Estruturas de dados built-in do python

DISCIPLINA: ESTRUTURAS DE DADOS - Turma 02
PROFESSOR: ALBERTO COSTA NETO
Equipe: Alexander Nunes Souza, Carlos Eduardo da Silva

Introdução

As estruturas de dados (EDs) são componentes fundamentais na programação, pois desempenham um papel crucial na organização, armazenamento e manipulação de dados em qualquer linguagem de programação. Em Python, existem várias estruturas de dados built-in, que são fornecidas como parte da linguagem e são amplamente utilizadas para resolver uma variedade de problemas. Neste contexto, exploraremos a importância das EDs built-in em Python, os problemas que elas resolvem e suas vantagens e desvantagens.

Importância das Estrutura de Dados

As estruturas de dados são essenciais porque permitem que os programadores armazenem e manipulem dados de maneira eficiente. A escolha adequada de uma estrutura de dados pode ter um impacto significativo no desempenho e na simplicidade do código. Ao utilizar as EDs built-in do Python, os desenvolvedores podem economizar tempo e recursos na implementação de estruturas personalizadas, aproveitando soluções eficazes e otimizadas que já estão disponíveis.

Problemas Resolvidos

As EDs built-in do Python resolvem uma variedade de problemas, incluindo:

- **Armazenamento de Dados**: Permitem armazenar dados de diferentes tipos (números, strings, objetos, etc.) de maneira organizada e acessível.
- **Ordenação e Busca**: Facilitam a ordenação de dados e a busca eficiente de elementos em listas, tuplas e dicionários.
- **Manipulação de Sequências**: Permitem a criação, extensão, redução e manipulação de sequências de dados, como listas e tuplas.
- **Mapeamento de Chaves e Valores**: Oferecem uma maneira eficiente de associar chaves a valores, como no caso dos dicionários.

Vantagens

- **Facilidade de Uso**: Python oferece uma sintaxe simples e intuitiva para trabalhar com EDs, tornando o código mais legível e fácil de escrever.
- **Eficiência**: As EDs built-in são implementadas em C (na implementação padrão, Python), o que as torna eficientes em termos de desempenho.
- **Versatilidade**: Python oferece uma variedade de EDs para atender às diferentes necessidades de programação.

Desvantagens

- **Limitações de Tipo**: Algumas EDs, como listas, podem armazenar elementos de tipos diferentes, o que pode levar a erros de tipo em tempo de execução.
- **Complexidade de Algoritmos**: Embora as EDs sejam eficientes, a complexidade de alguns algoritmos pode ser um desafio em problemas específicos.
- **Memória**: Em alguns casos, as EDs podem consumir mais memória do que estruturas de dados personalizadas otimizadas para um problema específico.

TIPOS SEQUÊNCIA: TUPLES, LISTS

Operações comuns de sequências

Operação	Resultado	Notas
`x in s`	`True` caso um item de *s* seja igual a *x*, caso contrário `False`	(1)
`x not in s`	`False` caso um item de *s* for igual a *x*, caso contrário `True`	(1)
`s + t`	a concatenação de *s* e *t*	(6)(7)
`s * n` ou `n * s`	equivalente a adicionar *s* a si mesmo *n* vezes	(2)(7)
`s[i]`	*i*-ésimo item de *s*, origem 0	(3)
`s[i:j]`	fatia de *s* de *i* até *j*	(3)(4)
`s[i:j:k]`	fatia de *s* de *i* até *j* com passo *k*	(3)(5)
`len(s)`	comprimento de *s*	
`min(s)`	menor item de *s*	
`max(s)`	maior item de *s*	
`s.index(x[, i[, j]])`	índice da primeira ocorrência de *x* em *s* (no ou após o índice *i*, e antes do índice *j*)	(8)
`s.count(x)`	numero total de ocorrência de *x* em *s*	

TUPLES

Tuplas são sequências imutáveis, tipicamente usadas para armazenar coleções de dados heterogêneos (como as tuplas de 2 elementos produzidas pelo função embutida enumerate()). Tuplas também são usadas para casos em que seja necessária uma sequência imutável de dados homogêneos (como permitir o armazenamento em uma instância set ou dict).

class tuple([*iterable*])

As tuplas podem ser construídas de várias maneiras:

