UFV/CCE/DPI
INF101 – Introdução à Programação II
Lista de Exercícios 1
16/09/2016

1. Faça um programa em Python que leia uma expressão com parênteses e, usando uma pilha, verifique se os parênteses estão balanceados. Isto é, verifique se o número de parênteses que abrem é igual ao número dos que fecham. Por exemplo:

```
(())          OK
()()(()())    OK
())           Errado
(()           Errado
```

2. Escreva uma função em Python para adicionar um dia a uma data. Considere que uma data será representada por uma 3-tupla da seguinte forma: (dia, mes, ano), onde todos os componentes são números inteiros que determinam uma data legal. Observe que há necessidade de considerar os anos bissextos. Um ano é *bissexto* se for divisível por 4, mas não por 100, ou se for divisível por 400. Por exemplo, 2016 é bissexto, 2100 **não** será bissexto, 2000 foi bissexto, 2015 não foi bissexto, nem 1900. Denomine a função `addOne(d, m, a)`. Assim alguns valores retornados por ela são:

```
addOne(16, 9, 2016) = (17, 9, 2016)
addOne(31, 12, 2016) = (1, 1, 2017)
addOne(28, 2, 2016) = (29, 2, 2016)
addOne(28, 2, 2015) = (1, 3, 2015)
addOne(30, 9, 2016) = (1, 10, 2016)
addOne(30, 8, 2016) = (31, 8, 2016)
addOne(28, 2, 2100) = (1, 3, 2100)
```

3. Escreva uma função que subtraia um dia de uma data. Denomine a função `subOne(d, m, a)`. Alguns valores retornados por ela são:

```
subOne(16, 9, 2016) = (15, 9, 2016)
subOne(1, 1, 2017) = (31, 12, 2016)
subOne(1, 3, 2016) = (29, 2, 2016)
subOne(1, 3, 2015) = (28, 2, 2015)
subOne(1, 9, 2016) = (31, 8, 2016)
subOne(1, 10, 2016) = (30, 9, 2016)
subOne(1, 3, 2100) = (28, 2, 2100)
```

4. Agora escreva uma função em Python que adicione um número (inteiro) qualquer de dias a uma data. Se o número de dias for negativo, significa que é para ser subtraído o dado número de dias. **Sugestão:** Use as duas funções, `addOne` e `subOne`, que foram escritas nos dois exercícios anteriores. Denomine a função `addDays(n, d, m, a)`. Então teremos:

```
addDays(10, 16, 9, 2016) = (26, 9, 2016)
addDays(20, 16, 9, 2016) = (6, 10, 2016)
addDays(-30, 16, 9, 2016) = (17, 8, 2016)
addDays(1000, 16, 9, 2016) = (13, 6, 2019)
addDays(-1000, 16, 9, 2016) = (21, 12, 2013)
addDays(67, 20, 12, 2016) = (25, 2, 2017)
```

5. Escreva uma função booleana em Python que retorne `True`, se a tupla (m, d, a) constitui uma data legal; caso contrário, retorne `False`. Considere 1600 como o ano mínimo para uma data legal. Denomine a função `checkDate(m, d, a)`. Assim teremos:

```
checkDate(16, 9, 2016) = True
checkDate(31, 9, 2016) = False
checkDate(29, 2, 2017) = False
checkDate(31, 7, 2000) = True
checkDate(32, 4, 2016) = False
checkDate(-3, 6, 2001) = False
checkDate(7, 13, 2016) = False
checkDate(31, 12, 1599) = False
checkDate(29, 2, 2000) = True
checkDate(29, 2, 2100) = False
```

6. Escreva uma função em Python que pesquise a primeira ocorrência de um determinado componente x em uma lista L. Se x ocorrer em L, a função retornará o índice da primeira posição em que x estiver; caso contrário, retornará `-1`. Denomine a função `pesquisa(x, L)`. Por exemplo, se `L = [36, 18, 43, 9, 18, 25, 14]`, então:

```
pesquisa(25, L) = 5
pesquisa(18, L) = 1
pesquisa(15, L) = -1
pesquisa(14, L) = 6
```

7. Escreva uma função **recursiva** em Python que faça a pesquisa de um dado componente x em uma lista L ordenada crescentemente. Pelo fato de que a lista está ordenada, podemos implementar um método de pesquisa muito eficiente, a saber, *pesquisa binária*. Denomine a função `pesqBin(x, L, inicio, fim)`. O algoritmo de pesquisa binária em altíssimo nível pode ser descrito assim:
   i. Determine o elemento do meio da lista com índice `k = (inicio + fim)//2`.
   ii. Se `x == L[k]`, o elemento procurado foi encontrado, retorne k.
   iii. Se `x > L[k]`, nenhum elemento da primeira metade do arranjo pode ser igual a x; logo a continuação da pesquisa pode se restringir à segunda metade da lista.
   iv. Se `x < L[k]`, analogamente, a pesquisa pode se restringir à primeira metade da lista.
   v. Em cada uma dessas metades, aplique o mesmo método novamente até que x seja encontrado ou as metades da (sub)lista em questão se tornem vazias. Neste caso, a pesquisa foi sem sucesso.

