, tais como **type**, **len**, etc.*

***NOTA**: Atenção que em Python 2 **print** não é uma função, mas sim uma palavra-reservada que indica uma instrução/comando.*

6. Continuando a inspeccionar os tipos de dados dos objectos, introduza ainda:

```
>>> type(True)
<class 'bool'>
>>> type(print)
<class 'builtin_function_or_method'>
>>> type(type(1))
<class 'type'>
```

7. Continuando, desta feita introduza:

```
>>> 3 > 2
True
>>> 3 != 2
True
>>> 3 == 2
False
>>> "Alberto" == "alberto"
False
>>> "Alberto" == "Alberto"
True
>>> not 3 == 2
True
>>> not 3 != 2
False
>>> 3 > 2 and 10 != 8
True
>>> 3 > 22 and 10 != 8
False
>>> 3 > 22 or 10 != 8
True
```

*Em cima escrevemos várias expressões - eg: 30 - 5*6 . Uma **expressão** é "tudo aquilo" que produz um valor. Resulta da combinação de objectos e de outras expressões, por meio de operadores. Um **operador** é um simbolo que representa uma operação. Exemplos de operadores: **+** (soma), **/** (divisão), **<<** (deslocamento binário à esquerda), etc. A expressão 3*(14 + 25) combina duas subexpressões - 3 e 14 + 25 - através do operador * para produzir o valor 117.*
Expressões não necessitam necessariamente de envolver valores numéricos:
*. "Bom dia," + "Alberto" -> expressão que junta duas strings (**string** é o termo que utilizamos para texto) "Bom dia, " e "Alberto" e produz o texto "Bom dia,Alberto"*
*. 10 > 20 -> expressão que produz o valor booleano **False***
*O operador **==** avalia se duas expressões produzem o mesmo valor. O operador **!=** avalia o oposto, isto é, se duas expressões produzem valores diferentes. Note-se que ambos os operadores produzem **True** ou **False**. Designam-se por **operadores booleanos** ou **lógicos**. Outro operador lógico, o **not**, devolve o valor lógico oposto ao da expressão à direita. Repare que este operador é representado por uma palavra. A linguagem Python reserva determinadas palavras para serem utilizadas pelo interpretador e essas palavras, designadas por **palavras-reservadas** ou **palavras-chave**, não podem ser utilizadas pelos programadores para dar nomes a objectos (algo que, veremos, é muito comum).*

```
>>> not (3 > 22 or 10 != 8)
False
>>> type(3) == type(4)
True
>>> True != False
True
>>> "abc" >= "xyz"
False
>>> "abc" >= "Xyz"
True
```

*Outros dois operadores lógicos que também são palavras-reservadas são os operadores **and** ("E lógico") e **or** ("OU" lógico). O primeiro devolve **True** apenas quando as expressões à esquerda e à direita do operador forem ambas verdadeiras. O segundo devolve **False** apenas quando as expressões à esquerda e à direita do operador forem ambas falsas.*
*Um pouco de terminologia: operadores **unários**, como o **not**, são os que avaliam apenas uma expressão, **binários**, os que avaliam duas (**and**, +, etc.), **ternários**, os que avaliam três (veremos mais à frente), etc.*

i + j	Soma **i** com **j**. Se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**.
i - j	**i** menos **j**. Se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**.
i * j	**i** vezes **j**. Se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**.
i / j	**i** a dividir por **j**. O resultado é sempre um **float**. Em Python 2 e noutras linguagens, se `i` e `j` forem ambos do tipo **int**, o resultado é um **int,** e se um dos valores for do tipo **float**, o resultado é um **float**. Nestas linguagens, `3/2` produz `1` e não `1.5` . Em Python 3 o resultado é de facto `1.5` .
i // j	Divisão inteira de **i** por **j**. A divisão inteira ignora o resto da divisão e devolve o quociente. Assim, `3//2` devolve `1`, `5//2` devolve `2` e `5.0//2` devolve `2.0` . Ou seja, se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**, mas em todos os casos é feita a divisão inteira.
i % j	O resto da divisão de **i** por **j**. Se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**. Exemplos: `5 % 2 == 1`, `5.0 % 2 == 1.0`, `5.5 % 2 == 1.5` .
i ** j	**i** levantado à potência **j**. Se `i` e `j` forem ambos do tipo **int**, o resultado é um **int**. Se algum for do tipo **float**, o resultado é um **float**.
>, >=, <, <=, ==, !=	Operadores relacionais e de comparação. Como operadores lógicos que são, produzem **True** ou **False**.

