Tipos Abstratos de Dados

Tipo Abstrato de Dados (TAD, em inglês *Abstract Data Type*) é uma abstração de um conceito matemático que define a estrutura de um dado e as operações que podem ser realizadas sobre ele. Em programação, os TADs são usados como uma forma de representar dados complexos e estruturados de maneira a facilitar a manipulação deles.

Alguns exemplos comuns de TADs incluem pilhas, filas, árvores, listas encadeadas e conjuntos. Cada um destes tipos define uma estrutura de dados específica e suas operações associadas, como inserção, remoção, busca e outras. Ao usar TADs, os programadores podem concentrar-se na solução do problema em questão, sem se preocupar com a implementação dos dados subjacentes.

Tipos Abstratos de Dados podem ser usados em uma variedade de contextos além de Estruturas de Dados. Eles são uma forma de encapsular informações e operações relacionadas em uma única entidade, tornando o código mais organizado e fácil de manter. Um exemplo de uso de TADs em outro contexto é na modelagem de objetos do mundo real.

Por exemplo, um TAD Pessoa pode conter informações como nome, idade, endereço e métodos para exibir essas informações e realizar operações como mudar de endereço.

Outro exemplo é na área de gráficos, onde um TAD pode ser criado para representar figuras geométricas como círculos, retângulos e polígonos, incluindo métodos para calcular área, perímetro e desenhar a figura em uma tela. De maneira geral, TADs são úteis em qualquer lugar onde seja necessário modelar entidades complexas com informações e operações relacionadas.

Mas antes de nos assustarmos com tantas possibilidades, e considerando que vamos trabalhar estes Tipos Abstratos de Dados em Python, proponho seguir um passo a passo de revisão para irmos entrando no mundo dos TADs com os pés firmados no chão.

Então, vamos começar com a implementação de uma função simples. Por ora não nos interessa sua utilidade. Vamos pensar somente em como implementá-la.

Esta função, que vamos chamá-la de `inserir`, recebe como entrada um número inteiro entre 0 e 10 e uma lista de 11 valores booleanos (True ou False). Ela altera o valor na posição correspondente ao número passado na lista para True e retorna a lista modificada. Se o número não estiver entre 0 e 10, a função retorna a mensagem "Valor inválido".

```
def inserir(num, lista):
    if num >= 0 and num <= 10:
        lista[num] = True
    else:
        print ("Valor inválido")
```

Não temos como saber se esta função está correta até testá-la. Uma boa forma seria criando uma lista e imprimindo-a após a chamada da função para verificar se a alteração foi feita.

Não era isso que eu pretendia, mas, ok, eu deixo – mas só esta vez:

```
lista = [False] * 11
print(lista)
inserir(2,lista)
print(lista)
```

O resultado será este

```
[False, False, False, False, False, False, False, False, False, False, False]
```

```
[False, False, True, False, False, False, False, False, False, False, False]
```

E veremos que na posição 2 (deixei em negrito) o valor realmente foi alterado.

Considerando que a ideia de TADs inclui o conceito de encapsulamento, ou seja, esconder informações dentro do próprio TAD, vamos trabalhar para o código em questão, criando duas funções: uma para criar a lista de valores booleanos (True ou False) e outra para apresentar os índices das posições que contêm o valor True. Esta última função não deve mostrar os valores da lista em si, mas sim os índices das posições True. E o nosso TAD começa a ficar com esta cara:

```
def criar():
    lista = [False] * 11
    return lista

def show(lista):
    for i in range(11):
        if lista[i]:
            print(i)

def inserir(num, lista):
    if num >= 0 and num <= 10:
        lista[num] = True
    else:
        print ("Valor inválido")
```

Agora nosso teste fica um pouco mais elegante.

```
lista = criar()
inserir(2,lista)
print('Apresentando lista após inserir o valor 2:')
show(lista)
inserir(6,lista)
print('Apresentando lista após inserir o valor 6:')
show(lista)
```


Como resultado, veremos:

```
Apresentando lista após inserir o valor 2:

2

Apresentando lista após inserir o valor 6:

2

6
```

Vamos acrescentar outras funções para incrementar nosso TAD. Vamos escrever a função `remover` que irá receber um número entre 0 e 10 e uma lista com 11 valores booleanos (True ou False). A função irá retornar a mesma lista, mas com a posição correspondente ao número fornecido como argumento agora sendo False. Caso o número não esteja dentro do intervalo de 0 a 10, deve ser informado que o valor é inválido.

