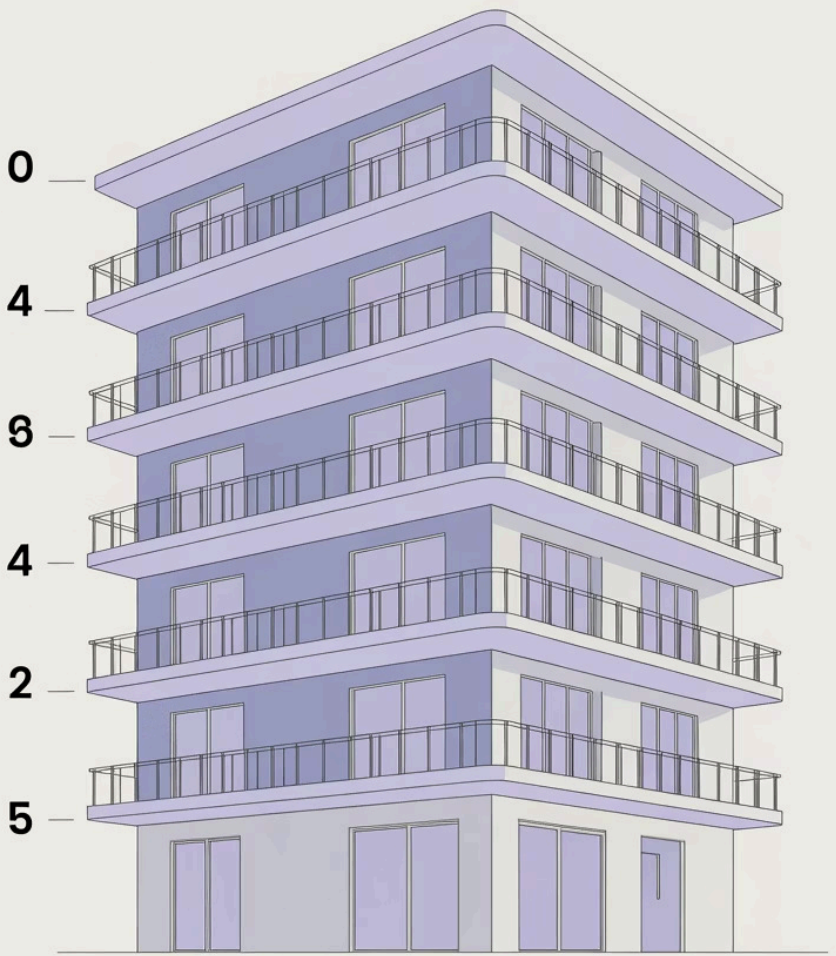
Listas em Python

Domine o poder das estruturas de dados mais versáteis da programação

Começar a aprender

Ver exemplos



O que são 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.

Podemos imaginar uma lista como um edifício de apartamentos, onde o térreo é o andar zero, o primeiro andar é o andar 1 e assim por diante. O índice é utilizado para especificarmos o "apartamento" onde guardaremos nossos dados.

Flexibilidade Total

Listas são mais flexíveis que prédios e podem crescer ou diminuir com o tempo durante a execução do programa.

Criando Listas em Python

01

Lista Vazia

Crie uma lista sem elementos usando colchetes vazios

```
L = []
```

02

Lista com Elementos

Inicialize uma lista com valores separados por vírgulas

```
Z = [15, 8, 9]
```

03

Acessando Elementos

Use índices entre colchetes para acessar valores

```
Z[0]  # Retorna 15
Z[1]  # Retorna 8
Z[2]  # Retorna 9
```

Os colchetes ([ ]) após o símbolo de igualdade servem para indicar que estamos criando uma lista. Em um prédio de seis andares, teremos números de andar variando entre 0 e 5. Se chamarmos nosso prédio de P, teremos P[0] como o endereço do térreo, P[1] como endereço do primeiro andar, continuando assim até P[5].

Modificando Elementos da Lista

Utilizando o nome da lista e um índice, podemos mudar o conteúdo de um elemento. Quando criamos a lista Z, o primeiro elemento era o número 15. Por isso, Z[0] era 15.