Operadores para os tipos int e float

8. Nem sempre podemos aplicar todos os operadores a todos os tipos de operandos. Por exemplo:

```
>>> 4 + 'bacalhau'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 10 / '5'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

*Obtivemos um erro de semântica estática. Ou seja, sintacticamente a expressão está correcta mas o Python recusa-se a determinar o seu significado e assinala o erro de uma forma especial: lança uma excepção do tipo **TypeError**. **Excepções** são um tipo de objecto com a propriedade de interromper a execução do programa se nós programadores não fizermos nada para isso. A excepção **TypeError** serve precisamente para assinalar que determinadas operações não se aplicam a determinados tipos de dados.*

9. No entanto, e nalguns casos talvez seja surpreendente, podemos fazer o seguinte:

```
>>> 4 * 'bacalhau'
'bacalhaubacalhaubacalhaubacalhau'
>>> "bacalhau" + ' com ' + "grão"
'bacalhau com grão'
>>> True + 2
3
>>> False * 19
0
```

10. Podemos converter de **int**s para **float**s ou **str**s e vice-versa. Vejamos:

```
>>> int(3.2)
3
>>> int(3.9)
3
>>> float(3)
3.0
>>> int(3.8)
3
>>> str(3)
'3'
>>> str(2) + " mais " + str(3) + " dá " + str(2 + 3)
'2 mais 3 dá 5'

>>> int('bacalhau')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'bacalhau'
>>> int('2A')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '02A'
>>> int('2A', 16)
42
>>> int('10', 2)
2

>>> int('3.8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '02A'
>>> int(float('3.8'))
3
```

*Os tipos de dados **int**, **float** e **str** designam também três funções que permitem construir um tipo de dados a partir de outro (se possível). Designamos estas funções por métodos construtores ou inicializadores, algo que aprofundaremos quando falarmos de classes. Ou seja, int é simultâneamente um tipo de dados e um construtor/inicializador, e o mesmo se passa com **float** e **str**.*
*Note-se que a função **int** permite construir números inteiros escritos em outras bases para além da base decimal (eg, binária, octal, hexadecimal).*

*A excepção **ValueError** ocorre quando um determinado valor é inválido para uma operação. A questão aqui não reside no tipo de dados, mas sim no valor em concreto.*

11. Números de vírgula flutuante são uma aproximação a números reais e não são mesmo números reais:

```
 >>> 5.9 - 2
3.9000000000000004
```

```
>>> 1.1 + 2.2
3.3000000000000003
>>> 1.0 % 0.1
0.09999999999999995
# devia dar resto 0 porque 1 é divisível por 0.1
```

*Como foi referido numa caixa mais acima, **float**s não são reais. São uma representação eficiente mas inexacta. Em situações em que a exactidão é crucial (eg, aplicações financeiras), devemos utilizar o tipo de dados `Decimal` definido no módulo `decimal` da biblioteca padrão.*
*Um **módulo** corresponde a um conjunto de definições relacionadas e guardadas num ficheiro com o nome do módulo e com extensão `.py` . Estas definições podem depois ser **import**adas para outros módulos ou para o REPL. De notar que, por exemplo, as funções print e type também foram definidas num módulo: o módulo **builtins**. Todavia, devido à sua importância não é necessário importar este módulo.*

12. E é bom termos consciência disso, caso contrário as consequências podem ser graves:

```
>>> (1.1 + 2.2) - 3.3 == 0
False
```

13. Vamos utilizar o tipo de dados `decimal.Decimal` para trabalharmos com uma representação exacta dos números reais. Comece por fazer:

```
>>> from decimal import Decimal
```

14. Agora faça:

```
>>> Decimal('1.1') + Decimal('2.2')
Decimal('3.3')
>>> Decimal('1.1') + Decimal('2.2') - \
    Decimal('3.3')
Decimal('0.0')
>>> Decimal('1.0') % Decimal('0.1')
Decimal('0.0')