Você não demorará a perceber que ela é muito semelhante à função inserir implementada anteriormente.

```
def remover(num, lista):
    if num >= 0 and num <= 10:
        lista[num] = False
    else:
        return "Valor inválido"
```


Vamos testá-la. Você já deve ter percebido que não estamos fazendo testes exaustivos. Não é este o nosso objetivo. Mais pra frente conversaremos sobre entender como pensar as situações que costumo chamar de periclitantes e como realizar testes para verificar se o código as vence.

```
lista = criar()
inserir(2,lista)
print('Apresentando lista após inserir o valor 2:')
show(lista)
inserir(6,lista)
print('Apresentando lista após inserir o valor 6:')
show(lista)
remover(2,lista)
print('Apresentando lista após remover o valor 2:')
show(lista)
```

E a saída, como você já previu, é a seguir, mostrando na última impressão que o número 2 não faz mais parte da lista.

```
Apresentando lista após inserir o valor 2:

2

Apresentando lista após inserir o valor 6:

2

6

Apresentando lista após remover o valor 2:

6
```


Vamos fazer umas funções um pouco diferentes agora. A função que chamaremos de `uniao` irá receber duas listas com 11 posições, cada uma preenchida com valores booleanos (True ou False). A função irá retornar uma nova lista, onde cada posição será True se pelo menos um dos valores das posições correspondentes das duas listas for True, e será False se ambos os valores das posições correspondentes forem False.

```
def uniao(listaA, listaB):
    listaC = criar()
    for i in range(11):
        listaC[i] = listaA[i] or listaB[i]
    return listaC
```


Como você viu, utilizamos uma estratégia de lógica booleana para atender a condição exigida. A descrição da função indicava que é necessário que ao menos um valor daquela posição entre as duas listas seja True para a mesma posição na lista destino seja True e que, sendo ambas False, no destino também seria False. Isso é o ou-lógico que foi utilizado na linha `listaC[i] = listaA[i] or listaB[i]`.

Testaremos com um código um pouco ampliado:

```
lista1 = criar()
lista2 = criar()
inserir(1,lista1)
inserir(2,lista1)
inserir(7,lista1)
inserir(5,lista2)
inserir(7,lista2)
lista3 = uniao(lista1,lista2)
print('Lista 1:')
show(lista1)
print('Lista 2:')
show(lista2)
print('Lista 3: (a união entre as duas listas)')
show(lista3)
```

E o resultado comprova que a estratégia funcionou:

```
Lista 1:

1

2

7

Lista 2:

5

7

Lista 3: (a união entre as duas listas)

1

2

5

7
```


Vamos fechar com mais uma função para nosso TAD. A função `inter` recebe duas listas com 11 posições cada, preenchidas com valores booleanos. A função irá retornar uma nova lista, onde o valor em cada posição será True somente se os valores correspondentes nas duas listas de entrada forem ambos True. Caso contrário, o valor equivalente na posição na nova lista será False.


```
def inter(listaA, listaB):
    listaC = criar()
    for i in range(11):
        listaC[i] = listaA[i] and listaB[i]
    return listaC
```

De forma análoga à função anterior, usamos aqui o e-lógico que vai indicar que o valor resultado será True somente quando os valores das posições equivalentes das duas listas forem True.

No teste acrescentamos as três linhas finais:

```
lista1 = criar()
lista2 = criar()
inserir(1,lista1)
inserir(2,lista1)
inserir(7,lista1)
inserir(5,lista2)
inserir(7,lista2)
lista3 = uniao(lista1,lista2)
print('Lista 1:')
show(lista1)
print('Lista 2:')
show(lista2)
print('Lista 3: (a união entre as duas listas)')
show(lista3)
lista4 = inter(lista1,lista2)
print('Lista 4: (a interseção entre as duas listas)')
show(lista4)
```


E, por fim, vemos que a única posição quem tem o valor True nas duas listas é a posição 7.

```
Lista 1:
1
2
7
Lista 2:
5
7
Lista 3: (a união entre as duas listas)
1
2
5
7
Lista 4: (a interseção entre as duas listas)
7
```


Como todas as pistas foram dadas, você viu que implementamos um TAD para trabalhar com um conjunto limitado de números naturais sem, no entanto, termos armazenado quaisquer números naturais nas listas. Este é o poder do encapsulamento: não há a necessidade de todos saberem os segredos escondidos nas implementações.

