Universidade de Brasília
Departamento de Ciência da Computação

Semana 13

Dicionários e Tuplas

CIC0004
Algoritmos e Programação de Computadores

Prof. Pedro Garcia Freitas https://pedrogarcia.gitlab.io/ pedro.garcia@unb.br

Brasilia

Objetivos

Esta aula introduz os tipos de dados "tupla" e dicionários.

1. Tuplas

- As tuplas em Python são usadas para armazenar múltiplos itens em uma única variável.

1. Tuplas

- Elas são semelhantes às listas, mas com uma diferença fundamental: as tuplas são imutáveis, o que significa que seus elementos não podem ser alterados após a criação da tupla.

1. Tuplas

- As tuplas são frequentemente utilizadas quando se deseja armazenar um conjunto de **valores relacionados que não devem ser modificados.**

1. Tuplas

- Além disso, as tuplas podem ser usadas como chaves em dicionários, pois são imutáveis.

1. Tuplas

1.1 Definição

- As tuplas são sequências ordenadas de elementos, separados por vírgulas e geralmente delimitados por parênteses. Por exemplo:

```
>>> tupla = (1, 2, 3, "abc")
```

1. Tuplas

1.2 Imutabilidade

- A principal característica das tuplas é que elas são **imutáveis**, ou seja, u**ma vez criadas, não é possível modificar seus elementos**. Isso significa que você não pode adicionar, remover ou alterar elementos em uma tupla após a sua criação.
- No entanto, é possível criar uma nova tupla a partir de uma existente ou concatenar tuplas.

1. Tuplas

1.2 Imutabilidade

- Exemplo:

```
>>> tupla = (1, 2, 3, "abc")
>>> tupla.append("Novo valor")
```

1. Tuplas

1.2 Imutabilidade

- Exemplo:

```
>>> tupla = (1, 2, 3, "abc")
>>> tupla.append("Novo valor")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

1. Tuplas

1.2 Imutabilidade

- Exemplo 2:

```
>>> tupla = (1, 2, 3)
>>> tupla += ("abc", "Novo valor")
>>> print(tupla)

    (1, 2, 3, 'abc', 4, 'novo valor')
```

1. Tuplas

1.2 Imutabilidade

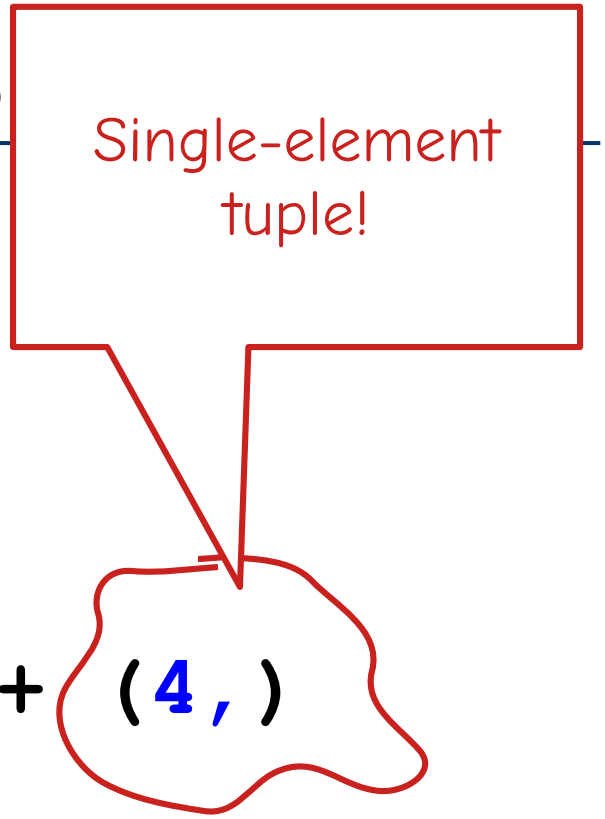
- Exemplo 3:

```
>>> concat = (1, 2, 3) + (4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate tuple
(not "int") to tuple
```

1. Tuplas

1.2 Imutabilidade

- Exemplo 4:




