(Introdução à) Programação

Vitor Beires Nogueira (vbn@uevora.pt)

Dezembro de 2020

Introdução

Este documento/*notebook* surgiu como resposta à necessidade tornar mais interativo o ensino/aprendizagem da componente teórica da unidade curricular **(Introdução à) Programação** da Universidade de Évora.

Nesse contexto, foi feita uma mudança de slides expositivos estáticos para conteúdos baseados em Jupyter lab/noteboook. Podemos descrever *Jupyter Notebook* como *web application* que permite criar e partilhar documentos que contém código dinâmico, equações, componentes gráficos e texto narrativo.

Estes *notebooks* podem ser utilizados *localmente* ou na *cloud*. Tendo em conta que as contas da Universidade de Évora tem acesso ao *Google Workspace* uma das opções na *cloud* que se destaca é a Google Colab. De modo a que seja possível usufruir em pleno destes *notebooks* é conveniente recorrer a um destes sistemas. De qualquer modo, e como complemento, também é disponibilizado em conjunto um PDF. Por último, todos os documentos deste projecto estão disponíveis no

Configuração

1. Tendo em conta a especificidade de lidar com ficheiros e para simplificar, o *notebook* denominado "Ficheiros: Input e Output" assume uma utilização local e num sistema Linux.

2. Para ter um comportamento mais interactivo sempre que executamos várias instruções consecutivas (e sem ter recorrer à função *print*) iremos utilizar a configuração abaixo. Importa ressalvar que a mesma poderá não resultar em todos os **ambientes**

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Variável, expressão e instrução

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Valores: elementos básicos

- Valor
  - Quantidade numérica
    - 1
    - 3.72 -20
  - Texto
    - "Hello World"
    - 'Hello World'
- Os valores têm um **Tipo**
  - Numérico inteiro: `int`
  - Numérico real com precisão simples: `float`
  - String (sequência de caracteres / texto): `str`
    - as strings podem ser delimitadas por "Aspas" ou 'Plicas'

In [2]:
```
3.73  + 2 * 2
```

Out[2]: 7.73

In [3]:
```
"Hello"
```

Out[3]: 'Hello'

In [4]:
```
'World'
```

Out[4]: 'World'

Tipo dos valores

- Função built-in `type()`: devolve o tipo de um valor ou de uma variável

In [5]:
```
type('exemplo')
```

Out[5]: str

In [6]:
```
type("12")
```

Out[6]: str

In [7]:
```
type(7.13)
```

Out[7]: float

Qual será o resultado de `type("12")` ?

Variável

- Variável: um nome que representa um valor
  - Mecanismo que permite múltiplas referências ao valor ao longo do programa
- Instrução de afetação `=`
  - Associa um valor (à direita) à variável (à esquerda)
- Exemplos:

In [8]:
```
x = 5 + 1
x = x + 1
x = x + 2
y = 2 * x
y
```

Out[8]: 18

Afetação

- Uma variável pode receber várias afetações,
  - inclusivamente com tipos diferentes da primeira afetação
    - mas convém evitar, para facilitar a análise do programa e minimizar a probabilidade de erro
- Exemplo no PythonTutor: https://goo.gl/czzExw

In [9]:
```
x = 3 + 1
x = x * 2
x
```

Out[9]: 8

Conteúdo da Variável

- O conteúdo de uma variável pode mostrar-se com a função `print()`
- Para consultar o tipo de uma variável existe a função `type()`
- Exemplo PythonTutor: https://goo.gl/wSw9sn

In [10]:
```
nome = 'Maria Albertina'
type(nome)
```

Out[10]: str

In [11]:
```
pi = 3.14159265359
print(pi)
type(pi)
```

```
3.14159265359
```

Out[11]: float

Forma geral para afetação

- Uso comum:

$$< variável > = < exp1 >$$

- Afetação múltipla

$$< variável1 >, < variável2 >, \ldots, < variávelN > = < exp1 >, < exp2 >, \ldots, < expN >$$

- Exemplo PythonTutor: https://goo.gl/Ni5ejx

In [12]:
```
a, b, c = 1, 2, 3
print(a)
print(b)
print(c)
```

```
1
2
3
```

In [13]:
```
a, b = 1, 2
a, b = b, a
print(a)
print(b)
```

```
2
1
```

Nomes

- Regra para o nome das variáveis
  - iniciar com letra (minúscula), seguida de outras letras, algarismos ou underscore _
- Nome inválido gera erro de sintaxe. Nomes são inválidos se:
  - iniciar com algarismo; incluir símbolos, coincidir com palavras reservadas da linguagem
- Nomes devem ser sugestivos
  - Facilitam a leitura; ajudam o programador a entender o código
  - Exemplos
    - comprimento, largura

In [14]:
```
'''invalido
1pi = 3
'''
```

Out[14]: 'invalido\n1pi = 3\n'

In [15]:
```
""" Error
1*pi = 1 * 3.14
"""
```

Out[15]: ' Error \n1*pi = 1 * 3.14\n'

Instrução (statement)

- Elemento no código fonte que o interpretador executa
  - Exemplos
    - Afetação: `x = 5`
    - Outros: `pass, return, raise, continue, break`
- Um script é um conjunto de instruções, num ficheiro .py
  - O modo interativo mostra o resultado após cada instrução
  - O mesmo não acontece em modo script

Operadores

- Aritmética simples
  - `+, -, /, *`
  - `%` - resto da divisão inteira
  - `//` - divisão inteira
- Comparação
  - `==, !=, <, <=, >, >=`
- Outros
  - Exponencial: `**`
  - XOR: `^`

In [16]:
```
7 ** 2 == 20
```

Out[16]: True

Expressões e Precedência

- Expressão: combinação de valores, operadores
  - Inseridas em modo interativo, são avaliadas pelo interpretador e o resultado é mostrado
- Precedência: regras para ordem de avaliação numa expressão composta
  - Parêntesis: prioridade máxima
    - `2 * (3-1) == 4`
  - Depois `**`
    - `(3 * 1 ** 3 == 3)`
  - Depois / e *, com prioridade maior que + e -
    - `6 + 4 / 2 == 8.0`
  - Entre operadores de igual prioridade
    - Avaliação da esquerda para a direita

In [17]:
```
a, b = 1, 2
a == b
```

Out[17]: False

In [18]:
```
a, b, c = 1, 2, 3
d = (a**2 + c)
print(d // b)
print(d % b)
```

```
3
1
```

Comentários

- São anotações, em linguagem natural, para ajudar a entender o código fonte.
  - comentário multi-linha

    ```
    '''the percentage of the hour that
    has elapsed'''
    percentage= (min * 100) / 60
    ```

  - comentário na própria linha

    ```
    percentage= (min*100) / 60 # percentage of an hour
    ```

- usar nomes sugestivos para variáveis reduz a necessidade de comentários
- `"# xxxx"` isto não é um comentário, é uma string... pois está entre " (ou ')

In [19]:
```
minimo = 0.4
# the percentage of the hour that has elapsed
percentagem = (minimo * 100) / 60
print(percentagem)
```

```
0.6666666666666666
```

Debugging

- Ver o apêndice A do livro
- Nota: o nome das variáveis deve ser escrito sempre da mesma forma. Cuidado com as maiúsculas:

In [20]:
```
pesoLiquido = 250
'''Error
valor = pesoliquido*100
'''
```

Out[20]: 'Error\nvalor = pesoliquido*100\n'

Instrução print

- Forma geral: `print( <expressão> )`
- Como é executada:
  - A é avaliada. O valor resultante (convertido para texto, se necessário) é escrito no output, seguido de uma mudança de linha (`"\n"`)
- Se não for indicada, apenas será escrito `"\n"`

In [21]:
```
print(11+1)

a = 12

print(a)

print(a+1)

print("o dobro de ")
print(a)
print("é")
print(a*2)

print("o dobro de {0} = {1}".format(a, 2*a))

print("a média entre {0} e {1} é {2}".format(a, 15, (a+15)/2))
```

```
12
12
13
o dobro de
12
é
24
o dobro de 12 = 24
a média entre 12 e 15 é 13.5
```

Tipos Numéricos e Booleanos, Passagem de Parâmetros e Conversão de Tipos

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Tipos numéricos: int

- Podem ser positivos ou negativos
- Em Python3, `int` não tem 'limite'

In [2]:
```
long_number = 123456789101112131415161718920
long_number + 1
```

Out[2]:
```
123456789101112131415161718921
```

Tipos numéricos: float

- valores numéricos com parte decimal
- a melhor aproximação para os reais
- Sintaxe
  - Simples (com separador decimal): 0.000578
  - Notação em forma exponencial ou científica: 5.78e-4 (== 0.000578)
  - Outros: 2 ** (-4) (== 0.0625)

In [3]:
```
5.78e-4 == 0.000578
```

Tipos numéricos: complex

- Tipo que representa números complexos, com parte real e parte imaginária (sufixo *J* ou *j*)

In [3]:
```
z = complex(2, 3)

print("número complexo {0} com parte real {1} e imaginária {2}".format(z, z.real, z.imag))

número complexo (2+3j) com parte real 2.0 e imaginária 3.0
```

Booleanos

- Representam um valor de verdade
  - Constantes: `True` ou `False`
- Operações booleanas: sem avaliar o outro operando (short-circuit operator)
  - `x or y`: se x é False devolve y, caso contrário devolve x
  - `x and y`: se x é False devolve x, caso contrário devolve y
  - `not x`: se x é False devolve True, caso contrário devolve False
  - Estes operadores têm prioridade inferior aos operadores não booleanos

In [4]:
```
x = not 1 == 4

y = (True or False) and True

print("Valor de x {0} e de y {1}".format(x, y))

Valor de x True e de y True
```

In [5]:
```
print('A' < 'a')

True
```

In [6]:
```
print(1==2)

print(1 < 2)

print('a' == 'A')

print('A' < 'a')

print(1== 1 and True)

a, b, c = 1, 2, 3

print(a == 1 and b < c)

print( not b == 1)

print(True or False and True)

False
True
False
True
True
True
True
True
```

Operadores sobre booleanos aceitam outros tipos

- Expressões cujo valor de verdade é equivalente a False:
  - 0 (em qualquer tipo numérico)
  - Sequência vazia: `''`, `()`, `[]`
  - ainda outro caso muito especial, não considerado
- Outras expressões serão interpretadas como True

In [7]:
```
not(1)
```

Out[7]:
```
False
```

In [8]:
```
not(0)
```

Out[8]:
```
True
```

In [9]:
```
x = 1
b = True
print(b and x)
print(b and x or 0)

1
1
```

Booleanos: muito úteis para

- Resultado de uma comparação ou teste
  - Igualdade?
    - `idade == 18` # dará True ou False
  - Desigualdade
    - `teste = altura < 1.80` # teste recebe True ou False
- Expressar um conjunto de características
  - Conjunções ou disjunções ou outras combinações
  - Exemplo: "ser um inteiro par e inferior a 10"

In [10]:
```
x = 8
x % 2 == int and x % 2 == 0 and x < 10 # inteiro e par e menor que 10
```

Out[10]:
```
True
```

In [11]:
```
x = 3
x % 2 != 0 and x > 5 # impar e maior que 5
```

Out[11]:
```
False
```

Conversão de tipos explícita

- float(`<expr>`) converte `<expr>` num valor de vírgula flutuante
- int(`<expr>`) converte `<expr>` num valor inteiro
- str(`<expr>`) devolve uma representação em "texto" (string) de `<expr>`
- eval(`<string>`) avalia a string como uma expressão

In [12]:
```
print(int(3.2))

print(int(3.9))

x = 2
xpto = float(x)
print(type(xpto))
print(xpto)

print(str(1 + 2 / 3))

print(eval('1 + 2 / 3'))

print(str('1 + 2 / 3'))

3
3
<class 'float'>
2.0
1.6666666666666665
1.6666666666666665
1 + 2 / 3
```

Conversão de tipos implicita

- Python permite aritmética entre operandos de tipo diferente
  - Converte automaticamente para o tipo "mais abrangente"
    - int → float
    - float → complex

In [13]:
```
x = 6

print(x / 2)

print(x / 2.5)

print(type( x / 2.5 ))

3.0
2.4
<class 'float'>
```

Parâmetros para o programa

- Como separar a parte funcional (ão cálculo) de valores específicos?
  - Preparar o programa para obter os valores ... em vez de os ter diretamente no código fonte.
    - Vantagem: podemos realizar N testes com diferentes valores sem editar o código fonte
  - Possibilidades:
    1. Passar argumentos ao executar o script
    2. Ler valores durante a execução

Argumentos para um script

- É se um script necessitar de valores de input para realizar um cálculo
  - Execução
    - `$ python file.py 'Maria' 20`
  - Código fonte
    - os argumentos estão na variável sys.argv, uma lista de strings
      - Comprimento >= 1
    - sys.argv[0] tem o nome do script (ou uma string vazia)
    - sys.argv[1] é o primeiro argumento ('Maria')
    - sys.argv[2] é o segundo argumento ('20', como string!)
- É necessário fazer `import sys`

Parâmetros: leitura de valores input

- Pedir valores ao utilizador a meio da execução do programa
  - Python 3: usar a função `input(prompt)`
    - Apresenta a prompt ao utilizador (opcional) e devolve uma string com o texto que ele insere

In [14]:
```
nome = input('Indica o seu nome: ')
print(nome)
print(type(nome))

Nogueira
<class 'str'>
```

Parâmetros: leitura de valores eval

- Ler um valor de outro tipo (inteiro, real, booleano) – Python 3
  1. Podemos avaliar uma string lida com a função anterior `eval(input(prompt))`
  2. Ou convertemos explicitamente o retorno de `input(prompt)`

In [15]:
```
idade = int(input("idade: "))
type(idade)
```

Out[15]:
```
int
```

In [16]:
```
idade = eval(input('idade: '))
print(idade)

100
```

In [17]:
```
ser_casado = bool(input('é casado? True/False'))
type(ser_casado)
```

Out[17]:
```
bool
```

Obtenção de Parâmetros:

Qual a melhor forma? Argumentos do script ou leitura?

