Programação Script

Tuplas, Sets e Dicionários

Aula 11

Prof. Felipe A. Louza

Roteiro

1. Tuplas
2. Sets
3. Dicionários
4. Exemplos
5. Referências

Roteiro

1. Tuplas
2. Sets
3. Dicionários
4. Exemplos
5. Referências

Tuplas

Uma **tupla** (`tuple`), como uma lista, é um sequência de items de qualquer tipo.

- Porém, ao contrário de listas, as tuplas são imutáveis.
- Em **Python**, tuplas são representadas por uma sequência de valores separados por vírgula, por exemplo:

```
>>> t = (1, 2.3, "abc", True)
```

```
>>> type(t)
<class 'tuple'>
```

  - Por convenção, envolvemos uma tupla entre parênteses.

Tuplas

As operações para acessar os elementos de um `lista` ou `string`, também funcionam em `tuplas`:

```
>>> t = (1, 2.3, "abc", True)
>>> t[2]
'abc'
>>> t[-1]
True
```

- Podemos percorrer todos os itens com um **laço**:

```
>>> for item in t:
  print(item, end=" ")
```

```
>>> i = 0
>>> while(i < len(t)):
  print(t[i], end=" ")
  i += 1
```

Tuplas

As operações para acessar os elementos de um `lista` ou `string`, também funcionam em `tuplas`:

```
>>> t = (1, 2.3, "abc", True)
>>> t[2]
'abc'
>>> t[-1]
True
```

- Podemos percorrer todos os itens com um **laço**:

```
>>> for item in t:
  print(item, end=" ")
```

```
>>> i = 0
>>> while(i < len(t)):
  print(t[i], end=" ")
  i += 1
```

Tuplas

- Podemos fazer fatiamento (slicing) com `tuplas`:

```
>>> t = (1, 2.3, "abc", True)
##
>>> t[1:3]
(2.3, 'abc')
##
>>> t[:2]
(1, 2.3)
##
>>> t[2:]
('abc', True)
##
>>> t[::2]
(1, 'abc')
##
>>> t[::-1]
(True, 'abc', 2.3, 1)
```

  - E outras operações de `listas` e `strings`: `count()`, `index()`, `+` ...

Tuplas

- Podemos fazer fatiamento (slicing) com tuplas:

```
>>> t = (1, 2.3, "abc", True)
##
>>> t[1:3]
(2.3, 'abc')
##
>>> t[:2]
(1, 2.3)
##
>>> t[2:]
('abc', True)
##
>>> t[::2]
(1, 'abc')
##
>>> t[::-1]
(True, 'abc', 2.3, 1)
```

  - E outras operações de listas e strings: count(), index(), + ...

Tuplas

Entretanto, tuplas são imutáveis, portanto:

- **Não** podemos modificar, adicionar ou remover elementos.

```
>>> t = (1, 2.3, "abc", True)
>>> t[2] = "xyz"
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    t[2] = "xyz"
TypeError: 'tuple' object does not support item assignment
```

Tuplas

- Podemos alterar o **objeto** `tupla` ao qual `t` se refere:

```
>>> t = (1, 2.3, "abc", True)
>>> id(t)
140663014115344
>>> t = ("xyz", 3)
>>> id(t)
140663014200256
```

- Um detalhe, `tuplas` de tamanho 1:

```
>>> t = (1)
>>> type(t)
<class 'int'>
>>> t = (1,)
>>> type(t)
<class 'tuple'>
```

Tuplas

- Podemos alterar o **objeto** `tupla` ao qual `t` se refere:

```
1 >>> t = (1, 2.3, "abc", True)
2 >>> id(t)
3 140663014115344
4 >>> t = ("xyz", 3)
5 >>> id(t)
6 140663014200256
```

- Um detalhe, `tuplas` de tamanho 1:

```
1 >>> t = (1)
2 >>> type(t)
3 <class 'int'>
4 >>> t = (1,)
5 >>> type(t)
6 <class 'tuple'>
```

Tuplas

Tuplas são úteis para armazenar diferentes itens em uma única variável.

```
>>> aluno1 = ("Carlos", "Daniel", 1993, 11, 30, "Belo Horizonte", \
          "MG", "Brasil")
##
>>> aluno2 = ("Maria", "Camila", 1989, 9, 21, "São Paulo", \
          "SP", "Brasil")
```

- Em outras linguagens de programação frequentemente chamamos de registros (*records*).
- Em **Python** não há descrição do que cada um desses campos (*fields*).

