TÉCNICO LISBOA

Fundamentos da Programação

Segundo Teste

10 de Janeiro de 2014

09:00–10:30

Nome: _____ Número: _____

- Esta prova, individual e sem consulta, tem **8** páginas com **12** perguntas. A cotação de cada pergunta está assinalada entre parêntesis.
- Escreva o seu número em todas as folhas da prova. O tamanho das respostas deve ser limitado ao espaço fornecido para cada questão. O corpo docente reserva-se o direito de não considerar a parte das respostas que excedam o espaço indicado.
- Pode responder usando lápis.
- Em cima da mesa devem apenas estar o enunciado, caneta ou lápis e borracha e cartão de aluno. Não é permitida a utilização de folhas de rascunho, telemóveis, calculadoras, etc.
- Boa sorte.

Pergunta	Cotação	Nota
1.	1.0	
2.	1.0	
3.	1.5	
4.	1.0	
5.	1.5	
6.	1.5	
7.	1.5	
8.	3.0	
9.	1.5	
10.	2.0	
11.	3.0	
12.	1.5	
Total	20.0	

1. **(1.0)** Diga quais as fases por que passa o desenvolvimento de um programa e o que se faz em cada uma delas.

   Resposta:

   - *Análise do problema.* O programador, juntamente com o cliente, estuda o problema a resolver com o objectivo de determinar exactamente o que o programa deve fazer.
   - *Desenvolvimento de uma solução*. Determinação de como vai ser resolvido o problema. Desenvolvimento de um algoritmo e definição abstracta dos tipos de informação usados. Deve usar-se a metodologia do topo para a base.
   - *Codificação da solução.* Tradução do algoritmo para uma linguagem de programação, e implementação dos tipos de informação. Depuração, i.e., correcção de erros sintácticos e semânticos.
   - *Testes.* Definição de uma bateria de testes com o objectivo de "garantir" que o programa funciona correctamente em todas as situações possíveis.
   - *Manutenção.* Fase que decorre depois do programa estar em funcionamento. A manutenção é necessária por dois tipos de razões: a descoberta de erros ou a necessidade de introduzir modificações e actualizações nas especificações do programa.

2. **(1.0)** Explique em que consiste a abstracção de dados, usando os termos barreiras de abstracção, encapsulação da informação e anonimato da representação.

   Resposta:

   A abstracção de dados consiste em separar a o modo como os dados são utilizados do modo como os dados são representados. Para isso definem-se camadas conceptuais lidando com cada um destes aspectos estando estas camadas separadas por "barreiras de abstracção" que definem o modo como os programas acima da barreira podem comunicar com os programas que se encontram abaixo da barreira. Idealmente, os programas que se encontram a um dado nível de abstracção contêm toda a informação necessária para lidar com um certo tipo de dados, a informação está "encapsulada" dentro desta camada conceptual, e escondem das restantes partes do programa o modo como a informação está representada, o que é conhecido por "anonimato da representação".

3. O Python permite a utilização de três paradigmas de programação, a programação imperativa, a programação por objectos e a programação funcional. Apresente de uma forma sucinta as características principais de cada um destes paradigmas.

   (a) **(0.5)** Programação imperativa.
   Resposta:

   Em programação imperativa, um programa é constituido por uma série de ordens dadas ao computador. A programação imperativa depende da instrução de atribuição e da utilização de ciclos.

   (b) **(0.5)** Programação por objectos.
   Resposta:

   A programação por objectos baseia-se na utilização de objectos, entidades com estado interno associados a um conjunto de métodos que manipulam esse estado.

   (c) **(0.5)** Programação funcional.
   Resposta:

   A programação funcional baseia-se na utilização de funções que devolvem valores que são utilizados por outras funções. Em programação funcional as operações de atribuição e os ciclos podem não existir.

4. **(1.0)** Considere as estruturas de informação cujo comportamento é caracterizado por LIFO e FIFO. Explique o significado destes termos e diga quais as estruturas de informação a que estão associados.

   Resposta:

   LIFO significa "Last In First Out" e está associado ao tipo pilha. FIFO significa "First In First Out" e está associado ao tipo fila.