- Depende do contexto do programa
  - Leitura com input
    - O programa pára e espera que o utilizador [*] insira valores, retomando a execução após o enter
    - É aceitável se o script é usado por uma pessoa
    - Mas e se o script é usado automaticamente por outros programas?
      - Seria mais conveniente passar os valores para a execução... como argumentos do script

[*] Há um procedimento especial para evitar essa espera, relacionado com uma opção avançada de redirecionamento do input.

Operações elementares sobre strings

- `+` : concatenação / junção de duas strings
- `*` : repetição da string

Que propriedade tem a adição (`+`) de números que não existe para a concatenação (`+`) de strings?

In [18]:
```
a = 'um' + 'Teste'; print(a)
b = 'foo'*'3 '; print(b)
c = a + b; print(c)
d = b + a; print(d)

umTeste
foofoofoo
umTesteFoofoofoo
FoofoofooumTeste
```

Conversão de string para inteiro

Ainda a função int()

- `int([x[, base]])`
  - converte uma string para um inteiro. Por omissão usa a base=10
  - Exemplo: `idade= int( sys.argv[1] )`
    - e agora o valor já pode ser usado em cálculos

In [19]:
```
a ='101';  b = int(a); print(b)

101
```

In [20]:
```
a ='101'; b = int(a,10); print(b)

101
```

In [21]:
```
a ='101'; b = int(a, 2);# print(b)
# 1*2**2 + 0*2**1 + 1*2**0 == 5

5
```

In [22]:
```
a ='210'; b = int(a, 3); print(b)
# 2_3 2**2 + 1 * 3**1 + 0*3 ** 0

21
```

Mais algumas funções built-in

- `round()`
  - Arredondar
  - retorna float
- `abs()`
  - Valor absoluto
  - incluivalamente um float
- `max(), min()`
  - Estas funções podem usar-se com um nº variável de argumentos

In [23]:
```
round(11.3)
```

Out[23]:
```
11
```

In [24]:
```
abs(-4)
```

Out[24]:
```
4
```

In [25]:
```
max(1, 2, 3, 4)
```

Out[25]:
```
4
```

In [26]:
```
min(10, 9, 8)
```

Out[26]:
```
8
```

In [27]:
```
max(1, 2, min(10, 9, 8), round(11.3))
```

Out[27]:
```
11
```

Nota: a função `round` tem algumas especificidades. Para mais informação consulta por exemplo Python Docs

Estruturas de controlo do fluxo de execução: condicionais

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```python
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Introdução ao controlo de fluxo

- Execução do Programa
  - fluxo sequencial
- Exemplos simples como programas para:
  - Imprimir a hora
  - Dados dois valores, mostrar a soma
- Há problemas cuja solução pode variar em função das circunstâncias ...
  - execução sequencial não basta

Decisão

- Avaliar as circunstâncias e tomar uma decisão
  - Optar por uma das possíveis soluções
    - ramo numa árvore de decisão
- Exemplo: programa para dizer se um aluno obteve aprovação a uma disciplina
  - Obter notas nos elementos de avaliação
  - Aplicar a fórmula para calcular a nota final arredondada
  - Decisão: avaliar "a nota é suficiente para aprovar"?
    - sim
      - mostrar "Aluno aprovado à disciplina."
    - não
      - mostrar "Aluno reprovado à disciplina."

Condicional: `if`

Instrução if

```
# executa a instrução se e só a condição for True
if <condição>:
    <instrução>
```

In [2]:
```python
x = eval(input('insira um valor: '))
print(' ... posição 1')
if x > 0:
    print('valor positivo')
print(' ... posição 2')
```

```
 ... posição 1
valor positivo
 ... posição 2
```

Condicional com alternativa: `if-else`

Instrução if-else

```
if <condição>:
    <instrução 1>
else:
    <instrução 2>
```

In [3]:
```python
x = eval(input('insira um valor: '))
print(' ... posição 1')
if x > 0:
    print('valor positivo')
else:
    print('valor negativo ou zero')
print(' ... posição 2')
```

```
 ... posição 1
valor negativo ou zero
 ... posição 2
```

Espaços em branco no código

- É possível ter linhas em branco (ou espaços à direita)
- Espaços no início das linhas podem gerar erros de sintaxe
- Espaços à esquerda das instruções:
  - indentação
  - significado especial

Blocos e indentação

- Bloco
  - Sequência de instruções
  - devem ter a mesma indentação
    - Avanço para a direita
    - Unidades de avanço:
      - tab
      - (2 ou 4) espaços
  - corpo do if-else (ou nos ciclos)
    - 1 ou mais instruções

```
<instrução 1>
if <condição>:
    <instrução 2>
    <instrução 3>
    <instrução 4> # última instrução do bloco
<instrução 5>
```

In [4]:
```python
## Exemplo com blocos
nota = eval(input('Insira a nota: '))
if nota >= 10:
    print('O aluno aprovou à disciplina.')
    print('Parabéns!')
else:
    print('O aluno reprovou à disciplina.')
    print('Ver datas de exame de recurso.')
print('--------------') # fora do bloco
```

```
O aluno aprovou à disciplina.
Parabéns!
--------------
```

Condicional encadeados: `elif`

múltiplos testes em sequência

```
if <condição 1>:
    <bloco 1>
elif <condição 2> :
    <bloco 2>
elif <condição 3> :
    <bloco 3>
...
else:
    <bloco alternativo>
```

Mais de duas possibilidades de decisão elif

- Pode haver várias sequências elif
- Em todo o if, apenas um ramo será executado (se algum for)
  - o ramo do primeiro teste a dar True
- O else é opcional. Se presente, vem no final da sequência
  - é a opção de último recurso
  - sem else, pode acontecer que nenhum ramo seja executado

In [5]:
```python
x, y = eval(input('Insira x e y:'))
if x < y:
    print('x é menor que y')
elif x > y:
    print('x é maior que y')
else:
    print('x é igual a y')
```

```
x é menor que y
```

Condicionais encaixados (nested)

Condicional pode surgir no corpo do ramo de outro condicional

```
if <condição 1>:
    if <condição 2> :
        <bloco 1>
    else :
        <bloco 2>
...
else:
    <bloco alternativo>
```

In [6]:
```python
x, y = eval(input('Insira x e y:'))

if x == y:
    print('x é igual a y')
else:
    if x < y:
        print('x é menor que y')
    else:
        print('x é maior que y')
```

```
x é maior que y
```

Condicionais encaixados (nested)

Por vezes podem reescrever-se com operador `and`

```
if <condição 1>:
    if <condição 2> :
        <bloco 1>
    else :
        <bloco 2>
...
else:
    <bloco alternativo>
```

---

por

```
if <condição 1> and <condição 2> :
    <bloco 1>
elif <condição 1>:
        <bloco 2>
...
else:
    <bloco alternativo>
```

In [7]:
```python
x = eval(input())
if 0 < x:
     if x < 10:
           print('x é positivo < 10')
print('continuação')
```

```
x é positivo < 10
continuação
```

In [8]:
```python
x = eval(input())
if 0 < x and x < 10:
    print('x é positivo < 10')
print('continuação')
```

```
x é positivo < 10
continuação
```

Instrução pass

- Instrução `pass`: não fazer nada
- Durante o desenvolvimento
  - No corpo de uma instrução composta
    - motivo: validação de sintaxe para executar uma implementação parcial
- No código final não devem existir instruções `pass`

In [9]:
```python
x = eval(input())
if x < 0:
     pass # caso negativo: POR IMPLEMENTAR
else:
    print('resolvido')
```

Condicionais na prática ...

Programa para mostrar o maior de 3 valores `python a, b, c = eval(input('insira três inteiros separados por vírgula: '))

comparações

...

imprimir o valor pretendido

print("O maior valor é: {}".format(maximo))

Condicionais na prática ...

- Há 3 resultados possíveis: a, b ou c
  - Abordagem:
  - Se a escolha acertada é a
    - maximo = a
  - Se a escolha acertada é b
    - maximo = b
  - Se a escolha acertada é c
    - maximo = c
- E o que é ser a escolha acertada? Ser maior (ou igual) aos outros valores

Condicionais na prática ...

- Como descrever "a é a escolha acertada"?
  - a,b,c estão por ordem decrescente ou a,c,b estão por ordem decrescente
- Como expressar "a é o maior"?
  - operador `and`
    - `a >= b and a >= c`

Condicionais na prática ...

- Hipótese:

  ```python
  if a >= b and a >= c:
      maximo = a
  elif b >= a and b >= c:
      maximo = b
  else:              # se não é a ou b, será c
      maximo = c
  ```

- Análise
  - Implementação correta
  - 2 a 4 comparações para determinar o máximo

Condicionais na prática ...

- Implementação Alternativa:

```python
if a >= b:
    if a >= c:
        maximo = a
    else:
        maximo = c
else;
    if b >= c:
        maximo = b
    else:
        maximo = c
```

- Análise
  - Implementação correta
  - 2 comparações para determinar o máximo
  - mais linhas de código... mas solução mais eficiente

Condicionais na prática ...

- Outra Alternativa: processamento sequencial
  - Assumir um deles, por exemplo `a`
    - e ver se algum dos outros o destrona

```python
maximo = a
if b > maximo:
    maximo = b
if c > maximo
    maximo = c
```

- Análise
  - Implementação correta
  - 2 comparações para determinar o máximo
  - A estrutura simples e repetitiva permite ainda outra evolução
    - ciclo facilmente aplicável a sequências de N valores

Considerações

- Há várias soluções para um problema
  - Prioridade: correção
- Analisar a solução
  - eficiência
  - legibilidade do código e simplicidade
- Generalizar ... abstrair
  - A solução deve ser abrangente, para todas as combinações possíveis de inputs

Condicionais: mais exemplos práticos

- Equação quadrática
  - Polinómio de grau 2
  - $f(x) = ax^2 + bx + c$
- Fórmula resolvente
  - Encontrar intersecções com eixo X
    - são as raizes da equação

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Condicionais: mais exemplos práticos

- Discriminante
  - Expressão abaixo da raiz: $d = b^2 - 4ac$
  - Determina as raízes
    - $d > 0$: há 2 raízes reais
    - $d == 0$: um só valor como raiz real (x, tal que $f(x) == 0$)
      - $x = \frac{-b}{2a}$
    - $d < 0$: não tem raízes reais (tem complexas)

Condicionais: mais exemplos práticos

- Aplicar diretamente a função math.sqrt() a um negativo dá erro
  - `math.sqrt( -2 )`
  - `ValueError: math domain error`
- solução com condicionais
  - avaliar o discriminante
    - operações apropriadas a cada caso:
      - só precisa calcular raiz num caso
      - cada caso tem um print apropriado

Finalização intencional do programa

Programar uma paragem na execução do

---

script

- `exit( [arg] )`
- `sys.exit( [arg] )`
  - o programa vai terminar
  - argumento arg é opcional
    - Inteiro
    - Representa o estado de finalização do programa
      - 0: normal
      - !=0 para denotar algum problema ou imprevisto
        - Exemplo: tentativa de execução sem argumentos, quando eles são obrigatórios

Estruturas de controlo do fluxo de execução: ciclos while

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Introdução

- Programa para imprimir uma contagem decrescente, de 5 a 0

```
print(5)
print(4)
print(3)
print(2)
print(1)
print(0)
```

- E se fosse deste 20?
- E se fosse deste um valor n?
  - O procedimento é idêntico!
  - Varia o nº de vezes que é repetido e o valor em cada passagem

