Definir funções

É possível criar uma função escrevendo **def**, seguido pelo nome da função.

Função básica

```
def function():
        print("Olá")
```

É possível também adicionar variáveis à função, obrigando o utilizador a adicionar algo para que a função execute.

Função com variáveis

```
def function(x,y):
        return (x*y+6*x)
```

return vs. print

Função com variáveis

```
def f(x,y):
        A = x*y+6*x
        return A
```

É necessário usar *return* em vez de *print* para que depois o resultado possa ser reutilizado.

Exemplo de reutilização incorreto

```
def retomar():
        return A+1 #o que acontece dentro da função fica na função
```

Exemplo de reutilização correto

```
valor = f(···,···)
def retomar():
        return (valor+1)
```

Importar bibliotecas

Para várias funcionalidades, pode ser necessário importar bibliotecas (funcionalidades adicionais). A mais comum é a biblioteca *math*. Comandos da biblioteca têm o nome da biblioteca como prefixo.

Cálculo de uma raiz quadrada (com e sem a biblioteca)

```
import math
def f(n):
        return (math.sqrt(n), n**(1/2))
```

Pode-se também importar uma biblioteca sem prefixos.

Cálculo de uma raiz quadrada

```
from math import * #vai retirar os prefixos
def f(n):
        return sqrt(n)
```

Biblioteca *turtle*

Uma outra biblioteca popular é a biblioteca *turtle*. Esta permite controlar um *robot* virtual (uma tartaruga) e desenhar imagens. Também é possível importar bibliotecas e dar-lhes um prefixo diferente.

Exemplo de um comando com *turtle*

```
import turtle as tt
def draw(n,α):
        tt.clear() #limpar janela
        tt.forward(n) #andar para a frente n unidades
        tt.left(α) #virar para a esquera α graus
        tt.forward(2n) #andar para a frente 2n unidades
```

Comandos *turtle* principais

- forward(n) andar para a frente n unidades
- backward(n) andar para trás n unidades
- left(α) virar para a esquerda α graus
- color(c) mudar a cor da caneta
- pensize(n) mudar a largura do traço
- penup() levantar caneta
- pendown() baixar caneta
- speed(n) mudar a velocidade da tartaruga (sendo $n = 0$ o mais rápido)
- clear() limpar janela
- reset() limpar janela e re-inicializar tartaruga

Biblioteca *random*

Dá-nos a capacidade de gerar números pseudo-aleatórios. Chamam-se pseudo-aleatórios pois são gerados com base numa *seed* que se comporta de uma determinada forma.

Importar biblioteca

```
import random as rd
```

Seguem-se abaixo os comandos mais úteis.

- rd.random() gera um número pseudo-aleatório entre 0 e 1
- rd.randint(*a*,*b*) gera um inteiro pseudo-aleatório entre *a* e *b*
- rd.seed(n) muda a *seed* para um determinado valor *n*

Ao definir uma determinada *seed* numa função, ao usar *rd.random()*, serão gerados sempre os mesmos valores.

Condicionais

Usam-se condicionais para testar se algo é verdadeiro.

Exemplo de condicionais

```
def f(n):
        if n==1:
                return True
        elif n<1 or n>2: #elif = else if
                return False
        else:
                return None
```

- == igual a
- != diferente de
- < menor que
- > maior que
- <= menor ou igual a
- >= maior ou igual a
- *and* e
- *or* ou
- *not* contrário de
- *in* contido em

Ciclos

É possível criar um ciclo (*loop*) em python. A primeira forma é usando o comando *for* (usado para repetir algo um número determinado de vezes - pára quando todos os itens tiverem a sua vez); a segunda é usando o comando *while* (corre o comando enquanto uma certa condição for verdadeira).

