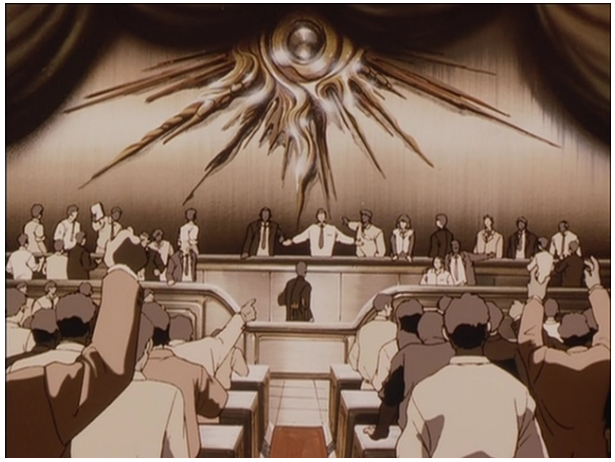
Conversão, Lógica, Controle de Fluxo por Condicionais, Operadores (2), Estrutura de Repetição, Sequências

Conversões entre tipos

Supondo uma situação da vida real:

- Em muitos congressos, (científicos, políticos, etc.), ocorrem à presença de diversas pessoas de diversas partes do mundo. Um problema que isto ocasiona é a dificuldade no entendimento entre as mesmas, dado o idioma e cultura de cada uma. Para estes casos, existe um profissional que faz o intermédio entre as conversas, o tradutor.



Analogamente, quando trabalhamos com variáveis que englobam um dado de um tipo específico, e necessitamos que este dado se apresente como de algum outro tipo, o que devemos fazer é utilizar as funções de conversão de tipos (também conhecido como “casting”).

Sendo mais claro com um exemplo, para a entrada de dados que serão usados em uma expressão matemática (no caso, através de raw_input), é interessante que façamos a conversão para algum tipo numérico.

Logicamente, a conversão deve ser possível para ser efetuada.

Praticamente todo e qualquer tipo definido em Python possui uma função para “realizar tal conversão”.

```
1  variavel_crua = 666
2
3  con_string = str(variavel_crua) #String
4  con_integer = int(variavel_crua) #Inteiro
5  con_float = float(variavel_crua) #Ponto Flutuante
6
```

No entanto, máxima atenção quando estiver trabalhando com dados que exijam conversão...

Lógica Booleana

Em nossa vida diária, muitas vezes nos deparamos com situações que exigem que façamos uma espécie de "comparação".

Por exemplo, quando saímos de casa, temos de verificar, "o fogão está desligado ou não?". Se sim, ok. Se não, temos de desligá-lo.

Isto pode ser interpretado como inerente às características da própria mente humana.

Na maioria das vezes, estas comparações são definidas pelo uso do que chamamos lógica (material extra).

Em programação, existem formas de podermos realizar tais operações, chamamos isto de lógica booleana (criada por George Boole).

A mesma se baseia no uso de números em base binária (1 e 0), porém no caso de python, não deve ser confundida com “operações bitwise”.

Em python, especificamente, logíca booleana pode ser realizada através dos operadores definidos à seguir:

```python
este_eh_o_melhor_curso = bool(1) #Verdadeiro, True ou Ligado
eu_jogo_bola_diariamente = bool(0) #Falso, False ou Desligado

print (eu_jogo_bola_diariamente)#Exibe "False"

#Operadores

print (este_eh_o_melhor_curso and eu_jogo_bola_diariamente) #And, operador "e" logico. Retorna True quando ambas as expressoes sao True
print (este_eh_o_melhor_curso or eu_jogo_bola_diariamente) #Or, operador "ou" logico. Retorna True quando pelo menos uma expressao eh True
print (not eu_jogo_bola_diariamente) #Not, operador "negacao" logica. Retorna True para expressoes False e vice-versa

```

Lógica Relacional

Já quando desejamos avaliar dois (ou mais) valores, de forma a comparar os mesmos, devemos fazer uso da lógica relacional.