Tuplas

Tuplas são úteis para armazenar diferentes itens em uma única variável.

```
>>> aluno1 = ("Carlos", "Daniel", 1993, 11, 30, "Belo Horizonte", \
          "MG", "Brasil")
##
>>> aluno2 = ("Maria", "Camila", 1989, 9, 21, "São Paulo", \
          "SP", "Brasil")
```

- Em outras linguagens de programação frequentemente chamamos de registros (*records*).
- Em **Python** não há descrição do que cada um desses campos (*fields*).

Tuplas

Além disso, algumas vantagens:

- Percorrer os elementos em uma tupla é mais rápido do que em uma lista[1].
- O espaço ocupado em uma tupla é menor do que em uma lista.

```
1 >>> a = (1, 2, 3, 4, 5)
2 >>> b = [1, 2, 3, 4, 5]
3 ##
4 >>> print(a.__sizeof__())
5 64 # (em bytes)
6 >>> print(b.__sizeof__())
7 80 # (em bytes)
```

- Tuplas garantem dados *read-only*.
- Tuplas podem ser usadas como chaves para dicionários (vamos ver ainda).

[1] Essa diferença ocorre quando manipulamos grandes volumes de dados.

Tuplas

Além disso, algumas vantagens:

- Percorrer os elementos em uma tupla é mais rápido do que em uma lista[1].
- O espaço ocupado em uma tupla é menor do que em uma lista.

```
>>> a = (1, 2, 3, 4, 5)
>>> b = [1, 2, 3, 4, 5]
##
>>> print(a.__sizeof__())
64 # (em bytes)
>>> print(b.__sizeof__())
80 # (em bytes)
```

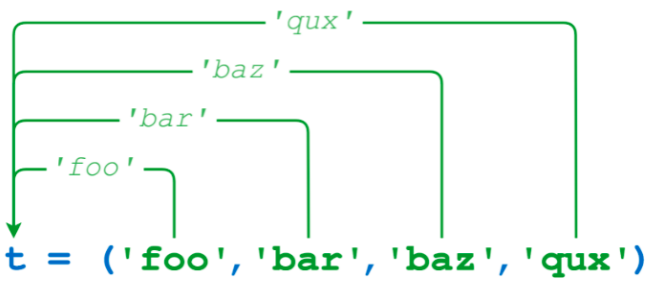
- Tuplas garantem dados *read-only*.
- Tuplas podem ser usadas como chaves para dicionários (vamos ver ainda).

[1] Essa diferença ocorre quando manipulamos grandes volumes de dados.

Tuplas

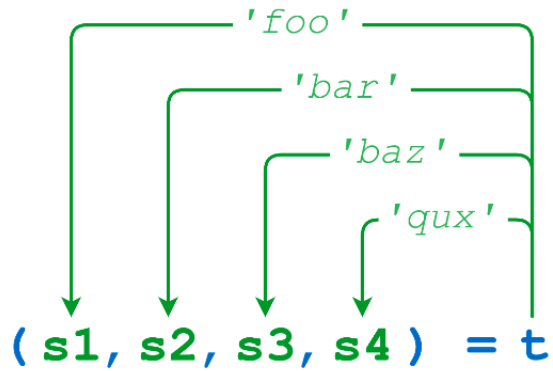
- A criação de uma tupla também é chamada de empacotamento:

```
1 >>> t = ('foo', 'bar', 'baz', 'qux')
```



- Podemos desempacotar uma tupla em diferentes variáveis:

```
1 >>> s1, s2, s3, s4 = t
2 >>> s1
3 'foo'
4 >>> s2
5 'bar'
6 >>> s3
7 'baz'
8 >>> s4
9 'qux'
```