- Usando um par de parênteses para denotar a tupla vazia: ()
- Usando uma vírgula à direita para uma tupla singleton: a, ou (a,)
- Separando os itens com vírgulas: a, b, c ou (a, b, c)
- Usando a função embutida tuple(): tuple() ou tuple(iterable)

O construtor constrói uma tupla cujos itens são iguais e na mesma ordem dos itens de *iterable*. *iterable* pode ser uma sequência, um contêiner que suporta iteração ou um objeto iterador. Se *iterable* já for uma tupla, este será retornado inalterado. Por exemplo, tuple('abc') retorna ('a', 'b', 'c') e tuple([1, 2, 3]) retorna (1, 2, 3). Se nenhum argumento for dado, o construtor criará uma tupla vazia, ().

Observe que, na verdade, é a vírgula que faz uma tupla, e não os parênteses. Os parênteses são opcionais, exceto no caso de tupla vazia, ou quando são necessários para evitar ambiguidades sintáticas. Por exemplo, f(a, b, c) é uma chamada da função com três argumentos, enquanto que f((a, b, c)) é uma chamada de função com uma tupla de 3 elementos com um único argumento.

As tuplas implementam todas as operações comuns de sequência.

Operações de sequências mutáveis

Operação	Resultado	Notas
`s[i] = x`	item *i* de *s* é substituído por *x*	
`s[i:j] = t`	fatias de *s* de *i* até *j* são substituídas pelo conteúdo do iterável *t*	
`del s[i:j]`	o mesmo que `s[i:j] = []`	
`s[i:j:k] = t`	os elementos de `s[i:j:k]` são substituídos por aqueles de *t*	(1)
`del s[i:j:k]`	remove os elementos de `s[i:j:k]` desde a listas	
`s.append(x)`	adiciona *x* no final da sequência (igual a `s[len(s):len(s)] = [x]`)	
`s.clear()`	remove todos os itens de *s* (mesmo que `del s[:]`)	(5)
`s.copy()`	cria uma cópia rasa de *s* (mesmo que `s[:]`)	(5)

`s.extend(t)` ou `s += t`	estende *s* com o conteúdo de *t* (na maior parte do mesmo `s[len(s):len(s)] = t`)	
`s *= n`	atualiza *s* com o seu conteúdo por *n* vezes	(6)
`s.insert(i, x)`	insere *x* dentro de *s* no índice dado por *i* (igual a `s[i:i] = [x]`)	
`s.pop()` ou `s.pop(i)`	retorna o item em *i* e também remove-o de *s*	(2)
`s.remove(x)`	remove o primeiro item de *s* sendo `s[i]` igual a *x*	(3)
`s.reverse()`	inverte os itens de *s* in-place	(4)

Lists

As listas são sequências mutáveis, normalmente usadas para armazenar coleções de itens homogêneos (onde o grau preciso de similaridade variará de acordo com a aplicação).

class list([*iterable*])

As listas podem ser construídas de várias maneiras:

- Usando um par de colchetes para denotar uma lista vazia: []
- Usando colchetes, separando itens por vírgulas: [a], [a, b, c]
- Usando uma compreensão de lista: [x for x in iterable]
- Usando o construtor de tipo: list() ou list(iterable)

O construtor produz uma lista cujos itens são iguais e na mesma ordem que os itens de *iterable*. *iterable* pode ser uma sequência, um contêiner que suporte iteração ou um objeto iterador. Se *iterable* já for uma lista, uma cópia será feita e retornada, semelhante a iterable[:]. Por exemplo, list('abc') retorna ['a', 'b', 'c'] e list((1, 2, 3)) retorna [1, 2, 3]. Se nenhum argumento for dado, o construtor criará uma nova lista vazia [].

Muitas outras operações também produzem listas, incluindo a função embutida sorted().

Listas implementam todas as operações de sequências comuns e mutáveis. As listas também fornecem o seguinte método adicional:

sort(*, *key=None*, *reverse=False*)
  Esse método classifica a lista in-place, usando apenas comparações < entre itens. As exceções não são suprimidas - se qualquer operação de comparação falhar, toda a operação de ordenação falhará (e a lista provavelmente será deixada em um estado parcialmente modificado).