>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(0.70 * 1.05, 2)
0.73
```

Importamos o módulo `xpto` fazendo:

```
import xpto
```

Esta instrução importa todas as definições exportáveis presentes em `xpto`. Depois se pretendermos utilizar a função `abc` definida em `xpto` temos que fazer

```
xpto.abc()
```

Repare que prefixámos o nome da função com o nome do módulo, separando ambas as partes com o operador de acesso `.` (ponto).
*Se apenas estivemos interessados na função `xpto`, temos com alternativa a instrução **from** `X` **import** `Y`:*

```
from xpto import abc
```

E agora podemos utilizar esta função sem a termos que prefixar com o nome do módulo:

```
abc()
```

***import** e **from**, à semelhança de **and**, **or**, **not**, etc., são também palavras-reservadas. A biblioteca padrão (e qualquer biblioteca em Python) não passa de uma colecção de módulos. Voltaremos aos módulos nos laboratórios seguintes.*
Apenas um esclarecimento para evitar confusões: no exemplo anterior, `decimal` (com 'd' pequeno) é o nome do módulo ao passo que `Decimal` é o nome do tipo de dados e respectivo construtor, definido no módulo `decimal`.

Variáveis

15. Introduza o seguinte:

```
>>> lado = 10
>>> comp = 20
>>> area = lado * comp
>>> print("Área do rectângulo é:", area)
Área do rectângulo é:  200
>>> type(area)
<class 'int'>
>>> import math
>>> raio = 3
>>> area = math.pi * raio ** 2
>>> print("Área da circunf. é:", area)
Área da circunf. é: 28.274333882308138
>>> type(area)
<class 'float'>
```

*lado, comp, area são variáveis. Em Python, uma **variável** é apenas e só um nome que referencia a um objecto. Este nome pode associar-se a diferentes objectos ao longo da "vida" do programa e daí a designação "variável" (varia o objecto ao qual a variável está associada). As variáveis, no fundo, representam a história do programa. Elas servem para guardar informação que pode ser utilizada mais tarde. Indicam em que "estado" é que um programa se encontra e permitem-nos tomar decisões em função do que já se passou.*
*Criamos uma variável quando atribuimos um valor/objecto com operador de atribuição **=**. Este operador indica que o objecto que está à direita está ligado (bounded) à variável que está à esquerda. Note-se que este operador _não_ serve para "perguntar" se o que está à direita é igual ao que está à esquerda (esse operador é o **==**).*
*Ao contrário de linguagens estáticas como C++ e Java, as variáveis em Python não têm tipo de dados; os objectos, sim. Deste modo, a variável area pode num dado instante referir uma "quantidade" **int**eira, e mais à frente, uma quantidade **float**.*

16. Podemos desassociar um nome a um objecto, através da instrução **del**.

```
>>> del raio
>>> area = 2 * math.pi * raio
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'raio' is not defined
>>> # del print   -> péssima ideia! mas é possível..
```

*Uma noção importante em Python é a noção de **espaço de nomes** (**namespace**). Um espaço de nomes é um contexto onde determinados nomes são válidos. Por exemplo, cada módulo possui o seu espaço de nomes.*
*No caso do REPL, o espaço de nomes é designado por __main__, mas para já isto não é importante de fixar. O que é importante é perceber que um espaço de nomes corresponde a uma ou mais associações entre nomes e objectos, e que determinados nomes só são válidos dentro desse espaço de nomes. A instrução **del** quebra essa associação dentro do espaço nomes.*
*Repare que quando tentamos invocar um nome não se encontra definido obtemos uma excepção **NameError**.*
*Já agora, o termo técnico para nome é **identificador**. Em Python, os identificadores podem conter maiúsculas, minúsculas, dígitos (mas não podem começar por um dígito) e o caractere especial _ (*underscore*). À semelhança de C, C++, Java, etc, a capitalização é importante: Idade é um nome e idade é outro nome. Determinados identificadores, designados por **palavras-reservadas**, estão reservados para a própria linguagem e já têm um significado. Não podemos utilizar esses nomes para nomear objectos.*

17. Todos os objectos podem ser referenciados através de variáveis:

