www.estudar.com.vc

Mac em Python

Fuja do Nabo P3 2019.1

Resumo e Lista de Exercícios

Resumo

1 - Recursão

Um tanto quanto mal compreendida, recursão é um jeito diferente de pensar os seus códigos. Ao invés de resolver tudo numa tacada só, você divide a solução em diversas partes partindo do caso mais trivial.

É muito útil para problemas essencialmente recorrentes (fibonacci e cálculo do determinante) e para problemas complexos cuja solução trivial é óbvia (potência, algumas ordenações).

In [23]:
```python
def potencia_recursiva(base, expoente):
    if expoente == 0:       # Caso mais trivial da potenciação é x^0(x**0)
        return 1.0   # Solução do caso trivial
    else:
        return base * potencia_recursiva(base, expoente-1) # Se não é o caso trivial, vamos mais fundo até chegar nele
                                                           # Aqui que ocorre de fato a recursão (chamada recursiva)

base_qualquer = 2
expoente_qualquer = 1000

print(potencia_recursiva(base_qualquer, expoente_qualquer))
```

```
1.0715086071862673e+301
```

Acima é feito um exemplo com a potenciação. Reparem que o "problema" (calcular a potencial de uma base num determinado expoente) é atacado primeiro observando o caso mais trivial e então realizando uma chamada recursiva ao final divindo o problema até chegar no caso trivial.

O lado ruim da recursão é que para contas muito grandes (aqui quero dizer com "muitas chamadas"), como o cálculo com a base 1 e expoente 10000, o algorítmo não executa por haver um limite de profundidade das recursões

2 - Tipo Dict

O tipo de dado Dict, para os "íntimos", ou simplesmente dicionários, trabalha de forma diferente em relação as listas, strings e tuplas. Pois enquanto estas eram sequenciais (elementos são acessados com índices inteiros e os valores estão ordenados da esquerda para direita), o dicionário faz uma associação entre **chave** e **valor**, sendo a **chave** única e imutável (logo strings, números e tuplas) e o **valor**, qualquer tipo de dado.

In [1]:
```python
dic = {}   # Cria um dicionário vazio
contas = {"telefone": 50, "gas": 40, "supermercado": 300}   # Num dict, conjuntos (chave, valor) são separados por vírgula
print(contas)

# Adicionando um conjunto (chave, valor) a um dict --> fazemos: dic[chave] = valor
contas["xerox_faculdade"] = 1000
print(contas)

# As chaves são imutáveis, mas os valores não!
contas["xerox_faculdade"]+=1000
print(contas)

# Podemos checar se há alguma chave existe ou não no nosso dicionário
if "telefone" in contas:
    print("Achei telefone nas minhas contas!!")

# Além disso podemos iterar sobre as chaves do dicionário, sobre os valores e sobre os conjuntos (chave,valor)!

for chaves in contas:
    print(chaves)
#OU
for chaves in contas.keys():
    print(chaves)

for valores in contas.values():
    print(valores)

for (chave, valor) in contas.items():
    print("Achei %s nas minhas contas com valor %d"%(chave, valor))
```

Output da célula dos dicionários:

```
{'telefone': 50, 'gas': 40, 'supermercado': 300}
{'telefone': 50, 'xerox_faculdade': 1000, 'gas': 40, 'supermercado': 300}
{'telefone': 50, 'xerox_faculdade': 2000, 'gas': 40, 'supermercado': 300}
Achei telefone nas minhas contas!!
telefone
xerox_faculdade
gas
supermercado
telefone
xerox_faculdade
gas
supermercado
50
2000
40
300
Achei telefone nas minhas contas com valor 50
Achei xerox_faculdade nas minhas contas com valor 2000
Achei gas nas minhas contas com valor 40
Achei supermercado nas minhas contas com valor 300
```

3 - Lidando com Arquivos

Olha só! Agora podemos abrir e criar arquivos em python! Basicamente, o que fazemos é criar uma instância, um objeto, de um certo arquivo em python e nesse objeto temos métodos interessantes para ajudar a lidar com arquivos. Para criar essa "instância" basta usarmos a função ***open("nome_do_arquivo.formato", "modo de abertura do arquivo")***

In [ ]:
```python
# Nessa cédula apresentarei diferentes formas de ler um arquivo .txt, não rode essa cédula com todos os módulos. Comente
# todos exceto o que você quiser testar

# Abrindo um arquivo aleatório no modo de leitura
arquivo_objeto = open("notes.txt", 'r')   # O arquivo_objeto é a tal instância e o parâmetro 'r' significa read, leitura.

# Podemos ler o conteúdo completo:
print(arquivo_objeto.read())
# Ou linha por linha:
for linha in arquivo_objeto:
    print(linha, end="")

for linha_denovo in arquivo_objeto.readlines():
    print(linha_denovo, end="")
    # Faz quase a mesma coisa que o de cima, porém a função readlines() cria um iterável das linhas e podemos passar
    # parâmetros interessantes para esse método, que infelizmente nao cabem tanto para esse curso.

linha_novamente = arquivo_objeto.readline()
while linha != "":  # Condição até que chegue no final do arquivo (linhas nulas)
    print(linha_novamente, end="")
    linha_novamente = arquivo_objeto.readline()

arquivo_objeto.close()
```