Quando executamos Z[0]=7, alteramos o conteúdo do primeiro elemento para 7. Isso pode ser verificado quando pedimos para exibir Z, agora com 7, 8 e 9 como elementos.

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

Exemplo Prático: Calculando Médias

Vejamos um exemplo onde um aluno tem cinco notas e desejamos calcular sua média aritmética. Uma grande vantagem desse programa é que não precisamos declarar cinco variáveis para guardar as cinco notas. Todas as notas foram armazenadas na lista, utilizando um índice para identificar ou acessar cada valor.

```
notas = [6, 7, 5, 8, 9]
soma = 0
x = 0
while x < 5:
    soma += notas[x]
    x += 1
print("Média: %5.2f" % (soma/x))
```

Criar Lista

Inicializar com 5 notas

Iterar

Percorrer cada elemento

Somar

Acumular valores

Calcular

Dividir pela quantidade

Lendo Notas do Usuário

Podemos modificar o exemplo anterior para ler as notas uma a uma do usuário. Criamos a lista de notas com cinco elementos, todos zero. Utilizamos a repetição para ler as notas do aluno e armazená-las na lista.

Terminada a primeira repetição, teremos a lista de notas preenchidas. Para imprimir a lista de notas, reinicializamos o valor da variável x para 0 e criamos outra estrutura de repetição.

```
notas = [0, 0, 0, 0, 0]
soma = 0
x = 0
while x < 5:
 notas[x] = float(input("Nota %d:" % x))
 soma += notas[x]
 x += 1
x = 0
while x < 5:
 print("Nota %d: %6.2f" % (x, notas[x]))
 x += 1
print("Média: %5.2f" % (soma/x))
```