Fluxo com Ciclos

- Execução de tarefas repetitivas
  - A tarefa programa-se uma só vez, num bloco
- Repete-se a execução do bloco
  - Tantas vezes quanto o necessário

Instrução `while`

```
while <condição>:
    <bloco>          # corpo do ciclo
```

1. A condição é avaliada (obtendo-se True ou False)
2. Se a condição é False, o corpo não é executado
   - O programa continua na instrução a seguir (com a mesma identação que o `while`)
3. Se a condição é True, o corpo é executado e o programa volta ao ponto 1, com a reavaliação da condição.
4. O corpo do ciclo deve ter efeitos numa ou mais variáveis na condição
   - Para existir **paragem** e o ciclo terminar

Instrução `while`

- Voltando à contagem decrescente
  - Tarefa: imprimir um valor
  - Repetir a tarefa com while
  - Condição para execução: não ultrapassar o valor 0

In [2]:
```
valor = 5

while valor >= 0:
    print(valor)
    valor = valor - 1
print("fora do ciclo")
print(valor)
```

```
5
4
3
2
1
0
fora do ciclo
-1
```

Outra saída do ciclo: `break`

```
while <condição>:
    <instrução 1>
    ...
    if <condição_especial>:
        break          # termina o ciclo, sai a meio
    ...
    <instrução N>
```

- Permite executar "uma iteração e meio"

Exemplo: programa para ler e contar letras

- Repete:
  - Pedir letra
  - Imprimir letra
  - Imprime o total
  - Sai se a letra é 'x'

In [4]:
```
letra = 'a'

total = 0                  # acumulador

while letra != 'x':
    letra = input('insira uma letra:')
    print(letra)
    total = total + 1      # contagem
    print('letras lidas: {}'.format(total))

print("terminou.")
```

```
a
letras lidas: 1

s
letras lidas: 2

x
letras lidas: 3
terminou.
```

Exemplo com break

- No mesmo programa, não exceder 10 letras
  - Interromper ciclo se total > 10

In [5]:
```
total = 0
letra = 'a'
while letra != 'x':
  letra= input('insira uma letra:')
  print(letra)
  total = total + 1
  if total > 10:
    break  # sai
  print('letras lidas: {}'.format(total))
print("terminou.")
```

```
a
letras lidas: 1

b
letras lidas: 2

c
letras lidas: 3

d
letras lidas: 4

e
letras lidas: 5

f
letras lidas: 6

g
letras lidas: 7

h
letras lidas: 8

i
letras lidas: 9

j
letras lidas: 10

k
terminou.
```

Saltar para a iteração seguinte: `continue`

- A meio do corpo de um ciclo
  - Ignorar o resto do bloco
  - Passar de imediato ao teste do ciclo, para nova iteração

```
while <condição>:
    <instrução 1>
    ...
    if <condição_especial>:
        continue    # passa para a avaliação da condição e iteração seguinte
    ...
    <instrução N>
```

- Se então o ciclo continua mas não é executada

Exemplo com `continue`

- Para ao mesmo programa
  - não imprimir nem contar as letras b e c

In [6]:
```
letra ='a'
total = 0
while letra != 'x':
    letra = input('insira uma letra:')
    if letra == 'b' or letra == 'c':
        continue
    print(letra)
    total = total + 1
    if total > 10:
        break  # sai
    print('letras lidas: {}'.format(total))
print("terminou.")
```

```
a
letras lidas: 1

d
letras lidas: 2

x
letras lidas: 3
terminou.
```

Ciclos: momento da saída

- Cuidado: a condição de paragem deve ser verificada
  - E comum haver uma iteração acidental a mais ou a menos
  - `break` ou `continue` : a interrupção do corpo não tem efeitos secundários?
- Qual o mais correcto: o exemplo acima ou abaixo ?

In [7]:
```
letra ='a'
total = 0
while letra != 'x':
    letra = input('insira uma letra:')
    if letra == 'b' or letra == 'c':
        continue
    print(letra)
    total = total + 1
    print('letras lidas: {}'.format(total))
    if total > 10:
        break  # sai
print("terminou.")
```

```
a
letras lidas: 1

d
letras lidas: 2

x
letras lidas: 3
terminou.
```

Cuidados a ter com os ciclos

O que há de errado com este código?

```
i = 0

while i <= 10:

    print (i)

print ('ok')
```

- Não termina!!!
- Falta uma alteração à variável i

In [8]:
```
# Calcular a média de determinado nº de valores
nval = int(input("quantos valores existem? "))
lidos = 0          # contador
total = 0          # acumulador
while lidos < nval:  # enquanto há mais valores para ler
    valor_actual = float(input("insira o valor: "))
    total = total + valor_actual
    lidos = lidos + 1
# final da parte interativa.. cálculo:
media = total/nval
print ( 'média: {}'.format(media) )
```

```
média: 12.666666666666666
```

Considerações

- Ciclo é uma instrução cujo corpo pode ter outras instruções
- O corpo de um ciclo
  - pode conter condicionais
  - pode conter outros ciclos
- Ciclos com while:
  - Atenção especial à condição de permanência no ciclo
  - Se essa condição tem uma variável
    - deve ser afetada no corpo do ciclo
- Há outras instruções para ciclos

In [9]:
```
# Exemplo: ciclo inserido em ciclo

linhas = 8   # nº de linhas
colunas = 8  # nº de colunas
l = 1
while l <= linhas:
    strLinha= ""                    # inicia com string vazia
    c =0
    while c <= colunas:             # ciclo interno
        strCelula = "{},{}) ".format(l,c)  # coordenadas
        strLinha = strLinha + strCelula  # junta na linha
        c = c + 1
    print (strLinha)  # mostrar cada linha
    l = l+1
```

```
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7)
(1,0) (1,1) (1,2) (1,3) (1,4) (1,5) (1,6) (1,7)
(2,0) (2,1) (2,2) (2,3) (2,4) (2,5) (2,6) (2,7)
(3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (3,6) (3,7)
(4,0) (4,1) (4,2) (4,3) (4,4) (4,5) (4,6) (4,7)
(5,0) (5,1) (5,2) (5,3) (5,4) (5,5) (5,6) (5,7)
(6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6) (6,7)
(7,0) (7,1) (7,2) (7,3) (7,4) (7,5) (7,6) (7,7)
```

Mais exemplos: Método de Newton para Raiz Quadrada

- Aproximação à raiz do valor a
- Estimativa x: iniciar com um valor
  - Por exemplo 3
  - Obter o valor y de acordo com a fórmula y é a nova estimativa

$$y = \frac{x + \frac{a}{x}}{2}$$

- Continuar recalculando y a partir do valor anterior
  - a cada iteração ocorre uma aproximação ao valor exato
  - quando terminar?
    - quando os valores estabilizam
      - diferença entre estimativas é desprezável (abaixo de um limiar)

In [10]:
```
# método de Newton
import math
valor = float(input("qual o valor?"))
estimativa0 = float(input("qual o valor para a 1a estimativa?"))

limiar = 0.0001 # precisão para decidir quando parar
x = estimativa0

while True:
    y = (x + valor / x ) / 2
    print('   estimativa actual %f' %(y))

    if abs(y-x) < limiar: # verificar se estabilizou
        break

    x = y # considerarmos a ultima tentativa

print('aprox a raiz: {}'.format(y))
print('math.sqrt: {}'.format(math.sqrt(valor)))
```

```
qual o valor?3
qual o valor para a 1a estimativa?1
   estimativa actual 1.500000
   estimativa actual 1.416667
   estimativa actual 1.414216
   estimativa actual 1.414214
aprox a raiz: 1.4142135623730951
math.sqrt: 1.4142135623730951
```

Funções: definição, argumentos, retorno

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Programação e repetição

- Tarefas são organizadas em blocos de código
- A repetição de execução de blocos é muito comum
  - Repetição consecutiva
    - Ciclos
      - `while`
      - `for`
  - Repetição de execução não necessariamente consecutiva
    - Blocos de código com um nome
      - **funções**

Funções: já disponíveis no Python

- Algumas funções da Linguagem Python:
  - Funções built-in
    - `abs(), eval(), input(), max(), min(), str(), type(), ...`
  - Funções de bibliotecas, como por exemplo no módulo math
    - `math.sqrt(), math.sin(), ...`
    - Módulo: ficheiro com um conjunto de funções relacionadas
  - Métodos: *Funções* aplicadas sobre uma variável objeto
    - surgem noutra secção
    - `lista.sort(), lista.insert(), ...`

In [2]:
```
import math
print(math.sin(0))
print(math.sqrt(2))
```

```
0.0
1.4142135623730951
```

Funções: motivação

- Código fonte do programa deve ser organizado
  - Evitar repetição de código ao longo do programa
    - Ganha-se legibilidade
    - Reduz-se a probabilidade de erro
      - Eventuais ajustes serão aplicados num só local
      - De outro modo seria necessário repetir o ajuste em cada instância do código redundante $\longrightarrow$ **má programação!**
    - Resultado mais modular
- Um programa executa sequências de tarefas
  - O código de cada tarefa deve ser arrumado numa função

Funções: motivação

- Vantagens de organizar cada tarefa numa função
  - Repetição
    - Uma tarefa muito comum é facilmente repetida
      - invocar a função correspondente: `nome_funcao()`. O código é escrito só uma vez, na definição de `nome_funcao()`
- Exemplo: encontrar o maior valor
  - `idadeMaior = maior_valor(idadeMaria, idadeJose)`
  - `maxAltura = maior_valor(alturaRita, alturaManuel)`
  - `maisPesado = maior_valor(pesoA, pesoB)`
- A rotina a executar é a mesma, só mudam os argumentos com que trabalha, e a partir dos quais produzirá um resultado

Funções: motivação

- Vantagens de organizar cada tarefa numa função
  - Composição
    - Uma tarefa composta é facilmente definida invocando a sequência de funções para cada tarefa constituinte
- Exemplo: contar os alunos aprovados numa disciplina pode dividir-se em 3 passos:

```
contar_alunos_aprovados(disciplina):
    alunos = obter_lista_alunos(disciplina)
    lista_notas = obter_lista_notas(alunos)
    aprovados = conta_maior_ou_igual(listaNotas, 10)
    print(“total aprovados: {}“.format(aprovados) )
```

Funções: motivação

- Vantagens de organizar cada tarefa numa função
  - Clareza
  - Maior facilidade
    - Implementação
    - Leitura
    - Manutenção ou atualização do programa

Função: o conceito

- Uma função é como um pequeno programa inserido em programas maior
  - Possui *código fonte próprio* para a execução de uma tarefa
- Uma função tem um **nome**, especificado na sua **definição**
  - Exemplo: `maximo()  # não esquecer os parênteses`
- Pode ter argumentos
- Pode retornar um valor ou não
- A função é executada sempre que o seu nome for invocado
  - Exemplo: `valor = maximo()`

Definição de Funções: introdução

- Antes de ser usada (invocada), uma função tem de estar definida
- Instrução `def` é usada para definir uma função

```
pythoin
def nome():              # cabeçalho

   <bloco de código>     # corpo da função
```

- Nome da função
  - pode ter letras, números, “_” mas deve iniciar com letra
  - convém evitar repetição de nomes já associados a variáveis
  - invocação: nome e parênteses: `nome()`
- Cabeçalho:
  - termina com :
- Corpo da função:
  - indentado para a direita, relativamente ao cabeçalho

In [3]:
```
# cumprimentar o utilizador Manuel
def cumprimentar():
    print ('Bom Dia Manuel')

print (' - inicio do programa - ')
# invocar a função para o cumprimento
cumprimentar()

# … e o programa continua
```

```
 - inicio do programa -
Bom Dia Manuel
```

Definição de Funções e o Fluxo

- Definição de função não altera o fluxo de Execução
  - Primeira passagem do interpretador:
    - A definição é considerada mas o corpo não é executado!
- O código da função é executado quando a mesa é invocada
  - `cumprimentar()`

Argumentos de Funções

- A tarefa realizada por uma função pode necessitar de parâmetros
  - `math.sqrt()` requer um valor para o qual irá calcular a raiz quadrada
- Para indicar que a função a definir tem argumentos
  - Colocamos o nome dos argumentos entre os parênteses

```
pythoin
def nome(arg1, arg2, ..., argn):              # cabeçalho

   <bloco de código>     # corpo da função
```

In [4]:

```python
# cumprimentar o utilizador
def cumprimentar(nome):
  print ("Bom Dia {}".format(nome))

print (" - inicio do programa - ")
# invocar a funcao, com um valor no argumento

cumprimentar("Maria")

# … e o programa continua
```

```
 - inicio do programa -
Bom Dia Maria
```

Funções com e sem retorno

- A função anterior não tem retorno explícito
  - Tal como o método `sort()`, nas listas, não devolve um valor
  - Devolvem `None` (constante que representa ausência de valor)
- Algumas funções têm retorno
  - Como `sqrt(), abs()`
- Para determinar o valor de resultado a devolver
  - Instrução `return`
    - A execução da função termina
      - É devolvido o valor indicado à direita de `return`
      - O programa continua após a invocação da função