   Use o mesmo esquema de retorno da função do exercício anterior, qual seja, se a pesquisa obtiver sucesso, retorne o índice k da posição em que x se encontra na lista; caso contrário, retorne `-1`. Por exemplo, se `L = [9, 14, 18, 18, 25, 36, 43]`, então:

```
pesqBin(25, L, 0, 6) = 4
pesqBin(18, L, 0, 6) = 3
pesqBin(36, L, 0, 6) = 5
pesqBin(15, L, 0, 6) = -1
```

8. A verificação de um número do CPF brasileiro é feita por meio de dois dígitos verificadores que aparecem no final da sequência dos onze dígitos que perfazem o número: os nove primeiros dígitos são o número realmente do cadastro e os dois últimos são os dígitos verificadores.

Escreva uma função em Python que retorne `True`, se a cadeia de caracteres dada como argumento é um CPF válido; caso contrário, retorne `False`. Você pode usar tantas funções auxiliares quantas quiser. A cadeia de caracteres contendo o CPF possui apenas dígitos sem pontos e sem hífen. O algoritmo de verificação de um CPF segue:

Cálculo do primeiro dígito verificador

i. Calcula-se o somatório $s = \sum_{i=0}^{8}(\text{cpf}_i \times \text{peso}_i)$, onde cpf_i é *i*-ésimo dígito do CPF, a partir da esquerda para a direita, e peso_i corresponde aos valores da sequência decrescente $\text{peso}_0 = 10$, $\text{peso}_1 = 9$, ..., $\text{peso}_8 = 2$.
ii. Calcula-se o valor *r* como o resto da divisão da soma *s* por 11.
iii. Se $r < 2$, o primeiro dígito verificador é $\text{dv}_1 = 0$; caso contrário, $\text{dv}_1 = 11 - r$.

Cálculo do segundo dígito verificador

i. Calcula-se o somatório $s = \sum_{i=0}^{8}(\text{cpf}_i \times \text{peso}_i) + (\text{dv}_1 \times \text{peso}_9)$, onde cpf_i são os mesmos de antes e dv_1 é o primeiro dígito verificador e a sequência decrescente de pesos agora é $\text{peso}_0 = 11$, $\text{peso}_1 = 10$, ..., $\text{peso}_9 = 2$. Observe que a sequência de pesos para cada dígito verificador é ligeiramente diferente da outra.
ii. Calcula-se o valor *r* como o resto da divisão da soma *s*, calculada no passo anterior, por 11.
iii. Se $r < 2$, o segundo dígito verificador é $\text{dv}_2 = 0$; caso contrário, $\text{dv}_2 = 11 - r$.

Após calculados os dígitos verificadores, a função deve compará-los com os respectivos dois últimos algarismos do CPF dado. Se os valores calculados corresponderem aos informados, o CPF dado é válido; caso contrário, não o é. Denomine a função `cpfval(cpf)`. Assim teremos:

```
cpfval("12345678909") = True
cpfval("12345678910") = False
cpfval("11144477723") = False
cpfval("12562377877") = True
```

9. (*O Crivo de Eratóstenes*) Escreva um programa em Python que imprima todos os números primos menores que 1000. Você pode escrever este programa criando uma lista de números primos. Para começar, a lista está vazia. Em seguida, escreva dois laços `for` aninhados. O laço externo percorre todos os números de 2 a 999. O laço interno percorre a lista de números primos. Se o próximo número do laço externo não for divisível por nenhum dos números primos até o momento, então ele também é primo e é adicionado à lista de números primos. Ao terminar o laço externo, imprima a lista dos números primos obtidos.

10. Escreva um programa que peça ao usuário entrar com uma lista e então reverter a lista *in situ,* de modo que, após a reversão, a lista original é que tenha sido revertida ao invés de criar uma nova lista.

11. Escreva uma função denominada `todosPares(L)` que, dada uma lista L de números inteiros, retorne uma nova lista contendo somente os inteiros pares. Use a função em um programa e teste seu código com diversos valores diferentes.