```
>>> concat = (1, 2, 3) + (4,)
>>> print(concat)
```

```
(1, 2, 3, 4)
```

1. Tuplas

1.3 Acesso aos elementos

- Os elementos em uma tupla podem ser acessados por meio de índices, semelhante ao acesso em listas. O primeiro elemento possui o índice 0, o segundo possui o índice 1 e assim por diante. Exemplo:

```
>>> tupla = (1, 2, 3)
>>> print(tupla[0])  # Saída: 1
```

1. Tuplas

1.4 Operações com tuplas

- Algumas operações comuns em tuplas incluem c**oncatenação de tuplas**, **repetição**, **fatiamento (*slicing*)** e obtenção do **tamanho da tupla**.

1. Tuplas

1.4 Operações com tuplas

Expressão	Resultados	Descrição
`len((1,2,3))`	3	Comprimento
`(1,2,3) + (4,5,6)`	(1, 2, 3, 4, 5, 6)	Concatenação
`('Hi!',) * 4`	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetição
`3 in (1,2,3)`	True	Membership
`for x in (1,2,3):` `  prints(x)`	1 2 3	Iteração

1. Tuplas

1.5 Indexação e Fatiamento

- Como as tuplas são sequências, a indexação e o fatiamento funcionam da mesma forma para as tuplas como funcionam para as strings. Supondo a seguinte entrada :

```
>>> L = ('one', 'two', 'three!')
```

1. Tuplas

1.5 Indexação e Fatiamento

Espressão	Resultado	Descrição
`L[2]`	'three!'	Offset começa do 0
`L[-2]`	'two'	Indexação negativa: conte da direita pra esquerda
`L[1:]`	['two', 'three!']	Fatiamento

1. Tuplas

1.6 Funções Integradas para Tuplas

Function with Description	Descrição
`len(tuple)`	Retorna o comprimento total da tupla.
`max(tuple)`	Retorna o item da tupla com o valor máximo.
`min(tuple)`	Retorna o item da tupla com o valor mínimo.
`tuple(seq)`	Converte uma lista em uma tupla.

1. Tuplas

1.7 Tuplas x Listas

- Mas por que usar tuplas se elas são imutáveis?

1. Tuplas

1.7 Tuplas x Listas

- Mas por que usar tuplas se elas são imutáveis?
- Bem, além de fornecerem acesso "somente leitura" aos valores dos dados, as **tuplas também são mais rápidas** do que as listas para operações de **apenas leitura**.

1. Tuplas

1.7 Tuplas x Listas

- Considere os seguintes trechos de código:

	Tempo de execução

`import timeit`
`timeit.timeit(`
`  'x=(1,2,3,4,5,6,7,8,9)',`
`  number=100000)`
	0.00089401300

`import timeit`
`timeit.timeit(`
`  'x=[1,2,3,4,5,6,7,8,9]',`
`  number=100000)`
	0.0039191639989

1. Tuplas

1.7 Tupla

0.0039191639989/0.00089401300 =
4.383788601396176 x mais rápido

- Considere [illegible] go:

	execução

`import timeit`
`timeit.timeit(`
`'x=(1,2,3,4,5,6,7,8,9)',`
`number=100000)`
	0 401300

`import timeit`
`timeit.timeit(`
`'x=[1,2,3,4,5,6,7,8,9]',`
`number=100000)`
	0.0039191639989

1. Tuplas

1.8 Considerações sobre imutabilidade

- De acordo com a documentação oficial do Python, imutável é "um objeto com um valor fixo", mas "valor" é um termo um tanto vago, o termo correto para tuplas seria `'id'`.

  `'id'` é a identidade da localização de um objeto na memória.

1. Tuplas

1.8 Considerações sobre imutabilidade

- Vamos analisar um pouco mais a fundo:

```
>>> n_tuple = (1, 1, [3,4])

#Items with same value have the same id.
>>> id(n_tuple[0]) == id(n_tuple[1])
```

1. Tuplas

1.8 Considerações sobre im

- Vamos analisar um pouco mai

Truec

```
>>> n_tuple = (1, 1, [3,4])

#Items with same value have the same id.
>>> id(n_tuple[0]) == id(n_tuple[1])
```

1. Tuplas

1.8 Considerações sobre imutabilidade

- Vamos analisar um pouco mais a fundo:

```
>>> n_tuple = (1, 1, [3,4])

#Items with same value have the same id.
>>> id(n_tuple[0]) == id(n_tuple[1])