Ciclo *for*

```
for i in "Olá":
        print(i)
```

Ciclo *while*

```
loop=1
while loop==1:
        print("Olá") #a condição será sempre verdadeira e o loop é infinito - é bastante comum usar: while True (condição sempre verdadeira)
```

Mais informação sobre ciclos

O mais comum é fazer com que o ciclo *for* se repita num dado intervalo; para isso, é usado o comando *range*.

range

```
for i in range(50): #vai-se repetir 50 vezes
        print("Olá")
```

- range(n): números inteiros de 0 a $n-1$
- range(i,n): inteiros de i a $n-1$
- range(i,n,d): inteiros i, $i+d$, $i+2d$, ... inferiores a n

Existem dois comandos bastante importantes quando se fala de ciclos:

- continue passar logo à próxima iteração
- break sair do ciclo (muito útil em comandos *while*)

Listas e cadeias de caracteres

Em python é possível trabalhar com listas e com cadeias de caracteres (*strings*); sendo que estes dois têm as mesmas propriedades básicas.

Operações com listas e com cadeias de caracteres

```
>>> len("Olá") #comprimento
3
>>> "Olá"+"Mundo" #somar
"OláMundo"
>>> 3*"Olá" #repetição
"OláOláOlá"
>>> l in "Olá" #pertença
True
>>> for i in "Olá": #iteração
        print(i)
"O"
"l"
"á"
```

Comandos com cadeias de caracteres

Para qualquer cadeia de caracteres associada a txt:

- txt[i] retoma o termo nessa posição (começando em 0)
- txt[-i] números negativos começam a contar do fim
- txt[i:j] retoma od termos de *i* a *j*
- txt[i:] retoma os termos a partir de *i*
- txt[:j] retoma os termos até *j*
- txt=txt[:2]+"n"+txt[3:] Substitui o 3º termo por "n"
- ord("A") indica o valor numérico associado com a letra
- "A"<"B" Compara os valores numéricos
- txt.find("...") procura a primeira vez que "..." aparece
- txt.replace("a","u") substitui todos os "a" por "u"
- txt.lower() coloca tudo em minúscula
- txt.upper() coloca tudo em maiúscula

Informação sobre listas e tuplos

Exemplo de uma lista: [1,2,3,4,5]

Ao contrário de cadeias de caracteres, pode-se manipular os termos facilmente: t[0]=1, ou até mesmo: t[0:1]=[1,2,3] para acrescentar n elementos. Fazer somente t[0]=[1,2,3] vai acrescentar uma lista dentro da lista. Os comandos usados para mostrar os termos também se aplicam a listas.

Uma lista pode ter dois nomes mas continuar a ser uma só lista. ou seja, se $a = [1, 2, 3]$ e se $b = a$, ao alterar a, b também será alterado. Para que isto não aconteça, deve-se fazer $b = a[:]$.

Exemplo de um tuplo: (1,2,3,4,5) ou 1,2,3,4,5

Ao contrário de listas, tuplos são imutáveis e não se podem manipular facilmente. Funções usam tuplos, ex: $f(x, y)$. Fazer $x, y = 5, 7$ é o mesmo que fazer $x = 5$ e $y = 7$.

Comandos com listas

Para qualquer lista associada a lst:

- sum(lst) soma todos os elementos da lista
- lst.append("...") acrescenta "..." ao fim da lista
- lst.remove("...") remove todos os "..." da lista
- del lst[n] apaga o elemento na posição *n*
- lst.insert(n,"...") acrescenta "..." à posição *n* na lista
- lst.sort() ordena a lista por ordem crescente

- list(a) converte o tuplo *a* numa lista
- tuple(a) converte a lista *a* num tuplo

Mais comandos essenciais

- Em vez de i=i+k, escrever: i += k
- Em vez de i=i-k, escrever: i -= k
- Em vez de i=i*k, escrever: i *= k
- Em vez de i=i**k, escrever: i **= k
- Em vez de i=i/k, escrever: i /= k
- Em vez de i=i//k, escrever: i //= k

Numa cadeia de caracteres, \t acrescenta tabulação e \n cria uma nova linha.