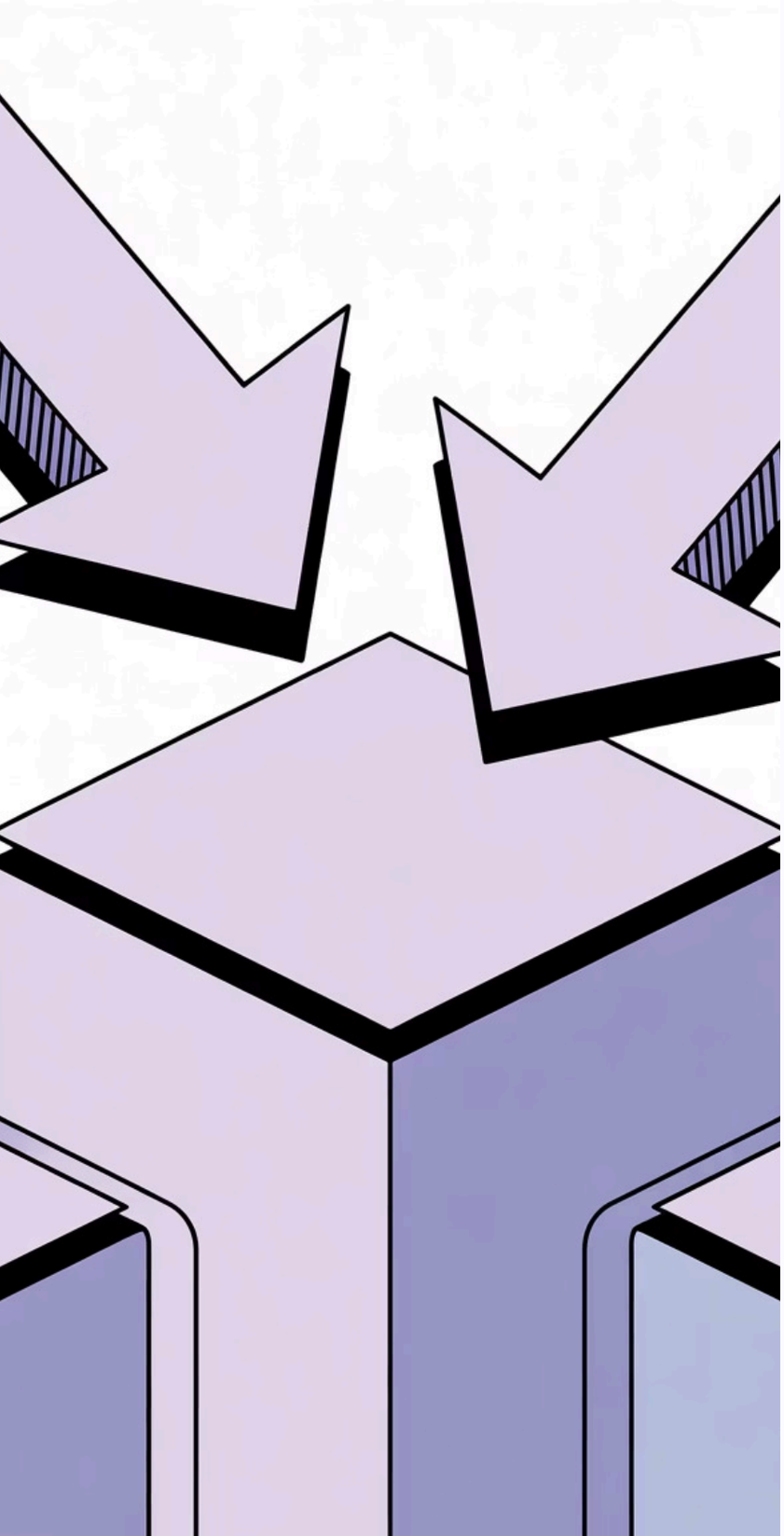
Trabalhando com Índices

Vejamos um programa que lê cinco números, armazena-os em uma lista e depois solicita que o usuário escolha um número a mostrar. O objetivo é ler 15, 12, 5, 7 e 9 e armazená-los na lista. Depois, se o usuário digitar 2, ele imprimirá o segundo número digitado.

Importante: Conversão de Índices

Observe que o índice do primeiro número é 0, e não 1. Começar a contar de 0 não é natural para a maioria das pessoas, por isso fazemos a conversão no programa.

```
números = [0, 0, 0, 0, 0]
x = 0
while x < 5:
 números[x] = int(input("Número %d:" % (x+1)))
 x += 1
while True:
 escolhido = int(input("Que posição você quer imprimir (0 para sair): "))
 if escolhido == 0:
 break
 print("Você escolheu o número: %d" % (números[escolhido-1]))
```



Cópia e Referências de Listas

Armadilha Comum

Ao atribuir uma lista a outra variável, você cria apenas uma referência, não uma cópia

```
L = [1, 2, 3, 4, 5]
V = L
V[0] = 6
# L também mudou!
```

Solução Correta

Use fatiamento completo para criar uma cópia independente

```
L = [1, 2, 3, 4, 5]
V = L[:]
V[0] = 6
# L permanece inalterado
```

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. Nesse caso, V funciona como um apelido de L, ou seja, V e L são a mesma lista.

Fatiamento de Listas

Podemos fatiar uma lista, da mesma forma que fizemos com strings. O fatiamento permite extrair partes específicas de uma lista usando índices de início e fim.

Fatiamento Completo

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

Fatiamento Parcial

```
L[1:3]  # [2, 3]
L[1:4]  # [2, 3, 4]
L[:3]   # [1, 2, 3]
L[3:]   # [4, 5]
```

Índices Negativos

```
L[:-1]  # [1, 2, 3, 4]
L[-1]   # 5
L[-2]   # 4
```

Índices negativos também funcionam. Um índice negativo começa a contar do último elemento, mas observe que começamos de -1. Assim L[0] representa o primeiro elemento; L[-1], o último; L[-2], o penúltimo, e assim por diante.

Função len(): Tamanho de Listas

Podemos usar a função len com listas. O valor retornado é igual ao número de elementos da lista.

