GUIA DE LABORATÓRIO 1.1
ELEMENTOS BÁSICOS DA LINGUAGEM - PARTE I (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 cientifica: 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
```

*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).*

```
>>> 3 > 22 or 10 != 8
True
>>> 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 **floats** 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: '3.8'
>>> 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
```

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
```

*Como foi referido numa caixa mais acima, **`float`**s não são reais. São uma representação eficiente mas inexacta. Em situaões 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.*

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 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
```

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
```

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
```

*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")
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.*

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
'AB'
>>> 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'
```

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. Com que valores ficam as variáveis nas seguintes atribuições:

4.1 `b = (2 > 3)`

4.2
```
x = 3
x, y = x + 1, x + 1
```

4.3 `c = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[10]`

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

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

4.6 `ck = 'U42'[2 != 2 + 1]`

5. Considerando que inicialmente `nome = "FERNANDO MANUEL"`, indique os valores das seguintes expressões:

5.1 `nome[3]` = ____

5.2 `nome[3:6]` = ____

5.3 `nome[3:]` = ____

5.4 `nome[-3]` = ____

5.5 `nome[-3:]` = ____

5.6 `nome[0:-3]` = ____

5.7 `nome[:-3]` = ____

5.8 `nome[-5:-3]` = ____

5.9 `nome[len(nome)-1]` = ____

5.10 `nome[-1]` = ____

5.11 `nome[-len(nome)]` = ____

5.12 `nome[-len(nome) + 2]` = ____

5.13 `nome[-10:10]` = ____

6. Quais das seguintes condições são sempre verdadeiras?

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

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

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

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

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

`x = 2 + "2"` `y = "abc".len()`	

`x = 19.0`
`#...`
`print("Valor de X:"  x, "Dobro X:"  2*x ]`	

`x = 2.9` `print(str[x] + ' 19')`	

`Import Decimal`
`x = 10`
`1+=x`
`y = x + '1'`
`input(y)`	

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

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

`x = 'Alberto'`
`print(x[0], x[3], x[-1], sep='/', end='$')`
`print(x[2], x[4])`	

`x, y = 3, 4`
`x, y = y + 1, x + 1`
`z = 2**y + x`
`print(x, y, z)`	

EXERCÍCIOS DE PROGRAMAÇÃO

9. 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.

10. Fazer um programa para calcular a contribuição para Segurança Social, IRS e o sindicato a partir do salário bruto, que ° 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.

11. Desenvolva um programa a solicitar a entrada de horas, minutos e segundos, calculando depois o tempo total em segundos.

12. Faça um programa que produza na saída padrão uma carta formatada semelhante ª indicada no exemplo que se segue. Deverá solicitar a introdução da informação em sublinhado, e dos restantes dados que achar necessários. A primeira linha e as últimas duas ("Caro Alberto/Armanda", "O seu," e "Arnaldo Antunes") deverão ser indentadas com 10 caracteres. A primeira linha de cada parágrafo deverá estar afastada 4 caracteres da margem esquerda. A linha a tracejado deverá conter 20 traços e estar indentada com 7 caracteres. Deverá dar as linhas de espaço indicadas no exemplo:

Caro Alberto/Armanda,

Venho por este meio convidá-lo/la para a cerimónia a realizar pelas 16h00 do dia 31/05/2036. Caro Alberto/Armanda, o código de vestimento ° formal, o que significa que deverá usar um fato com gravata/vestido e saltos altos.

O dia 31/05/2036 ° uma data muito especial para mim e contamos com a sua presença. O convite ° extensível ª sua companheira/companheiro.

Aguardamos a sua confirmação

O seu,

Arnaldo Antunes