TÉCNICO LISBOA

Fundamentos da Programação

Solução do Exame

31 de Janeiro de 2018

09:00–11:00

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

   (a) **(0.5)** Deve-se usar um ciclo `while` em vez de um ciclo `for` se for conhecido o número de elementos do iterável que se pretende percorrer.
   Resposta:
   Falso. O ciclo `for` deve ser utilizado quando se pretende percorrer um iterável. O ciclo `while` deve ser utilizado se não se souber, à partida, o número de iterações necessárias.

   (b) **(0.5)** Em classes, o objetivo do método `__repr__` é obter uma representação única do objecto.
   Resposta:
   Verdadeiro. Os objectivos dos métodos em questão estão correctos.

   (c) **(0.5)** A abstração de dados corresponde em abstrair que o programa manipula dados, concentrando-nos apenas no algoritmo do programa.
   Resposta:
   Falso. A abstração de dados corresponde a abstrair-nos da representação dos dados concentrando-nos apenas nas suas propriedades.

2. **(1.0)** Escreva a função `inverte` que recebe um número inteiro positivo e devolve esse número invertido. Não pode recorrer a cadeias de caracteres nem a listas. A sua função deve verificar a correção do argumento. Por exemplo,

```
>>> inverte(12345)
54321
>>>inverte(3.0)
ValueError: O argumento deve ser um inteiro positivo
```

   Resposta:

```
def inverte(num):
    if isinstance(num, int) and num >= 0:
        res = 0
        while num != 0:
            dig = num % 10
```

```
            num = num // 10
            res = res * 10 + dig
        return res
    else:
        raise ValueError ('O argumento deve ser um inteiro positivo')
```

3. **(1.0)** Escreva a função `remove_repetidos` que recebe uma lista e devolve a lista obtida da lista original em que todos os elementos repetidos foram removidos. A ordem da lista original deve ser mantida. Por exemplo,

```
>>> remove_repetidos([2, 4, 3, 2, 2, 2, 3])
[2, 4, 3]
>>> remove_repetidos([2, 5, 7])
[2, 5, 7]
```

Resposta:

```
def remove_repetidos(l):
    for i in range(len(l)-1, 1, -1):
        if l[i] in l[0:i-1]:
            del(l[i])
    return l
```

4. **(2.0)** Escreva a função `numero_occ_lista`, que recebe uma lista e um número, e devolve o número de vezes que o número ocorre na lista e nas suas sublistas, se existirem. Não é necessário validar os argumentos. Por exemplo,

```
>>> num_occ_lista([1, 2, 3, 4, 3], 3)
2
>>> num_occ_lista([1, [[[1]], 2], [[[2]]], 2], 2)
3
```

Resposta:

```
def num_occ_lista(lst, n):
    if lst == []:
        return 0
    elif isinstance(lst[0], list):
        return num_occ_lista(lst[0], n) + \
               num_occ_lista(lst[1:], n)
    elif lst[0] == n:
        return 1 + num_occ_lista(lst[1:], n)
    else:
        return num_occ_lista(lst[1:], n)
```

5. Considere a função de ordem superior `soma_fn` que recebe um número inteiro positivo `n` e uma função de um argumento inteiro `fn`, e devolve a soma de todos os valores da função entre `1` e `n`. A função `soma_fn` não verifica a correção do seu argumento nem usa funcionais sobre listas. Por exemplo,

```
>>> soma_fn(4, lambda x: x * x)
30
>>> soma_fn(4, lambda x: x + 1)
14
```

(a) **(1.0)** Escreva a função `soma_fn` usando iteração linear.
Resposta:

```
def soma_fn(n, fn):
    res = 0
    for i in range(1, n + 1):
        res = res + fn(i)
    return res
```

(b) **(1.0)** Escreva a função `nenhum` usando recursão com operações adiadas.
Resposta:

```
def soma_fn(n, fn):
    if n == 0:
        return 0
    else:
        return fn(n) + soma_fn(n - 1, fn)
```

(c) **(1.0)** Escreva a função usando recursão de cauda.
Resposta:

```
def soma_fn(n, fn):

    def soma_fn_aux(n, soma):
        if n == 0:
            return soma
        else:
            return soma_fn_aux(n - 1, fn(n) + soma)

    return soma_fn_aux(n, 0)
```

6. **(1.0)** Usando um ou mais dos funcionais sobre listas (`filtra`, `transforma`, `acumula`), escreva a função `conta_pares`, que recebe uma lista de inteiros, e devolve o número de elementos pares da lista. A sua função deve conter apenas uma instrução, a instrução `return`. Por exemplo:

```
>>> conta_pares([1, 2, 3, 4, 5, 6])
3
```

