

Folha 9 - Testar e depurar

Python v 3.*

Exercícios

- 1) Recorde a função que calcula a soma dos divisores de um número inteiro:

```
def soma_divisores(n):  
    """  
    Soma dos divisores de um número inteiro dado  
  
    Requires: n int > 0  
    Ensures: um int correspondente à soma dos divisores  
    de n que sejam maiores que 1 e menores que n  
    >>> soma_divisores(4)  
    2  
    """  
    soma = 0  
    for i in range(2, n):  
        if n % i == 0:  
            soma += i  
    return soma
```

- a) Identifique na *docstring* um exemplo de um caso de utilização da função.

- i) O que está a ser testado?
- ii) Qual o valor de entrada para o teste?
- iii) Qual o valor esperado?

- b) O que significa a seguinte sequência de comandos?

```
import doctest  
doctest.testmod()
```

- c) Alterando-se o valor esperado do teste para 4 obtemos o seguinte resultado.
O que pode concluir?

```
*****
File "/Users/fmartins/Desktop/soma_divisores.py", line 6, in
__main__.soma_divisores
Failed example:
    soma_divisores(4)
Expected:
    4
Got:
    2
*****
1 items had failures:
  1 of  1 in __main__.soma_divisores
***Test Failed*** 1 failures.
```

- d) Efetuemos testes unitários de cobertura de código baseados no fluxo de execução da função:
- i) Escreva um teste que faça com que a função não entre no ciclo;
 - ii) Escreva outro teste que faça com que o fluxo da função execute um só passo do ciclo;
 - iii) Escreva um último teste em que a função execute mais de um passo ao ciclo.

- 2) As funções seguintes determinam se um dado número é perfeito (cuja definição já foi apresentada numa aula anterior) e a lista de números perfeitos até um dado número, recorrendo à função `soma_divisores`.

```
def perfeito(numero):
    return soma_divisores(numero) + 1 == numero

def lista_perfeitos(numero):
    perfeitos = []
    for n in range(numero):
        if perfeito(n):
            perfeitos.append(n)
    return perfeitos
```

- a) Escreva os contratos das funções `perfeito` e `lista_perfeitos`.
- b) As funções `perfeito` e `lista_perfeitos` satisfazem os contratos das funções que invocam?
- c) O que significa os seguintes testes unitários da função `perfeito` em termos de valores de entrada e valores esperados e qual o seu resultado?

```
>>> perfeito(4)
False
>>> perfeito(6)
True
```

- d) Escreva testes para a função `lista_perfeitos` que cubra o fluxo de execução da função, em particular:
 - i) Um teste que não entre no ciclo;
 - ii) Um teste que entre no ciclo, execute um único passo e que não entre no `if`;
 - iii) Um teste que entre no ciclo, execute um único passo e que entre no `if`;
 - iv) Um teste que entre no ciclo, execute vários passos do ciclo, entre no `if` e depois termine;
 - v) E por último, um teste que entre no ciclo, execute vários passos do ciclo, não entre no `if` e depois termine;

Notas: a) para escrever os valores esperados dos testes tenha em atenção a forma como o Python escreve listas de inteiros. Execute, por exemplo, o comando `print([1,2,3])` e observe cuidadosamente o resultado, incluindo os espaços produzidos;

b) não introduza espaços após cada valor esperado;

c) Como disse E. W. Dijkstra, "*Testing shows the presence, not the absence of bugs*", portanto, testar só aumenta o nível de confiança na correção de um programa.

d) Por vezes não é possível escrever os testes que pretendemos. Estes testes dizem-se "inviáveis".

3) Considere a seguinte função

```
def contar_repetidos_consecutivos(colecao, elemento):  
    """  
    requires: colecao um iterável; elemento um objeto qualquer  
    ensures: um inteiro correspondente ao maior numero de vezes  
    que elemento aparece consecutivamente em colecao  
    >>> contar_repetidos_consecutivos([1,4,4,3,4,2,5,1,1,1,2], 4)  
    2  
    """  
    iguais = 1  
    maior = 0  
    for valor in elemento:  
        if valor == elemento:  
            iguais += 1  
        else:  
            if iguais < maior:  
                maior = valor  
            iguais = 1  
    return maior if maior <= iguais else iguais
```