In [5]:

```python
import math

def quadrado(valor):
    ''' Função para obter o quadrado do valor '''
    return valor*valor

a = 5

print(quadrado(a))

cateto1 = 4
cateto2 = 7

hipotenusa = math.sqrt(quadrado(cateto1) + quadrado(cateto2))
print(hipotenusa)


help(quadrado)
```

```
25
8.06225774829855
Help on function quadrado in module __main__:

quadrado(valor)
    Função para obter o quadrado do valor
```

Funções com retorno

- O bloco pode ter *várias* opções para retornar o valor
  - o primeiro a ser executado determina o valor a retornar
  - código da função, após a execução de `return`, já não será executado

In [6]:
```
def valor_absoluto(x):
    if x >= 0:
        return x
    else:
        return -x
    # se existisse código aqui seria código morto
```

Funções e invocação

O corpo de uma função pode invocar funções

In [7]:
```
def quadrado(valor):
  return valor * valor

def cubo(valor):
  return valor * quadrado(valor)

print(cubo(3))
```

```
27
```

In [8]:
```
# funcao que mostra um verso da canção
def happy():
    print ("Happy birthday to you!")

def sing(person):
    happy()
    happy()
    print ("Happy birthday, dear {}!".format(person) )
    happy()

sing("Fred")
```

```
Happy birthday to you!
Happy birthday to you!
Happy birthday, dear Fred!
Happy birthday to you!
```

Avaliação de Expressões com invocação encadeada de funções

- Avaliar uma expressão com invocação de função:
  - Avaliar a expressão em cada argumento
  - Executar o código da função
  - Apurar o valor retornado (se existir)
- `f(g(1 + 2, h(3)), 4)`
  - O que envolve avaliar `g(1 + 2, h(3))`
    - Avaliar `1+2` # 1º argumento
    - Avaliar `h(3)` # 2º argumento
      - Executar a função h com argumento 3 e apurar o resultado
    - Apurar resultado (x) de executar g com aqueles argumentos
    - Apurar resultado de f, executada com os argumentos x e 4

Documentação

- Ao definir uma função podemos (e devemos) adicionar uma linha com **documentação**
  - É opcional.
  - Na 1ª linha do corpo da função.
  - Delimitada por `""" """` (triple double quotes)
- Esta documentação pode ser consultada com a função `help()`
- Fica armazenada em `<nome da função>.__doc__`

In [9]:
```
def happy():
    """Uma função que mostra um verso da canção de aniversário."""
    print ("Happy birthday to you!")
```

```
help(happy)
```

```
Help on function happy in module __main__:

happy()
    Uma função que mostra um verso da canção de aniversário.
```

Variáveis Locais: visibilidade

- Variáveis definidas no corpo de função não são visíveis de fora

In [10]:
```
def soma_dois_valores(a, b):
    soma = a + b
    return soma

s = soma_dois_valores(2, 3)
print(s)
# no programa principal, fora da função:
# print(soma)
```

```
5
```

Variáveis Locais: sobreposição

- Uma variável local esconde outra com o mesmo nome
  - Dentro da função, é usada a definição mais próxima do nome
  - Fora da função, é usada a definição visível

In [11]:
```
soma = 8

def soma_dois_valores(a, b):
    soma = a + b
    print(soma)
    return soma

resultado = soma_dois_valores(2, 3)
print(resultado)
print(soma)
```

```
5
5
8
```

Argumentos: passagem por valor

- Ao invocar uma função, é passado o valor a usar nos argumentos
  - Esses argumentos são trabalhados no corpo da função como variáveis locais
  - A alteração dessas variáveis locais relativas aos argumentos não tem efeito fora da função excepto certas manipulações de **tipos referenciados**, como listas

In [12]:
```
def sucessor(n):
  n = n + 1
  return n

valor = 4
print(sucessor(valor))         # imprime 5
print(valor)                   # imprime 4

# A afetação n=n+1 (dentro da função)
# não alterou a variável valor (fora da função).
```

```
5
4
```

Para pensar

- O matemático Srinivasa Ramanujan encontrou uma série que pode ser usada para estimar uma aproximação ao valor π

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

- Pense num programa em Python
  - com várias funções
  - visualização de estimativas com aproximação do valor de pi

In [13]:
```
# definição e uso de duas funções:
```

```python
def area(largura, comprimento):
    return largura * comprimento

def volume(largura, comprimento, altura):
    return area(largura, comprimento) * altura

l = 5        # largura, em metros
c = 8        # comprimento, em metros
a = 3        # altura, em metros
print('área de uma sala {}x{}: {} m2'.format(l, c, area(l,c)))
print('área de um campo {}x{}: {} m2'.format(80, 110, area(80,110)))
print('volume de sólido {}x{}x{}: {} m3'.format(l, c, a, volume(l,c,a)))
```

```
área de uma sala 5x8: 40 m2
área de um campo 80x110: 8800 m2
volume de sólido 5x8x3: 120 m3
```

Outros exemplos

- Índice de Massa Corporal
- Implemente 2 funções:
  - `imc(peso,altura)`
    - devolve (ou *retorna*) o índice de massa corporal de uma pessoa com aquele peso e altura
  - `classifica(peso, altura)`
    - Não devolve nenhum valor
    - Imprime A, B ou C, como resultado de classificar o IMC em:
      - A: até 18.5
      - B: entre 18.5 e 25
      - C: acima de 25
    - Esta função não faz o cálculo, em vez disso deve invocar a função imc()

Ciclos `for` e função `range`

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Tipos de Natureza Sequencial

- Tipos de Dados
  - Constituídos por diversos elementos
  - A ordem entre os elementos importa
- Exemplos
  - String
  - Lista
  - Range
- Tal como a lista, uma string é **indexável**
  - permite acesso direto aos seus elementos

In [2]:
```
s = 'um exemplo'
print(s[3])
```

```
e
```

Introdução às Listas

- Lista
  - Tipo de dados
  - Guarda uma sequência de valores (chamados elementos da lista)
  - Notação:
    - `[]  # uma lista vazia`
    - `[1]  # lista com um elemento: o inteiro 1`
    - `[1, 'exemplo']  # lista com 2 elementos`
- Outro exemplo: a lista com os argumentos passados a um script:
  - `sys.argv    # lista de str`

In [3]:
```
l = [1,2]
type(l)
```

Out[3]:
```
list
```

Operações elementares sobre listas

- Tal como nas strings, temos:
  - Concatenação
    - operador `+`
    - entre duas listas
  - Repetição
    - operador `*`
    - entre uma lista e um inteiro

In [4]:
```
a = [1, 2]; b = [3, 4, 5]; c = a + b
print(type(c))
c
```

```
<class 'list'>
```

Out[4]:
```
[1, 2, 3, 4, 5]
```

In [5]:
```
lista = [1, 2] * 3
lista
```

Out[5]:
```
[1, 2, 1, 2, 1, 2]
```

Função built-in len()

- `len(obj)`

- Devolve um inteiro com o comprimento (nº de elementos) de obj
- O argumento deve ter um tipo de natureza sequencial

In [6]:
```
len('teste')

len([])

len([0,1,2,3])
s1 = 'outro'
s2 = 'exemplo'
l = [s1, ' ', s2]
len(l)
```

Out[6]: 5

Out[6]: 0

Out[6]: 4

Out[6]: 3

Ciclo for

- instrução `for`
  - ciclo vocacionado para iteração sobre uma sequência
- Semântica:
  - *para cada* elemento numa coleção
    - executar um bloco de instruções
- Sintaxe:

```
for <variável> in <sequência>:
  <bloco>        # corpo do ciclo
```

  - Em cada iteração, `<variável>` assume cada um dos elementos na sequência, pela ordem em que os mesmos surgem

In [7]:
```
# imprimir os elementos de uma lista:
for i in [2, 4, 6]:
    print(i)
```

```
2
4
6
```

In [8]:
```
moveis = ['mesa', 'camiseiro', 'cadeira', 'sofa']

for movel in moveis:
    print(movel)        # imprime elemento movel

print(len(movel) )   # imprime comprimento da string movel
```

```
mesa
camiseiro
cadeira
sofa
4
```

Função built-in range()

- `range([start,] stop[, step])`
  - Obter uma *progressão aritmética*, na forma de uma sequência
  - Aceita 1, 2 ou 3 argumentos inteiros (dois são opcionais)
    - `stop` : limite superior da sequência, não incluído no resultado
    - `start` : início da sequência (por omissão é 0)
    - `step` : passo da progressão (por omissão é 1)
- Muito útil nos ciclos for (e não só)

In [9]:
```
print(list(range(3)))

print(list(range(10)))

print(list(range(1, 11)))

print(list(range(0, 30, 5)))

print(list(range(0, 8, 3)))

print(list(range(0, -10, -1)))
```

```
print(list(range(0)))

print(list(range(1, 0, 1)))
```

```
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 5, 10, 15, 20, 25]
[0, 3, 6]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
[]
[]
```

O tipo `range` no Python 3

- Em Python 3, `range` é um tipo sequencial imutável, com valores numéricos
  - A função `range( )`
    - É construtor do tipo/classe `range`.
    - Devolve valor do tipo `range`, em vez de lista
  - Indexável
  - Aplicável em ciclos `for`, como qualquer sequência

In [10]:
```
# indexavel
range(1, 5)[2]
```

Out[10]: 3

In [11]:
```
for x in range(3):
    print(x)
```

```
0
1
2
```

Exemplo

- Suponhamos que pretendemos a média de n valores
  - `n` é variável, inserido pelo utilizador

In [12]:
```
n = int(input('quantos valores são? '))

soma = 0.0        # para acumular a soma dos valores

for i in range(n):      # ciclo: para i, de 0 a n-1
  valor = float(input('insira o valor -> '))
  soma = soma + valor

print('soma: {:.2f}'.format(soma))
print('média: {:.2f}'.format(soma / n))
```

```
soma: 37.00
média: 12.33
```

Mais exemplos

- Dada uma lista, imprimir cada elemento

In [13]:
```
moveis = ['mesa', 'camiseiro', 'cadeira', 'sofa']

print('versão while:')
i = 0
while i < len(moveis):
    print(moveis[i])
    i = i+1
print()
print('versão for in range:')
for i in range(len(moveis)): # cada i, 0 a comprim.-1
    print(moveis[i])

print()
print('versão for in lista:')
for m in moveis:  # para cada m, de 'mesa' a 'sofa'
    print(m)
```

```
versão while:
mesa
camiseiro
cadeira
sofa
```

```
versão for in range:
mesa
camiseiro
cadeira
sofa

versão for in lista:
mesa
camiseiro
cadeira
sofa
```

Exemplo: números pares

- Mostrar n nºs pares acima de um valor y

In [14]:
```
y = int(input())
n = int(input())
if y % 2 != 0: # se ímpar, ajustar para par seguinte
    y += 1

for par in range(y, y + (n * 2), 2):  # ...de 2 em 2
    print(par)
```

```
12
14
16
```

In [15]:
```
# alternativa 1:
y = int(input())
n = int(input())
mostrados = 0
while mostrados < n:
    if y % 2 == 0:
        print(y)    # mostra par
        mostrados += 1
    y = y + 1
```

```
12
14
16
```

In [16]:
```
# alternativa 2:
y = int(input())
n = int(input())
mostrados = 0
if y % 2 != 0:
    y = y + 1
# y é par
while mostrados < n:
    print(y)   # mostra par
    mostrados += 1
    y = y + 2   # próximo par
```

```
12
14
16
```

`break` em ciclos for

- efeito idêntico ao previsto para os ciclos `while`

```
for <variável> in <sequência>:
  <instrução 1>
   ...
  if <condição_especial>:
      break        # sai imediatamente do ciclo
  ...
  <instrução N>
```

- **Nota**: se existirem ciclos dentro de ciclos (encaixados)
  - `break` vai terminar apenas o ciclo em que se insere
    - o mais recente

`continue` em ciclos for

- efeito idêntico ao previsto para os ciclos `while`

```
for <variável> in <sequência>:
  <instrução 1>
   ...
  if <condição_especial>:
      continue        # passa à iteração seguinte
  ...
  <instrução N>
```

- **Nota**: se existirem ciclos dentro de ciclos (encaixados)
  - `continue` vai terminar apenas o ciclo em que se insere
    - o mais recente

In [17]:
```
def primo_for_break(n):
    primo = True
    for i in range(2, n):
        if n % i == 0:
            primo = False
            break
    return primo

primo_for_break(13)
```

Out[17]:
```
True
```

In [18]:
```
def primo_while_break(n):
    primo = True
    i = 2
    while (i < n):
        if n % i == 0:
            primo = False
            break
        i += 1
    return primo

primo_while_break(13)
```

Out[18]:
```
True
```

In [19]:
```
def primo_while(n):
    primo = True
    i = 2
    while (primo and i < n):
        if n % i == 0:
            primo = False
        i += 1
    return primo

primo_while(13)
```

Out[19]:
```
True
```

`for` sobre strings

- Semelhante
  - Os elementos são os caracteres no texto

In [20]:
```
for char in 'Programação':
    print(char)
```

```
P
r
o
g
r
a
m
a
ç
ã
o
```

------## Considerações

- `while` e `for` são instruções para ciclos
  - `while` é mais genérico
    - também pode usar-se para percorrer listas
  - `for` é vocacionado para iterar sobre sequências (como *listas* e *strings*)
    - Simplifica a sintaxe do programa
    - Fácil de ler

- Não requer **manipulação** explícita de uma variável envolvida na condição de permanência no ciclo (usual no while)

In [21]:
```
l = [1,2,3]
i = 0
while i < len(l):
  print(l[i])
  i += 1
```

```
1
2
3
```

In [22]:
```
l = [1,2,3]
for e in l:
  print(e)
```

```
1
2
3
```

Função `range()` e ciclo `for`: outros exemplos

- Como fazer:
  - Lista de inteiros entre 1 e 20
  - Lista de inteiros entre 5 e 15 que sejam divisíveis por 3
  - Imprimir o quadrado dos m primeiros valores inteiros da sequência
    - n, n+k, n+2k, n+3k...