```
>>> exibe = print   # agora print também se chama 'exibe'
>>> exibe("Valor de PI:", math.pi)
Valor de PI: 3.141592653589793
>>> matematica = math
>>> exibe("Valor de PI:", matematica.pi)
Valor de PI: 3.141592653589793
```

18. Podemos inspecionar os nomes definidos em espaço de nomes através da função *built-in* **`dir`**:

```
>>> dir()     # devolve nomes no espaço de nomes do REPL
... resultado não indicado ...
>>> dir(math)
... resultado não indicado ...
>>> dir("uma string qualquer")
... resultado não indicado ...
>>> dir(1)    # numeros são objectos com espaço de nomes...quem diria?!
```

19. O operador = permite atribuir vários nomes a vários objectos:

```
>>> x, y = 10, 4
>>> z, x = x, 20
>>> print(x, y, z)
20 4 10
```

20. Isto é especialmente conveniente se quisermos trocar o valor de duas variáveis:

```
>>> a, b = 1, 11
>>> print(a, b, sep=', ')
1, 11
>>> a, b = b, a
>>> print(a, b, sep=', ')
11, 1
```

Em Python podemos fazer múltiplas atribuições numa só instrução. Essas atribuições são feitas em paralelo e sempre utilizando os valores antigos das variáveis. Trocar o valor de duas variáveis `var1` e `var2` é muito fácil. Basta fazer:

```
var1, var2 = var2, var1
```

Noutras linguagens que não suportam atribuição múltipla de valores teríamos que definir uma terceira variável para guardar temporariamente um dos objectos das referenciados por var1 ou var2. Por exemplo:

```
tmp = var1
var1 = var2
var2 = tmp
```

*A função **`print`** possui dois importante parâmetros com nome (*) mas que são opcionais: `sep` e `end`. Vejamos alguns exemplos auto-explicativos:*

```
>>> print(2, 3, 4)
2 3 4
>>> print(2, 3, 4, sep='->')
2->3->4
>>> print(2, 3, 4, '->')
2 3 4 ->
>>> print(2, 3, 4, end="FIM!")
2 3 4FIM!>>>
>>> print(2, 3, 4, end="FIM!  ", sep="->")
2->3->4FIM!  >>>
```

(*) - Quando falarmos de funções em detalhe, entre outras coisas, vamos ficar a saber o que são argumentos, parâmetros e parâmetros com nome.

Strings

21. Introduza o seguinte:

```
>>> "Alberto"    # podemos utilizar " (aspas) ou ' (plicas) para delimitar strings
'Alberto'
>>> 'Armando'
'Armando'
>>> marca = 'Levi\'s'    # a 2a plica não é para ser interpretada; prefixamos com \
```

```
>>> marca
"Levi's"
>>> print(marca)
Levi's
>>> "Aqui está uma string delimitada com \""
'Aqui está uma string delimitada com "'

>>> marca = "Levi's"    # hmmm...afinal a barra não é necessária
>>> marca
"Levi's"
>>> 'Aqui está uma string delimitada com "'
'Aqui está uma string delimitada com "'
```

> ***Strings**, à semelhança de listas e tuplos (tipos de dados que ainda não abordámos), são sequências **imutáveis** de 0 ou mais caracteres. Note-se que em Python não existe um tipo de dados para trabalhar com caracteres individuais, como sucede em C ou Java, onde o tipo de dados `char` identifica uma localização de memória com espaço para exactamente um caractere.*
> *Como quaisquer sequências, as strings são numeradas a partir de 0 e podem ser indexadas. O primeiro caractere está no índice, ou **posição**, 0, o segundo no índice ou posição 1, e assim sucessivamente. Se a string tiver N caracteres, o último está na posição N-1. Em Python também é possível numerar as strings "de trás para a frente" utilizando índices negativos. O último caractere está na posição -1, o penúltimo na posição -2, etc. Podemos visualizar isto através do seguinte esquema adaptado do tutorial em http://www.python.org .*

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
   0   1   2   3   4   5
  -6  -5  -4  -3  -2  -1
```

> *Muitas operações estão disponíveis para trabalhar com sequências. Adicionalmente, cada tipo de sequência disponibiliza operações específicas. Neste laboratório vamos começar por conhecer as básicas...*