- Por exemplo: “Qual veículo custa mais caro?” É um problema que pode usufruir de lógica relacional.

  Em python, também existem operadores específicos para o trato de tais operações. Vale lembrar ainda, que mesmo os operadores de lógica relacional, em python, trabalham alguma lógica booleana em seu interior.

Operadores:

```
1   cinco = 5
2   treze = 13
3
4   print (cinco < treze) # Operador "Menor"
5   print (treze > cinco) #Operador "Maior"
6   print (treze == cinco) #Operador "Igual" ou "Equivalencia"
7   print (treze != cinco) #Operador "Diferente" ou "Nao Equivalencia".
8   #Em versoes anteriores, "Diferente" era tido como "<>"
9   print (treze <= cinco) #Operador "Menor ou Igual"
10  print (cinco <= cinco)
11  print (treze >= cinco)# Operador "Maior Igual"
12  print (treze >= treze)
```

Controle de fluxo

Ainda tendo em mente o exemplo do aluno saindo de casa, pode ter percebido que há uma espécie de "tomada de decisão" nele (Se fogão desligado , ok. Senão, temos de desligá-lo.).

Isto pode ser entendido como uma alteração no fluxo de execução do programa, dado que temos duas "linhas possíveis" para serem executadas.

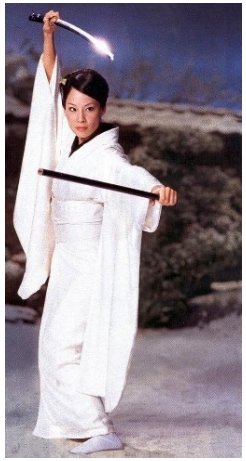
- Outra Analogia: Jogo de Videogame com múltiplos finais.

Em programação, novamente, isto não é diferente. Muitas vezes desejamos que dada ação seja executada apenas se certos requerimentos forem cumpridos. Por exemplo, na validação de uma senha.

Sintaxe:

```
1   senha_segura = raw_input("Senha: ")
2
3   if(senha_segura == "senha1234"):
4       print "Bem-vindo"
5   else:
6       print "Senha incorreta. Este sistema ira se autodestruir."
```

```
1  print "RPG Batalha do Terminal - Versao 0.0.1 - Fase de Teste"
2
3  qtd_ataque = 10
4  qtd_hp = 100
5
6  print "Voce esta andando pela mansao da terrivel Cottonmouth. Um espadachim inimigo avista voce."
7  print "Ele vem em sua direcao."
8  if(qtd_ataque > 5): # Se ataque for maior que 5, executa o bloco de codigo no "interior" do if
9      print("Voce desfere um golpe letal no inimigo!")
10 elif(qtd_hp > 20): # Senao, se hp maior que 20, executa o bloco de codigo no interior do "else if"
11     print("O inimigo desfere um golpe em voce, porem voce so leva um corte.")
12     qtd_hp -= 20
13 else: # Se tudo "falhar", ele ira para este bloco de codigo
14     print("O inimgo desfere um golpe. Voce morreu!")
```



- O conteúdo do bloco “if/elif” será avaliado se a condição passada for verdadeira.
- Caso contrário, a mesma não será avaliada, e o conteúdo a ser executado será, caso contenha, o do bloco “else”.
- Podem existir quantos “elif” aninhados em um bloco quanto se queira.
- Para cada “if” pode ou não haver apenas um “else”.
- Não existe bloco “else” sem “if”.

Para aqueles que já tem algum conhecimento de Shell Script ou já cursaram S.O., vale notar que tais estruturas de decisão no python se assemelham sintaticamente com àquelas do Bash.

```
1  #!/bin/bash
2  echo "Decisao em Shell"
3
4  if [ $1 -gt 10 ]
5  then
6      echo "Maior que 10"
7  else
8      echo "Menor que 10"
9  fi
10
```

Blocos if/elif/else são muito úteis para adição de lógica aos nossos programas, não somente em python.