In [23]:
```
# dada uma lista de valores com notas de alunos,
# devolver o nº de positiva

def conta_positivas(lista_de_notas):
   total = 0
   for nota in lista_de_notas:          # para cada valor
      if nota >= 10:
         total += 1
   return total       # contagem de notas >=10 na lista


# exemplo de invocação da função
notas_de_prog= [9, 12, 15, 8, 10]
pos = conta_positivas(notas_de_prog)
print("Em Programação houve {} positivas".format(pos))
```

```
Em Programação houve 3 positivas
```

Tipo Lista

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Tipo `list`

- Tipo de dados
  - Sequencial
    - Há noção de posição relativa entre os elementos
  - Indexável
    - Permite acesso direto ao elemento numa determina posição
  - Mutável
    - É possível mudar os elementos da lista
      - Valor de um elemento
      - Adicionar ou remover elementos

Os elementos da lista

- Quantos são?
  - Usar a função `len()`
  - O nº de elementos corresponde ao comprimento da lista
    - `[]` é a lista vazia, tem `0` elementos
      - `len([]) == 0`
- Cada elemento é independente dos restantes
  - Podem ter tipos diferentes. Como por exemplo:
    - inteiros e strings
      - `[ 1, 'exemplo' ]`
    - inteiros e listas
      - `[ 2, ['lista como elemento... e tem uma string'] ]`

In [2]:
```
l = [ 2, ['lista como elemento', 'tem uma string'] ]
len(l)
```

Out[2]: 2

Índices

- Índices simples: correspondem diretamente aos elementos
  - Compreendidos entre `0` e (tamanho-1)
- Índices para facilitar a manipulação das extremidades da lista: índices negativos `-1:` corresponde ao último elemento
  - Em geral, `-n` determina o posicionamento desde o fim da lista
    - Corresponde ao índice normal `len(lista) - n`

In [3]:
```
l = [1, 2, 3, 4, 5]

print(l[-4])
```

```
2
```

Operações elementares sobre listas

- Tal como nas strings, termos:
  - Concatenação: operador `+`
    - Entre duas listas
  - Repetição: operador `*`
    - Entre uma lista e um inteiro

In [4]:
```
a =[1, 2]; b = [3, 4, 5]; c = a + b
c
```

Out[4]: [1, 2, 3, 4, 5]

In [5]:
```
lista = [1, 2] * 3
lista
```

Out[5]:
```
[1, 2, 1, 2, 1, 2]
```

Listas como objetos: métodos úteis

- Python trata listas como objetos
  - Têm associadas funções muito úteis (métodos)
- Podemos aplicar um destes métodos sobre qualquer variável do tipo lista
  - `var.metodo()`
  - `var.metodo(argumento)`
- Alguns métodos úteis:
  - `append(), insert(), pop(), remove(), index(), count(), reverse(), sort(), extend()`

Operações via método: append

- `append(e)`
  - Adicionar um elemento à lista, no fim
  - altera a lista

In [6]:
```
lista = [1, 2]

lista.append(3)

lista
```

Out[6]:
```
[1, 2, 3]
```

Operações via método: insert

- `insert(i, e)`
  - inserir novo elemento e na posição/índice `i`
  - A lista é alterada
  - o novo elemento é colocado na posição indicada no primeiro argumento
  - se (índice == comprimento), então é como `append()`

In [7]:
```
lista = [1, 2]
lista.insert(1, 1.5)
lista
```

Out[7]:
```
[1, 1.5, 2]
```

Operações via método: `remove`

- `remove(valor)`
  - remove o primeiro elemento da lista igual a `valor`
  - a lista é alterada

In [8]:
```
lista = [11, 12, 13, 14, 15]
lista.remove(13)
lista
```

Out[8]:
```
[11, 12, 14, 15]
```

In [9]:
```
# só remove a 1ª ocorrência
lista = [1, 2, 3, 2, 4]
lista.remove(2)
lista
```

Out[9]:
```
[1, 3, 2, 4]
```

Operações via método: `pop`

- `pop(i)`
  - O argumento é opcional e representa um índice (`i`).
  - o elemento nessa posição será removido

- sem argumentos, retira o elemento do fim da lista
- a lista é alterada e é **devolvido** o valor removido

In [10]:
```
lista = ['a', 'b', 'c', 'd', 'e']
e1 = lista.pop(1) # remove o 2o elemento
print(e1)
print(lista)

e2 = lista.pop()  # remove o último elemento
print(e2)
print(lista)
```

```
b
['a', 'c', 'd', 'e']
e
['a', 'c', 'd']
```

Operações via método: `reverse`

- `reverse()`
  - Altera a ordem dos elementos, deixando-os do último para o primeiro
  - Não devolve nada, mas altera a lista

In [11]:
```
lista = ['sofa', 'cadeira', 'mesa']
lista.reverse()
lista
```

Out[11]:
```
['mesa', 'cadeira', 'sofa']
```

In [12]:
```
l = [1, 2, 3, 4, 5]
l.reverse()
l
```

Out[12]:
```
[5, 4, 3, 2, 1]
```

Operações via método: `index`

- `index(valor)`
  - devolve o índice do primeiro elemento igual a `valor`
  - Dá erro se não houver aquele `valor` na lista

In [13]:
```
lista = ['sofa', 'mesa', 'cadeira', 'mesa']
i = lista.index('mesa')
print(i)
```

```
1
```

Operações via método: `extend`

- `extend(outra_lista)`
  - Estende a lista alvo, acrescentando-lhe todos os elementos da lista `outra_lista`, no fim.

In [14]:
```
l1 = [1, 2]
l2 = [3, 4]

l2.extend(['hello', 'world'])
print(l2)

l1.extend(l2)
print(l1)
```

```
[3, 4, 'hello', 'world']
[1, 2, 3, 4, 'hello', 'world']
```

Operações via método: count

- `count(valor)`
  - conta o número de elementos da lista que são iguais a `valor`

In [ ]:
```
lista = [1, 2, 3, 2, 1]

# print(lista.count(3))

c1 = lista.count(1)
print(c1)
```

```
2
```

In [15]:
```python
def conta_lista(lista, elemento):
    conta = 0
    for e in lista:
        if e == elemento:
            conta += 1
    return conta



lista =  ['a', 'b', 'c', 'a', 'a']

c1 = conta_lista(lista, 'a')

print(c1)
```

```
3
```

Métodos: alteração e retorno

- Alteram a lista
  - sem retorno
    - `sort, reverse, insert, append, extend`
  - com retorno (devolvem um valor diferente de `None`)
    - `remove, pop`
- Não alteram a lista, devolvem um valor
  - `index, count`

Comparação entre listas

- Equivalência entre listas
  - elementos com valores idênticos: `==`

In [16]:
```python
lista1 = [1, 2, 3]

lista2 = []

print(lista1 == lista2) # False

lista3 = [1, 3, 2]

print(lista1 == lista3) # False

lista4 = [1, 2, 3]

print(lista1 == lista4) # True

print([1, 3, 2] == lista3) # True
```

```
False
False
True
True
```

Pertença a lista: operador `in`

- `<expressão> in <lista>`
  - dá `True` se a `lista` contém um elemento igual ao valor da `expressão` caso contrário devolve `False`

In [17]:
```python
listaDeCompras = ['pao', 'queijo', 'fruta', 'manteiga']

print('leite' in listaDeCompras) # False

print('pao' in listaDeCompras) # True
```

```
False
True
```

In [18]:
```python
# Considerar a precedência
a = [1, 1+1, 4-1]
print(a) # [1, 2, 3]
print(2 in a) # True
print(1+1 in a) # True
```

```
[1, 2, 3]
True
True
```

Exemplos: tira números pares

Função que recebe uma lista de valores e devolve uma nova lista com os pares

In [19]:
```
def devolve_pares( lista ):
    '''devolve uma nova lista com os n°s pares recebidos no argumento'''
    pares = []              # lista para colecionar os pares
    for valor in lista:
        if valor % 2 == 0:     # se é par
            pares.append(valor)
    return pares

# teste:
l1 = [3, 4, 5, 6, 7]
p1 = devolve_pares(l1)
print(p1)                     # imprime [4, 6]
print(devolve_pares([3, 5]))          # imprime []
```

```
[4, 6]
[]
```

In [20]:
```
# Exemplos: remover valores de uma lista

def remove_abaixo(lista, limiar):
     tamanho = len(lista)     # referência para o maior índice da lista
     i = 0
     while i < tamanho:
          if lista[i] < limiar:   # este deve ser removido
               removido = lista.pop(i)   # é retirado um elemento da lista
               print("foi tirado o valor: {}".format(removido))
               tamanho -= 1    # atualizar o tamanho
               continue          # continuar para a iteração seguinte
          i += 1                    # incremento apenas quando não se remove

lista = [1, 4, 2, 3, 5]
remove_abaixo(lista, 3)
print(lista)
```

```
foi tirado o valor: 1
foi tirado o valor: 2
[4, 3, 5]
```

Identidade da lista: operador is

In [21]:
```
# Duas variáveis com a mesma lista têm sempre o mesmo valor

a = [1, 2]
b = a # b é um alias ou sinónimo de a

print(a == b) # True

print(a is b) # True

a.append(3)
print(b)
```

```
True
True
[1, 2, 3]
```

In [22]:
```
# Duas variáveis com listas separadas podem ter o mesmo conteúdo

a = [1, 2]
b = [1, 2]
print(a == b) # True validam a comparação de equivalência ==
print(a is b) # False mas não a identidade

a.append(3)
print(b)
print(a)
```

```
True
False
[1, 2]
[1, 2, 3]
```

Comparação rápida dos 1ºs elementos

- Operador `>` ou `<` entre listas
  - Comparam o primeiro elemento
    - Apurada a desigualdade, os restantes elementos não são considerados
    - Se o 1º elemento for igual, compara-se o seguinte

- ...