#Different values have different id.
>>> id(n_tuple[0]) == id(n_tuple[2])
```

1. Tuplas

1.8 Considerações sobre imu

- Vamos analisar um pouco mais

```
>>> n_tuple = (1, 1, [3,4])

#Items with same value hav  the same id.
>>> id(n_tuple[0]) == id( _tuple[1])

#Different values have different id.
>>> id(n_tuple[0]) == id(n_tuple[2])
```

False

1. Tuplas

1.8 Considerações sobre imutabilidade

Embora as tuplas sejam imutáveis, elas têm algumas vantagens em certos contextos:

1. **Integridade dos dados:** Como as tuplas são imutáveis, os dados nelas armazenados não podem ser alterados acidentalmente. Isso pode ser útil em situações em que se deseja garantir a integridade dos dados, evitando alterações indesejadas.

1. Tuplas

1.8 Considerações sobre imutabilidade

Embora as tuplas sejam imutáveis, elas têm algumas vantagens em certos contextos:

2. **Uso como chaves de dicionário:** As tuplas podem ser usadas como chaves em dicionários. Diferentemente das listas, que são mutáveis, as tuplas podem ser usadas como identificadores únicos e imutáveis para mapear valores correspondentes.

1. Tuplas

1.8 Considerações sobre imutabilidade

Embora as tuplas sejam imutáveis, elas têm algumas vantagens em certos contextos:

3. Desempenho: As tuplas são mais eficientes em termos de desempenho em comparação com as listas. Como elas são imutáveis, o acesso aos elementos e a realização de operações nessas estruturas de dados podem ser mais rápidos em comparação com as listas.

1. Tuplas

1.8 Considerações sobre imutabilidade

Embora as tuplas sejam imutáveis, elas têm algumas vantagens em certos contextos:

4. **Convenção semântica:** Em alguns casos, o uso de tuplas pode transmitir a intenção de que os dados são apenas leitura e não devem ser modificados. Isso pode tornar o código mais legível e compreensível para outros desenvolvedores.

1. Tuplas

1.8 Considerações sobre imutabilidade

Embora as listas sejam mais flexíveis e frequentemente utilizadas, as tuplas têm seu lugar em situações em que a imutabilidade e as propriedades específicas mencionadas são desejáveis.

1. Tuplas

1.8 Considerações sobre imutabilidade

Para resumir o que aprendemos até agora:

- Não podemos adicionar elementos a uma tupla devido à sua propriedade imutável. Não há método **`append()`** ou **`extend()`** para tuplas.

1. Tuplas

1.8 Considerações sobre imutabilidade

Para resumir o que aprendemos até agora:

- Não podemos remover elementos de uma tupla, também devido à sua imutabilidade.
- As tuplas não possuem os métodos **`remove()`** ou **`pop()`**.

1. Tuplas

1.8 Considerações sobre imutabilidade

Para resumir o que aprendemos até agora:

- Podemos encontrar elementos em uma tupla, pois isso não altera a tupla.
- Podemos também usar o operador **`in`** para verificar se um elemento existe na tupla.

1. Tuplas

1.8 Considerações sobre imutabilidade

Para resumir o que aprendemos até agora:

- Portanto, se você estiver definindo um conjunto de valores constantes e a única coisa que você fará com eles é **iterar**, use uma tupla em vez de uma lista. Será mais rápido do que trabalhar com listas e também mais seguros.

1. Tuplas

1.8 Considerações sobre imutabilidade

Caso de Uso	Tuplas	Listas
Valores constantes	Ideal para definir constantes	Não recomendado, pois as listas são mutáveis
Iteração	Iterar através dos elementos	Iterar através dos elementos
Acesso a elementos específicos	Acessar elementos por índice	Acessar elementos por índice
Uso como chaves de dicionário	Podem ser usadas como chaves de dicionário	Não podem ser usadas como chaves de dicionário
Necessidade de adicionar/remover elementos	Imutáveis, não é possível adicionar ou remover elementos	Mutáveis, podem adicionar ou remover elementos
Manipulação de dados dinâmicos	Limitado, devido à imutabilidade	Flexível, pois podem ser modificadas

2. Dicionários

2.1 Motivação

Suponha que temos uma lista de números de telefone, como por exemplo:

```
[['CS','713-743-3350'],['UHPD','713-743-3333']]
```

ou inventário de latas de refrigerante, como por exemplo:

```
[['Coke',12],['Diet Coke',8],['Coke Zero',2]]
```

Encontrar o número de telefone da UHPD ou o número de latas de 'Coke Zero' que temos em estoque requer pesquisar na lista.z

- **Problema:** busca constante é insere intenso *overhead.*

2. Dicionários

2.2 Introdução

- Dicionários armazenam **pares** de entradas chamadas de ***itens***.
- Cada par de entradas contém:
  - Uma **chave**
  - Um **valor**
- A **chave** e o **valor** são **separados por dois pontos** (":")
- **Os pares de entradas** são s**eparados por vírgulas** (",")
- O **dicionário** é delimitado por **chaves** ({})

2. Dicionários

2.2 Introdução

```
my_dict = {
    "apple" : 3,
    "banana": 5,
    "orange": 2
}
```

Neste exemplo, temos um dicionário chamado **`my_dict`** que armazena informações sobre a quantidade de frutas. As chaves são os nomes das frutas (''maçã'', ''banana'', ''laranja''), e os valores correspondentes representam a quantidade de cada fruta (3 maçãs, 5 bananas, 2 laranjas).

2.2 Introdução

```
my_dict = {
    "apple" : 3,
    "banana": 5,
    "orange": 2
}
```

2. Dicionários

2.2 Introdução

```
my_dict = {
    "apple" : 3,
    "banana": 5,
    "orange": 2
}
```

Chaves

2. Dicionários

2.2 Introdução

```
my_dict = {
    "apple" : 3,
    "banana": 5,
    "orange": 2
}
```

Valores

Chaves

2. Dicionários

2.2 Introdução

Podemos acessar os valores no dicionário usando as **chaves**. Por exemplo:

```
print(my_dict["apple"])   # Output: 3
print(my_dict["banana"]) # Output: 5
```

2. Dicionários

2.2 Introdução

- Dicionários são **mutáveis**.
- Podemos também modificar os valores no dicionário ou adicionar novos pares de chave-valor.
- Por exemplo:

```
my_dict["orange"] = 4  # Modifying the value for the key "orange"
my_dict["grape"] = 6  # Adding a new key-value pair for "grape"
```

2. Dicionários

2.2 Introdução

- ```
  {'CS': '713-743-3350',
   'UHPD': '713-743-3333'}
  ```
- Também pode ser construído passo a passo:
  - `number = {}`
  - `number['CS'] = '713-743-3350'`
  - `number['UHPD'] = '713-743-3333'`
  ```

2. Dicionários

2.3 Acessando valores

- Sendo o dicionário

```
age = {'Alice':25, 'Bob':28}
```

então

`age['Alice']` is **25**

e

`age['Bob']` is **28**

2. Dicionários

2.3 Mutabilidade

Dicionários são **mutáveis!**

```
>>> age = {'Alice': 25, 'Bob': 28}
>>> saved = age
>>> age['Bob'] = 29
>>> age
{'Bob': 29, 'Alice': 25}
>>> saved
{'Bob': 29, 'Alice': 25}
```

2. Dicionários

2.4 Unicidade de chaves

Chaves devem ser **únicas**!

```
>>> age = {
        'Alice' : 25,
        'Bob' : 28,
        'Alice' : 26
    }
>>> age
{'Bob': 28, 'Alice': 26}
```

2. Dicionários

2.5 Operações com membros de dicionário

```
>>> age = {'Bob': 28, 'Alice': 26}
>>> len(age)
2
>>>age['Bob'] += 1
>>>age
{'Bob': 29, 'Alice': 26}
>>>age['Bob'] + age ['Alice']
55
```

2. Dicionários

2.6 Adição de conteúdo

```
age = {}
age['Alice'] = 25
```

2. Dicionários

2.6 Adição de conteúdo

```
age = {}
age['Alice'] = 25
```

Não é possível fazer o mesmo com uma lista

- `alist=[]`
- `alist[10]= 5 # NO NO`

2. Dicionários

2.7 Métodos globais de dicionário.

Método	Argumento	Descrição
`keys()`	none	Retorna uma visão das chaves no dicionário.
`values()`	none	Retorna uma visão dos valores no dicionário.
`items()`	none	Retorna uma visão dos itens (pares chave-valor) no dicionário.

