ESTRUTURA DE DADOS

Revisão

Variáveis

Nomes de variáveis
Em Python, nomes de variáveis devem iniciar obrigatoriamente com uma letra, mas podem conter números e o símbolo sublinha (_).

A versão 3 da linguagem Python permite a utilização de acentos em nomes de variáveis, pois, por padrão, os programas são interpretados utilizando-se um conjunto de caracteres chamado UTF-8, capaz de representar praticamente todas as letras dos alfabetos conhecidos.

Variáveis

- Variáveis numéricas
  Dizemos que uma variável é numérica quando armazena números inteiros ou de ponto flutuante. Os números inteiros são aqueles sem parte decimal, como 1, 0, -5, 550, -47, 30000. Números de ponto flutuante ou decimais são aqueles com parte decimal, como 1.0, 5.478, 10.478, 30000.4. Observe que 1.0, mesmo tendo zero na parte decimal, é um número de ponto flutuante.
  Em Python, e na maioria das linguagens de programação, utilizamos o ponto, e não a vírgula, como separador entre a parte inteira e fracionária de um número.

Variáveis

- Variáveis do tipo Lógico
  Muitas vezes, queremos armazenar um conteúdo simples: verdadeiro ou falso em uma variável. Nesse caso, utilizaremos um tipo de variável chamado tipo lógico ou booleano. Em Python, escreveremos **True** para verdadeiro e **False** para falso (observe que o T e o F são escritos com letras maiúsculas).

```
resultado = True
aprovado = False
```

- Variáveis string
  Variáveis do tipo string armazenam cadeias de caracteres como nomes e textos em geral. Chamamos cadeia de caracteres uma sequência de símbolos como letras, números, sinais de pontuação etc. Para possibilitar a separação entre o texto do seu programa e o conteúdo de uma string, utilizaremos aspas (") para delimitar o início e o fim da sequência de caracteres.

Operadores

- Operadores relacionais - Para realizarmos comparações lógicas, utilizaremos operadores relacionais. A lista de operadores relacionais suportados em Python (veja quadro).
- O resultado de uma comparação é um valor do tipo lógico, ou seja, **True** (verdadeiro) ou **False** (falso).
- Utilizaremos o verbo "avaliar" para indicar a resolução de uma expressão.

Operador	Operação	Símbolo matemático
==	igualdade	=
>	maior que	>
<	menor que	<
!=	diferente	≠
>=	maior ou igual	≥
<=	menor ou igual	≤

- Operadores lógicos
  Para agrupar operações com lógica booleana, utilizaremos operadores lógicos. Python suporta três operadores básicos: **not** (não), **and** (e), **or** (ou). Esses operadores podem ser traduzidos como não (¬ negação), e (Λ conjunção) e ou (V disjunção).

Operador Python	Operação
not	não
and	e
or	ou

Entrada de Dados

- A função **input** é utilizada para solicitar dados do usuário.

```
x = input("Digite um número: ")
print(x)
```

- Conversão da entrada de dados - A função **input** sempre retorna valores do tipo string, ou seja, não importa se digitamos apenas números, o resultado sempre é string. Utilizamos então a **função int** ou a **função float**, conforme nossa necessidade.

```
anos = int(input("Anos de serviço: "))
valor_por_ano = float(input("Valor por ano: "))
```

Condições

- **if**
  As condições servem para selecionar quando uma parte do programa deve ser ativada e quando deve ser simplesmente ignorada. Em Python, a estrutura de decisão é o **if**.
- Formato da estrutura de condicional if:
  if `<condição>:`
  `bloco verdadeiro`
- Python é uma das poucas linguagens de programação que utiliza o deslocamento do texto à direita (recuo) para marcar o início e o fim de um bloco. Outras linguagens contam com palavras especiais para isso, como `BEGIN` e `END`, em Pascal; ou as famosas chaves (`{` e `}`), em C e Java.