In [23]:
```
a = [1, 2, 40]
b = [1, 3, 5]
print(a < b) # True

print(a > b) # False
b = [0, 3]
print(a > b) # True
```

```
True
False
True
```

Slicing: manipulação de partes

- Slicing significa obter uma "fatia" da lista
  - Forma geral: `lista[i:j:k]`
    - Copiar os elementos desde o índice `i` até ao `j-1`, tirando elementos de `k` em `k` posições
    - Se `k` é omitido, os elementos são escolhidos de `1` em `1` # step/passo
    - Se `i` é omitido, o início é no índice `0`
    - Se `j` é omitido, o final coincide com o fim de lista
    - Se ambos `i` e `j` são omitidos, todos os elementos de lista são selecionados, formando uma cópia idêntica numa nova lista

In [24]:
```
a = ['a', 'b', 'c', 'd']

# print(a[:3])

print(a[1:3])
print(a[:3])
print(a[2:])
b = a[:] # b recebe uma cópia da lista a
print(b) # ['a', 'b', 'c', 'd']
print(b == a) # True
print(b is a) #  False  b é cópia, não é alias de a
a.append('e')
print(b)
```

```
['b', 'c']
['a', 'b', 'c']
['c', 'd']
['a', 'b', 'c', 'd']
True
False
['a', 'b', 'c', 'd']
```

Slicing à esquerda da afetação

- Slicing à esquerda da afetação (sem o argumento `k`)
  - Substituir fatia da lista pela sequência à direita da afetação

In [25]:
```
a1 = [0, 1, 2, 3, 4, 5]
a1[0:3] = [9]
print(a1) # [9, 4, 5]
a2 = [1, 2, 3, 4, 5]
a2[0:3] = []
print(a2) # [4, 5]
```

```
[9, 3, 4, 5]
[4, 5]
```

Manipulação de listas

- Manipulação da lista envolve operações sobre os elementos
  - A posição do elemento é conhecida?
    - *Sim*: acesso direto com `lista[indice]`
    - *Não*: é necessário percorrer a lista, por exemplo com um ciclo
  - A operação mais frequente é a procura/pesquisa do valor dos elementos
    - Os elementos são lidos mas **não alterados**
  - Operações que modificam a lista
    - Adicionar ou retirar elementos
    - Alteração do valor de elementos
  - Se a lista original não se pode perder, estas operações devem fazer-se sobre uma cópia: `copia = lista[:]`

Argumentos de Função: passagem por referência

- Tipos referenciados (como **listas**)
  - o valor passado funciona como um sinónimo (alias) para a variável passada como argumento
  - *alterar* um elemento da lista irá refletir-se após a função terminar

In [26]:
```
def muda_lista(lista):
    lista[0] = 'outro'     # colocar outro valor na 1ª posição

l = [1, 2, 3, 4]

# muda_lista(l[:])       # o argumento é cópia da lista l
# print(l)               # imprime [1, 2, 3, 4]

muda_lista(l)       # passando l, a função usa sinónimo
print(l)
```

```
['outro', 2, 3, 4]
```

Argumentos de Função: passagem por referência

- Tipos referenciados (como `listas`)

In [27]:
```
def appendar(lista, elemento):
    lista.insert(len(lista), elemento)

lista1 = [1, 2]
lista2 = lista1[:]
appendar(lista2, 3)
print(lista1)
print(lista2)
```

```
[1, 2]
[1, 2, 3]
```

Formas de percorrer a lista

- Pesquisa sequencial
  - Do início para o fim
    - Com ciclo `for` (o mais usual)
    - Com ciclo `while` e variável para índice: `0`, `1`, ...
  - Do fim para o início
    - Com ciclo `while` e variável para índice: `len()-1`, ..., `1`, `0`
    - `reverse()`, seguido de ciclo `for`
- Pesquisa não sequencial
  - Tentativas em "zonas" da lista, quando esta está ordenada
    - Para encontrar um elemento por aproximação
      - Primeiro procurar no índice do meio
        - se é o valor procurado, então já está
        - se elemento > procurado, ver à esquerda,
        - se não, ver à direita

Listas de listas para representar matrizes

In [28]:
```
matriz = [
          ['a', 'b'],
          ['c', 'd'],
          ['e', 'f']
          ]

# print(len(matriz))
# print(len(matriz[1]))

print(matriz[1][1])
# print(len(matriz[2]))
```

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Tuplos

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

- Tipo de dados: `<type 'tuple'>`
- Tipo sequencial
  - Tuplo é uma sequência de valores separados por vírgula
    - Usualmente entre `( )` # mas nem sempre

In [2]:
```
# Exemplo: par ordenado
coordenadas = (4, 7)
type(coordenadas)
```

Out[2]: tuple

Tuplo: indexável e imutável

- Sendo uma sequência, tem um comprimento: o nº de itens
- Acesso direto a cada item pelo seu índice
- Imutável: não podemos alterar o valor de um item do tuplo

In [3]:
```
coordenadas = (4, 7)
print(len(coordenadas))
print(coordenadas[0])
print(coordenadas[1])
# coordenadas[0] = 3 # -> erro
```

```
2
4
7
```

Tuplos: itens de qualquer tipo

Afetação a uma sequência de valores separados por vírgula (com ou sem parêntesis) gera um tuplo

In [4]:
```
tup1 = 2011,
tup2 = 9, 8, 7
tup3 = (1, 'dois', [3], (4), (5,))

type(tup1) # <class 'tuple'>
type(tup2) # <class 'tuple'>
type(tup3[2]) # <class 'list'>
type(tup3[3]) # <class 'int'>
type(tup3[4]) # <class 'tuple'>
type(tup3[1][1])
```

Out[4]: tuple

Out[4]: tuple

Out[4]: list

Out[4]: int

Out[4]: tuple

Out[4]: str

Tuplos vazios, índices e slices

- Tuplo como sequência de zero itens
- Índices negativos e slices têm a mesma leitura que vimos para outros tipos sequenciais... As slices de tuplos geram tuplos!

In [5]:
```
tuplo_vazio = ( )

type(tuplo_vazio) # <class 'tuple'>

t = ('a','b','c','d','e')

t[-1] # 'e'

t[0::2] # ('a', 'c', 'e')

t[-2:] # ('d', 'e')
```

Out[5]: tuple

Out[5]: 'e'

Out[5]: ('a', 'c', 'e')

Out[5]: ('d', 'e')

Função `tuple()`

- Função built-in que *gera* um tuplo
  - cujos itens são os elementos do argumento (um valor iterável)

In [6]:
```
a = tuple()
a # ()      # TUPLO VAZIO

b = tuple('exemplo')
b # ('e', 'x', 'e', 'm', 'p', 'l', 'o')

c = tuple([1, 2, 3, 4, 5])
c # (1, 2, 3, 4, 5)
```

Out[6]: ()

Out[6]: ('e', 'x', 'e', 'm', 'p', 'l', 'o')

Out[6]: (1, 2, 3, 4, 5)

Outras operações comuns em sequências

- `+`: Concatenação entre tuplos gera novo tuplo
- `*`: Repetição do valor de um tuplo, formando um novo tuplo

In [7]:
```
t1 = (1, 2) + tuple("ola")
t1 # (1, 2, 'o', 'l', 'a')

t2 = ('a', 'b', 1) * 2
print(t2) # ('a', 'b', 1, 'a', 'b', 1)
```

Out[7]: (1, 2, 'o', 'l', 'a')

```
('a', 'b', 1, 'a', 'b', 1)
```

Tuple packing & unpacking

- Packing: passar de uma sequência de valores para um tuplo
  - O que se faz com uma afetação
- Unpacking: operação inversa. Afetação do valor de cada elemento do tuplo a uma variável
- Packing e unpacking na mesma expressão de afetação:

In [8]:
```
t = 1, 2, 3     # packing: formar tuplo por afetação de sequência
type(t) # <class 'tuple'>

a, b, c = t   # unpacking
b # 2

a, b = b, a # packing e unpacking
b # 1
```

Out[8]: tuple

Out[8]: 2

Out[8]: 1

Tuplos e funções

- Tuplos tornam possível:
  1. Devolver mais que um valor como resultado da função
     - Envolvendo todos os valores num tuplo
  2. Passar um número variável de valores como argumentos

In [9]:
```
# função: max()
max(5, 6) # 6

max(5, 6, 1) # 6

max(5, 6, 1, 9) # 9
```

Out[9]: 6

Out[9]: 6

Out[9]: 9

In [10]:
```
def divide(a, b):
    q = 0
    r = 0
    while a > b:
        a = a - b   # divisão por subtrações
        q += 1
        r = a
    return (q, r)    # devolve 1 expr com 2 valores

quociente, resto = divide(18, 10)
print(quociente)
print(resto)
```

```
1
8
```

Tuplos como argumento

- Permitindo N itens como argumento da função

In [11]:
```
def maximo(*valores):
    if len(valores) == 0:
        return None
    else:
        m = valores[0]        # suponhamos que o máximo é o primeiro elemento do tuplo
        for v in valores[1:]:
            if v > m:
                m = v
        return m

print(maximo())
print(maximo(1, 2, 3, 4, 5))
print(maximo(1, 2, 3))
```

```
None
5
3
```

Função `zip()`

- `zip(s1, …, sn)`
- Forma uma sequência de tuplos obtidos obtidos a partir das sequências nos argumentos `s1, ... sn`
  - cada tuplo `i` inclui o i-ésimo elemento de cada sequência, nos argumentos
  - o comprimento da sequência devolvida é o comprimento do menor argumento
  - retorno:
    - tipo sequencial; iterável; não indexável – exatamente com os mesmos elementos

In [12]:
```
numeros = [1, 2, 3]
letras = ['a', 'b', 'c']
simbolos = ['!', '#', '$']
zipped = zip(numeros, letras, simbolos)
print('Tipo: {} e conteúdo convertido para lista: {}'.format(type(zipped),list(zipped)))
```

```
Tipo: <class 'zip'> e conteúdo convertido para lista: [(1, 'a', '!'), (2, 'b', '#'), (3, 'c', '$')]
```

Exemplo: atravessar 2 sequências (`s1` e `s2`) num só ciclo, para saber se `s1[i] == 2*s2[i]`

In [13]:
```
# versão com while
def compara(s1, s2):
    i = 0
    while i < len(s1):
        if s1[i] != 2*s2[i]:
            return False
        i = i+1
    return True

print(compara([4, 8], [2, 4]))
```

```
True
```

In [14]:
```
# zip e iteradores em for
def compara(s1, s2):
    for a, b in zip(s1, s2):   # mais de 1 var iteradora
        if a != 2 * b:    # se um não valida
            return False
    return True

print(compara([4, 8], [2, 4]))
```

```
True
```

In [15]:
```
# zip e iteração com o ciclo for
for a, b in zip(['a', 'b', 'c'], ['A', 'B', 'C']):
    print('valor: {} ... e o outro: {}'.format(a, b))
```

```
valor: a ... e o outro: A
valor: b ... e o outro: B
valor: c ... e o outro: C
```

Funções: recursividade, argumentos opcionais com valor por omissão

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Recursividade: critério de paragem

- Tal como nos ciclos, a definição recursiva de funções requer um ponto de paragem
  - **Perigo**: infinitas invocações sucessivas
    - Esgotam recursos de memória do computador

Recursividade

- O corpo de uma função é uma sequência de instruções
  - Afetações
  - Invocação de funções
  - Outras expressões
- É possível ter uma invocação à própria função
  - Exemplo: uma função para fazer a contagem decrescente desde `n` é
    - Se `n >= 0`:
      - Imprimir `n`
      - Invocar a mesma função com `n-1` (continuar a contagem decrescente com `n-1`)

In [2]:
```
def contagem_decrescente(n):
    if (n >= 0):
        print(n)
        contagem_decrescente(n-1)
    else:
        return None

contagem_decrescente(10)
```

```
10
9
8
7
6
5
4
3
2
1
0
```

In [3]:
```
def contagem_decrescenteRInf(n): # mau exemplo
   print(n)
   contagem_decrescenteRInf(n-1)
```

Recursividade: outro exemplo

Função para contar os elementos numa lista, supondo que não podemos usar a função built-in `len()`

In [4]:
```
def comprimento(lista):
    if lista == []:        # paragem da recursão
        return 0
    # se nao fez return, a lista tem elementos
    # 1 + comprimento da slice sem o primeiro elemento
    else:
        return 1 + comprimento(lista[1:])


print( comprimento([]) )              # 0
print( comprimento([1]) )             # 1
print( comprimento([1,2,3,4,5]) )     # 5

def comprimentoWhile(lista):
    comprimento = 0
    while(lista != []):
        comprimento += 1
        lista = lista[1:]

comprimentoWhile([1,2,3])
```

```
0
1
5
```

Argumentos: valor por omissão

- É possível especificar um valor por omissão para os argumentos
  - Esse valor é usado apenas se a invocação não passa o argumento

In [5]:
```
def despedida(mensagem = 'Adeus!'):
    print(mensagem)

# invocar normalmente, com um valor
despedida("Ate Breve!")       # imprime Ate Breve!

# e podemos invocar sem passar o argumento
despedida()                   # imprime Adeus!
```

```
Ate Breve!
Adeus!
```

Argumentos: invocação e pares nome=valor

Ao invocar, podemos também indicar o nome do argumento para o qual indicamos um valor, com pares `argumento = valor`

In [6]:
```
def repete_frase(frase = 'teste', vezes = 2):
   while vezes > 0:
      print(frase)
      vezes = vezes-1

# sintaxe válida para invocar a função:
repete_frase('outra frase')
repete_frase(frase = 'teste', vezes = 2) # repete_frase()

repete_frase(vezes = 4)
repete_frase(vezes = 4,frase = 'outra frase')
```

```
outra frase
outra frase
teste
teste
teste
teste
teste
teste
outra frase
outra frase
outra frase
outra frase
```

In [7]:
```
def vogal(l):
    return l.lower() in list('aeiou')

def conta_vogais(frase):
     if len(frase)==0:              # caso base
          return 0
     else:
          x = conta_vogais(frase[1:])  # o resto
          if vogal(frase[0]):  # se 1ª letra é vogal
               x += 1
          return x
```

```
conta_vogais("A Maria e o Pedro")
```

Out[7]: 8

In [8]:
```
# Função para devolver o n-ésimo elemento da sequência de Fibonacci

def fib(n):

     if n < 2:     # caso base
          return 1
     else:
          return fib(n-1) + fib(n-2)

fib(5)
```

Out[8]: 8

Outros exemplos

- Factorial:
  - $n! = n * (n-1)!$
  - Tem uma definição recursiva...
- Abordagem em soluções **recursivas**:
  1. Cadeia de invocações da mesma função
     - $factorial(n) \rightarrow factorial(n-1) \rightarrow factorial(n-2) \ldots$
  2. Critério de paragem
     - $n < 2$
  3. Em cada invocação, tratar parte do problema e juntar com o que resulta do resto das invocações.
     - $n*$ valor_deixado_pela_função

Dicionários

Vitor Beires Nogueira (vbn@uevora.pt)


```
In [1]: from IPython.core.interactiveshell import InteractiveShell
        InteractiveShell.ast_node_interactivity = "all"
```

Dicionário

- Tipo de dados
  - Python 3: `<class 'dict'>`
- Mutável: podemos alterar o seu conteúdo
- Associativo: guarda correspondências entre valores
- Não é sequencial: os seus itens não são organizados por uma ordenação explícita.
- É uma estrutura de armazenamento mais complexa que as listas
- Um dicionário tem entradas ou itens
  - `{<entrada1>, <entrada2>,... , <entradaN>}`

Dicionário

- As sequências (como listas ou strings) são indexadas por inteiros
  - Item na posição 0, item na posição 1...
- Os Dicionários são indexados por **chaves**
  - Chave pode ser um valor de tipo **imutável**
    - strings, valores numéricos...
    - tuplos podem ser chave se não incluírem itens mutáveis
    - ~~listas não~~
- Um dicionário é um conjunto não ordenado de associações do tipo
  - *chave: valor*
    - valor pode ser de qualquer tipo (mutável ou não).
    - há uma única entrada para cada chave.