Resposta:

```
def conta_pares(lst):
        return len(filtra(lambda x: x % 2== 0, lst))
```

7. Considere a seguinte gramática em notação BNF, cujo símbolo inicial é ⟨expressão⟩:

⟨expressão⟩ ::= `(`⟨arg⟩ ⟨op⟩ ⟨arg⟩`)`

⟨op⟩ ::= `+` | `-` | `*` | `/`

⟨arg⟩ ::= ⟨digito⟩$^+$

⟨digito⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

(a) **(0.5)** Indique os símbolos terminais e os símbolos não terminais da gramática.
Símbolos terminais:
Resposta:
`(`, `)`, `+`, `-`, `*`, `/`, `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`
Símbolos não terminais:
Resposta:
⟨expressão⟩, ⟨op⟩, ⟨arg⟩, ⟨digito⟩

(b) **(0.5)** Indique, justificando no caso de não pertencer, quais das seguintes expressões pertencem ou não pertencem ao conjunto de frases da linguagem definida pela gramática:
`(2 + -)`
`(24 * 00006)`
`2 * 0`
`(84 + 3.0)`
`(0 / 0)`

(c) **(1.0)** Escreva a função `reconhece`, que recebe como argumento uma cadeia de caracteres e devolve verdadeiro se o seu argumento corresponde a uma frase da linguagem definida pela gramática e falso em caso contrário.
Resposta:

```
def reconhece(cad):
    if not (cad[0] == '(' and cad[-1] == ')'):
        return False
    cad = cad[1:-1]

    pos = 0
    while pos < len(cad) and '0' <= cad[pos] <= '9':
        pos = pos + 1

    if pos < len(cad) and cad[pos] not in '+-*/':
        return False

    pos = pos + 1
    if pos == len(cad):
        return False

    while pos < len(cad) and '0' <= cad[pos] <= '9':
        pos = pos + 1
    return pos == len(cad)
```

8. **(1.5)** Recorrendo à estrutura de dados pilha, escreva uma função balanceados que recebe uma cadeia de caracteres contendo uma expressão aritmética e devolve verdadeiro se os parênteses estiverem balanceados, e falso em caso contrário. A expressão apenas contém parênteses curvos. Não necessita de verificar o argumento da função.

   Recorde que as pilhas apresentam as seguintes operações básicas: `nova_pilha`, `empurra`, `tira`, `topo`, `e_pilha`, `pilha_vazia` e `pilhas_iguais`. Por exemplo:

```
>>> balanceado('(1 * (2 / (4 + 2)))')
```

```
True
>>> balanceado('( 1 + (3 + 2)')
False
```

Resposta:

```
def balanceado(cad):
    p = nova_pilha()
    for c in cad:
        if c == '(':
            p = empurra(p, c)
        if c == ')':
            if not pilha_vazia(p):
                p = tira(p)
            else:
                return False
    return pilha_vazia(p)
```

9. Suponha que desejava criar o tipo `produto` em Python. Um `produto` é caracterizado por um nome (uma cadeia de caracteres) e um preço (um real positivo). Podemos considerar as seguintes operações básicas para produtos:

   (a) *Construtor*:

   - $cria\_produto : cad\_caracteres \times \mathbb{R}_+ \mapsto produto$
     $cria_produto(nome, preco)$ tem como valor o produto com nome $nome$ e preço $preco$.

   (b) *Seletores*:

   - $nome(produto) : produto \mapsto cad\_caracteres$
     $nome(produto)$ tem como valor o nome do $produto$.
   - $preco(produto) : produto \mapsto \mathbb{R}_+$
     $preco(produto)$ tem como valor o preço do $produto$.

   (c) *Reconhecedor*:

   - $eh\_produto : universal \mapsto lógico$
     $eh\_produto(arg)$ tem o valor $verdadeiro$ se arg é um produto e tem o valor $falso$ em caso contrário.

   (d) *Testes:*

   - $produtos\_iguais : produto \times produto \mapsto lógico$
     $produtos\_iguais(p_1, p_2)$ tem valor $verdadeiro$ se os produtos p_1 e p_2 tiverem nome e preços iguais e tem o valor $falso$ em caso contrário.
   - $mais\_caro : produto \times produto \mapsto lógico$
     $mais\_caro(p_1, p_2)$ tem o valor $verdadeiro$ no caso do preço de p_1 seja superior ao preço de p_2 e tem o valor $falso$ em caso contrário.

   (a) **(0.5)** Defina uma representação para produto baseada em dicionários.
   Resposta:
   $\Re[nome, preco]$ = `{'nome':` $nome$`, 'preco':` $preco$`}`.