- **else**
  Quando há problemas, onde a segunda condição é simplesmente o inverso da primeira, podemos usar outra forma de **if** para simplificar os programas. Essa forma é a cláusula **else** para especificar o que fazer caso o resultado da avaliação da condição seja falso, sem precisarmos de um novo **if**.
- A vantagem de usar **else** é deixar os programas mais claros, uma vez que podemos expressar o que fazer caso a condição especificada em **if** seja falsa.

- **elif**
  Python apresenta uma solução muito interessante ao problema de múltiplos **if**s aninhados. A cláusula **elif** substitui um par **else if**, mas sem criar outro nível de estrutura, evitando problemas de deslocamentos desnecessários à direita.

Repetições

- Repetições representam a base de vários programas. São utilizadas para executar a mesma parte de um programa várias vezes, normalmente dependendo de uma condição.
- Uma das estruturas de repetição do Python é o **while**, que repete um bloco enquanto a condição for verdadeira.
- Formato da estrutura de repetição com while
  while `<condição>:`
  `bloco`
- Interrompendo a repetição - Embora muito útil, a estrutura **while** só verifica sua condição de parada no início de cada repetição. Dependendo do problema, a habilidade de terminar **while** dentro do bloco a repetir pode ser interessante. A instrução **break** é utilizada para interromper a execução de **while** independentemente do valor atual de sua condição.

Repetições

- Python apresenta uma estrutura de repetição especialmente projetada para percorrer listas. A instrução **for** funciona de forma parecida a **while**, mas a cada repetição utiliza um elemento diferente da lista.
  A cada repetição, o próximo elemento da lista é utilizado, o que se repete até o fim da lista.
- Exemplo:

```
L=[8,9,15]
for e in L:
        print(e)
```

- Embora a instrução **for** facilite nosso trabalho, ela não substitui completamente **while**. Dependendo do problema, utilizaremos **for** ou **while**. Normalmente utilizaremos **for** quando quisermos processar os elementos de uma lista, um a um. **while** é indicado para repetições nas quais não sabemos ainda quantas vezes vamos repetir ou onde manipulamos os índices de forma não sequencial. Vale lembrar que a instrução **break** também interrompe o **for**.

Listas

- Listas são um tipo de variável que permite o armazenamento de vários valores, acessados por um índice. Uma lista pode conter zero ou mais elementos de um mesmo tipo ou de tipos diversos, podendo inclusive conter outras listas. O tamanho de uma lista é igual à quantidade de elementos que ela contém.
- Uma lista vazia
  `L=[]`
- Uma lista com três elementos
  `Z=[ 15, 8, 9 ]`

Listas

- Acesso a uma lista (índice: sempre se inicia por 0 (zero))

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[1]
8
>>> Z[2]
9
```

Listas

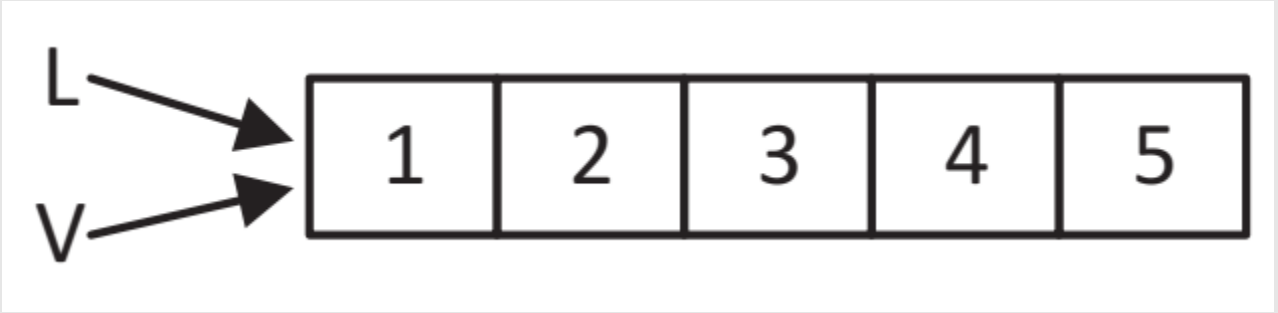
- Modificação de uma lista (utilizando o nome da lista e o índice)