- Outro Exemplo (C++ X Python):

```python
senha_segura = raw_input("Senha: ")

if(senha_segura == "senha1234"):
    print "Bem-vindo"
else:
    print "Senha incorreta. Este sistema ira se autodestruir."
```

```cpp
#include <iostream>
#include <string>

using namespace std;

int main(){

    cout<<"Senha: ";
    string senha = "";
    cin>>senha;

    if(senha == "senha1234"){
        cout<<"Bem-Vindo!"<<endl;
    }else{
        cout<<"Senha incorreta. Este sistema ira se autodestruir."<<endl;
        }
    return 0;
    }
```

Outro detalhe, que já foi citado mas que à partir de agora devemos tomar cuidado é a questão da identação. Observe o próximo exercício.

Exercício #5 (5 min.)

(Analítico) Atente aos códigos abaixo:

```
1  #1
2  pontuacao = 61
3
4  if(pontuacao > 60):
5      print "Voce foi aprovado, parabens!"
6  else:
7      print "Voce foi reprovado!"
8      print "Vai ter de ficar estudando durante mais duas semanas e refazer a prova."
9
```

```
1  #2
2  pontuacao = 61
3
4  if(pontuacao > 60):
5      print "Voce foi aprovado, parabens!"
6  else:
7      print "Voce foi reprovado!"
8  print "Vai ter de ficar estudando durante mais duas semanas e refazer a prova."
```

- Qual a diferença entre os resultados de um e do outro?
- Porque isto ocorre?

- Dica: Lembre-se do que foi explanado anteriormente, quanto à questão da estruturação de um programa em python.

Exercício #6 (10 min.)

Neste exercício, iremos implementar o clássico decisor para abastecimento de carros “flex”.

O mesmo consiste em verificar: Caso o preço do àlcool exceda 70% do preço da gasolina, abasteceremos com gasolina. Senão, àlcool.

Entradas: Preço da gasolina e do àlcool.

Saída: Com qual abasteço.

- Desafio: Deixar o programa bem interativo com o usuário.
- Dica: Cálculo de porcentagem: 0.7 * valor_da_gasolina

Faça uso de If..else para decisão. Lembre-se da conversão.

Exercício #7 (5 min.)

Elabore um programa que dada a temperatura de entrada (valor numérico), classifique-a de acordo com uma das categorias abaixo:

- Temperatura < 0 : “Sub-Zero!”
- Temperatura >= 0 e < 15 : “Trolla-Solteiros”
- Temperatura >= 15 e <= 35: “Sorvete Tropical”
- Temperatura >= 35: “Ta pegando fogo bicho!”

Operadores de Associação

Eis aqui uma peculiaridade que simplifica diversos programas em python. Os operadores de associação são operadores que servem para testar se um elemento “está contido” em determinada sequência.

Por hora, podemos pensar numa sequência como um termo genérico para um conjunto de itens que possuam certa “ordem”. Um exemplo de tipo que é baseado em sequência são as strings que já apresentamos.

- O operador básico de associação é o “in”. O mesmo é avaliado como verdadeiro caso encontre o membro especificado dentro da sequência especificada.
- O mesmo pode ser combinado da forma “not in”, para testar o contrário.

```
1   # Exemplos que consideram o tipo String
2   frase = "Words have no power to impress the mind without the exquisite horror of their reality." # Frase de Poe
3
4   if ("reality" in frase): # Operador in, ira checar se a palavra reality esta contida na string que eh o valor de "frase"
5       print "Esta frase contem a palavra reality."
6   if ("green" in frase):
7       print "Esta frase contem a palavra green"
8   if ("ghost" not in frase): # Operador not in, checa se ghost nao estara em "frase"
9       print "Esta frase nao contem a palavra ghost"
10
11  # Exemplo com apenas um caractere
12  palavra = "poe"
13
14  if("p" in palavra):
15      print "Esta contida a letra p"
16
```

Exercício #8 (10 min.)