5. **(1.5)** Escreva uma função *recursiva* chamada `soma_quadrados_digitos` que recebe como argumento um número inteiro não negativo `n`, e devolve a soma dos quadrados dos dígitos de `n`. A sua função não tem que verificar a correcção do argumento. Por exemplo,

```
>>> soma_quadrados_digitos(322)
17
```

   Resposta:

```
def soma_quadrados_digitos(num):
    if num == 0:
        return 0
    else:
        digito = num % 10
        return digito * digito +\
               soma_quadrados_digitos(num // 10)
```

6. **(1.5)** Escreva uma função *recursiva*, `numero_ocorrencias_lista`, que recebe uma lista e um inteiro, e devolve o número de vezes que esse inteiro ocorre na lista e nas suas sublistas, se existirem. A sua função não tem que verificar a correcção dos argumentos. Por exemplo,

```
>>> numero_ocorrencias_lista([4, 5, 6],  5)
1
>>> numero_ocorrencias_lista([3, [[3], 5], 7, 3, [2, 3]], 3)
4
```

   Resposta:

```
def numero_ocorrencias_lista2(lst, num):
    if lst == []:
        return 0
    elif lst[0] == num:
        return 1 + numero_ocorrencias_lista2(lst[1:], num)
    elif isinstance(lst[0], list):
        return (numero_ocorrencias_lista2(lst[0], num) +
                numero_ocorrencias_lista2(lst[1:], num))
    else:
        return numero_ocorrencias_lista2(lst[1:], num)
```

7. **(1.5)** Escreva uma função `todos_lista` que recebe uma lista e um predicado unário, e devolve verdadeiro caso todos os elementos da lista satisfaçam o predicado e falso no caso contrário. A sua função não tem que verificar a correcção dos argumentos. Por exemplo,

```
>>> todos_lista([4, 5, 6], lambda x: x > 5)
False
>>> todos_lista([4, 5, 6], lambda x: x >= 4)
True
```

Resposta:

```
def todos_lista(lst, pred):
    for el in lst:
        if not(pred(el)):
            return False
    return True
```

8. Escreva uma função `soma_divisores` que recebe um número inteiro não negativo `n`, e devolve a soma de todos os divisores de `n`. No caso de `n` ser 0 deverá devolver 0. A sua função não tem que verificar a correcção do argumento. Por exemplo,

```
>>> soma_divisores(20)
42
>>> soma-divisores(13)
14
```

   (a) **(1.0)** Usando recursão com operações adiadas.
   Resposta:

```
def soma_divisores(n):

    def soma_aux(n, d):
        if d == 0:
            return 0
        elif n % d == 0:
            return d + soma_aux(n, d - 1)
        else:
            return soma_aux(n, d - 1)

    return soma_aux(n, n)
```

   (b) **(1.0)** Usando recursão sem operações adiadas ou de cauda.
   Resposta:

```
def soma_divisores(n):

    def soma_aux(n, d, soma):
        if d == 0:
            return soma
        elif n % d == 0:
            return soma_aux(n, d - 1, soma + d)
        else:
            return soma_aux(n, d - 1, soma)

    return soma_aux(n, n, 0)
```

   (c) **(1.0)** Usando um processo iterativo e um ciclo `while`.
   Resposta:

```
def soma_divisores(n):

    soma = 0
    d = n
    while d > 0:
        if n % d == 0:
            soma = soma + d
        d = d - 1
    return soma
```

9. **(1.5)** Escreva uma função, cujo argumento é o nome de um ficheiro de texto, que devolve o número de *emoticons* "`:-)`" que existem no ficheiro que é seu argumento. As linhas do ficheiro podem conter mais do que um *emoticon*. Por exemplo, se `tst1.txt` corresponder ao ficheiro:

```
Teste com alguns destes emoticons :-) :-) :-) numa :-) linha
e outro :-) na linha seguinte
```

e se `tst2.txt` corresponder ao ficheiro:

```
Teste com alguns destes emoticons :-) :-) :-) numa :-
) linha
e outro :-) na linha seguinte
```

Podemos obter a interacção:

```
>>> emoticons('tst1.txt')
5
>>> emoticons('tst2.txt')
4
```

A sua função deverá abrir o ficheiro (não é necessário usar encoding) e fechá-lo antes de terminar.