22. Determinados caracteres são especiais e não é possível introduzi-los em strings através do teclado:

```
>>> texto = "Primeira linha\nSegunda linha"
>>> texto
'Primeira linha\nSegunda linha'
>>> print(texto)
Primeira linha
Segunda linha
>>> texto = "Primeira linha" \
            "Segunda linha"
>>> print(texto)
Primeira linhaSegunda linha
>>> texto = ("Primeira linha"
             "Segunda linha")
>>> print(texto)
Primeira linhaSegunda linha
>>> print("abc\ndef")
abc
def
>>> print(r"abc\ndef")
```

> *Quando o último caractere de uma instrução é a barra \, isto indica que a instrução ainda não acabou e continua na linha seguinte.*

> *Em Python, tal como em C e C++, uma barra \ dentro de uma string indica que o vamos introduzir um caractere especial (não confundir com uma barra \ fora de uma string, tal como referido noutra caixa). Exemplos de caracteres especiais:*
> *\n -> nova linha*
> *\t -> tabulação horizontal*
> *\v -> tabulação vertical*
> *\a -> um som*
> *Se prefixarmos a string com o caractere `r`, então todos os caracteres dentro da string são interpretados literalmente.*

```
abc\ndef
```

23. Também podemos utilizar um *triplete* de aspas ou de plicas para definir strings. A única diferença reside no facto de os fins de linha serem incluídos nas ditas :

```
>>> texto = """
Segunda linha
Terceira linha
"""
>>> texto
'\nSegunda linha\nTerceira linha\n'
>>> print(texto)

Segunda linha
Terceira linha

>>> texto = '''
Segunda linha e chega'''
>>> print(texto)

Segunda linha e chega
>>>
```

Quando trabalhamos com """ ou com ''' , podemos suprimir uma nava linha com a barra \. Por exemplo:

```
>>> txt = """
Segunda linha""
>>> print(txt)
'\nSegunda linha'
>>> txt = """\
Primeira linha"""
>>> print(txt)
'Primeira linha'
```

24. Podemos juntar strings literais com o operador + (quando aplicado a strings é designado de operador de concatenação) ou podemos simplesmente juntá-las:

```
>>> 'ABC' + 'DEF'
'ABCDEF'
>>> 'ABC' 'DEF'
'ABCDEF'
>>> 'ABC' \
    'DEF'
'ABCDEF'
```

25. Porém, esta segunda hipótese já não resulta se uma das strings não for literal, ou seja, se acedermos a ela através de uma variável:

```
>>> txt = 'ABC'
>>> txt + 'DEF'
'ABCDEF'
>>> txt 'DEF'
File "<stdin>", line 1
    txt 'DEF'
            ^
SyntaxError: invalid syntax
```

26. Outro operador que também está disponível para strings é o operador += :

```
>>> txt = 'ABC'
>>> txt += 'DEF'
>>> txt
'ABCDEF'
```

O operador += é um operador de atribuição que acrescenta ao objecto à esquerda o conteúdo da expressão à direita. Strings, números, listas, etc., suportam este operador.
*Neste exemplo, acrescentámos a string 'DEF' à string txt e o Python constrói uma nova string com o resultado da concatenação e coloca a variável txt a referenciar essa nova string. É muito importante perceber que a string anterior não é alterada - as strings são **imutáveis** em Python -, mas que uma nova é criada. Por exemplo, quando trabalharmos com listas vamos verificar que o comportamento não é o mesmo: aí o operador += altera a lista existente e não cria uma nova.*
*O += também pode ser utilizado com **int**s, **float**s, etc.:*

```
>>> x = 12
>>> x += 1
>>> x
13
```

*Além do +=, também temos -=, *=, /=, //=, %= e outras variações. Ao contrário das linguagens derivadas de C, Python não disponibiliza os operadores ++ e --.*

27. A string original não muda por acção do += :

```
>>> txt1 = 'ABC'
>>> txt2 = txt1    # txt2 referencia o mesmo objecto que txt1
>>> txt2 += 'DEF'  # mas agora já não... uma nova string foi criada
>>> txt1, txt2
('ABC', 'ABCDEF')
```