Implemente o seguinte programa:

- Entrada: Seu primeiro nome;
- Saída:

  Caso seu nome possua a letra “a”, deve imprimir: “A de Astronomy”;

  Caso possua a letra “e”, “E de Enter Sandman”

  Caso possua a letra “i”, “I de I Disappear ”

  Caso possua a letra “o”, “O de One”

  Caso possua a letra “u”, “U de Unforgiven”

  Caso não possua nenhuma das letras, a saída deve ser: “Certamente prefere Megadeth...”

- Dica: Simples, porém atenção ao enunciado, pois o único caso excludente é o último...

Estruturas de Repetição (I’ll be Back!)

Uma das coisas mais fascinantes na computação é a habilidade de controlar e designar tarefas à serem executadas de forma a poupar o esforço humano.

Muitas vezes, encontraremos situações onde é necessário repetir uma mesma tarefa diversas e diversas vezes. Por exemplo, quando desejamos cadastrar várias pessoas em um mesmo sistema.

Python, tal qual outras linguagens, nos permite trabalhar com 2 tipos de estrutura de repetição:

- For
- While

Estruturas de repetição nos permitem “iterar” sobre elementos do código tanto quanto executar instruções múltiplas vezes.

- Uma iteração = Uma passagem da repetição.

Duas coisas que devemos tomar cuidado são: Identação, tal qual tratamos nas instruções “if” e também, a questão de “loops infinitos” (possíveis erros semânticos).

- Vale lembrar que...

For

Trabalhar com “loops for” em python é um tanto quanto diferente de outras linguagens.

Em primeiro lugar, em python, um loop for é uma entidade que opera sobre uma sequência.

Em segundo lugar, for em python admite o uso de “else”.

Finalmente, existe uma função específica em python com a capacidade de gerar sequências, e que é utilizada frequentemente para o trato de loops for. É a função range().

- Exemplo:

```
1   for cont in range(5): # Para CADA ITEM gerado por Range(5), o codigo abaixo sera executado.
2       # Linhas abaixo serao executadas a cada iteracao
3       print cont
4
5   for cont in range (1, 6): #A saber, podemos iterar em um limite bem definido
6       print cont
7
8   # Um exemplo mais pratico
9
10  #Checagem de paridade
11  for cont in range (10):
12      if(cont % 2 == 0):
13          print "O numero " + str(cont) + " eh par."
```

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/For Loop$ python forloop.py
0
1
2
3
4
1
2
3
4
5
O numero 0 eh par.
O numero 2 eh par.
O numero 4 eh par.
O numero 6 eh par.
O numero 8 eh par.
```

O loop for irá executar o bloco interior tantas vezes quanto for necessário, até iterar sobre o último elemento determinado.

Loops for são indicados quando se conhece o fim do conjunto sobre o qual está se iterando ou operando durante às iterações.

While

Realiza as operações contidas no interior de seu bloco, enquanto a condição passada ao mesmo seja avaliada como verdadeira. É ideal para os casos onde o fim é desconhecido.

Note que, apesar de sintaticamente diferentes, ambos for e while possuem às mesmas capacidades.

- Exemplo:

```
1   x = 0 #Inicializamos uma variavel que sera nossa contadora
2   while(x < 5):#Inicio do loop While, com nosso criterio de parada sendo "execute ENQUANTO x < 5"
3       print x
4       x+=1 #Incrementamos nossa contadora #NAO PODEMOS ESQUECER DISSO!
5
6   #Um exemplo mais pratico
7   #Somatorio
8   #Funcao: 2X + 4
9   somatorio = 0
10  limite_inf = 0
11  limite_sup = 5
12
13  while(limite_inf <= limite_sup):
14      somatorio += 2*limite_inf + 4
15      limite_inf += 1
16  print somatorio
17  #Resultado deve ser 54
```