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

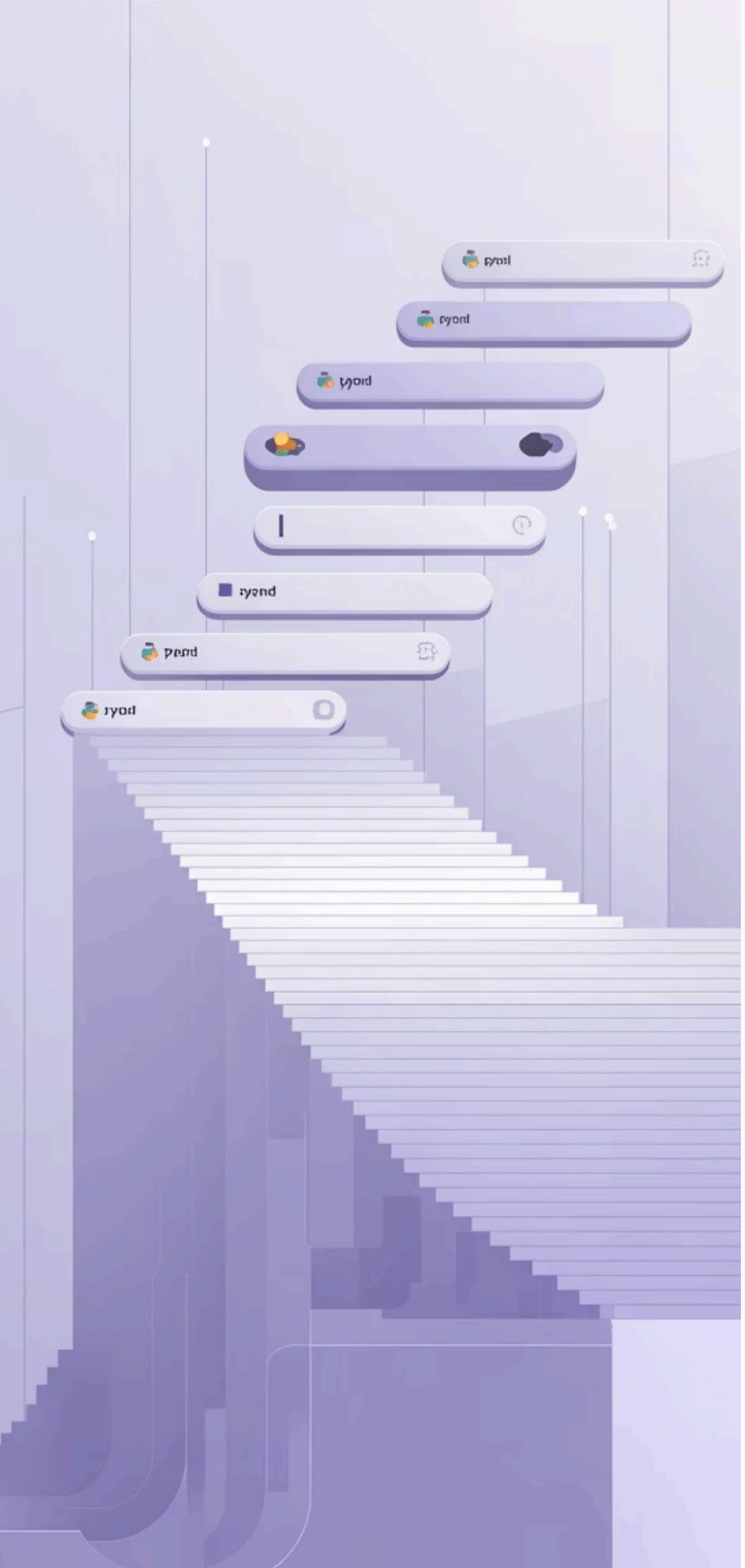
Vantagem em Repetições

A função len pode ser utilizada em repetições para controlar o limite dos índices. A vantagem é que se trocarmos o tamanho da lista, o resto do programa continuaria funcionando.

```
L = [1, 2, 3]
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

Atenção: Índices Válidos

O valor retornado pela função len não pode ser utilizado como índice. Os índices válidos de uma lista (L) variam de 0 até o valor de len(L)-1.



Adicionando Elementos com append()

Uma das principais vantagens de trabalharmos com listas é poder adicionar novos elementos durante a execução do programa. Para adicionar um elemento ao fim da lista, utilizaremos o método append.

1 Lista Vazia