```
>>> Z=[15,8,9]
>>> Z[0]
15
>>> Z[0]=7
>>> Z[0]
7
>>> Z
[7, 8, 9]
```

Listas

- Tentativa de copiar listas (Quando modificarmos V, modificamos também o conteúdo de L. Isso porque uma lista em Python é um objeto e, quando atribuímos um objeto a outro, estamos apenas copiando a mesma referência da lista, e não seus dados em si)

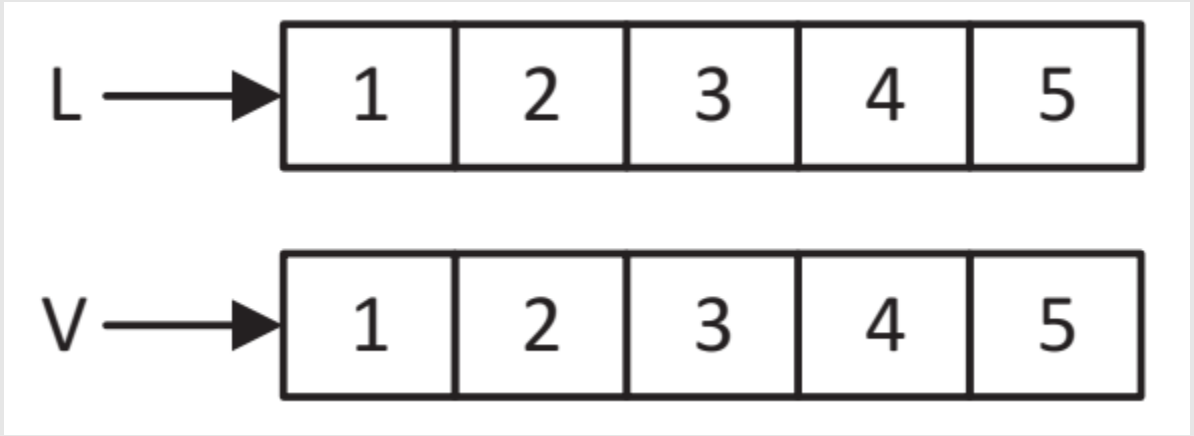
```
>>> L=[1,2,3,4,5]
>>> V=L
>>> L
[1, 2, 3, 4, 5]
>>> V
[1, 2, 3, 4, 5]
>>> V[0]=6
>>> V
[6, 2, 3, 4, 5]
>>> L
[6, 2, 3, 4, 5]
```



Listas

- Cópia de listas (Ao escrevermos `L[:]`, estamos nos referindo a uma nova cópia de `L`)

```
>>> L=[1,2,3,4,5]
>>> V=L[:]
>>> V[0]=6
>>> L
[1, 2, 3, 4, 5]
>>> V
[6, 2, 3, 4, 5]
```



Listas

- Fatiamento de listas (Veja que índices negativos também funcionam. Um índice negativo começa a contar do último elemento, mas observe que começamos de `-1`.