```
1  contador = 0
2  while (contador < 44):
3      print "Volta o cao arrependido, com suas orelhas tao fartas, com seu osso moido, e com o rabo entre as patas..."
4      contador+=1
5
6  for i in range(0, 44, 1):
7      print "Volta o cao arrependido, com suas orelhas tao fartas, com seu osso moido, e com o rabo entre as patas..."
8
```

range()

A “range()” é uma função com capacidade de gerar um sequência numérica iterável (Futuramente, trataremos isto como uma “Lista Numérica”). Geralmente, é usada com loops for, porém seu uso não é exclusivo de tal estrutura.

Sintaxe:

```python
valores = range(5)
print valores # [0, 1, 2, 3, 4]

#range pode ser definida, em sua forma completa, da seguinte forma: range(inicio, fim, passo), onde:
#Inicio: De onde o intervalo de geracao devera partir. O intervalo sera fechado para este valor, ou seja, o mesmo sera incluido no conjunto
#Fim: Onde o intervalo devera terminar. O intervalo sera aberto para este valor, logo, o mesmo nao sera incluido no conjunto
#Passo: De quanto em quanto o valor ira progredir (aumentar ou decrementar)
#Em outras formas desta funcao, sao aceitos 2 parametros ou apenas 1

print "Exemplos"
print range (1, 10, 2) #De 1 a 9, "contando" de 2 em 2
print range (0, 5, 1) #Equivalente a range(5)
print range (15, 0, -5) #Decremental

#Forma de 2 parametros: inicio e fim
print range (1, 20)

#Forma de 1 parametro: apenas o fim (considera o inicio como 0)
print range(10)
```

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/Range$ python funcrange.py
[0, 1, 2, 3, 4]
Exemplos
[1, 3, 5, 7, 9]
[0, 1, 2, 3, 4]
[15, 10, 5]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

len()

Existe uma função em python (aplicável à tipos baseados em sequência) que retorna a quantidade de valores que compõe o elemento. É a função len().

Será útil para iterarmos sobre listas, strings, dicionários e etc.

Sintaxe:

```
1  umafrase = "Qual o sentido da vida, do universo e etc?"
2  print(len(umafrase))#Ira retornar um valor inteiro, nesse caso, 42
3
4  print(len(range(0,5,1)))#Combinacao interessante esta...
5
```

- Um exemplo mais prático, um contador de letras para uma palavra:

```
1  nome = "Irineu"
2  for i in range(len(nome)):
3      print "Achei", i+1, "letra(s)"
4
```

break e continue

Algumas vezes, podemos desejar que o nossa repetição termine antes do esperado. Isto pode vir a permitir ganho em eficiência, pois caso já tenha sido alcançado o objetivo do loop, não há razão para manter o mesmo rodando.

- Observe o exemplo abaixo

```
1  #Este programa pretende encontrar a primeira ocorrencia da letra "T" em uma palavra
2  nome = "Alan Turing"
3  for letra in nome: #Lembre-se, podemos iterar sobre qualquer sequencia
4      print "Li a letra: ", letra
5      if(letra == "T"):
6          print "Encontrei a letra T, ja posso terminar"#Ja encontrei, porem talvez ainda tenha que executar mais coisas
7
```

Outra situação possível é quando desejamos que uma das iterações simplesmente “não ocorra”, mas ainda assim, há outras que devem ocorrer.

- Execução:

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/BreakContinue$ python breakcontinue.py
Li a letra:  A
Li a letra:  l
Li a letra:  a
Li a letra:  n
Li a letra:
Li a letra:  T
Encontrei a letra T, ja posso terminar
Li a letra:  u
Li a letra:  r
Li a letra:  i
Li a letra:  n
Li a letra:  g
```

Python provê duas instruções para controle de execução de loops:

- break: termina a execução do loop “mais interno”;
- continue: “Pula” um ciclo de iterações do loop “mais interno”;

Aplicando ao exemplo anterior:

```
1  #Este programa pretende encontrar a primeira ocorrencia da letra "T" em uma palavra
2  nome = "Alan Turing"
3  for letra in nome: #Lembre-se, podemos iterar sobre qualquer sequencia
4      print "Li a letra: ", letra
5      if(letra == "T"):
6          print "Encontrei a letra T, ja posso terminar"#Ja encontrei, porem talvez ainda tenha que executar mais coisas
7          break;
8
```