Vamos agora arrumar os códigos, com nomes mais adequados, para finalizarmos nosso primeiro momento com este TAD.

```
# Definição das funções

def criarConjunto():
    conjunto = [False] * 11
    return conjunto

def show(conjunto):
    for i in range(11):
        if conjunto[i]:
            print(i)

def inserir(num, lista):
    if num >= 0 and num <= 10:
        lista[num] = True
    else:
        print ("Valor inválido")

def remover(num, conjunto):
    if num >= 0 and num <= 10:
        conjunto[num] = False
    else:
        print ("Valor inválido")

def uniao(conjuntoA, conjuntoB):
    conjuntoC = criarConjunto()
    for i in range(11):
        conjuntoC[i] = conjuntoA[i] or conjuntoB[i]
    return conjuntoC


def interseccao(conjuntoA, conjuntoB):
    conjuntoC = criarConjunto()
    for i in range(11):
        conjuntoC[i] = conjuntoA[i] and conjuntoB[i]
    return conjuntoC

# Codigo teste

conjunto1 = criarConjunto()
conjunto2 = criar()
inserir(1,conjunto1)
inserir(2,conjunto1)
inserir(7,conjunto1)
inserir(5,conjunto2)
inserir(7,conjunto2)
conjunto3 = uniao(conjunto1,conjunto2)
print('Conjunto 1:')
show(conjunto1)
print('Conjunto 2:')
show(conjunto2)
print('Conjunto 3: (a união entre os dois conjuntos)')
show(conjunto3)
conjunto4 = inter(conjunto1,conjunto2)
print('Conjunto 4: (a interseção entre os dois conjuntos)')
show(conjunto4)
```


Agora, ampliando nosso conhecimento, podemos entender que um TAD pode ser pensado como uma classe, onde sua estrutura interna é escondida dos usuários e apenas as operações definidas para o tipo são expostas. Isso

garante uma maior segurança, já que o usuário não tem acesso direto aos dados internos, evitando a possibilidade de erros. Além disso, as operações que são definidas para o tipo garantem uma maior coesão e consistência dos dados.

Então, vamos remodelar o nosso TAD para que ele seja implementado como uma classe. Não discutiremos conceitos mais densos sobre orientação a objetos aqui, mas utilizaremos alguns de seus recursos para facilitar o entendimento e a implementação de nossos TADs.

Assim, a classe Conjunto que implementaremos a seguir é uma representação concreta do conceito de TAD. Ela define uma estrutura de dados que representa um conjunto de números naturais e as operações que podem ser realizadas sobre esse conjunto.

De início temos que começar a construir nossa classe e também entender que faremos um outro código, em outro arquivo, para criar os objetos (instanciar a classe) e usar os métodos (sim, as funções são chamadas de métodos aqui) desta classe.

Iniciamos implementando o que seria o construtor da classe. É praticamente equivalente ao nosso criar, porém ele remete a conceitos de orientação a objetos que não abordaremos aqui. Salve o código a seguir em um arquivo com o mesmo nome da classe: `Conjunto.py`. Não é obrigatório, mas é uma boa prática deixar os nomes semelhantes, pois ajuda na manutenção e organização do código.

```
class Conjunto:
    def __init__(self):
        self.numeros =[False] * 11
```

O nome `init` que você vê ali substituindo o nosso `criarConjunto` é o nome de um método especial em Python, conhecido como o método construtor da classe. Ele é chamado automaticamente quando uma nova instância da classe é criada. O método `init` é usado para inicializar os atributos da classe quando uma nova instância é criada. Neste caso, o método `init` está inicializando o atributo "`numeros`" da classe "`Conjunto`" como uma lista de 11 elementos com valores False.

No código, os traços duplos (__) indicam que o método `init` é um método especial, também conhecido como método construtor, na programação orientada a objetos em Python. Eles indicam que esse método é executado automaticamente quando um objeto é criado a partir da classe.

O `self` que aparece como argumento e depois dentro da função é uma referência ao objeto atual da classe. Ele permite que o objeto acesse seus próprios atributos e métodos. Em Python, é convenção passar `self` como o primeiro argumento em métodos de classe.