TIPO CONJUNTOS: SETS

SETS

class set([*iterable*])
class frozenset([*iterable*])

Retorna um novo objeto set ou frozenset, cujos elementos são obtidos a partir de um *iterable*. Os elementos de um conjunto devem ser hasheável. Para representar conjuntos de sets, os sets internos devem ser objetos frozenset. Se *iterable* não for especificado, um novo conjunto vazio é retornado.

Conjuntos podem ser criados de várias formas:

- Usar uma lista de elementos separados por vírgulas entre chaves: {'jack', 'sjoerd'}
- Usar uma compreensão de conjunto: {c for c in 'abracadabra' if c not in 'abc'}
- Usar o construtor de tipo: set(), set('foobar'), set(['a', 'b', 'foo'])

Instâncias de set e frozenset fornecem as seguintes operações:Retorna um novo conjunto com elementos no conjunto que não estão nos outros.

len(s) Retorna o número de elementos no set *s* (cardinalidade de *s*).

x in s Testa se *x* pertence a *s*.

x not in s Testa se *x* não pertence a *s*.

isdisjoint(*other*) Retorna True se o conjunto não tem elementos em comum com *other*. Conjuntos são disjuntos se e somente se a sua interseção é o conjunto vazio.

issubset(*other*) Testa se cada elemento do conjunto está contido em *other*.

issuperset(*other*) Testa se cada elemento em other está contido no conjunto.

union(others*)** Retorna um novo conjunto com elementos do conjunto e de todos os outros.

intersection(others*)** Retorna um novo conjunto com elementos comuns do conjunto e de todos os outros.

difference(others*)** Retorna um novo conjunto com elementos no conjunto que não estão nos outros.

symmetric_difference(*other*) Retorna um novo conjunto com elementos estejam ou no conjunto ou em other, mas não em ambos.

copy() Retorna uma cópia rasa do conjunto.

A seguinte tabela lista operações disponíveis para set que não se aplicam para instâncias imutáveis de frozenset:

update(others*)** Atualiza o conjunto, adicionando elementos dos outros.

intersection_update(others*)** Atualiza o conjunto, mantendo somente elementos encontrados nele e em outros.

difference_update(others*)** Atualiza o conjunto, removendo elementos encontrados em outros.

symmetric_difference_update(*other*) Atualiza o conjunto, mantendo somente elementos encontrados em qualquer conjunto, mas não em ambos.

add(*elem*) Adiciona o elemento *elem* ao conjunto.

remove(*elem*) Remove o elemento *elem* do conjunto. Levanta KeyError se *elem* não estiver contido no conjunto.

discard(*elem*) Remove o elemento *elem* do conjunto se ele estiver presente.

pop() Remove e retorna um elemento arbitrário do conjunto. Levanta KeyError se o conjunto estiver vazio.

clear() Remove todos os elementos do conjunto.

tipo Mapeamento: Dicts

DICTS

Um objeto mapeamento mapeia valores hasheáveis para objetos arbitrários. Mapeamentos são objetos mutáveis. Existe no momento apenas um tipo de mapeamento padrão, o *dicionário*. (Para outros contêineres, veja as classes embutidas list, set e tuple, e o módulo collections.)

As chaves de um dicionário são *quase* valores arbitrários. Valores que não são hasheáveis, ou seja, valores contendo listas, dicionários ou outros tipos mutáveis (que são comparados por valor e não por identidade de objeto) não podem ser usados como chaves. Valores que comparam iguais (como 1, 1.0 e True) podem ser usados alternadamente para indexar a mesma entrada do dicionário.

DICTS

class dict(***kwargs*)
class dict(*mapping*, ***kwargs*)
class dict(*iterable*, ***kwargs*)

Retorna um novo dicionário inicializado a partir de um argumento posicional opcional, e um conjunto de argumentos nomeados possivelmente vazio.