- Além do teste apresentado, escreva dois testes para contar o número de uns e de dois.
 - Escreva um teste para contar o número de "s" consecutivos na palavra "possível".
 - Escreva dois testes que verifiquem se a função funciona corretamente quando aparecem repetidos no início e no fim da coleção.
 - E como se comportará a função quando a coleção dada estiver vazia.
 - E quando o elemento a procurar não se encontra na sequência?
 - Apresente o resultado da execução de cada teste.
 - Como procederia para corrigir os problemas da função?
 - Onde incluiria instruções de impressão para averiguar os problemas da exibidos pela função durante a execução de testes?
 - Faça uma sessão de depuração utilizando o *idle* para identificar os problemas da função.
- 4) A função seguinte informa a posição onde um elemento se encontra numa sequência ou devolve `None` caso o elemento não pertença à sequência.

```
def encontra(sequencia, valor):  
    i = 0
```

```
for elemento in sequencia:
    if valor == elemento:
        return valor
    else:
        return None
```

- a) Elabore testes à função que explore os vários valores possíveis que os parâmetros `sequencia` e `valor` podem tomar, em particular,
- i) `sequencia` sem elementos;
 - ii) `sequencia` com um só elemento e `valor` não pertencente a `sequencia`;
 - iii) `sequencia` com um só elemento e `valor` a pertencer a `sequencia`;
 - iv) `sequencia` com vários elementos e `valor` não pertencente a `sequencia`;
 - v) `sequencia` com vários elementos e `valor` o primeiro valor da `sequencia`;
 - vi) `sequencia` com vários elementos e `valor` um dos valores do meio;
 - vii) `sequencia` com vários elementos e `valor` o último valor da `sequencia`.
- Nota:** a) nos testes use diversos tipos de sequências, como por exemplo, lists, tuples, strings, dictionaries; b) nos testes, o valor esperado `None` é indicado não se incluindo nada no valor esperado.
- b) Qual o resultado da execução de cada teste?
- c) Identifique quais os testes que não têm sucesso; para cada um destes encontre o defeito da função que provoca a falha do teste.
- d) Apresente a função corrigida.

Problemas

- 1) Considere a seguinte função que efetua a diferença de duas listas tal como se descreve a seguir.

```
def diferenca(l1, l2):  
    """  
    Diferença entre duas listas dadas  
  
    requires: l1 list; l2 list  
    ensures: devolve uma list que contem os elementos de l1 que não estão em  
    l2. As listas l1 e l2 não são alteradas.  
    """  
    result = l1  
    i = 1  
    while i < len(l1):  
        if i in l2:  
            result.remove(i)  
        i += 1  
    return result
```

- a) Escreva testes unitários para a função tendo em conta um critério apropriado.
- b) Ao executar os testes, o que pode concluir sobre a correção da função? Consegue escrever algum teste que falhe?
- c) Faça o teste e depuração da função com vista a determinar o defeito (ou defeitos) que a função possui. Recorra à impressão de valores intermédios para tentar identificar os defeitos.

- 2) Considere os seguintes testes unitários de uma função f .

```
>>> f (1,1,0)  
[[0]]  
>>> f (3,3,1)  
[[1, 1, 1], [1, 1, 1], [1, 1, 1]]  
>>> f (1, 5, 0)  
[[0, 0, 0, 0, 0]]  
>>> f (5, 1, 2)  
[[2], [2], [2], [2], [2]]
```

- a) Implemente uma função `f` que passe os testes acima.
- b) Descreva o que faz a função que desenvolveu e elabore o seu contrato.

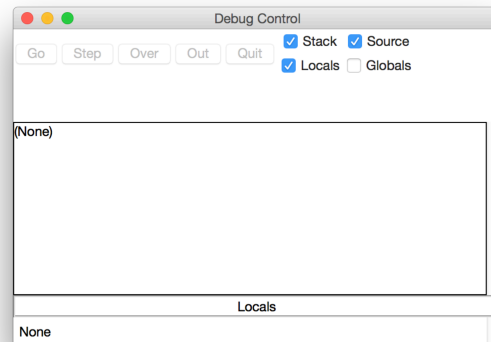
3) Considere a função seguinte que obtém uma dada coluna de uma matriz. Uma matriz pode ser representada com uma lista de listas todas com o mesmo comprimento.

```
def obter_coluna(matriz, col):  
    """  
    Coluna de uma matrix dada num índice dado  
  
    Requires: matriz uma lista de listas em que todas as listas internas  
    tem o mesmo comprimento; col int>=0 e < len(matriz[0]), representando  
    o índice de uma coluna  
    Ensures: uma lista correspondente à col-ésima coluna da matriz  
    """  
    return [matriz[i][coluna] for i in range(len(matriz))]
```