No outro arquivo, que gosto de salvar como usaConjunto (você é livre, salve como você quiser, mas não se perca), vamos começar instanciando esta classe com dois objetos, para podermos melhor trabalhar depois.

```
import Conjunto

# Criação dos objetos da classe Conjunto
conjunto1 = Conjunto.Conjunto()
conjunto2 = Conjunto.Conjunto()
```

A repetição `Conjunto.Conjunto()` está acontecendo porque a classe está sendo importada como um módulo e, posteriormente, o objeto da classe é criado acessando-a através da notação `módulo.classe`. Dessa forma, a primeira `Conjunto` se refere ao módulo importado e a segunda `Conjunto` acessa a classe que está dentro deste módulo.

Agora vamos criar nossos métodos. Começaremos com o inserir e o remover.

```
class Conjunto:
      def __init__(self):
        self.numeros = [False] * 11

      def inserir(self, num):
        if num >= 0 and num <= 10:
             self.numeros[num] = True
        else:
             return "Valor inválido"

      def remover(self, num):
         if num >= 0 and num <= 10:
             self.numeros[num] = False
         else:
             return "Valor inválido"
```

A diferença bem visível é que não é necessário passar a lista como argumento. Isso porque a lista é um atributo da própria classe e ela será acessada por meio do self. Isso refletirá na execução deste método, como será visto a seguir. Como dá para perceber, toda a parte lógica do código se mantém.

```
import Conjunto

# Criação dos objetos da classe Conjunto
conjunto1 = Conjunto.Conjunto()
conjunto2 = Conjunto.Conjunto()

# Inserção de valores nos conjuntos
conjunto1.inserir(5)
conjunto1.inserir(7)
conjunto1.inserir(8)
conjunto2.inserir(5)
conjunto2.inserir(9)
conjunto2.inserir(0)
```

```
# Remoção de valores nos conjuntos
conjunto1.remover(7)
conjunto2.remover(0)
```

Para ver a mágica acontecer precisamos implementar o método de impressão. O mesmo show criado anteriormente com adaptação pelo fato de agora ser parte de uma classe.

```
    def show(self):
        for i in range(11):
            if self.numeros[i]:
                print(i)
```


Veja que o `self` se faz presente entre os parênteses, mesmo não havendo argumentos para passar para o interior do método.

Já vamos adequar nossos métodos de união e de intersecção para podermos ver tudo funcionando em conjunto (com o perdão do trocadilho).

```
    def uniao(self, conj2):
        conj_uniao = Conjunto()
        for i in range(11):
            conj_uniao.numeros[i] = self.numeros[i] or conj2.numeros[i]
        return conj_uniao

    def interseccao(self, conj2):
        conj_inter = Conjunto()
        for i in range(11):
            conj_inter.numeros[i] = self.numeros[i] and conj2.numeros[i]
        return conj_inter
```


De novo, vê-se que a modificação mais visível é o fato de não passarmos a lista que está sendo trabalhada como argumento. No caso da união ou intersecção entre duas listas, uma é passada como argumento – no caso, `conj2` – e outra é representada pelo `self`.

Para ninguém se perder, eis abaixo o código completo de nosso TAD `Conjunto`.

```
class Conjunto:
    def __init__(self):
        self.numeros = [False] * 11

    def inserir(self, num):
        if num >= 0 and num <= 10:
            self.numeros[num] = True
        else:
            print ("Valor inválido")
```

```
    def remover(self, num):
        if num >= 0 and num <= 10:
            self.numeros[num] = False
        else:
            print ("Valor inválido")

    def uniao(self, conj2):
        conj_uniao = Conjunto()
        for i in range(11):
            conj_uniao.numeros[i] = self.numeros[i] or conj2.numeros[i]
        return conj_uniao

    def interseccao(self, conj2):
        conj_inter = Conjunto()
        for i in range(11):
            conj_inter.numeros[i] = self.numeros[i] and conj2.numeros[i]
        return conj_inter

    def show(self):
        for i in range(11):
            if self.numeros[i]:
                print(i)
```


