TÉCNICO LISBOA

Fundamentos da Programação

Solução do segundo exame

29 de Janeiro de 2016

1. Indique se cada uma das seguintes afirmações é verdadeira ou falsa. No caso de ser falsa, justifique de forma sucinta.

   (a) **(0.5)** Uma das características da abstração procedimental é a exigência de sabermos como uma função executa a sua tarefa.
   Resposta:
   Falsa, a abstracção procedimental consiste exactamente no contrário, sabermos o que uma função faz, sem nos preocuparmos do modo como ela executa a sua tarefa.

   (b) **(0.5)** Um objecto é uma entidade com estado interno cujo comportamento é ditado por um conjunto de métodos pertencentes ao objecto.
   Resposta:
   Verdadeira.

2. Considere a seguinte definição de função:

```
def soma_me(x):
    return x + x
```

   Indique, justificando, qual o resultado obtido nas seguintes chamadas à função. Se uma destas chamadas der origem a um erro, explique a razão do erro.

   (a) **(0.5)**

```
>>> soma_me(3)
```

   Resposta:

```
6
```

   O "+" corresponde à adição de inteiros.

   (b) **(0.5)**

```
>>> soma_me('to')
```

   Resposta:

```
'toto'
```

   O "+" corresponde à concatenação de cadeias de caracteres.

(c) **(0.5)**

```
>>> soma_me({'a': 3})
```

Resposta:

```
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

A operação "+" não está definida para dicionários.

3. Quando se efectua um depósito a prazo de uma quantia q com uma taxa de juros j ($0 < j < 1$), o valor do depósito ao fim de n anos é dado por:

$$q \times (1 + j)^n$$

(a) **(1.0)** Escreva em Python a função `valor` que recebe como argumentos um número inteiro positivo q correspondente à quantia depositada, um real j no intervalo $]0, 1[$ correspondente à taxa de juros e um inteiro positivo n correspondente ao número de anos que o dinheiro está a render, e, verificando a correcção dos argumentos, devolve um real correspondente ao valor do depósito ao fim desse número de anos. Caso os argumentos não estejam correctos, deverá gerar um erro. Por exemplo,

```
>>> valor(100, 0.03, 4)
112.55088100000002
```

Resposta:

```
def valor(q, j, n):
    if not (isinstance(q, int) and q > 0):
        raise ValueError ('valor: a quantia deve ser um inteiro > 0')
    if not (isinstance(j, float) and 0 < j < 1):
        raise ValueError ('valor: a taxa deve ser um real entre 0 e 1')
    if not (isinstance(n, int) and n > 0):
        raise ValueError ('valor: anos deve ser um inteiro > 0')

    return q * (1 + j) ** n
```

(b) **(1.5)** Usando a função da alínea anterior, escreva uma função que calcula ao fim de quantos anos consegue duplicar o seu dinheiro. Não é necessário validar os dados de entrada. Por exemplo,

```
>>> duplicar(100, 0.03)
24
```

Resposta:

```
def duplicar(q, j):
    n = 1
    quant = q
    while quant < 2 * q:
        n = n + 1
        quant = valor(q, j, n)
    return n
```

4. Um método básico para codificar um texto corresponde a isolar os caracteres nas posições pares para um lado e os caracteres nas posições ímpares para outro, juntando depois as duas partes anteriormente obtidas. Por exemplo, o texto `abcde` é codificado por `acebd`.

(a) (1.5) Defina uma função que codifica uma cadeia de caracteres de acordo com o algoritmo apresentado. Não é necessário validar os dados de entrada. Por exemplo,

```
>>> codifica('abcde')
'acebd'
```

Resposta:

```
def codifica(texto):
    pares = ''
    impares = ''

    for i in range(len(texto)):
        if i % 2 == 0:
            pares = pares + texto[i]
        else:
            impares = impares + texto[i]

    return pares + impares
```

(b) (1.5) Defina uma função que descodifica uma cadeia de caracteres de acordo com o algoritmo apresentado. Não é necessário validar os dados de entrada. Por exemplo,

```
>>> descodifica('acebd')
'abcde'
```

Resposta:

```
def descodifica(texto):
    res = ''
    pares = 0
    if len(texto) % 2 == 0:
        impares = len(texto) // 2
    else:
        impares = len(texto) // 2 + 1

    while impares < len(texto):
        res = res + texto[pares] + texto[impares]
        pares = pares + 1
        impares = impares + 1

    if len(texto) % 2 != 0:
        res = res + texto[pares]

    return res
```

5. Um número diz-se perfeito se for igual à soma dos seus divisores (não contando o próprio número). Por exemplo, 6 é perfeito porque $1 + 2 + 3 = 6$.

(a) **(1.5)** Usando recursão de cauda, escreva a função `perfeito` que recebe como argumento um número inteiro e tem o valor `True` se o seu argumento for um número perfeito e `False` em caso contrário. Não é necessário validar os dados de entrada.

Resposta:

```
def perfeito(n):

    def perfeito_aux(n, div, soma):
        if div == n:
            return n == soma
        elif n % div == 0:
            return perfeito_aux(n, div + 1, soma + div)
        else:
            return perfeito_aux(n, div + 1, soma)

    return perfeito_aux(n, 1, 0)
```

(b) **(1.5)** Usando recursão com operações adiadas e a função `perfeito` da alínea anterior, escreva a função `perfeitos_entre` que recebe dois inteiros positivos e devolve a lista dos números perfeitos entre os seus argumentos, incluindo os seus argumentos. Não é necessário validar os dados de entrada. Por exemplo:

```
>>> perfeitos_entre(6, 30)
[6, 28]
```

Resposta:

```
def perfeitos_entre(inicio, fim):
    if inicio > fim:
        return []
    elif perfeito(inicio):
        return [inicio] + perfeitos_entre(inicio + 1, fim)
    else:
        return perfeitos_entre(inicio + 1, fim)
```

6. **(1.0)** Considere os tipos abstractos de dados (TADs) utilizados no segundo projecto e a seguinte implementação de algumas das operações básicas desses TADS:

```
def cria_coordenada(x, y):
    if not(isinstance(x, int) and isinstance(y, int)):
        raise ValueError('cria_coordenada: argumentos invalidos')
    if x <= 0 or y <= 0:
        raise ValueError('cria_coordenada: argumentos invalidos')
    return (x, y)

def coordenada_linha(c):
    if not e_coordenada(c):
        raise ValueError('coordenada_linha: argumento invalido')
    return c[0]

def coordenada_coluna(c):
    if not e_coordenada(c):
        raise ValueError('coordenada_linha: argumento invalido')
    return c[1]

def cria_tabuleiro(conf):
    if not verifica_conf(conf):
        raise ValueError('cria_tabuleiro: argumentos invalidos')
```

```
    # Um tabuleiro e' representado por uma lista com a dimensao, a
    # especificacao e a matriz quadrada (inicializada a 0).
    return [len(conf[0]), conf, \
        [[0 for x in conf[1]] for x in conf[0]]]

def tabuleiro_preenche_celula(t, c, v):
    if not(e_tabuleiro(t) and e_coordenada(c)):
        raise ValueError('tabuleiro_preenche_celula: args invalidos')
    dim = tabuleiro_dimensoes(t)[0]
    if not(1 <= c[0] <= dim and 1 <= c[1] <= dim and v in (0, 1, 2)):
        raise ValueError('tabuleiro_preenche_celula: args invalidos')
    t[2][c[0] - 1][c[1] - 1] = v
    return t
```

Assinale no código onde é que são violadas as barreiras de abstracção em relação ao TAD `coordenada`. Justifique, indicando como corrigir o código.

Resposta:

No código acima ocorre violação das barreiras de abstracção na função `tabuleiro_preenche_celula` quando recorremos à representação interna da coordenada. Basta substituir essas ocorrências por chamadas aos selectores `coordenada_linha` e `coordenada_coluna`.

7. **(1.5)** Escreva uma função, `metabolismo`, que recebe um dicionário cujas chaves correspondem a nomes de pessoas e cujos valores correspondem a tuplos, contendo o género, a idade, a altura e o peso dessa pessoa. A sua função devolve um dicionário que associa a cada pessoa o seu índice de metabolismo basal. Não é necessário validar os dados de entrada. Sendo s o género, i a idade, h a altura e p o peso de uma pessoa, o metabolismo basal, m, é definido do seguinte modo:

$$m(s,i,h,p) = \begin{cases} 66 + 6.3 \times p + 12.9 \times h + 6.8 \times i & \text{se } s = M \\ 655 + 4.3 \times p + 4.7 \times h + 4.7 \times i & \text{se } s = F \end{cases}$$

Por exemplo:

```
>>> d = {'Maria' : ('F', 34, 1.65, 64), 'Pedro': ('M', 34, 1.65, 64),
         'Ana': ('F', 54, 1.65, 120), 'Hugo': ('M', 12, 1.82, 75)}
>>> metabolismo(d)
{'Ana': 1458.955, 'Hugo': 675.078, 'Maria': 1109.755, 'Pedro': 736.685}
```

Resposta:

```
def metabolismo(dic_p):
    res = {}
    for e in dic_p:
        if dic_p[e][0] == 'M':
            res[e] = 66 + 6.3 * dic_p[e][1] + 12.9 * dic_p[e][2] \
                     + 6.8 * dic_p[e][3]
        else:
            res[e] = 655 + 4.3 * dic_p[e][1] + 4.7 * dic_p[e][2] \
                     + 4.7 * dic_p[e][3]
    return res
```

8. Defina os seguintes funcionais sobre listas:

(a) **(0.5)** `filtra(fn, lst)` que devolve uma lista obtida a partir de `lst` mas que apenas contém os elementos que satisfazem o predicado `fn`.
Resposta:

```
def filtra(fn, lst):
    if lst == []:
        return lst
    elif fn(lst[0]):
        return [lst[0]] + filtra(fn, lst[1:])
    else:
        return filtra(fn, lst[1:])
```

(b) **(0.5)** `transforma(fn, lst)` que devolve uma lista obtida a partir de `lst` cujos elementos correspondem à aplicação da função `fn` aos elementos de `lst`.
Resposta:

```
def transforma(fn, lst):
    if lst == []:
        return lst
    else:
        return [fn(lst[0])] + transforma(fn, lst[1:])
```

(c) **(0.5)** `acumula(fn, lst)` que devolve o valor obtido da aplicação da função `fn` a todos os elementos de `lst`.
Resposta:

```
def acumula(fn , lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return fn(lst[0], acumula(fn, lst[1:]))
```

9. **(1.5)** Usando alguns ou todos os funcionais sobre listas da pergunta anterior, escreva a função `soma_quadrados_impares`, que recebe uma lista de inteiros e devolva a soma dos quadrados dos seus elementos ímpares. A sua função deve conter apenas uma instrução, a instrução `return`. Não é necessário validar os dados de entrada. Por exemplo:

```
soma_quadrados_impares([1, 2, 3, 4, 5, 6])
35
```

Resposta:

```
def soma_quadrados_impares(lst):
    return acumula(lambda x, y: x + y, \
                   transforma(lambda x: x * x, \
                              filtra(lambda x: x % 2 != 0, lst)))
```

10. Considere a seguinte assinatura para o TAD *conjunto* com os significados óbvios:

- $novo\_conjunto : \{\} \mapsto conjunto$
- $insere : elemento \times conjunto \mapsto conjunto$
- $elem\_conj : conjunto \mapsto elemento$ (devolve um elemento escolhido aleatoriamente do conjunto)

- $retira\_conj : elemento \times conjunto \mapsto conjunto$
- $cardinal : conjunto \mapsto inteiro$
- $e\_conjnto : universal \mapsto lógico$
- $e\_conj\_vazio : conjunto \mapsto lógico$
- $conj\_iguais : conjunto \times conjunto \mapsto lógico$
- $pertence : elemento \times conjunto \mapsto lógico$

(a) **(0.5)** Defina uma representação para conjuntos.
Resposta:
O conjunto vazio é representado pela lista vazia. Um conjunto com os elementos e_1, $\ldots$, e_n ($n \geq 1$) é representado pela lista com os elementos $\Re[e_1], \ldots \Re[e_n]$.

(b) **(1.5)** Defina o tipo `conjunto` com base na representação escolhida.
Resposta:

```
from random import random

def conjunto():
    return []

def insere(e, c):
    if e in c:
        return c
    else:
        return c + [e]

def elem_conj(c):
    if c == []:
        raise ValueError('elem_conj: conjunto vazio')
    else:
        return c[int(random()*len(c))]

def retira_conj(e, c):
    if e in c:
        return retira(e, c)
    else:
        return c

def cardinal(c):
    return len(c)

def e_conjunto(arg):

    def sem_els_repetidos(lst):
        if lst == []:
            return True
        elif lst[0] in lst[1:]:
            return False
        else:
            return sem_els_repetidos(lst[1:])

    return isinstance(arg, list) and sem_els_repetidos(arg)

def e_conj_vazio(c):
```

```
    return c == []

def pertence(e, c):
    return e in c

def conj_iguais(c1, c2):
    if c1 == [] and c2 == []:
        return True
    elif c1 == [] or c2 == []:
        return False
    elif c1[0] in c2:
        return conj_iguais(c1[1:], retira(c1[0], c2))
    else:
        return False

# função auxiliar

def retira(e, l):
    if l == []:
        return l
    elif e == l[0]:
        return l[1:]
    else:
        return [l[0]] + retira(e, l[1:])
```

(c) **(1.5)** O usando a abstracção adequada, escreva a função `subconjunto` de dois argumentos correspondentes a conjuntos e que devolve `True` se o primeiro conjunto é um subconjunto do segundo e `False` em caso contrário. Caso os argumentos não estejam correctos, deverá gerar um erro.
Resposta:

```
def subconjunto(c1, c2):
    if e_conjunto(c1) and e_conjunto(c2):
        return subconjunto_aux(c1, c2)
    else:
        raise ValueError ('subconjunto: argumentos não são conjuntos')

    def subconjunto_aux(c1, c2):
        if e_conj_vazio(c1):
            return True
        else:
            e = elem_conj(c1)
            if pertence(e, c2):
                return subconjunto(retira(e, c1), retira(e, c2))
            else:
                return False
```