Tuplas

- No desempacotamento a quantidade de variáveis do lado esquerdo deve ser igual ao tamanho da tupla.

```
>>> t = ('foo', 'bar', 'baz', 'qux')
>>> s1, s2, s3 = t
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    s1, s2, s3 = t
ValueError: too many values to unpack (expected 3)
```

Tuplas

- Com isso, podemos escrever a troca de valores (swap) sem utilizar variável auxiliar:

```
>>> a = 1
>>> b = 0
##
>>> temp = a
>>> a = b
>>> b = temp
##
>>> a, b
(0, 1)
```

Tuplas

- Com isso, podemos escrever a troca de valores (swap) sem utilizar variável auxiliar:

```
1 >>> a = 1
2 >>> b = 0
3 ##
4 >>> temp = a
5 >>> a = b
6 >>> b = temp
7 ##
8 >>> a, b
9 (0, 1)
```

```
1 >>> a = 1
2 >>> b = 0
3 ##
4 >>># Magic time!
5 >>> a, b = b, a # (b, a)
6 ##
7 >>> a, b
8 (0, 1)
```

Tuplas

Funções podem retornar tuplas:

```
1 def comprimento_e_area(raio):
2   c = 2 * 3.14159 * raio       # 2*pi*r
3   a = 3.14159 * raio * raio    # pi*r^2
4   return c, a #(c, a)
```

```
1 >>> raio = int(input("Digite o valor do raio: "))
2 >>> c, a = comprimento_e_area(raio)
3 ##
4 >>> c
5 31.4159
6 >>> a
7 78.53975
```

Tuplas

Funções podem retornar tuplas:

```
1 def comprimento_e_area(raio):
2   c = 2 * 3.14159 * raio       # 2*pi*r
3   a = 3.14159 * raio * raio    # pi*r^2
4   return c, a #(c, a)
```

```
1 >>> raio = int(input("Digite o valor do raio: "))
2 >>> c, a = comprimento_e_area(raio)
3 ##
4 >>> c
5 31.4159
6 >>> a
7 78.53975
```

Tuplas

- Podemos criar outra tupla, com valores da primeira alterados:

```
>>> x = ("maça", "banana", "cereja")
##
>>> y = list(x)
>>> y[1] = "kiwi"
##
>>> x = tuple(y)
>>> x
("maça", "kiwi", "cereja")
```

  - Um novo **objeto** `tupla` é criado.
  - Podemos fazer isso para `append()` e `remove()`.

Tuplas

- Podemos criar outra tupla, com valores da primeira alterados:

```
1 >>> x = ("maça", "banana", "cereja")
2 ##
3 >>> y = list(x)
4 >>> y[1] = "kiwi"
5 ##
6 >>> x = tuple(y)
7 >>> x
8 ("maça", "kiwi", "cereja")
```

  - Um novo **objeto** `tupla` é criado.
  - Podemos fazer isso para `append()` e `remove()`.

Roteiro

1. Tuplas
2. Sets
3. Dicionários
4. Exemplos
5. Referências

Sets

Em **Python** podemos representar conjuntos de elementos com `sets`:

- Conjuntos são representados por valores separados por chaves, por exemplo:

```
>>> s = {"maça", "banana", "cereja"}
```

```
>>> type(s)
<class 'set'>
```

Sets

- Em um `set` não temos repetições, ordem ou indexação.

```
>>> s = {"maça", "banana", "cereja", "cereja"}
>>> s
{'banana', 'maça', 'cereja'}
```

```
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
```

  - Também não é possível alterar um elemento de um `set`, mas podemos adicionar novos elementos.

Sets

- Em um `set` não temos repetições, ordem ou indexação.

```
>>> s = {"maça", "banana", "cereja", "cereja"}
>>> s
{'banana', 'maça', 'cereja'}
```

```
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
```

  - Também não é possível alterar um elemento de um `set`, mas podemos adicionar novos elementos.