```
L = []
```

2 Primeiro Elemento

```
L.append("a")
# L = ['a']
```

3 Segundo Elemento

```
L.append("b")
# L = ['a', 'b']
```

4 Terceiro Elemento

```
L.append("c")
# L = ['a', 'b', 'c']
```

Em Python, chamamos um método escrevendo o nome dele após o nome do objeto. Como listas são objetos, sendo L a lista, teremos L.append(valor). Métodos são recursos de orientação a objetos, suportados e muito usados em Python.

Programa: Lendo Números Dinamicamente

Vejamos um programa que lê números até que 0 seja digitado e depois os imprime na mesma ordem em que foram digitados. Esse simples programa é capaz de ler e imprimir um número inicialmente indeterminado de valores.

```
L = []
while True:
    n = int(input("Digite um número (0 sai):"))
    if n == 0:
        break
    L.append(n)
x = 0
while x < len(L):
    print(L[x])
    x = x + 1
```

Entrada

Usuário digita números

Adição

Números são adicionados à lista

Iteração

Percorre todos os elementos

Saída

Imprime cada número

Adição de Listas: Operador +

Adição Simples

```
>>> L = []
>>> L = L + [1]
>>> L
[1]
>>> L += [2]
>>> L
[1, 2]
>>> L += [3, 4, 5]
>>> L
[1, 2, 3, 4, 5]
```

Método extend()

```
>>> L = ["a"]
>>> L.extend(["c"])
>>> L
['a', 'c']
>>> L.extend(["f", "g", "h"])
>>> L
['a', 'b', 'c', 'f', 'g', 'h']
```

Quando adicionamos apenas um elemento, tanto L.append(1) quanto L+[1] produzem o mesmo resultado. Quando adicionamos uma lista a outra, o interpretador executa um método chamado extend que adiciona os elementos de uma lista a outra.

append( ) vs extend( ): Diferenças Importantes

Usando append() com Lista

Se você utilizar o método append com uma lista como parâmetro, em vez de adicionar os elementos no fim da lista, append adicionará a lista inteira como apenas um novo elemento.

```
>>> L = ["a"]
>>> L.append(["b"])
>>> L.append(["c", "d"])
>>> len(L)
3
>>> L[2]
['c', 'd']
```

Listas Dentro de Listas

Teremos então listas dentro de listas. Esse conceito permite a utilização de estruturas de dados mais complexas, como matrizes, árvores e registros.

```
>>> L[2][1]
'd'
>>> len(L[2])
2
```

Com extend os elementos são adicionados como elementos não como uma lista.

Removendo Elementos com del

Como o tamanho da lista pode variar, permitindo a adição de novos elementos, podemos também retirar alguns elementos da lista, ou mesmo todos eles. Para isso, utilizaremos a instrução del.

1

Remoção Individual

```
>>> L = ["a", "b", "c"]
>>> del L[1]
>>> L
['a', 'c']
```

O elemento excluído não ocupa mais lugar na lista

2

Reorganização Automática

```
>>> del L[0]
>>> L
['c']
```

Os índices são reorganizados automaticamente

3

Remoção de Fatias

```
>>> L = list(range(101))
>>> del L[1:99]
>>> L
[0, 99, 100]
```

Podemos apagar fatias inteiras de uma só vez

Resumo: Dominando Listas em Python

Estrutura Versátil

Listas armazenam múltiplos valores de qualquer tipo, com tamanho dinâmico e acesso por índice começando em 0.

Operações Essenciais

Domine append(), extend(), pop(), del, len() e fatiamento para manipular listas com eficiência.

Cuidados Importantes

Lembre-se: atribuição cria referências, não cópias. Use [:] para copiar. Índices válidos vão de 0 a len(L)-1.

Listas são fundamentais em Python e permitem criar programas poderosos e flexíveis. Com prática, você dominará essa estrutura de dados essencial e poderá construir aplicações cada vez mais sofisticadas.

Praticar exercícios | Próxima lição