Lembre-se sempre de fechar a instância do arquivo com o ***.close()*** . Se quiser eliminar essa necessidade, podemos usar o ***with ... as ...***

In [27]:
```python
with open("notes.txt", 'r') as arquivo_objeto:
    # Aqui fazemos o que queríamos antes fazer e sem precisar do .close()
    for linha in arquivo_objeto:
        print(linha, end="")
```

```
O legal/ruim é que pra vocês usarem direitinho a apostila, precisam desse arquivo.
Por quê?
Bom, porque quando fizemos o open("filename.txt", 'r'), a gente não designou nenhum "caminho", para ele achar o arquivo, então obrigatoriamente o arquivo tem que estar na mesma pasta da apostila, senão o python não vai encontrar.
Porém, para as implementações em MAC2166 isso está ótimo. Se tiverem dúvidas em como avançar mais, é só mandar e-mail!
Abraços!
```

Só que lidar com arquivos não é só ler, não é mesmo? Para escrever em um arquivo temos os modos **'w'** e **'a'** . O modo **'w'** permite criar um arquivo com o nome fornecido, caso o arquivo com tal nome não existe, porém, se o arquivo existir, ele sobrescreverá o conteúdo (apaga o que já houver lá). O modo **'a'** serve justamente para esse segundo caso, pois permite escrever sobre um conteúdo anterior sem apagá-lo.

In [28]:
```
# Usando o modo 'w' para criar um arquivo (nome de arquivo fornecido não existe) e escrever nele:
arquivo = open("novo_arquivo.txt", 'w')

for indices in range (35, 65):
    arquivo.write(chr(indices))   # Escreve a string no arquivo
    # Se quiser colocar quebra de linha precisaria de: arquivo.write('\n')

arquivo.close()
```

In [31]:
```
# Se fizermos o mesmo código, mudando o range dos indices, o conteúdo do arquivo será apagado e haverá um novo conteúdo
arquivo = open("novo_arquivo.txt", 'w')

for indices in range (65, 99):
    arquivo.write(chr(indices))

arquivo.close()
```

In [32]:
```
arquivo = open("novo_arquivo.txt", 'a')

arquivo.write('\n')   # Insere quebra de linha no arquivo

for indices in range (35, 65):
    arquivo.write(chr(indices))   # Escreve a string no arquivo
    # Se quiser colocar

arquivo.close()
```

Exercícios

1. Recursão

P3 2017 Adaptada

Simule o código abaixo e forneça a impressão do programa.

In [1]:
```
def f (L, e, d):
    if e>d:
        return
    c = (e + d)//2
    print(L[c])
    f(L, e, c-1)
    f (L, c + 1, d)

def g (L, e, d):
    if e > d:
        return
    c = (e + d)//2
    g (L, e, c-1)
    print (L[c])
    g (L, c+1, d)

def main ():
    f ([8, 57, 18], 0, 2)
    g ([8, 57, 18], 0, 2)

main()
```

2. Dicionários e Arquivos de Texto

P3 2017

Preencha as lacunas do código abaixo ($L1$ até $L8$) de forma a obter um programa que lê um arquivo de texto e determine a palavra mais frequente com pelo menos 3 letras e sua frequência relativa de ocorrência no texto. As opções para cada lacuna estão presentes na tabela abaixo:

	A	B	C	D
L1	file(nome, 'r')	open(nome, 'w')	open(nome, 'r')	file(nome, 'w')
L2	[0]*len(a)	[]	[0]*len(nome)	{}
L3	a.readline()	a.read()	l.split('.')	l.split()
L4	f[p] != 0	p in f	p not in f	f[p] == 0
L5	f[p] = 0	f.append(p)	f[p] -= 1	f[p] = 1
L6	p in f	p in l.keys()	p in f.values()	p in l
L7	f[p] < f[m]	f[p] < t	f[p] > f[m]	f[p] > t

As lacunas estão no código abaixo representadas por $\#Li$ sendo i de 1 até 8.