Sets

- Em um `set` não temos repetições, ordem ou indexação.

```
>>> s = {"maça", "banana", "cereja", "cereja"}
>>> s
{'banana', 'maça', 'cereja'}
```

```
>>> s[0]
Traceback (most recent call last):
  File "<pyshell#46>", line 1, in <module>
    s[0]
TypeError: 'set' object is not subscriptable
```

  - Também não é possível alterar um elemento de um `set`, mas podemos adicionar novos elementos.

Sets

Adicionando novos elementos com o método `add()`:

```
>>> s = {"maça", "banana", "cereja"}
>>> s.add("laranja")
>>> s
{'banana', 'maça', 'cereja', 'laranja'}
```

- Podemos adicionar **todos** os itens de um outro `set`:

```
>>> tropical = {"abacaxi", "manga", "banana"}
##
>>> s.update(tropical)
>>> s
{'abacaxi', 'manga', 'banana', 'maça', 'laranja', 'cereja'}
```

  - Valores repetidos são ignorados.
  - O método `update()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Adicionando novos elementos com o método `add()`:

```
>>> s = {"maça", "banana", "cereja"}
>>> s.add("laranja")
>>> s
{'banana', 'maça', 'cereja', 'laranja'}
```

- Podemos adicionar **todos** os itens de um outro `set`:

```
>>> tropical = {"abacaxi", "manga", "banana"}
##
>>> s.update(tropical)
>>> s
{'abacaxi', 'manga', 'banana', 'maça', 'laranja', 'cereja'}
```

  - Valores repetidos são ignorados.
  - O método `update()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Adicionando novos elementos com o método `add()`:

```
>>> s = {"maça", "banana", "cereja"}
>>> s.add("laranja")
>>> s
{'banana', 'maça', 'cereja', 'laranja'}
```

- Podemos adicionar **todos** os itens de um outro `set`:

```
>>> tropical = {"abacaxi", "manga", "banana"}
##
>>> s.update(tropical)
>>> s
{'abacaxi', 'manga', 'banana', 'maça', 'laranja', 'cereja'}
```

  - Valores repetidos são ignorados.
  - O método `update()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Podemos remover elementos de um `set()`:

```
1 >>> s = {"maça", "banana", "cereja"}
2 >>> s.remove("banana")
3 >>> s
4 {'maça', 'cereja'}
```

- Se o elemento não existir, teremos um erro:

```
1 >>> s.remove("laranja")
2 Traceback (most recent call last):
3   File "<pyshell#71>", line 1, in <module>
4     s.remove("laranja")
5 KeyError: 'laranja'
```

  - O método `discard()` também remove, sem lançar uma exceção quando o item não está no `set`.

Outros métodos: `clear()` e `del`.

Sets

Podemos remover elementos de um `set()`:

```
>>> s = {"maça", "banana", "cereja"}
>>> s.remove("banana")
>>> s
{'maça', 'cereja'}
```

- Se o elemento não existir, teremos um erro:

```
>>> s.remove("laranja")
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    s.remove("laranja")
KeyError: 'laranja'
```

  - O método `discard()` também remove, sem lançar uma exceção quando o item não está no `set`.

Outros métodos: `clear()` e `del`.

Sets

Podemos remover elementos de um `set()`:

```
>>> s = {"maça", "banana", "cereja"}
>>> s.remove("banana")
>>> s
{'maça', 'cereja'}
```

- Se o elemento não existir, teremos um erro:

```
>>> s.remove("laranja")
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    s.remove("laranja")
KeyError: 'laranja'
```

  - O método `discard()` também remove, sem lançar uma exceção quando o item não está no `set`.

Outros métodos: `clear()` e `del`.

Sets

Podemos percorrer todos os itens com um **for**:

```
>>> s = {"maça", "banana", "cereja"}
>>> for item in s:
  print(item, end=" ")

banana maça cereja
```

- Podemos verificar a pertinência de um elemento no `set`:

```
>>> "banana" in s
True
>>> "Banana" in s
False
```

É possível reescrever esse laço com um `while`?