Por vezes, pode ser muito útil converter números em cadeias de caracteres para os poder manipular (e só depois os voltar a converter em números). Para um número *n*: int(str(n)) para números inteiros ou float(str(n)) para decimais.

Dicionários

Dicionários são tabelas de associação - cada chave corresponde um só valor. Ao colocar um certo elemento no dicionário, este pode ser seguido por ":", associando-o ao termo seguinte. Listas são criadas com [], tuplos com (), dicionários com {}. Tal como listas, estes podem ser alterados.

Exemplo de um dicionário: dic = {"A":3,"B":7}.
Dicionários, em geral, não mantêm a ordem em que foram introduzidos; porém, não interessa, pois dicionários servem para criar correspondências - ordem não importa.

- dic.keys() mostra somente as chaves do dicionário
- dic[...] indica o item correspondente à chave (ex: dic["A"] retoma 3 e dic["B"] retoma 7)

O comando for i,x in dic.items() irá percorrer todas as chaves e correspondências do dicionário ao mesmo tempo.

Formatação de texto

É possível formatar texto fazendo algo como: "%05.1f" % 24

Dentro de aspas (cadeia de caracteres) começa-se por colocar %. De seguida, indica-se o tamanho da indentação (no exemplo acima, a indentação será de 5 caracteres; porém, como vamos querer indentar o número 24 - 2 caracteres - só será indentado 3 caracteres). Se quisermos que a indentação seja preenchida por zeros, colocamos 0 entre o % e o tamanho da indentação.
Podemos colocar um ponto e segui-lo pelo número desejado de casas decimais.
Acabamos por colocar uma letra: **d** para números inteiros, **f** para *float* e **s** para cadeias. Podemos depois seguir com texto normal que será mostrado. Para terminar, coloca-se % e o item a formatar.

Exemplo de uma formatação

```
>>> "Esta é uma data: %02d/%02d%4d" % (1,6,2013)
Esta é uma data: 01/06/2013
```

Recursividade

Uma função pode executar outras funções no seu processo; porém, também se pode executar a si mesma.

Se a função B estiver dentro da função A, é necessário chegar ao *return* da função B para regressar à parte da função A em que se parou.

Método de resolução de problemas

- **Compreender**: deve-se entender o que está a ser pedido;
- **Planear**: antes de passar à construção do código, pensar como se poderia resolver o problema sem ajuda de computadores e criar um modelo de resolução (algoritmo) - deve-se decompor o problema em funções mais simples e objetivas;
- **Executar (programar)**: deve-se usar a linguagem de programação para criar uma função que execute o algoritmo definido - é aqui que ocorre a maior parte dos erros;
- **Avaliar**: existe possibilidade do programa não fazer o pretendido (pode-se testar isso com a biblioteca *doctest*); sendo assim, deve-se reler e simplificar o código o mais possível - também se deve pensar noutras formas de resolver o problema, criar um código para cada uma dessas e comparar com o código que tínhamos.

Correção de erros

Muitas vezes, o python vai retomar um erro. Nesses momentos, é importante ter **calma** lembrar: o computador está sempre certo; se existe um erro, a culpa é do programador. Aqui seguem algumas dicas de modo a resolver tais erros:

- O python vai indicar a linha onde o erro foi detetado, mas a causa do erro pode estar em linhas anteriores;
- Deve-se tentar encontrar onde pode estar o erro, entender a sua causa e pensar como pode ser resolvido;
- Manipular o código, não à sorte, mas sim a pensar como isso irá resolver o erro.

Criar erros

Podemos, no entanto, querer que a função retome um erro, mesmo quando tal não era suposto acontecer.

É possível colocar um condicional na função que force a aparecimento de um erro usando o comando: raise *tipo de erro*.

Exemplo do comando *raise*

```
if lista==[]:
        raise ValueError("A lista não pode estar vazia")
```

Pode-se também criar um *AssertError*. O comando *assert* testa se a condição é verdadeira e, se for falsa, retoma um erro.

Exemplo do comando *assert*

```
assert n>0, "Os valores têm de ser positivos"
```