Dicionário

- Possibilidades para criar um dicionário vazio:
  1. Afetação:
     - `d = {}  # cria um dicionário vazio`
  2. Construtor / função dict ( )
     - `d = dict() # cria um dicionário vazio`
     - `dict()` aceita uma sequência de valores com dois elementos, usando o primeiro para chave e o segundo para valor
       - `dict([('two', 2), ('one', 1)])`

```
In [2]: d1 = {}                 # cria um dicionário vazio
        type(d1)

        d2 = dict()     # cria um dicionário vazio
        d1 == d2 # True
        d1 is d2 # False

Out[2]: dict

Out[2]: True

Out[2]: False
```

Dicionário: entradas

- Adicionar entradas num dicionário
  - `d[chave] = valor`
    - acrescenta (ou atualiza) uma entrada com chave valor
    - `d = {}; d['arvore']='vegetal de tronco lenhoso'`
- Remover entrada:
  - instrução `del d[chave]`
- Acesso a um elemento:
  - `d[chave]`

```
In [3]: d = dict([('um', 'one')])
        # d
        d['um'] = 'cone'
        #d
        d['dois'] = 'two'
        d
        d['dois']
        del d['um']
        d

Out[3]: {'um': 'cone', 'dois': 'two'}

Out[3]: 'two'

Out[3]: {'dois': 'two'}
```

```
In [4]: d1 = dict([(1, 'um')])
        d1
        # 2 -> dois
        d1[2] = 'dois'
        d1
        d1[1] = 'UM'
        d1
        del d1[1]
        d1

Out[4]: {1: 'um'}

Out[4]: {1: 'um', 2: 'dois'}

Out[4]: {1: 'UM', 2: 'dois'}

Out[4]: {2: 'dois'}
```

Tamanho e inclusão de chaves

- Tamanho / comprimento: número de itens no dicionário
  - `len()`
- Ver se o dicionário `d` tem uma entrada para `chave`
  - `chave in d`
    - `True` se `d` inclui aquela chave, `False` caso contrário

```
In [5]: len({}) # 0
        len({1:'um', 2:'dois', 3:'tres'}) # 3

        "donald" in {'tweety': 'passaro', 'willy': 'baleia', 'donald': 'pato'} # True
        "mickey" not in {'tweety': 'passaro', 'willy': 'baleia', 'donald': 'pato'} # True

Out[5]: 0

Out[5]: 3

Out[5]: True

Out[5]: True
```

Métodos de Dicionários

- `d.copy()`
  - devolve um novo dicionário que é uma cópia de `d`
- `d.clear()`
  - Remove todos os itens do dicionário `d`.

```
In [6]: d1 = {1: 'um', 2: 'dois', 3: 'tres'}
        d2 = d1.copy()
        del d1[1]
        len(d1) # 2
        len(d2) # 3

        d2.clear()
        len(d2) # 0

Out[6]: 2

Out[6]: 3

Out[6]: 0
```

Métodos de Dicionários

- `d.get(chave[, default])`
  - Devolve o valor associado a `chave` no dicionário `d`.
  - Se aquela chave não existir
    - devolve o valor por omissão, no argumento default.
      - o `default` é opcional
    - se o argumento não for passado, devolve `None`. Não gera um erro.

```
In [7]: d = {"tweety": "passaro", "willy": "baleia", "donald": "pato"}

        d.get("donald") # "pato"

        d.get("mickey","rato") # "rato"

        valor = d.get("mickey")

        print(valor) # None

Out[7]: "pato"

Out[7]: "rato"

        None
```

Métodos de Dicionários

- `d.pop(chave[, default])`
  - Remove a entrada com aquela chave (se existir) e
  - Devolve o valor que lhe está associado (ou em alternativa, o valor default – que é opcional, ou gera um KeyError, se não houver argumento default e não existir aquela chave no dicionário)

```
In [8]: d = {1:"a", 2:"b", 3:"c", 4:"d"}

        d.pop(3) # "c"

        d # {1: 'a', 2: 'b', 4: 'd'}

        d.pop(3, "e") # "e"


        d.pop(2,"e") # "b"

        len(d) # 2

        d.pop(2, "e") # "e"

        # d.pop(2) # --> erro

Out[8]: "c"

Out[8]: {1: "a", 2: "b", 4: "d"}

Out[8]: "e"

Out[8]: "b"

Out[8]: 2

Out[8]: "e"
```

Métodos de Dicionários

- `d.keys()`
  - Devolve uma sequência iterável (`dict_keys`) de chaves no dicionário `d`
- `d.values()`
  - Devolve uma sequência iterável (`dict_values`) de valores no dicionário `d`
- `d.items()`
  - Devolve uma sequência iterável (`dict_items`) com as entradas do dicionário `d` em tuplos (chave,valor)

```
In [9]: d = {1:'um', 2: 'dois', 3: 'tres'}

        d.keys() # dict_keys([1, 2, 3])

        d.values() # dict_values(['um', 'dois', 'tres'])

        d.items() # dict_items([(1, 'um'), (2, 'dois'), (3, 'tres')])

Out[9]: dict_keys([1, 2, 3])

Out[9]: dict_values(['um', 'dois', 'tres'])

Out[9]: dict_items([(1, 'um'), (2, 'dois'), (3, 'tres')])
```

Percorrer um dicionário

- Ciclos para atravessar o conteúdo de um dicionário

```
In [10]: d = {1: 'um', 2: 'dois', 3: 'tres'}

         for chave in d:
             print('{} <=> {}'.format(chave, d[chave]))

         for chave in d.keys():
             print('{} <=> {}'.format(chave, d[chave]))

         for chave, valor in d.items(): # obter os pares
             print('{} <=> {}'.format(chave, valor))

1 <=> um
2 <=> dois
3 <=> tres
1 <=> um
2 <=> dois
3 <=> tres
1 <=> um
2 <=> dois
3 <=> tres
```

Pesquisa inversa valor --> chave

- Dado um valor e um dicionário, encontrar uma chave que esteja associada aquele valor

```
In [11]: def obter_uma_chave_com_valor_em_dicionario(val, d):
             for c in d:
                 if d[c] == val:
                     return c
             return None

         d1 = {1: 'um', 2: 'dois', 3: 'tres'}

         print(obter_uma_chave_com_valor_em_dicionario('dois', d1))

2
```

Aplicações de Dicionários

```
In [12]: def contagem(str):
             '''> Contagem de letras numa string'''
             d = {}
             for c in str:
                 if c in d:  # já foi visto
                     d[c] += 1
                 else:       # primeira ocorrência
                     d[c] = 1
             return d    # devolve dicionário com as frequências

         d_palavra = contagem('banana')
         d_palavra
         type(d_palavra)

Out[12]: {'b': 1, 'a': 3, 'n': 2}

Out[12]: dict
```

```
In [13]: def barra(n):
             ''' > função que escreve uma sequencia de n símbolos =''''
             s = ''  # string vazia
             for i in range(n):
                 s += '='        # adiciona um bloco
             return s
         # poderiamos fazer de um modo bem mais simples
         help(barra)
         barra(4)

         Help on function barra in module __main__:

         barra(n)
             função que escreve uma sequencia de n símbolos =

Out[13]: '===='
```

```
In [14]: def histograma(str):
             d = contagem(str)
             for c, n in d.items():
                 graph = barra(n)
                 print('{}: {}'.format(c, barra(n)))

         histograma('banana')

b: =
a: ===
n: ==
```

Inverter um dicionário

- As chaves não se repetem, mas os valores podem repetir-se
- Ao inverter um dicionário, convém ter uma lista para os novos valores (que antes eram chaves)

```
In [15]: def inverte_dict(d):
             '''devolve o dicionário resultante de inverter o dicionário d'''
             d2 = {}
             for chave, val in d.items():
                 if val not in d2: # primeira ocorrencia
                     d2[val] = [chave]
                 else:             # se existir uma lista, adicionar...
                     d2[val].append(chave)
             return d2

         d_jogadores= {'filipe':'portugal', 'ronaldo' : 'portugal', 'sidane':'franca', 'guardiola':'espanha' }
         d_jogadores_inv = inverte_dict(d_jogadores)
         d_jogadores_inv

Out[15]: {'portugal': ['filipe', 'ronaldo'],
          'franca': ['sidane'],
          'espanha': ['guardiola']}
```

Outros usos do tipo dict

- Em qualquer problema que necessite de uma estrutura associativa, como
  - Lista telefónica
    - associa nomes a telefones
  - Índice Remissivo
    - associa termos a uma lista de páginas em que ocorrem
  - Stock
    - Produtos e respetivas quantidades

Mais Exemplos

- Dicionário como estrutura para os dados de uma agência de aluguer de carros

```
carros= {"11-10-AB": ["ford", 5, "gasolina"],
         "12-10-BC": ["vw", 5, "diesel"],
         "13-10-CD": ["mercedes", 2, "gasolina"] }
clientes= {"ana": [12345678, "Évora", 24],
           "bruno": [12340002, "Abrantes", 21],
           "carlos": [12340003, "Évora", 18] }
alugueres= {"11-10-AB": ["ana", "2012-11-19"] }
```

- Como responder a
  - O carro "12-10-BC" está alugado?
    - Há uma entrada com essa chave, em alugueres?
  - Qual a idade do cliente "bruno"?
    - Ver o índice 2 da lista associada à chave "bruno" em clientes.

Ficheiros: Input e Output

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Ficheiros: persistência

- Até aqui: dados inseridos pelo utilizador ficam em variáveis existem apenas durante a execução do programa
- Ficheiros
  - permitem armazenamento persistente de dados
    - podemos ler os valores a cada execução do programa: input
    - durante a execução podemos escrever dados no ficheiro: output
- Tipo file

```
f = open('texto.txt')
type(f) # <class '_io.TextIOWrapper'>
```

Operações para manipular ficheiros

- *Aceder* a um ficheiro, iniciando um objeto para aplicar as operações.
- Ler dados do ficheiro
  - Todo conteúdo como uma string
  - Ler uma linha (com o tipo string)
  - Ler todo o conteúdo obtendo uma lista de linhas
- *Escrever* para o ficheiro
- *Obter a posição*, no conteúdo do ficheiro, onde ocorre a próxima operação (como um apontador)
- *Alterar o posicionamento* dentro do ficheiro (avançar, recuar)
  - Por exemplo, para ler a partir do 10º caracter
- *Fechar* um ficheiro
  - terminar as operações libertando o apontador para o ficheiro

Função built-in open()

- `file = open(name, mode)`
  - Função built-in que dá acesso a um ficheiro cujo nome é passado no 1º argumento.
  - 2º argumento é opcional, para especificar o modo de uso:
    - 'r': Apenas para leitura (caso por omissão)
    - 'w': Apenas para escrita (faz truncate(), apaga o conteúdo)
    - 'a': Modo append (o que se escrever é adicionado no fim)
    - 'r+': Acesso para leitura e escrita.
- Se o ficheiro `name` não é encontrado:
  - **Errno**

Método close()

- `file.close()`
  - Fecha o ficheiro, libertando os recursos associados ao ficheiro.
  - Pode invocar-se mais que uma vez, sem erro.
  - Depois de fechar, uma tentativa de leitura gera um erro.
  - É importante fechar cada ficheiro aberto pelo programa.