```
>>> L=[1,2,3,4,5]
>>> L[0:5]
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[:-1]
[1, 2, 3, 4]
>>> L[1:3]
[2, 3]
>>> L[1:4]
[2, 3, 4]
>>> L[3:]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[-1]
5
>>> L[-2]
4
```

Listas

- Tamanho de listas (A função **`len`** pode ser utilizada em repetições para controlar o limite dos índices)

```
>>> L=[12,9,5]
>>> len(L)
3
>>> V=[]
>>> len(V)
0
```

Listas – Métodos de objetos do tipo lista

- list.append(x)
- Adiciona um item ao fim da lista. Equivalente a a[len(a):] = [x].

- list.extend(iterable)
- Prolonga a lista, adicionando no fim todos os elementos do argumento iterable passado como parâmetro. Equivalente a a[len(a):] = iterable.

- list.insert(i, x)
- Insere um item em uma dada posição. O primeiro argumento é o índice do elemento antes do qual será feita a inserção, assim a.insert(0, x) insere um elemento na frente da lista e a.insert(len(a), x) e equivale a a.append(x).

Listas – Métodos de objetos do tipo lista

- list.remove(x)
- Remove o primeiro item encontrado na lista cujo valor é igual a x. Se não existir valor igual, uma exceção ValueError é levantada.

- list.pop([i])
- Remove um item em uma dada posição na lista e o retorna. Se nenhum índice é especificado, a.pop() remove e devolve o último item da lista. (Os colchetes ao redor do i na demonstração do método indica que o parâmetro é opcional, e não que é necessário escrever estes colchetes ao chamar o método).

Listas – Métodos de objetos do tipo lista

- list.clear()
- Remove todos os itens de uma lista. Equivalente a del a[:].

- list.index(x[, start[, end]])
- Devolve o índice base-zero do primeiro item cujo valor é igual a x, levantando ValueError se este valor não existe.
- Os argumentos opcionais start e end são interpretados como nas notações de fatiamento e são usados para limitar a busca para uma subsequência específica da lista.

Listas – Métodos de objetos do tipo lista

- list.count(x)
- Devolve o número de vezes em que x aparece na lista.

- list.sort(*, key=None, reverse=False)
- Ordena os itens na lista (os argumentos podem ser usados para personalizar a ordenação, veja a função sorted() para maiores explicações).

- list.reverse()
- Inverte a ordem dos elementos na lista.

- list.copy()
- Devolve uma cópia rasa da lista. Equivalente a a[:].

Listas como Filas

- Uma lista pode ser utilizada como fila se obedecermos a certas regras de inclusão e eliminação de elementos. Em uma fila, a inclusão é sempre realizada no fim, e as remoções são feitas no início. Dizemos que o primeiro a chegar é o primeiro a sair (*FIFO - First In First Out*).
- Porém, listas não são eficientes para esta finalidade. Embora appends e pops no final da lista sejam rápidos, fazer inserts ou pops no início da lista é lento (porque todos os demais elementos têm que ser deslocados).
- Para implementar uma fila, use a classe ***collections.deque*** que foi projetada para permitir appends e pops eficientes nas duas extremidades (como veremos em aulas futuras).

Listas como Pilhas

- Uma pilha tem uma política de acesso bem definida: novos elementos são adicionados ao topo. A retirada de elementos também é feita pelo topo.
- Na pilha, o último elemento a chegar é o primeiro a sair (*LIFO - Last In First Out*).
- Os métodos de lista tornam muito fácil utilizar listas como pilhas, onde o item adicionado por último é o primeiro a ser recuperado. Para adicionar um item ao topo da pilha, use append(). Para recuperar um item do topo da pilha use pop() sem nenhum índice.

Pesquisa

- Podemos pesquisar se um elemento está ou não em uma lista, verificando do primeiro ao último elemento se o valor procurado estiver presente.
- Exemplo: Pesquisa sequencial

```
L=[15,7,27,39]
p=int(input("Digite o valor a procurar:"))
achou=False
x=0
while x<len(L):
    if L[x]==p:
        achou=True
          break
     x+=1
if achou:
     print("%d achado na posição %d" % (p,x))
else:
    print("%d não encontrado" % p)
```

Usando for

- Python apresenta uma estrutura de repetição especialmente projetada para percorrer listas. A instrução **for** funciona de forma parecida a **while**, mas a cada repetição utiliza um elemento diferente da lista. A cada repetição, o próximo elemento da lista é utilizado, o que se repete até o fim da lista.
- ```
  L=[8,9,15]
  for e in L:
      print(e)
  ```

  Impressão de todos os elementos da lista com **while:**