- Execução:

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/BreakContinue$ python breakcontinue2.py
Li a letra:  A
Li a letra:  l
Li a letra:  a
Li a letra:  n
Li a letra:
Li a letra:  T
Encontrei a letra T, ja posso terminar
```

- Já para o uso do continue, uma possível situação seria:

```
1  #Imprimir apenas consoantes
2  nome = "Alan Kay"
3  for letra in nome:
4      if(letra in "aeiouAEIOU"): # Se a letra estiver dentro deste conjunto
5          continue # Uso do continue
6      print letra
7
```

```
antonio@natsuki:~/Documentos/Ificina/Educacional/Minicursos/MinicursoPythonBásico/Slides/Exemplos/BreakContinue$ python breakcontinue3.py
l
n

K
y
```

Exercício #9 (5 min.)

(Analítico) Observe os trechos de código a seguir:

```
for i in range(1, 9):
    if(i%2 == 0):
        print i
        break
    print "Proximo!"


i = 0
while i < 9:
    i+=1
    if(i%2 == 0):
        print i
        continue
    print "Proximo!"
```

- Possuem alguma diferença quanto à semântica (aquilo que “fazem”)?
- O que acontecerá, no caso do primeiro em especial, quando mudarmos a instrução de “break” para “continue”?

Exercício #10 (15 min.)

- Dados nossos novos conhecimentos sobre loops, elaborem um programa que receba um valor “n” e retorne o n! (fatorial). Lembrando:

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

- Implemente o mesmo através de for;
- Implemente o mesmo através de while;

- Dica: Possivelmente o exercício mais complexo até o momento, mas nada impossível. Use uma variável auxiliar para o valor final. Atenção também na função range(). Por simplicidade, considere apenas valores pequenos.

Sequências

Já citamos algumas vezes ao longo deste curso esta nomenclatura, “sequências”, mas o que seriam?

Sequências, mais formalmente, são agrupamentos de dados com alguma “afinidade definida”, armazenados sob uma estrutura mais homogênea. Basicamente, são estruturas que permitem o armazenamento de múltiplos valores, de forma que os mesmos sejam reconhecidos como uma entidade única.

Alguns autores também nomeiam sequências como “coleções” ou também, “containers” (não confundir com containers a nível de sistema, por favor!).

List

Listas são uma das mais importantes estruturas de dados em python, e podemos facilmente notar sua utilidade. Uma lista consiste em uma coleção heterogênea de objetos, estes objetos sendo ditos “elementos da lista”.

Esta heterogeneidade diz respeito ao fato que os valores atribuídos em uma lista, não precisam necessariamente, ser do mesmo tipo.

Funcionalmente, uma lista é bastante parecida com um outro tipo já estudado: Strings.

Outro detalhe sobre listas é que, como iremos perceber, as mesmas estão intimamente ligadas às estruturas de repetição.

Em uma analogia com o mundo real, uma lista poderia ser interpretada, literalmente, como uma lista de compras, de amigos, de tarefas, etc...

Sintaxe:

```
1   lista_amigos = ["Lucoa", "Kobayashi", "Fafnir", "Kanna"] # Uma lista de Strings, com nomes
2
3   lista_compras = ["Frango", "Peixe", "Ovos"] # Lista de compras
4
5   lista_numeros = [0, 5, 10, 15] # Lista de numeros
6
7   lista_numeros_gerada = range(0, 10, 1) # Lista de numeros, gerada por range()
8
9   lista_de_coisas_aleatorias = ["Python", 3.14, 66] # Lista de valores aleatorios, String, float e int
10
11  palavra = ['M', 'A', 'I', 'D'] # Lista de caracteres. Pode ser interpretada como uma unica String
```

Para aqueles que já possuem conhecimento de alguma outra linguagem de programação, podem estranhar o fato de trabalharmos com listas antes de arrays (vetores e matrizes), mas, em python, por convenção, tais tipos não são muito comuns de se usar. Além disso, temos que a lista é uma estrutura bem mais generalista que os arrays, sendo que não há nenhuma implementação baseada em arrays que não possa ser trabalhada da mesma forma com listas.

- Para trabalhar necessariamente com arrays, são necessárias outras técnicas.

Diferentemente de outras linguagens, em Python, não é necessário especificar muitos parâmetros de uma Lista.

Assim como em Java, listas podem ser redimensionadas facilmente, além de o acesso a seus itens ser bastante simplificado.

Diferentemente de outros tipos bem parecidos, como vetores, por exemplo, não é necessário a especificação de um “tamanho” para a lista.

Valores em uma lista podem ser acessados por um índice (numérico). Tais índices começam em 0, como na maioria das linguagens.

- Acesso a uma lista:

```
1  lista_compras = ["Macarrao Instantaneo","Frango","Mortadela", "Peixe", "Ovos"] # Lista de compras
2
3  oquetemprahoje = lista_compras[0] # Acesso pelo indice.
4
5  print oquetemprahoje # Imprime "Macarrao Instantaneo"
6
7  oquetempraamanha = lista_compras[3]
8
9  print oquetempraamanha # Imprime "Peixe"
10
```

Operações sobre listas

Listas em python admitem algumas operações. Dentre elas:

- Inclusão;
- Remoção;
- Ordenação;
- Reversão;

```
so_bons = ["Princess Mononoke", "Fargo", "Predator", "Terminator 2"] # Nossa lista, alguns titulos bons aqui...

print so_bons # ["Princess Mononoke", "Fargo", "Predator", "Terminator 2"]

#ADICAO

so_bons.append("Batman Forever") # Operacao de adicao. Devera adicionar "Batman Forever" como ultimo item da lista

print so_bons # ["Princess Mononoke", "Fargo", "Predator", "Terminator 2", "Batman Forever"]

so_bons.append("Batman Vs. Superman")
#DELECAO

so_bons.remove("Batman Forever") # Com remove(), podemos remover um objeto da lista diretamente pela sua referencia

print so_bons # ["Princess Mononoke", "Fargo", "Predator", "Terminator 2", "Batman Vs. Superman"]

del(so_bons[4]) # Ja utilizando a Keyword del(), eh necessario passarmos o objeto completo, ou seja, o item da lista com seu indice para remover

print so_bons

#ORDENACAO

so_bons.sort() # Ordena a lista (nesse caso, por ordem alfabetica)

print so_bons

lista_num = [5, 2, 3, 7, 11]

print lista_num

lista_num.sort() # Ordena a lista de numeros

print "Ordenada: ", lista_num

#REVERSAO

print so_bons

so_bons.reverse() # Coloca a lista "ao contrario"

print so_bons

```

Trabalhando com Listas

- Iterando sobre os elementos de uma lista:

```
1  series_computacao = ["Black Mirror", "Mr. Robot", "Westworld", "Halt And Catch Fire"]
2
3  for iterador in series_computacao:
4      print iterador
```

- Concatenando Listas (similarmente à strings):

```
1  series_computacao = ["Black Mirror", "Mr. Robot", "Westworld", "Halt And Catch Fire"]
2
3  filmes_computacao = ["O Jogo da Imitacao", "Piratas do Vale do Silicio", "Revolution OS", "Ex_Machina", "Summer Wars", "Jurassic Park"]
4
5  filmes_e_series = series_computacao + filmes_computacao
6
7  print filmes_e_series # Ambas listas concatenadas
```

Exercício #11 (10 min.)

- Dadas as listas a seguir:

  lista_charme = [“casa”, “bonito”, “lanterna”]

  lista_funk = [“casa”, “lanterna”, “elegante”]

- Elabore um algoritmo que faça a diferença entre as duas listas, isto é, compare ambas elemento a elemento e diga quais elementos estão contidos apenas em um dos conjuntos.
- Dica: O jeito mais simples de fazer isso é utilizando um loop para ambos os conjuntos.

Dúvidas?
Sugestões?
Críticas?