Método read()

- `str = file.read()`
  - Devolve uma string lida de `file`, a partir da posição atual e até ao fim do ficheiro.
- `str = file.read(size)`
  - o argumento `size` é um inteiro para limitar a leitura.
  - devolve uma string lida do ficheiro `file`, a partir da posição atual e até um comprimento máximo de size bytes.

In [2]:
```
# Assumindo o ficheiro f.txt:
# abcdefghijkl

f = open('aux/f.txt')
s = f.read()
f.close()
s

f = open('aux/f.txt')
s1 = f.read(4)
s2 = f.read(4)
f.close()
s1
s2
```

Out[2]: 'abcdefghijkl\n'

Out[2]: 'abcd'

Out[2]: 'efgh'

Método readline()

- `str = file.readline()`
  - Lê uma linha inteira e devolve a string correspondente, incluindo o separador de linhas (`\n`), se estiver presente. (A última linha do ficheiro pode não ter este separador.)
- `str = file.readline(size)`
  - Lê uma linha devolve a string correspondente, incluindo o separador de linhas (`\n`) Se a linha é mais comprida que `size`, a leitura termina naquele ponto e é retornada a parte da linha que já foi lida.
- Uma linha em branco é devolvida como `"\n"`
- Se `str` é uma string vazia, atingiu-se o final do ficheiro.

Método readline()

- Exemplo de leitura de um ficheiro linha a linha
  - Com ciclo while

In [3]:
```
f = open('aux/f.txt')
s = f.readline()     # lê primeira linha
while s != '':       # enquanto não termina: str vazia
    print(s)
    s = f.readline()  # e lê a próxima linha
# após a leitura de todas as linhas
f.close()
```

```
abcdefghijkl
```

Método readlines()

- `line_list = file.readlines()`
  - Faz a leitura do conteúdo entre a posição atual e o final do ficheiro, devolvendo-o numa lista de strings.
  - Cada string representa uma linha, terminando com o respetivo `'\n'`.
  - A última pode ter ou não.

In [4]:
```
f = open('aux/f.txt')
line_list = f.readlines()
f.close()

# já com o ficheiro fechado
for strLinha in line_list:
    print(strLinha)      # imprime uma linha
```

```
abcdefghijkl
```

Método write()

- `file.write(str)`
  - Escrever a string `str` para o ficheiro `file`.
    - Dependendo do buffering do sistema operativo, se o ficheiro permanecer aberto, o conteúdo escrito poderá demorar algum tempo até surgir no ficheiro ...
    - A escrita será garantidamente processada com `file.flush()` ou `file.close()`.
- `f.write('Expressao para exemplo de escrita em ficheiro...\nOutra linha...')`
  - Conteúdo escrito:

```
Expressão para exemplo de escrita em ficheiro...
Outra linha...
```

Método writelines()

- `file.writelines(seq)`
  - Escrever uma sequência de strings para o ficheiro `file`.
    - O argumento `seq` pode ser uma lista ou tuplo de strings.
    - Não é adicionado nenhum separador.

```
f.writelines(['um','dois','tres'])
f.writelines(('quatro','cinco'))
```

    - Conteúdo escrito:

```
umdoistresquatrocinco
```

Métodos `tell()` e `flush()`

- `file.tell()`
  - devolve um inteiro com a posição atual dentro do ficheiro.
- `file.flush()`
  - Pedido ao Sistema Operativo para efetuar flush ou esvaziamento do buffer de saída associado ao ficheiro
  - Operação de baixo nível
  - Para que uma escrita pendente se concretize imediatamente

In [5]:
```
f = open('aux/f.txt')
print(f.tell())          # 0
s1 = f.read(4)
print(f.tell())          # 4
s2= f.read(4)
print(f.tell())          # 8
f.close()
```

```
0
4
8
```

Método `seek()`

- `file.seek(pos)`
  - Altera o posicionamento no ficheiro `file` para a posição `pos`.

In [6]:
```
f = open('aux/f.txt')
pos = f.tell()    # obter a posição atual
s = f.readline()  # lê uma string
print(s)
f.seek(pos)       # voltar à posição anterior
s = f.readline()  # lê a mesma string
print(s)
```

```
abcdefghijkl
abcdefghijkl
```

Iterar sobre o conteúdo do ficheiro

- Outro modo de ler o conteúdo de um ficheiro, linha a linha:
  - iteração com um ciclo for

```
inputFile= open(name)
for linha in inputFile:
print(linha)      # string, inclui eventual \n
inputFile.close()
```

- Como retirar o último '\n', no final da linha?
  - `linha = linha.strip()`
  - Ou, tendo já verificado que realmente termina com `"\n"`:
    - `linha = linha[:-1]`

Aplicações: copiar um ficheiro

- Efetuar uma cópia de um ficheiro com determinado nome:

In [7]:
```
# ler o nome do ficheiro existente
in_name = input('nome do ficheiro existente')
# ler o nome da copia
out_name = input('nome do ficheiro copia')

# inicias ficheiros, um para leitura outro para escrita
inputFile = open(in_name,'r')
outputFile = open(out_name,'w')

s = inputFile.readline()
while s != '':                # enquanto há mais conteúdo
    outputFile.write(s)       # escreve na cópia
    s = inputFile.readline()  # e lê a próxima linha

inputFile.close()
outputFile.close()
```

Out[7]: 13

Formatação de texto e ficheiros

Suponhamos que temos um ficheiro com o conteúdo:

```
manuel rosado: 19
maria ABRANTES:       31
Carlos gomes:159
MANUELA moTA: 0
```

Que queremos **normalizar** para:

```
NOME: manuel rosado,     VALOR:  19
NOME: maria abrantes,    VALOR:  31
NOME: carlos gomes,      VALOR: 159
NOME: manuela mota,      VALOR:   0
```

In [8]:
```
in_name = input('nome do ficheiro original')
out_name = input('nome do ficheiro normalizado')

infile= open(in_name)
outfile= open(out_name, 'w')

for linha in infile:
    l = linha.split(':')   # separar nome do valor
    nome = l[0].strip().lower()
    num = l[1].strip()
    outfile.write('NOME: {},\tVALOR: {:4s}\n'.format(nome, num))

infile.close()
outfile.close()
```

Out[8]: 33

Out[8]: 34

Out[8]: 32

Out[8]: 32

Outras aplicações: armazenamento persistente de dados

- Até agora. os dados usados num programa
  - Em variáveis – apenas em memória
  - Se o programa reiniciar perde novos dados
- Os ficheiros permitem escrever os dados que resultam da atividade do programa (lidos ou calculados)
  - Ao reiniciar pode ler os dados necessários dos ficheiro
  - Os ficheiros permitem o armazenamento persistente -Criação de uma memória que permanece no tempo
    - Pode evoluir
- Exemplo: lista telefónica
  - Pode adicionar novos contactos
  - Se o programa reiniciar, não perderá as últimas alterações

Modules/Módulos

Vitor Beires Nogueira (vbn@uevora.pt)

In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Considerações Gerais

- Um **módulo** é um ficheiro que contém definições e declarações cujo objectivo é serem usados noutros programas Python.
- Existem imensos módulos Python que são distribuídos conjuntamente com o Python e fazem parte da **biblioteca standard**
- Após importarmos um módulo, podemos usar as definições, etc que tal módulo define
- Aonde podemos encontrar informações sobre os módulos standard?
  - Python Module Index
- Podemos instalar outros módulos
  - pip

Ainda sobre Módulos

- Devemos considerar os módulos como "data objects"
  - Módulos contém simplesmente outros elementos do Python
- A 1a coisa que temos de fazer para usar um módulo é `import`
  - Como por exemplo, `import math`
    - cria um novo nome `math` e faz com que esse nome se refira ao módulo
- Usamos a notação `.` (ponto) para usarmos um módulo
  - Como por exemplo, `math.pi` que deve ser lido como: no módulo `math` aceda ao elemento `pi`

Módulo math

- O módulo `math` contém as funções expectáveis numa calculadora assim como várias contantes como

$$\pi, e$$

- Quando fazemos `import math`, criamos uma referência ao objecto módulo que contém tais elementos
- A *notação usual* para aceder às definições do módulo é *nome_do_modulo.nome_da_função*
  - O prefixo é o nome do módulo
  - O sufixo é o nome da função, etc
- https://docs.python.org/3/library/math.html

Exemplo simples

In [2]:
```
import math

print(math.pi)
print(math.e)

print(math.sqrt(2.0))

print(math.sin(math.radians(180)))    # sin of 90 degrees
```

```
3.141592653589793
2.718281828459045
1.4142135623730951
1.2246467991473532e-16
```

Módulo random

O módulo `random` fornece um conjunto de funções para gerar números (pseudo) aleatórios.

Funções

A função `random` devolve um float aleatório entre 0.0 (inclusivé) e 1.0 (exclusivé), i. e. um valor no intervalo

$$[0.0, 1.0[$$

In [3]:
```
import random
for i in range(10):
  print(random.random())
```

```
0.2480273524132972
0.446770557497947
0.17125961366047093
0.6009300572645073
0.24304203179562649
0.5367915191781825
0.646372198953277
0.5237311080292181
0.7708379218875707
0.32318253882227325
```

A função `randint(low, high)` recebe os parâmetros `low` e `high` e devolve um inteiro aleatório entre `low` (inclusivé) e `high` (inclusivé), i. e. um inteiro no intervalo

$$\{low, \ldots, high\}$$

In [4]:
```
import random
for i in range(5):
  print(random.randint(1, 10))
```

```
6
2
8
7
6
```

A função `choice` escolhe um elemento de uma sequência.

In [5]:
```
import random
lista = ['a', 'e', 'i', 'o', 'u']
for i in range(5):
  print(random.choice(lista))
```

```
u
o
o
u
u
```

Módulo mathplotlib (gráficos)

No exemplo abaixo iremos usar várias funções do módulo `mathplotlib`, em particular do seu interface `pyplot`

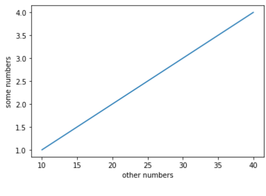
In [6]:
```
'''
matplotlib.pyplot is a state-based interface to matplotlib.
It provides a MATLAB-like way of plotting.
'''
import matplotlib.pyplot as plt
plt.plot([10, 20, 30, 40], [1, 2, 3, 4])
plt.ylabel('some numbers')
plt.xlabel('other numbers')
plt.show()
```

Out[6]:
```
'\nmatplotlib.pyplot is a state-based interface to matplotlib. \nIt provides a MATLAB-like way of plottin
g.\n'
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x7fdb959ad6d0>]
```

Out[6]:
```
Text(0, 0.5, 'some numbers')
```

Out[6]:
```
Text(0.5, 0, 'other numbers')
```



Módulo NumPy

O módulo NumPy é utilizado para computação científica e contém entre outras coisas:

- um objecto `array` N-dimensional
- ferramentas para integrar com código C/C++ e Fortan
- funcionalidades de álgebra linear, transformada de Fourier e números aleatórios
- funções "sofisticadas"

Conhecimentos Básicos

- O objecto principal do NumPy é o array homogéneo multidimensional
  - tabela elementos (normalmente números)
  - todos do mesmo tipo
  - indexado por um tuplo de inteiros não-negativos
  - dimensões são designadas por "axes" (eixos)
    - one axis: `[1, 2, 1]`
    - two axis: `[[ 1., 0., 0.], [ 0., 1., 2.]]`
    - ...

In [7]:
```
import numpy as np
a = np.arange(15).reshape(3, 5)
print(a)
print('shape: {}'.format(a.shape))
print('axis/ndimensions: {}'.format(a.ndim))
print('data type: {}'.format(a.dtype.name))
print('itemsize: {}'.format(a.size))
print('type: {}'.format(type(a)))
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]
shape: (3, 5)
axis/ndimensions: 2
data type: int64
itemsize: 15
type: <class 'numpy.ndarray'>
```

In [8]:
```
import numpy as np
b = np.array([6.5, 7.2, 8])
print(type(b))
print(b.dtype)
```

```
<class 'numpy.ndarray'>
float64
```

In [9]:
```
import numpy as np
a = np.array( [20, 30, 40, 50] )
b = np.arange( 4 )
print(a-b)
```

```
[20 29 38 47]
```

Módulos definidos pelo utilizador

- Um ficheiro que contém código Python, por exemplo `codigo.py`, define um módulo (neste caso seria o módulo `codigo`
- Usamos módulos para
  - particionar programas grandes
  - reutilizar o código
  - organizar em termos "lógicos" o código