  ```
  L=[8,9,15]
  x=0
  while x<len(L):
     e=L[x]
     print(e)
     x+=1
  ```
  ```

Range

- Podemos utilizar a função `range` para gerar listas simples. A função `range` não retorna uma lista propriamente dita, mas um gerador ou *generator*.
- **for** `v` **in** `range(10):`
  print`(v)` `#A função range gerou números de 0 a 9 porque passamos 10 como parâmetro.`
- **for** `v` **in** `range(5, 8):`
  print`(v)` `#Usando 5 como início e 8 como fim, vamos imprimir os números 5, 6 e 7.`
- **for** `t` **in** `range(3,33,3):`
  print`(t, end=" ")`
  print`()` `#Se acrescentarmos um terceiro parâmetro à função range, teremos como saltar entre os valores gerados, por exemplo, range(0,10,2) gera os pares entre 0 e 10, pois começa de 0 e adiciona 2 a cada elemento.`
- `L=list(range(100,1100,50))`
  print`(L)` `#Para transformar um gerador em lista, utilize a função list`

Enumerate

- Com a **função `enumerate`** podemos ampliar as funcionalidades de **for** facilmente.
- Impressão de índices sem usar a função enumerate

```
L=[5,9,13]
x=0
for e in L:
        print("[%d] %d" % (x,e))
        x+=1
```

- Impressão de índices usando a função enumerate

```
L=[5,9,13]
for x, e in enumerate(L):
        print("[%d] %d" % (x,e))
```

A função `enumerate` gera uma tupla em que o primeiro valor é o índice e o segundo é o elemento da lista sendo enumerada. Ao utilizarmos `x, e` em **for**, indicamos que o primeiro valor da tupla deve ser colocado em `x`, e o segundo, em `e`.

Listas com strings

- Strings podem ser indexadas ou acessadas letra por letra. Listas em Python funcionam da mesma forma, permitindo o acesso a vários valores e se tornando uma das principais estruturas de programação da linguagem.
- Listas com strings

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(len(S))
3
>>> print(S[0])
maçãs
>>> print(S[1])
peras
>>> print(S[2])
kiwis
```

Listas dentro de listas

- Podemos acessar as strings dentro da lista, letra por letra, usando um segundo índice.
- Listas com strings, acessando letras

```
>>> S=["maçãs", "peras", "kiwis"]
>>> print(S[0][0])
m
>>> print(S[0][1])
a
>>> print(S[1][1])
e
>>> print(S[2][2])
w
```

Listas com elementos diferentes

- Os elementos de uma lista não precisam ser do mesmo tipo.
- Listas com elementos de tipos diferentes

```
produto1 = [ "maçã", 10, 0.30]
produto2 = [ "pera", 5, 0.75]
produto3 = [ "kiwi", 4, 0.98]
```

Ordenação

- Para ordenar uma lista, realizaremos uma operação semelhante à da pesquisa, mas trocando a ordem dos elementos quando necessário.

Ordenação Bolha ou “Bubble Sort”

- A ordenação pelo método de bolhas consiste em comparar dois elementos a cada vez. Se o valor do primeiro elemento for maior que o do segundo, eles trocarão de posição.
- Essa operação é então repetida para o próximo elemento até o fim da lista. O método de bolhas exige que percorramos a lista várias vezes.
- Por isso, utilizaremos um marcador para saber se chegamos ao fim da lista trocando ou não algum elemento.
- Esse método tem outra propriedade, que é posicionar o maior elemento na última posição da lista, ou seja, sua posição correta. Isso permite eliminar um elemento do fim da lista a cada passagem completa.

- Ordenação pelo método de bolhas