Os dicionários podem ser criados de várias formas:

- Usar uma lista de pares key: value separados por vírgula com chaves: {'jack': 4098, 'sjoerd': 4127} ou {4098: 'jack', 4127: 'sjoerd'}
- Usar uma compreensão de dicionário: {}, {x: x ** 2 for x in range(10)}
- Usar o construtor de tipo: dict(), dict([('foo', 100), ('bar', 200)]), dict(foo=100, bar=200)

Estas são as operações que dicionários suportam (e portanto, tipos de mapeamento personalizados devem suportar também):

list(d) Retorna uma lista de todas as chaves usadas no dicionário *d*.

len(d) Retorna o número de itens no dicionário *d*.

d[key] Retorna o item de *d* com a chave *key*. Levanta um KeyError se *key* não estiver no mapeamento.

d[key] = value Define d[key] para *value*.

del d[key] Remove d[key] do *d*. Levanta uma exceção KeyError se *key* não estiver no mapeamento.

key in d Retorna True se *d* tiver uma chave *key*, caso contrário False.

key not in d Equivalente a not key in d.

iter(d) Retorna um iterador para as chaves do dicionário. Isso é um atalho para iter(d.keys()).

clear() Remove todos os itens do dicionário.

copy() Retorna uma cópia superficial do dicionário

get(*key*[, *default*]) Retorna o valor para *key* se *key* está no dicionário, caso contrário *default*. Se *default* não é fornecido, será usado o valor padrão None, de tal forma que este método nunca levanta um KeyError.

items() Retorna uma nova visão dos itens do dicionário (pares de (key, value)). Veja a documentação de objetos de visão de dicionário.

keys() Retorna uma nova visão das chaves do dicionário. Veja a documentação de objetos de visão de dicionário.

pop(*key*[, *default*]) Se *key* está no dicionário, remove a mesma e retorna o seu valor, caso contrário retorna *default*. Se *default* não foi fornecido e *key* não está no dicionário, um KeyError é levantado.

popitem() Remove e retorna um par (key, value) do dicionário. Pares são retornados como uma pilha, ou seja em ordem LIFO.

reversed(d) Retorna um iterador revertido sobre as chaves do dicionário.

setdefault(*key*[, *default*]) Se *key* está no dicionário, retorna o seu valor. Se não, insere *key* com o valor *default* e retorna *default*. *default* por padrão usa o valor None.

update([*other*]) Atualiza o dicionário com os pares chave/valor existente em *other*, sobrescrevendo chaves existentes. Retorna None.

values() Retorna uma nova visão dos valores do dicionário. Veja a documentação de objetos de visão de dicionário.

IMPLEMENTAÇÃO

DICTS

Os dicionários são implementados utilizando tabela hash.

E como posso provar isso?

- *Dispersão na distribuição dos dados*
- *Observando que a complexidade temporal é semelhante a uma tabela hash*
- *Verificando que existe uma função de cálculo de hash internamente*
- *Documentação*

DICTS

Codigo que mostra que o dict usa tabela hash:

- *Python implementa funções de hash que tendem a produzir resultados regulares em casos comuns, em vez de buscar aleatoriedade extrema.*
- *Python utiliza uma técnica de LINEAR PROBING modificada que leva em consideração tanto os bits de hash quanto os bits não utilizados para espalhar as colisões.*
- *A escolha do valor para PERTURB_SHIFT é um equilíbrio entre eficiência e capacidade de lidar com casos incomuns.*

DICT_HASH.C

SETS

Os sets herdam algumas funcionalidades do objeto dict, isto é, utiliza também a implementação de tabela hash.

- *Os sets são mutáveis*
- *Frozenset são imutáveis*
- *Os sets também utilizam keys, entretanto, essas keys não são visíveis externamente, apenas internamente dentro do core do python.*

SET_HASH.C

Tuplas

As tuplas são basicamente um array de ponteiros.

- *No header existe uma técnica chamada struct hack para alocação eficiente de memória.*
- *Tuplas são imutáveis, por isso um tamanho fixo no array.*
- *Caso queira adicionar itens numa tupla será necessário realocar novamente.*

TUPLA.C

Listas

As listas é um ponteiro que aponta para um array de ponteiros de objetos.

- *Listas são mutáveis e dinâmicas.*

lista.c

REFERÊNCIAS

- https://github.com/python/cpython/blob/main/Objects/tupleobject.c
- https://github.com/python/cpython/blob/main/Objects/listobject.c
- https://github.com/python/cpython/blob/main/Objects/setobject.c#L954
- https://github.com/python/cpython/blob/main/Objects/dictobject.c
- https://docs.python.org/3/