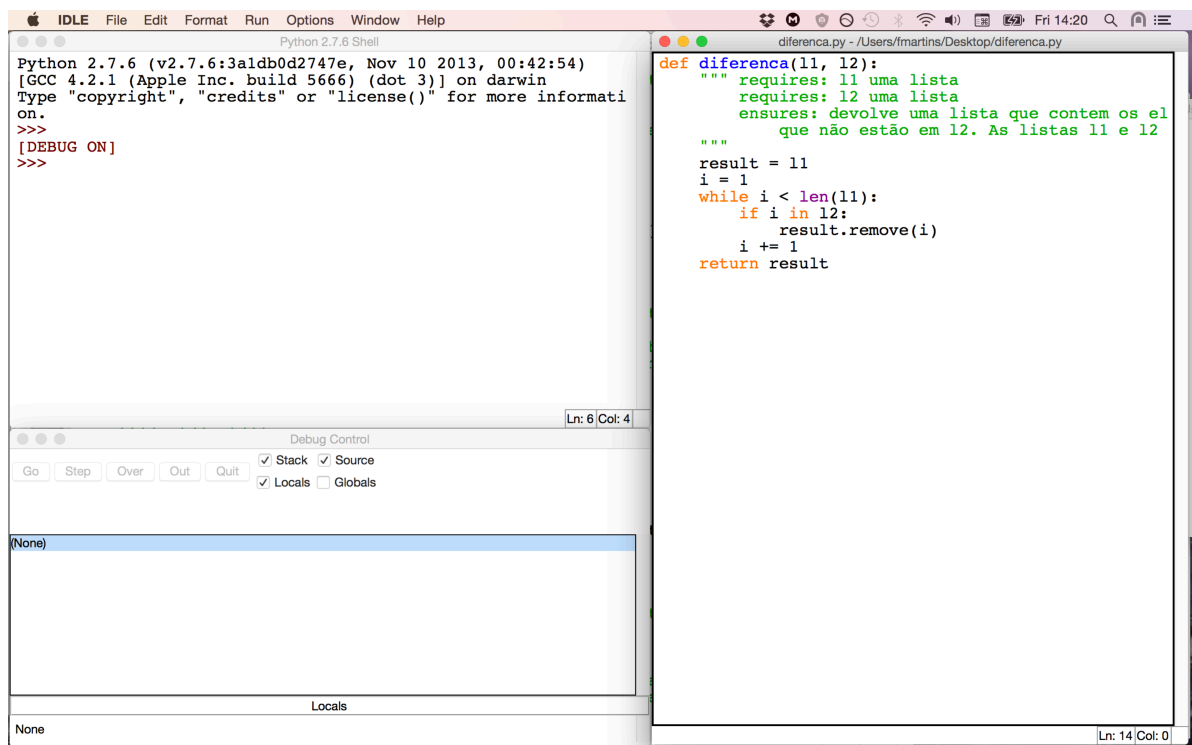
- a) Escreva testes convenientes para a função, explorando os valores possíveis para matriz e coluna, sem violar o contrato. Por exemplo, tente com a lista vazia, com a lista que só tem uma lista de comprimento 0 como elemento, uma lista só com um elemento com comprimento maior do que zero, uma lista com vários elementos todos com comprimento 1, etc.
- b) Reescreva a função utilizando ciclos `for`.
- c) Elabore testes que exercitem o maior número de fluxos de execução possíveis da função desenvolvida em b). Para os ciclos, elabore um teste que não entre no ciclo, outro que entre no ciclo e execute um passo, e outro que execute vários passos.

4) Vamos efetuar uma sessão de depuração da função do exercício 1) recorrendo ao programa `Idle`.

- a) Carregue o ficheiro com a função do exercício anterior;
- b) Para iniciar o modo depuração no programa `Idle`, deve seleccionar a janela do interpretador (*Shell*) e no menu `Debug` escolher a opção `Debugger`. Deverá obter a seguinte janela. Selecione as opções *Stack*, *Locals* e *Source*, caso ainda não se encontrem seleccionadas;