Como pode-se ver, a estrutura interna da classe `Conjunto` é escondida dos usuários, o que garante uma maior segurança e consistência dos dados. No caso, a representação do conjunto como uma lista de valores booleanos é interna à classe e não é exposta aos usuários. E isso pode ser observado no programa teste a seguir:


```
import Conjunto

# Criação dos objetos da classe Conjunto
conjunto1 = Conjunto.Conjunto()
conjunto2 = Conjunto.Conjunto()

# Inserção de valores nos conjuntos
conjunto1.inserir(5)
conjunto1.inserir(7)
conjunto1.inserir(8)
conjunto2.inserir(5)
conjunto2.inserir(9)
conjunto2.inserir(0)

# Remoção de valores nos conjuntos
conjunto1.remover(7)
conjunto2.remover(0)


# Apresentação dos conjuntos
print('Conjunto 1:')
conjunto1.show()
print('Conjunto 2:')
conjunto2.show()
```

```
# União dos conjuntos
uniao = conjunto1.uniao(conjunto2)
print('União dos conjuntos: ')
uniao.show()

# Intersecção dos conjuntos
inter = conjunto1.interseccao(conjunto2)
print('Interseção dos conjuntos: ')
inter.show()
```


Vamos abandonar nosso primeiro TAD e conversar sobre outros. Alguns exemplos de TADs simples incluem, só para exemplificar:

- Relógio: é um tipo que representa o tempo e suas operações, como adicionar e subtrair intervalos de tempo, ajustar a hora e exibir o tempo atual.
- Conta Bancária: é um tipo que representa uma conta bancária e suas operações, como depositar, retirar, consultar saldo e transferir dinheiro para outra conta.
- Endereço: é um tipo que representa um endereço e suas informações, como rua, número, cidade, estado, país e CEP.
- Produto: é um tipo que representa um produto e suas informações, como nome, descrição, preço, categoria e quantidade disponível.
- Agenda: é um tipo que representa um calendário e suas operações, como adicionar, remover e consultar compromissos, eventos e lembretes.

O código abaixo apresenta uma implementação da TAD Relogio. Vou deixa-lo aí para você copiar e colar, executar e buscar construir relações com tudo o que foi apresentado até agora.

```
class Relogio:
    def __init__(self, horas, minutos, segundos):
        self.horas = horas
        self.minutos = minutos
        self.segundos = segundos

    def adicionar_tempo(self, horas, minutos, segundos):
        self.horas += horas
        self.minutos += minutos
        self.segundos += segundos

        self.horas += self.minutos // 60
        self.minutos = self.minutos % 60

        self.horas = self.horas % 24

    def subtrair_tempo(self, horas, minutos, segundos):
        total_segundos = (horas * 3600) + (minutos * 60) + segundos
```

```
        tempo_atual_em_segundos = (self.horas * 3600) + (self.minutos * 60) + self.segundos
        tempo_atual_em_segundos -= total_segundos
        tempo_atual_em_segundos = tempo_atual_em_segundos % 86400
        self.horas = tempo_atual_em_segundos // 3600
        self.minutos = (tempo_atual_em_segundos % 3600) // 60
        self.segundos = tempo_atual_em_segundos % 60

    def ajustar_tempo(self, horas, minutos, segundos):
        self.horas = horas
        self.minutos = minutos
        self.segundos = segundos

    def exibir_tempo(self):
        print f"{self.horas:02d}:{self.minutos:02d}:{self.segundos:02d}"
```


O código a seguir serve para testá-lo e pode ajudar no seu entendimento. Brinque com este código, mude valores, chamadas de métodos. Faça o teste de mesa e confira com o resultado da execução do código.

```
import Relogio

# Criando um relógio com uma determinada
relogio = Relogio.Relogio(10, 35, 15)

# Adicionando 1 hora e 15 minutos ao relógio
relogio.adicionar_tempo(1, 15, 0)
print(relogio.exibir_tempo()) # 11:50:15

# Subtraindo 30 minutos do relógio
relogio.subtrair_tempo(0, 30, 0)
print(relogio.exibir_tempo()) # 11:20:15

# Ajustando o relógio para uma hora específica
relogio.ajustar_tempo(15, 45, 30)
print(relogio.exibir_tempo()) # 15:45:30
```