2. Dicionários

2.8 Atualizando dicionários

```
>>> age = {'Alice': 26 , 'Carol' : 22}
>>> age.update({'Bob' : 29})
>>> age
{'Bob': 29, 'Carol': 22, 'Alice': 26}
>>> age.update({'Carol' : 23})
>>> age
{'Bob': 29, 'Carol': 23, 'Alice': 26}
>>> age['Bob'] = 30
>>> age
{'Bob': 30, 'Carol': 23, 'Alice': 26}
```

2. Dicionários

2.9 Retorno de valores

```
>>> age = {'Bob':29,'Liz':23,'Ann':26}
>>> age.get('Bob')
29
>>> age['Bob']
29
```

2. Dicionários

2.10 Remoção de item específico

```
>>> a = {'Ann':24, 'Liz':'twenty-two'}
>>> a
{'Liz': 'twenty-two', 'Ann': 24}
>>> a.pop('Liz')
'twenty-two'
>>> a
{'Ann': 24}
```

2. Dicionários

2.10 Remoção de item específico

- `>>> a.pop('Alice')`
  `26`
- `>>> a`
  `{}`
- `>>>`

2. Dicionários

2.11 Remoção de item aleatório

```
>>> age = {'Bob': 29, 'Liz': 22, 'Ann': 27}
>>> age.popitem()
('Bob', 29)
>>> age
{'Liz': 22, 'Ann': 27}
>>> age.popitem()
('Liz', 23)
>>> age
{'Ann': 27}
```

2. Dicionários

2.12 Percorrendo um dicionário

Suponha que tenhamos um inventário, como por exemplo:

```
cans = {'Coke':12, 'Diet Coke':8, 'Coke Zero': 2}
```

- Como podemos calcular o número total de latas que temos?
  - Precisamos percorrer todos os itens no dicionário.
  - Para isso, podemos iterar da seguinte forma:

```
for akey in cans.keys()
```

2. Dicionários

2.12 Percorrendo um dicionário

Exemplo:

```
cans = {'Coke':12,'Diet Coke':8,'CokeZero':2}
total = 0
for akey in cans.keys() :
    total += cans[akey]
print(total, 'cans of soft drink')
```

- A saida: `22 cans of soft drink`

2. Dicionários

Exemplo: matrizes esparsas

Considere a matriz algébrica:

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \\ 20 & 21 & 22 \end{pmatrix}$$

Essa matriz pode ser representada como uma lista de listas!

- `A = [[0,1,2],[10,11,12],[20,21,22]]`

2. Dicionários

Exemplo: matrizes esparsas

Considere agora a matriz D, que possui alguns poucos elementos não-nulos.

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A represetação convencional de listas de listas desperdiça muito espaço!

- `A = [[1,0,0],[0,1,0],[0,0,1]]`

2. Dicionários

Exemplo: matrizes esparsas

Considere agora a matriz
poucos elementos não-nu

$$D = \begin{pmatrix} 1 & 0 & \\ 0 & 1 & \\ 0 & 0 & 1 \end{pmatrix}$$

Pode se tornar um problema para matrizes muito grande!

A represetação convencional de listas de listas **desperdiça muito espaço!**

- `A = [[1,0,0],[0,1,0],[0,0,1]]`

2. Dicionários

Exemplo: matrizes esparsas

- Nesse caso, poderíamos usar um **dicionário**!

  `D = {(0,0):1, (1, 1):1, (2, 2):1}`

  que conteria apenas os elementos não nulos da matriz.

  Em seguida, usaríamos `d[(i,i)]` em vez de `d[i][i]` para acessar os elementos não nulos da matriz diagonal.


THANK
YOU
GRACIAS
ARIGATO
SHUKURIA
DANKSCHEEN
TASHAKKURATU
YAQHANYELAY
SUKSAMA
EKHMET
BİYAN
SHUKRIA
TINGKI
GRAZIE
MEHRBANI
MAAKE
KOMAPSUMNIDA
GOZAIMASHITA
EFCHARISTO
JUSPAXAR
PALDIES
BOLZÏN
MERCI

Dúvidas?