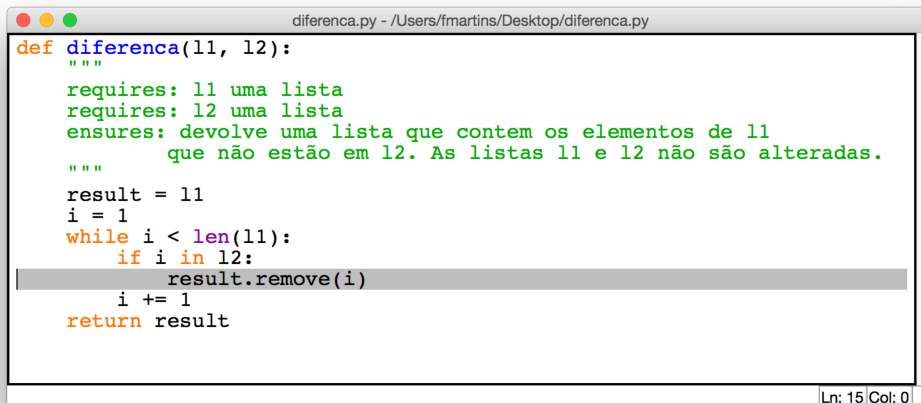


- c) Selecione a janela onde está a editar o seu programa e execute o programa (F5);
- d) Arranje as janelas de forma semelhante à próxima figura;



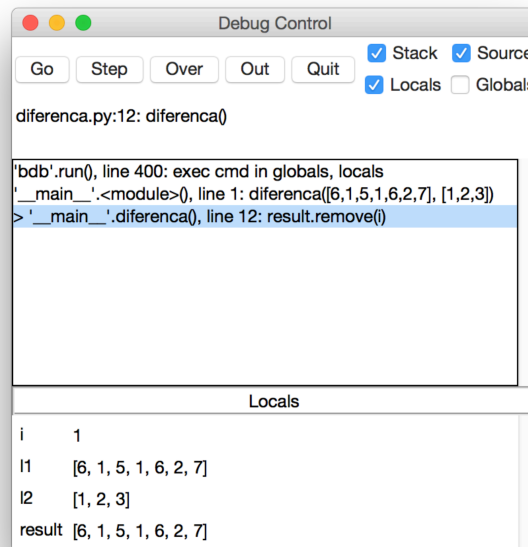
- e) Invoque a função como se segue:
`diferenca([6,1,5,1,6,2,7], [1,2,3])`

- f) Utilize o botão *Step* para fazer o programa avançar passo a passo. Note que à medida que o programa avança a sua execução uma linha sombreada marca a próxima instrução do código fonte que será executada.
- g) Prima agora o botão *Go* para que o programa execute até ao fim. O quê, uma mensagem de erro? O objetivo da nossa próxima execução será descobrir o que causou o defeito usando este método de depuração.
- h) Invoque de novo a função com os mesmos argumentos.
- i) Prima cinco vezes no botão *Step*. Analisemos em detalhe cada uma das janelas envolvidas na depuração.
 - i) A janela de edição de código fonte marca a linha que está a ser executada



```
def diferenca(l1, l2):  
    """  
    requires: l1 uma lista  
    requires: l2 uma lista  
    ensures: devolve uma lista que contem os elementos de l1  
             que não estão em l2. As listas l1 e l2 não são alteradas.  
    """  
    result = l1  
    i = 1  
    while i < len(l1):  
        if i in l2:  
            result.remove(i)  
            i += 1  
    return result
```

- ii) A janela de controlo de depuração ("*Debug control*") está dividida em 3 partes. A parte superior contém os botões de controlo da depuração: *Go*, executa a depuração até ao próximo ponto de interrupção (ver um exemplo mais tarde) ou até ao fim do programa; *Step*, executa a próxima instrução (no caso de ser a chamada a uma função, continua a depuração dentro da função); *Over*, executa a próxima instrução (no caso da ser a chamada a uma função, executa a função, sem mostrar a depuração do corpo da função) e continua na instrução seguinte; *Out*, executa a depuração até ao próximo ponto de interrupção, ou até a função terminar, continuando na instrução a seguir à chamada da função; *Quit*, termina a execução imediata do programa. Do lado direito, no topo, pode selecionar-se a informação a mostrar durante a depuração. No presente exemplo vemos o pilha de chamada, as variáveis locais e está assinalada a próxima instrução fonte a ser executada. A visualização de variáveis globais não está selecionada, porque os nossos programas não têm variáveis globais! ;)



A informação central indica-nos o nome do ficheiro, a linha e o nome da função a ser executada e a seguir, dentro do retângulo, a pilha de chamada. Neste caso, o `__main__`, na linha 1, chamou a função `diferenca` com os argumentos `[6,1,5,1,6,2,7]` e `[1,2,3]`. A função a ser executada no momento é a função `diferenca` (do módulo `__main__`), e a sua execução vai na linha 12, que corresponde à instrução `result.remove(i)`. Por último, a parte inferior da janela lista as variáveis locais e os seus valores.