Sets

Podemos percorrer todos os itens com um **for**:

```
1 >>> s = {"maça", "banana", "cereja"}
2 >>> for item in s:
3   print(item, end=" ")
4
5 banana maça cereja
```

- Podemos verificar a pertinência de um elemento no `set`:

```
1 >>> "banana" in s
2 True
3 >>> "Banana" in s
4 False
```

É possível reescrever esse laço com um `while`?

Sets

Em **Python**, `sets` podem ter elementos de tipos diferentes:

```
>>> s = {42, "foo", 3.14159, None}
>>> s
{None, 'foo', 42, 3.14159}
```

- Todos os elementos devem ser imutáveis:

```
>>> s = {[1, 2, 3], "a", 1}
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    s = {[1, 2, 3], 42, 'foo', 3.14159, None}
TypeError: unhashable type: 'list'
```

  - O mesmo vale para `s = {[1, 2, 3]}`.

Sets

Em **Python**, `sets` podem ter elementos de tipos diferentes:

```
>>> s = {42, "foo", 3.14159, None}
>>> s
{None, 'foo', 42, 3.14159}
```

- Todos os elementos devem ser imutáveis:

```
>>> s = {[1, 2, 3], "a", 1}
Traceback (most recent call last):
  File "<pyshell#87>", line 1, in <module>
    s = {[1, 2, 3], 42, 'foo', 3.14159, None}
TypeError: unhashable type: 'list'
```

  - O mesmo vale para `s = {[1, 2, 3]}`.

Sets

Podemos converter uma lista ou tupla em um `set`:

```
>>> lista = [1, 1, 2, 3, 3, 4, 5]
>>> set(lista)
{1, 2, 3, 4, 5}
##
>>> tupla = (1, 1, 2, 3, 3, 4, 5)
>>> set(tupla)
{1, 2, 3, 4, 5}
```

- Dicionários também (somente as chaves entram no `set`).

Sets

Podemos converter uma lista ou tupla em um `set`:

```
>>> lista = [1, 1, 2, 3, 3, 4, 5]
>>> set(lista)
{1, 2, 3, 4, 5}
##
>>> tupla = (1, 1, 2, 3, 3, 4, 5)
>>> set(tupla)
{1, 2, 3, 4, 5}
```

- Dicionários também (somente as chaves entram no `set`).

Sets

- Como definimos o conjunto vazio $\varnothing$?

```
>>> s = set()
>>> x
set()
```

```
>>> type(x)
<class 'set'>
```

  - Cuidado, `s = {}` representa um dicionário vazio.

Sets

- Como definimos o conjunto vazio $\varnothing$?

```
>>> s = set()
>>> x
set()
```

```
>>> type(x)
<class 'set'>
```

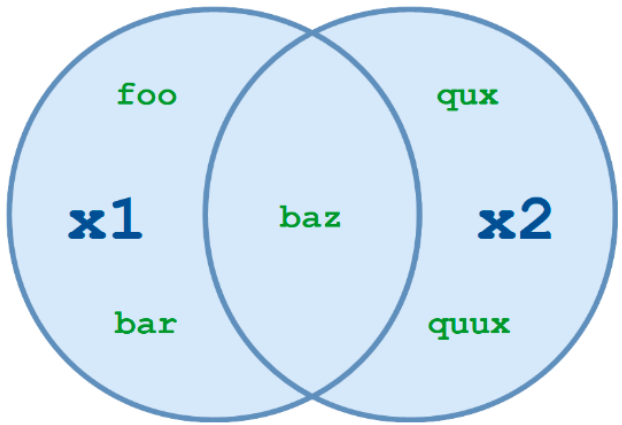
  - Cuidado, `s = {}` representa um dicionário vazio.