Resposta:

```
def emoticons(ficheiro):

  cont=0
  entrada = open(ficheiro, 'r')
  linha = entrada.readline()
  while (linha != ''):
    i=0
    l=len(linha)
    while i < l-3:
      if linha[i] == ':' and \
         linha[i+1] == '-' and \
         linha[i+2] == ')':
        cont = cont+1
      i=i+1
    linha = entrada.readline()
  entrada.close()
  return cont
```

10. **(2.0)** Crie a classe garrafa cujo construtor recebe a capacidade da garrafa em litros. A garrafa inicialmente será criada vazia. Os outros métodos suportados pela classe são:

    - `capacidade`, que devolve a capacidade total da garrafa;
    - `nivel`, que devolve o volume de líquido presente na garrafa;
    - `despeja`, que recebe a quantidade de líquido a remover da garrafa, em litros. Se a quantidade exceder o volume presente na garrafa, o volume presente na garrafa passa a ser 0.
    - `enche`, que recebe a quantidade de líquido a colocar na garrafa. Se a quantidade de líquido a colocar na garrafa fizer com que esta ultrapasse a sua capacidade, o volume presente na garrafa passa a ser igual à sua capacidade.

    Por exemplo,

```
>>> g1 = garrafa(1.5)
>>> g1.nivel()
0
>>> g1.capacidade()
1.5
>>> g1.despeja(2)
>>> g1.nivel()
0
```

    Resposta:

```
class garrafa:

    def __init__(self, capacidade):
        if isinstance(capacidade, (int, float)) and \
           capacidade > 0:
            self.nivel_actual = 0
            self.volume_total = capacidade
        else:
            raise ValueError('garrafa: capacidade deve ser numero positivo')

    def nivel(self):
        return self.nivel_actual

    def capacidade(self):
        return self.volume_total

    def despeja(self, volume):
        if volume > self.nivel_actual:
            self.nivel_actual = 0
        else:
            self.nivel_actual = self.nivel_actual - volume

    def enche(self, volume):
        if self.nivel_actual + volume > self.volume_total:
            self.nivel_actual = self.volume_total
        else:
            self.nivel_actual = self.nivel_actual + volume
```

11. Considere que tem disponíveis as operações básicas do tipo abstracto árvore: `nova_arv`, `cria_arv`, `raiz`, `arv_esq`, `arv_dir`, `arv_vazia`, `arv_iguais`, `escreve_arv`.

    (a) **(1.5)** Escreva a função `conta_nos` que recebe como argumento uma árvore e devolve o número total de nós que esta contém.
    Resposta:

```
def conta_nos(a):
    if arv_vazia(a):
        return 0
    else:
        return 1 + conta_nos(arv_esq(a)) + conta_nos(arv_dir(a))
```

    (b) **(1.5)** Uma árvore binária diz-se *equilibrada* se e só se, para cada uma das suas sub-árvores a diferença entre o número de nós da árvore esquerda e o número de nós da árvore direita nunca é superior a um. A arvore vazia é, por definição, equilibrada. Usando a função `conta_nos` da alínea anterior, escreva o predicado `equilibrada` que recebe como argumento uma árvore e devolve verdadeiro apenas se a árvore é equilibrada.
    Resposta:

```
def equilibrada(a):
    if arv_vazia(a):
        return True
    else:
        ne  = conta_nos(arv_esq(a))
        nd = conta_nos(arv_dir(a))
        if -1 <= ne-nd <= 1:
            return equilibrada(arv_esq(a)) and equilibrada(arv_dir(a))
        else:
            return False
```

12. **(1.5)** Considerando a existência do tipo pilha, realizada através de funções, com as operações `nova_pilha`, `topo`, `tira` e `empurra`, escreva um predicado com o nome `parenteses_balanceados` que recebe uma cadeia de caracteres correspondente a uma expressão aritmética (não verificando a correcção da entrada) e que determina se os seus parêntesis estão balanceados, ou seja cada parênteses que é aberto tem um parênteses fechado correspondente e os pares de parêntesis estão correctamente encadeados. Considere apenas os parêntesis curvos, rectos e as chavetas. Por exemplo,

```
>>> balanceados('2 * [(5 + 7) * a + b]')
True
>>> balanceados('3)')
False
>>> balanceados('({)}')
False
>>> balanceados('({[[()]]}())')
True
```

    Resposta:

```
def balanceados (exp):
    balanc = True
    pars = nova_pilha()
    for c in exp:
        if c in ['(', '[', '{']:
            pars = empurra(pars, c)
        elif c in [')', ']', '}']:
            if not pilha_vazia(pars):
                outro = topo(pars)
                if (c == ')' and outro == '(') or \
                   (c == ']' and outro == '[') or \
                   (c == '}' and outro == '{') :
                    pars = tira(pars)
                else:
                    return False
            else:
                return False
    return pilha_vazia(pars)
```