   (b) **(1.5)** Escreva as operações básicas, de acordo com a representação escolhida.
   Resposta:

```
def cria_produto(nome, preco):
    if not isinstance(nome,str) or not isinstance(preco, float) or \
           preco <= 0:
              raise ValueError('cria_produto: argumento(s) invalido(s)')
    return {'nome': nome, 'preco' : preco}

def nome(produto) :
    return produto['nome']

def preco(produto):
    return produto['preco']

def eh_produto(univ):
    return isinstance(univ,dict) and len(univ) == 2 and \
           nome in univ and isinstance(univ['nome'], str) and \
           preco in univ and istinstance(univ['preco'], float)

def produtos_iguais(produto1,produto2):
    return nome(produto1) == nome(produto2) and \
           preco(produto1) == preco(produto2)

def mais_caro(produto1,produto2):
    return preco(produto1) >= preco(produto2)
```

(c) **(1.0)** Escreva a função `preco_produtos` que recebe como argumento uma lista cujos elementos são listas com dois elementos, contendo um produto e a sua quantidade e que devolve o valor total dos produtos na lista. Por exemplo,

```
>>> choc = cria_produto('chocolate', 1.5)
>>> agua = cria_produto('agua', 0.5)
>>> hamb = cria_produto('hamb', 3.)
>>> p_nata = cria_produto('pastel_de_nata', 1.2)
>>> compras = [[choc, 3], [agua, 1], [hamb, 2], [p_nata, 1], \
               [cria_produto('rebucado', 0.2), 12]]
>>> preco_produtos(compras)
14.6
```

Resposta:

```
def preco_produtos(lst_produtos):
        res = 0
        for p_q in lst_produtos:
                res = res + preco(p_q[0]) * p_q[1]
        return res
```

(d) **(1.0)** Que alterações teria que fazer na função `preco_produtos` se a representação de produtos fosse um tuplo em lugar de um dicionário? Justifique a sua resposta.

Resposta:

Não teria que fazer qualquer alteração, pois sendo esta uma função de alto nível é independente da representação

10. **(1.5)** Considere um dicionário que contém as notas finais dos alunos de FP. O dicionário tem como chave a nota, um número natural entre 0 e 20. Para cada chave do dicionário, o seu valor é uma lista com os números dos alunos com essa nota. Por exemplo, um dicionário poderá ter o seguinte formato:

```
notas_dict = {1 : [96592, 99212, 90300, 99312], \
              15 : [92592, 99212], 20 : [98323]}
```

Escreva a função `resumo_FP` que recebe um dicionário com as notas finais dos alunos de FP e devolve um tuplo com dois elementos contendo a média dos alunos aprovados e o número de alunos reprovados. Não é necessário verificar a correção dos dados de entrada. Por exemplo:

```
>>> resumo_FP(notas_dict)
(16.666666666666668, 4)
```

Resposta:

```
def resumo_FP(notas):
    n_aprovados = 0
    n_reprovados = 0
    soma_notas = 0
    for nota in notas:
        if nota < 10:
            n_reprovados = n_reprovados + len(notas[nota])
        else:
            n_aprovados = n_aprovados + len(notas[nota])
            soma_notas = soma_notas + nota * len(notas[nota])
    return (soma_notas / n_aprovados, n_reprovados)
```

11. **(1.5)** Considere a função `g`, definida para inteiros não negativos do seguinte modo:

```
def g(n):
    if n == 0:
        return 0
    else:
return n - g(g(n-1))
```

Como pode verificar, a função calcula várias vezes o mesmo valor quando chamada com diferentes argumentos. Para evitar este problema, podemos definir uma classe, `mem_g`, cujo estado interno contém informação sobre os valores de `g` já calculados, apenas calculando um novo valor quando este ainda não é conhecido. Esta classe possui um método `val` que calcula o valor de `g` para o inteiro que é seu argumento e um método `mem` que mostra os valores memorizados. Por exemplo,

```
G = mem_g()
G.val(12)
8
G.mem()
{0: 0,
 1: 1,
 2: 1,
 3: 2,
 4: 3,
 5: 3,
 6: 4,
```

```
 7: 4,
 8: 5,
 9: 6,
 10: 6,
 11: 7,
 12: 8}
```

Defina a classe `mem_g`.

Resposta:

```
class mem_g:

    def __init__(self):
        self.G = {0 : 0}


    def val(self, n):
        if (n) in self.G:
            return self.G[n]
        else:
            g_n_1 = self.val(n-1)
            g_g_n_1 = self.val(g_n_1)
            self.G[n] = n - g_g_n_1
            return self.G[n]

    def mem(self):
        return self.G
```