```
L=[7,4,3,12,8]
fim=5
while fim > 1:
    trocou=False
    x=0
    while x<(fim-1):
        if L[x] > L[x+1]:
            trocou=True
            temp=L[x]
            L[x]=L[x+1]
           L[x+1]=temp
       x+=1
    if not trocou:
        break
fim-=1

for e in L:
    print(e)
```

Dicionários

- Dicionários consistem em uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes. Um dicionário é composto por um conjunto de chaves e valores.
- O dicionário em si consiste em relacionar uma chave a um valor específico.
- Em Python, criamos dicionários utilizando chaves ({}). Cada elemento do dicionário é uma combinação de chave e valor.
- Criação de um dicionário

```
tabela = { "Alface": 0.45,
"Batata": 1.20,
"Tomate": 2.30,
"Feijão": 1.50 }
```

Produto	Preço
Alface	R$ 0,45
Batata	R$ 1,20
Tomate	R$ 2,30
Feijão	R$ 1,50

Dicionários

- Um dicionário é acessado por suas chaves. No exemplo do slide anterior, para obter o preço da alface, digite no interpretador, depois de ter criado a tabela, `tabela["Alface"]`, onde `tabela` é o nome da variável do tipo dicionário, e “Alface” é nossa chave. O valor retornado é o mesmo que associamos na tabela, ou seja, 0,45.
- Diferentemente de listas, onde o índice é um número, dicionários utilizam suas chaves como índice. Quando atribuímos um valor a uma chave, duas coisas podem ocorrer:
  1. Se a chave já existe: o valor associado é alterado para o novo valor.
  2. Se a chave não existe: a nova chave será adicionada ao dicionário.
- Se a chave não existir, uma exceção do tipo `KeyError` será ativada. Para verificar se uma chave pertence ao dicionário, podemos usar o operador **in.**

Dicionários

- Exemplo:
- ```
  >>> tabela = { "Alface": 0.45,
  ...            "Batata": 1.20,
  ...            "Tomate": 2.30,
  ...            "Feijão": 1.50 }
  >>> print("Manga" in tabela)
  False
  ```
  ```

Dicionários

- Podemos também obter uma lista com as chaves do dicionário, ou mesmo uma lista dos valores associados:
- Obtenção de uma lista de chaves e valores

```
>>> tabela = { "Alface": 0.45,
... "Batata": 1.20,
... "Tomate": 2.30,
... "Feijão": 1.50 }
>>> print(tabela.keys())
dict_keys(['Batata', 'Alface', 'Tomate', 'Feijão'])
>>> print(tabela.values())
dict_values([1.2, 0.45, 2.3, 1.5])
```

Dicionários

- Para apagar uma chave, utilizaremos a instrução **del** .
-
```
>>> tabela = { "Alface": 0.45,
... "Batata": 1.20,
... "Tomate": 2.30,
... "Feijão": 1.50 }
>>> del tabela["Tomate"]
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Feijão': 1.5}
```

Tuplas

- Tuplas podem ser vistas como listas em Python, com a grande diferença de serem **imutáveis**.
- Tuplas são ideais para representar listas de valores constantes e também para realizar operações de empacotamento e desempacotamento de valores.
- Tuplas são criadas de forma semelhante às listas, mas utilizamos parênteses em vez de colchetes.

Tuplas

- Podemos também criar tuplas vazias, escrevendo apenas os parênteses:

```
>>> t4=()
>>> t4
()
>>> len(t4)
0
```

Tuplas também podem ser criadas a partir de listas, utilizando-se a função `tuple`:

```
>>> L=[1,2,3]
>>> T=tuple(L)
>>> T
(1, 2, 3)
```

Embora não possamos alterar uma tupla depois de sua criação, podemos concatená-las, gerando novas tuplas:

```
>>> t1=(1,2,3)
>>> t2=(4,5,6)
>>> t1+t2
(1, 2, 3, 4, 5, 6)
```