Continuando com a proposta de apresentar outros exemplos para você copiar, colar, executar e analisar, o código abaixo apresenta a implementação do TAD ContaBancaria:

```
class ContaBancaria:
    def __init__(self, nome_titular, numero_conta, saldo_inicial=0.0):
        self.nome_titular = nome_titular
        self.numero_conta = numero_conta
        self.saldo = saldo_inicial

    def depositar(self, valor):
        self.saldo += valor

    def retirar(self, valor):
        if valor > self.saldo:
            print("Saldo insuficiente")
        else:
            self.saldo -= valor
```

```
    def consultar_saldo(self):
        return self.saldo

    def transferir(self, valor, conta_destino):
        if valor > self.saldo:
            print("Saldo insuficiente")
        else:
            self.retirar(valor)
            conta_destino.depositar(valor)
```

Exemplo de uso

```
import ContaBancaria

# Criando contas bancárias
conta1 = ContaBancaria.ContaBancaria("João Silva", "0001", 1000.0)
conta2 = ContaBancaria.ContaBancaria("Maria Oliveira", "0002", 2000.0)

# Depositando dinheiro na conta1
conta1.depositar(500.0)

# Verificando saldo da conta1
print(f"Saldo da conta1: R$ {conta1.consultar_saldo()}")

# Transferindo dinheiro da conta1 para a conta2
conta1.transferir(300.0, conta2)

# Verificando saldo da conta1 e conta2
print(f"Saldo da conta1: R$ {conta1.consultar_saldo()}")
print(f"Saldo da conta2: R$ {conta2.consultar_saldo()}")

# Tentando retirar dinheiro da conta1 além do saldo disponível
conta1.retirar(3000.0)
```


Agora sugiro que você implemente o TAD Endereço, que segue a seguinte estrutura:

```
class Endereco:
    def __init__(self, rua, numero, cidade, estado, pais, cep):
    // insira sua implementação aqui

    def exibir_endereco(self):
        // insira sua implementação aqui

    def mudar_cidade(self, nova_cidade):
        // insira sua implementação aqui

    def mudar_estado(self, novo_estado):
        // insira sua implementação aqui
```

```
    def mudar_pais(self, novo_pais):
        // insira sua implementação aqui

    def mudar_cep(self, novo_cep):
        // insira sua implementação aqui
```

Como ele será usado:


```
import endereço
endereco = Endereco.Endereço("Rua das Flores", "123", "São Paulo", "SP",
"Brasil", "01234-567")
print(endereco.exibir_endereco())
endereco.mudar_cidade("Rio de Janeiro")
endereco.mudar_estado("RJ")
endereco.mudar_pais("Brasil")
endereco.mudar_cep("56789-123")
print(endereco.exibir_endereco())
```

Para ampliar seu conhecimento, crie os códigos para os TADs Produto e Agenda, descritos anteriormente.

Feito isso, você pode dar o próximo passo, pois agora vamos começar a pensar dentro , entrando no nosso contexto. Os TADs mais comuns em Estruturas de dados são:

- Lista encadeada (*Linked List*): é uma estrutura de dados linear que permite adicionar e remover elementos em qualquer posição. É comumente usada para implementar listas dinâmicas e para resolver problemas de alocação de memória.
- Pilha (*Stack*): é uma estrutura de dados que permite adicionar e remover elementos apenas no topo. É comumente utilizada para implementar algoritmos recursivos e para resolver problemas como a avaliação de expressões matemáticas.
- Fila (*Queue*): é uma estrutura de dados que permite adicionar elementos no fim e remover elementos do início. É comumente usada para implementar sistemas de atendimento em ordem, como em filas de banco ou em processamento de tarefas em sistemas operacionais.
- Deque: ou fila duplamente terminada, é uma estrutura de dados dinâmica que combina as características de pilhas e filas. Assim como uma lista encadeada, o Deque permite adicionar e remover elementos em qualquer posição, sem restrições. Ao mesmo tempo, ele também oferece as funcionalidades de uma pilha, permitindo adicionar e remover elementos no topo (frente) da estrutura, e de uma fila, permitindo adicionar elementos no fim (cauda) e remover elementos do início.

- Árvore (*Tree*): é uma estrutura de dados hierárquica que permite representar relações de contêiner entre elementos. É comumente utilizada para implementar árvores de busca, como árvores de decisão e árvores de busca binárias.

Material elaborado pelo professor
Fabiano Fagundes
Favor citar a fonte quando utilizá-lo