28. Se pretendermos saber quantos caracteres tem uma string, então acedemos à função **len** que é uma *built-in* comum a todas as sequências.

```
>>> len('Alberto')
7
>>> nome = "Alberto"
>>> len(nome)
7
>>> nome = input("Qual o seu nome? ")
Qual o seu nome? Armando
>>> len(nome)
7
```

*A função **input** aguarda que o utilizador introduza texto a partir de um "canal" de informação associado ao teclado. Opcionalmente podemos indicar uma mensagem (que foi o que fizemos neste exemplo).*
*O tal canal de informação associado ao teclado é designado de **entrada padrão** (**standard input**). Por seu turno, a informação exibida com print é enviada por um canal associado ao ecrã designado de **saída padrão** (**standard output**).*
*Esta função devolve sempre texto. Se pretendermos obter um valor como **int** ou **float**, então temos que o converter o texto introduzido, utilizando as funções **int** ou **float**. Na secção de "Tópicos variados..." damos outras alternativas*

29. Podemos indexar strings com o operador de indexação ([]) com índices positivos e negativos:

```
>>> nome = "António"
>>> nome[0], nome[1]
('A', 'n')
>>> nome[6], nome[len(nome) - 1]
('o', 'o')
>>> nome[-1], nome[-2], nome[-3], nome[-4], nome[-5], nome[-6], nome[-7]
('o', 'i', 'n', 'ó', 't', 'n', 'A')

>>> nome[7]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> nome[-8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

30. Como as strings são imutáveis, é um erro tentar modificar um caractere da string:

```
>>> nome[0] = 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

31. As strings (e outros tipos de sequência) podem ser fatiadas (*slicing*) com o operador de indexação:

```
>>> txt = 'ABCDEF'
>>> txt[0:2]  # caracts. 0 e 1
'AB'
>>> txt[2:4]  # caracts. 2 e 3
'CD'
>>> txt[0:len(txt)]  # caracts. 0 a 6 (exclusive, ou seja, 0 a 5)
'ABCDEF'
>>> txt[0:]   # por omissão 2o valor é len(txt)
'ABCDEF'
>>> txt[:2]   # por omissão 1o valor é 0
'ABCDEF'
>>> txt[:]    # uma cópia de txt de forma sucinta
'ABCDEF'
>>> txt[1:]   # todos os caracts. menos o primeiro
'BCDEF'
>>> txt[10:]  # fora do índice, devolve '' (e não dá erro)
''
```

> ***Fatias*** *(**slices**) são subsequências da sequência original. Por exemplo, dada a string* `xyz` *,*
>
> `xyz[inicio:fim]` *-> nova string contendo os caracteres de* `xyz[inicio]` *a* `xyz[fim-1]`*.*
>
> *Ou seja todos caracteres cujos índices estejam no intervalo aberto* `[inicio, fim)`*. Obtemos uma cópia da string fazendo:* `xyz[0:len(xyz)]`*. Por omissão,* `inicio` *tem o valor* `0` *e* `fim` *o valor* `len(...)`*. Deste modo,* `xyz[:]` *também devolve uma cópia da string.*
>
> *Para ajudar a visualizar o fatiamento pode ser útil olhar para os índices dos caracteres como estando entre os caracteres:*
>
> ```
>  +---+---+---+---+---+---+
>  | P | y | t | h | o | n |
>  +---+---+---+---+---+---+
>  0   1   2   3   4   5   6
> -6  -5  -4  -3  -2  -1
> ```
>
> *Note-se que tudo o que aprendemos sobre slicing aplica-se aos outros tipos de sequência que vamos dar (listas, tuplos, etc.)*

32. "Fatiamento" também funciona com índices negativos. A lógica é a mesma:

```
>>> txt[-6:-4]  # caracts. -6 e -5 (ou 0 e 1)
'AB'
>>> txt[-4:-2]  # caracts. -4 e -2 (ou 2 e 3)
'CD'
>>> txt[-6:]    # caracts. -6 até ao fim
'ABCDEF'
>>> txt[-6:-1]  # caracts. -6 até -1 (exclusive, ou seja -6 a -2 ou 0 a 4)
'ABCDE'
>>> txt[-6:2]   # caracts. -6 e 1 (isto é, 0 e 1)
'AB'
```