- iii) A janela do interpretador contem a informação usual e neste caso em particular a novidade é a menção a `[DEBUG ON]` a indicar que o modo de depuração está ativado.
- j) Continue a depuração (ou repita-a novamente) até identificar o que está a causar a exceção. Caso prima `Step` ou `Over` para além do “fim” do seu programa e comece a fazer depuração do próprio *Python Shell*, prima `Go` e feche a janela, pois não queremos que descubra defeitos no *Python Shell* durante a aula ;)
- k) Vamos de seguida incluir um ponto de interrupção na função. Para tal, seleccione a linha onde se pretende interromper a execução do programa. Em Windows, premir o botão do lado direito do rato, em OS X fazer *ctrl-click*, em Linux usar o terceiro botão (mas depende da distribuição). O objetivo é fazer abrir o menu de contexto seguinte

Cut
Copy
Paste
Set Breakpoint
Clear Breakpoint

neste é possível definir ou eliminar pontos de interrupção. Os pontos de interrupção são importante porque evitam perder tempo a executar passo a passo o programa até chegarmos ao ponto que se pretende analisar.

- l) Estabeleça um ponto de interrupção na instrução `i += 1` da função. Invoque a função, prima o botão *Go* e note que o programa para na instrução requerida. Continue a depuração a partir deste ponto.
- m) Quando terminar a depuração anterior, elimine o ponto de interrupção, invoque novamente a função, prima o botão *Go* e verifique que desta vez a função executou até ao fim.
- n) Efetue a depuração da função de forma a eliminar o maior número de problemas. Tome especial atenção ao facto da função ter de honrar o seu contrato, em particular a sua pós-condição.

5) Considere a seguinte especificação de uma função:

```
def obter_diagonal(matriz, linha, coluna, direc):  
    """  
    Requires: matriz uma lista de listas em que todas as listas internas  
    tem o mesmo comprimento.  
    Requires: linha um inteiro >= 0 e < len(matriz)  
    Requires: coluna um inteiro >= 0 e < len(matriz[0])  
    Requires: direc um inteiro = 1, que significa uma diagonal direita que  
    se prolonga de cima para baixo, da esquerda para a direita; ou -1,  
    que significa uma diagonal esquerda que se prolonga de cima para  
    baixo, da direita para a esquerda  
    Ensures: uma lista que corresponde a diagonal de matriz que passa pela  
    posicao linha, coluna com a direcao direc  
    """
```

- a) Sem programar a função escreva casos de teste para todas as situações possível para a matriz `[[1,2,3,4], [5,6,7,8], [9,10,11,12]]` e guarde-os num ficheiro próprio. Ao todo são 24 casos de teste. Segue o exemplo do caso de teste para a posição 0, 0.

```
>>> obter_diagonal ([[1,2,3,4],[5,6,7,8],[9,10,11,12]], 0, 0, 1)  
[1, 6, 11]
```

b) Verifique se a função seguinte passa os testes que escreveu.

```
def obter_diagonal(matriz, linha, coluna, direc):  
    if direc == 1:  
        l_i = linha - min (linha, len(matriz[0]) - 1 - coluna)  
        c_i = coluna + min (linha, len(matriz[0]) - 1 - coluna)  
        c_f = coluna - min (len(matriz) - linha, coluna)  
    else:  
        l_i = max (0, linha - coluna)  
        c_i = max (0, coluna - linha)  
        c_f = coluna + min(len(matriz) - linha, len(matriz[0]) - coluna)  
    return [matriz[l_i+i][c_i+i*direc] for i in range(abs(c_f-c_i))]
```

- c) Faça depuração por bissecção do código para tentar identificar os problemas da função, recorrendo apenas a comandos de escrita (e sem recorrer ao depurador do *idle*).
- d) Programe a sua própria versão da função e garanta que esta passa os testes que escreveu.

Terminologia

Português

- sequência de testes
- teste de caixa negra
- teste de caixa transparente
- caminho-completo
- teste unitário
- testes de integração
- piloto de teste
- coto
- defeito
- depurar
- teste de regressão

Inglês

test suite
black-box testing
glass-box testing
path-complete
unit testing
integration testing
test driver
stub
bug
debugging
regression testing