In [ ]:
```
def maxfreq (nome):
    a = #L1
    f = #L2
    for l in a:
        l = #L3
        for p in l:
            if len(p) >= 3:
                if #L4:
                    f[p]+=1
                else:
                    #L5
    m = None
    for p in f:
        m = p
    t = 0
    for #L6:
        t += f[p]
        if #L7:
            m = p
    a.close()
    return #L8

def main():
    arq = input("Nome do arquivo: ")
    p, f = maxfreq(arq)
    print("palavra: ", p, "frequência relativa: ", f)
```

3. Ordenação de Listas

P3 2018

As funções abaixo foram projetadas para ordenar qualquer lista de modo a resultar em uma lista de ordem decrescente. Entretanto, alguns falham para algumas configurações de lista. Os códigos de cada modo seguem abaixo:

In [ ]:
```
#I
def ordena(lista):
    range(10)
    L[0:2]
    for i in range(len(lista)-1):
        for j in range(len(lista)-1):
            if lista[j+1] >= lista[j]:
                q = lista[j]
                lista[j] = lista[j+1]
                lista[j+1] = q
```

In [5]:
```
#II
def ordena (lista):
    i = 0
    while (i < len(lista)):
        x = lista[i]
        j = i-1
        while (j >= 0 and lista[j] < x):
            lista[j+1] = lista[j]
            j = j-1
        print(j)
        lista[j+1] = x
        print(lista)
        i += 1
```

In [ ]:
```
#III
def ordena (lista):
    for i in range(len(lista)-1):
        for j in range(len(lista)-2):
            if (lista[j+1] > lista[j]):
                q = lista[j]
                lista[j] = lista[j+1]
                lista[j+1] = q
```

In [7]:
```
#IV
def ordena (lista):
    for i in range(len(lista)):
        maior = lista[i]
        indice = i
        for j in range(i, len(lista)):
            if (lista[j] > maior):
                maior = lista[j]
                indice = j
                lista[indice] = lista[i]
```

In [ ]:
```
#V
def ordena (lista):
    for i in range(len(lista)):
        maior = lista[i]
        indice = i
        for j in range(i, len(lista)):
            if (lista[j] > maior):
                maior = lista[j]
                indice = j
        if (indice != i):
            x = lista[indice]
            lista[indice] = lista[i]
            lista[i] = x
```

Quais das alternativas abaixo contém códigos que ordenam (de forma decrescente) qualquer configuração inicial da lista?

A. V
B. II
C. I
D. III
E. IV

4. Problemas de Código

P3 2017

O programa abaixo deveria calcular, para um valor x do usuário, a função a seguir:

$$f(x) = \sum_{n=1}^{10} (-1)^{(n+1)} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots - \frac{x^{10}}{10}$$

Porém, o programa contém erros e não funciona como esperado. Marque as alternativas que contêm as correções necessárias para que o programa efetue o cálculo corretamente.

In [ ]:
```
1  def main():
2      x = float(input("x: "))
3      t, i, s = 1, 1, 1
4      while i < 10:
5          t*=x/(i-s)
6          i+=1
7          s+=t/(10-i)
8      print(t)
9
10 main()
```

Assinale as alternativas corretas:

A. Substituir a linha 3 por $t, i, s = x, 0, x$.
B. Substituir a linha 3 por $t, i, s = x, 1, x$.
C. Substituir a linha 8 por $print(s)$.
D. Substituir a linha 7 por $s+= t/i$.
E. Substituir a linha 5 por $t *= -x$.
F. Substituir a linha 3 por $t, i, s = -1, -1, 0$.
G. Trocar as linhas 5 e 6 de lugar.
H. Substituir a linha 4 por $while\ i <= 10$.
I. Substituir a linha 6 por $i *= (i + 1)$.
J. Substituir a linha 7 por $s+= t//(10 - 1)$.

5. Algoritmo de Busca e Recursão

Adaptado de exemplo fornecido em aula

Escreva uma função que implemente o algoritmo da Busca Binária, mas de forma recursiva. A função deve receber como parâmetro uma lista ordenada e um número real x e devolver a posição em que x ocorre na lista ou $None$ caso x não esteja na lista.

Gabarito

1. 57, 8, 18, 8, 57, 18.

2.
L1. Alternativa C.
L2. Alternativa D.
L3. Alternativa D.
L4. Alternativa B.
L5. Alternativa D.
L6. Alternativa A.
L7. Alternativa C.
L8. Alternativa B.

3. Alternativas A, B e C.

4. Alternativas B, C, D, E.

5.

In [ ]:
```python
# Gabarito

def busca_binaria_rec(seq, inicio, fim, x):
    if inicio > fim:
        return None

    meio = (inicio + fim) // 2

    if x == seq[meio]:                                  # achou x na posição meio
        return meio

    if x < seq[meio]:                                   # x pode estar na primeira metade da sequência
        return busca_binaria_rec(seq, inicio, meio-1, x)

    # x pode estar na segunda metade da sequência
    return busca_binaria_rec(seq, meio+1, fim, x)


def busca_binaria(seq, x):
    return busca_binaria_rec(seq, 0, len(seq)-1, x)
```