33. 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
>>> # split: divide nome "à volta" de
>>> # uma string (neste caso, de 'm')
>>> nome.split('m')
['ar', 'ando']
>>> nome.split('a')
['', 'rm', 'ndo']
```

*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.*

34. 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). 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

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

*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

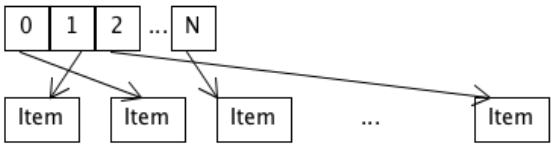
Listas

35. 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.*

36. 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']
>>> nomes2
['Alberta', 'Antónia', 'Armanda', 'Arnalda', 'Andreia']

>>> # 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']
```

Dicionários

37. 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')
>>> # 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)
```

Consultar estas e outras operações em:
https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

*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**.*
*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 {}.

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

Modo Interactivo e *Scripts*

38. 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.

39. 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)
```

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

```
$ python3 ola.py
```

41. 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
```

*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.

42. 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.

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

```
$ ola.py
```

Controlo da Execução: Breve Introdução

44. 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

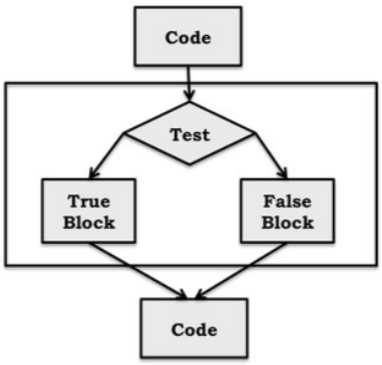
Crie o ficheiro `lista_pasta.py`:

```
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", conteudo, "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 **print**s) 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.*

45. 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.).

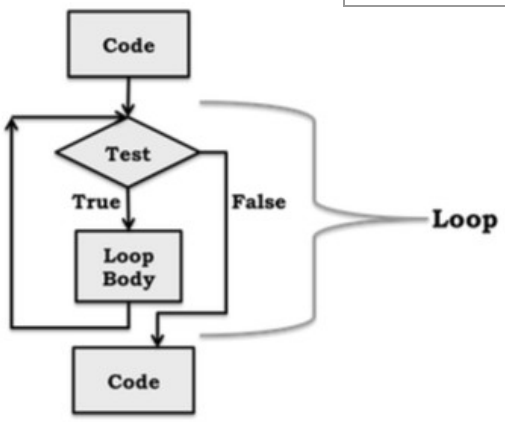
*Podemos utilizar o **if** para tomar mais do que duas decisões. Neste caso utilizamos o formato*
***if** `cond1`: … **elif** `cond2`: … **elif** `condN`: … **else**: …*
*Note-se que o bloco **else** não é obrigatório. Cada uma das **cláusulas elifs** é designada por **alternativa condicional**, ao passo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

46. 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: "))
i = soma = 0
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`***

47. 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_pasta.py` e acrescente:

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.

```
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
      # (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

48. 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 comIVA(valor, taxa_iva):
  return (valor * (100 + taxa_iva)) / 100

valor = float(input("Indique um montante: "))
print("{:.2F}".format(comIVA(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 nome(param1, ... , paramN):
    inst1
    ...
    instN
```

***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*

```
nome(arg1, ..., argN)
```

***Argumentos** são os valores dos parâmetros, e devem ser indicados entre parênteses.*

Tópicos Variados...

49. 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
```

50. 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)
```

51. 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 d e **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'>)
```

52. 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:]
```

```
>>> 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)
```

```
>>> str1, str2
('abc', 'abc')
>>> str1 is str2, str1 == str2
(False, True)
>>> id(str1), id(str2)
(4300399536, 4302751648)
```

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

53. E aqui ficam todas as palavras-reservadas da linguagem Python:

```
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. O que é um "identificador"?

2. Quais os tipos de dados primitivos do Python?

3. Indique o que fazem os seguintes operadores: `%`, `-= , /=`

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

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

5.1
```
b = (2 > 3)
```

5.2
```
x = 3
x, y = x + 1, x + 1
```

5.3
```
x , y, z = 2 is 2, [2] is [2], 2 == 2
```

5.4
```
c = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[10]
```

5.5
```
ci = ord("ABCDEFGHIJKLMNOPQRSTUVWXYZ"[-16])
```

5.6
```
d = (2.5 * (4.0 ** (11 % 4))) + ord('a')
```

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

6.1 `vals[2]` = ____

6.2 `vals[2:]` = ____

6.3 `vals[-len(vals)]` = ____

6.4 `vals[-len(vals) + 1]` = ____

6.5 `vals[2:-1]` = ____

6.6
```
nums = vals[1:4]
nums[1:3] = [4, 5]
nums = _____    vals = ____
```

7. Quais das seguintes condições são sempre verdadeiras?

7.1 `(x < y) or (x >= y)`

7.2 `(x == y) and (x != y)`

7.3 `not ((x == y) and (x != y))`

7.4 `(x <= 1) and (x >= 1)`

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

`x = 2 + "2"` `c = 3: 2`	

`x = 19.0`
`#...`
`print("Valor "  x,`
`  "Dobro "  2*x "\n")`	

`#...`
`def f(y)`
`  "{1:.2f}".format(y)`
`#...`
`x = 2.9`
`print(f[x])`	

`Import Decimal`
`def main():`
`  x = 10`
`  x++`
`  y = x + '1';`
`  return y`	

`num = input("Num? ")`
`if num = 19`
`  print("Introduziu dezanove")`	

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

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

Para cada instrução (`str.`) `format` 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`**.

`"{} ".format(y)`																			
`"{0:f} {0:.1f}".format(y, y+1)`																			

`"{:5}\n".format(x)`
`"{1:^5}{0:<5}".format(x, x*2)`

`"->({0:4.2f}+{1:.2f})/3=\n"`
`"{2:.2f}".format(x, y, (x+y)/3)`

`"X:{0:<5}\nY:{1:<5.2f}\n".`
`format(x, y)`

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

`x, y = 2, 3`
`print("XY -> " + str(x) + str(y))`
`"X+Y -> {0}".format(x+y)`
`x *= 6; y *= 2`
`print("X/Y -> ", x/y)`	

`p = 2.3`
`print("{:f}".format(p))`
`print(p*2,'\n')`
`print("{0}{1}".format(p*2, "\n"[0]))`	

`v1 = [0]*4`
`i = 0`
`v1[i] = 7; i+=1; v1[i] = 14; v1[len(v1) - 1] = 15`
`print(v1[i]*v1[i+1] + v1[1])`	

EXERCÍCIOS DE PROGRAMAÇÃO

11. Faça 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`**.

12. Um grupo de pessoas participou num jantar em que todos encomendaram o menu turístico e pretende fazer um programa para calcular a conta. Para tal, o programa deve começar por ler o número de pessoas envolvidas no jantar e, de seguida, calcular o valor da conta. O menu custa 15,00 € + IVA por pessoa. Assuma que o IVA é 23% e a gorjeta para o empregado é de 10% sobre o montante total com IVA. O programa deve exibir a despesa total sem IVA e sem gorjeta, o montante de IVA, o valor da gorjeta e a despesa total final.

13. 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.

14. Descubra como exibir os valores na unidade correcta (ie, a exibir euros).

15. Fazer um programa para calcular a contribuição para Segurança Social, IRS e o sindicato a partir do salário bruto, que é um atributo de entrada.

- SS – 11,5%
- IRS - 25%
- Sindicato – 0,5 %

O programa deve imprimir o valor das contribuições e o valor do salário líquido.

16. Faça um programa para converter euros para dólares. Pesquise o câmbio actual ou utilize o seguinte: €1 ▫ $1,39. Exemplo:

```
Montante em euros: 1250
Dolares -> 1737.5
```

17. Acrescente a conversão inversa ao programa anterior. 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
```

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

19. Vamos agora fazer um programa que lê três valores introduzidos na linha de comandos e indica o maior deles.

20. Pretendemos um programa que solicita um número ao utilizador e indica se esse número é primo (relembrando, um número é primo apenas se for divisível por 1 ou por ele próprio).

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 dois argumentos, duas datas, e indica o número de dias entre estas datas.