Sets

Alguns operadores e métodos:

- União: `|` e `union()`

```
1 >>> x1 = {'foo', 'bar', 'baz'}
2 >>> x2 = {'baz', 'qux', 'quux'}
3 ##
4 >>> x1 | x2
5 {'foo', 'qux', 'quux', 'baz', 'bar'}
6 ##
7 >>> x1.union(x2)
8 {'foo', 'qux', 'quux', 'baz', 'bar'}
```



Set Union

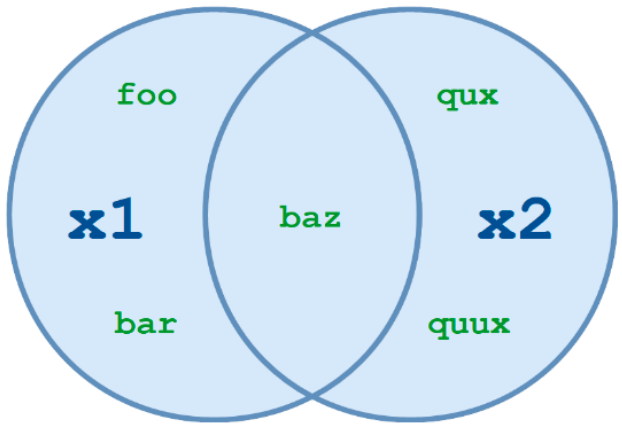
  - Ambos retornam a união dos conjuntos.
  - O método `union()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Alguns operadores e métodos:

- União: `|` e `union()`

```
1 >>> x1 = {'foo', 'bar', 'baz'}
2 >>> x2 = {'baz', 'qux', 'quux'}
3 ##
4 >>> x1 | x2
5 {'foo', 'qux', 'quux', 'baz', 'bar'}
6 ##
7 >>> x1.union(x2)
8 {'foo', 'qux', 'quux', 'baz', 'bar'}
```



Set Union

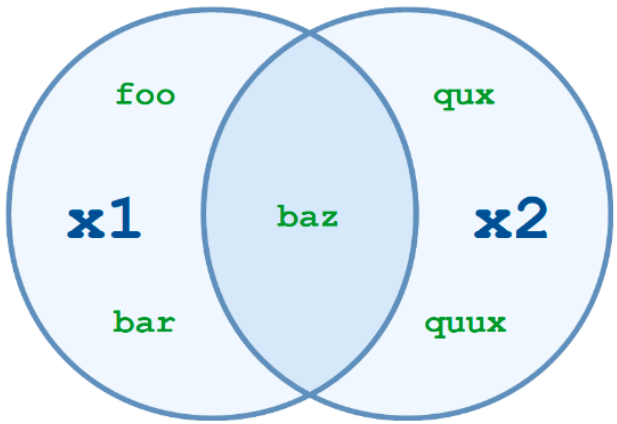
  - Ambos retornam a união dos conjuntos.
  - O método `union()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Alguns operadores e métodos:

- Intersecção: `&` e `intersection()`

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}
##
>>> x1 & x2
{'bar'}
##
>>> x1.intersection(x2)
{'bar'}
```



Set Intersection

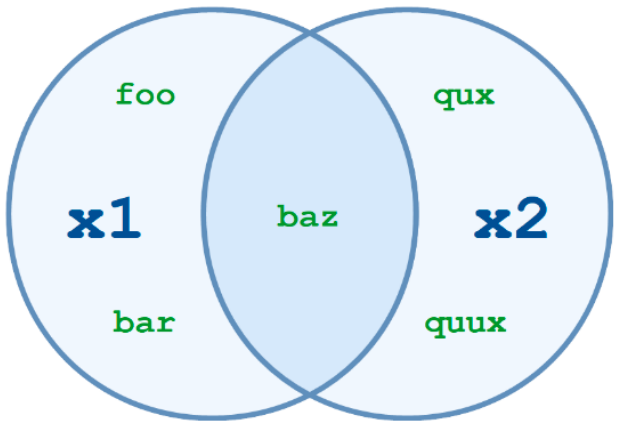
  - Ambos retornam a intersecção dos conjuntos.
  - O método `intersection()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Alguns operadores e métodos:

- Intersecção: `&` e `intersection()`

```
1 >>> x1 = {'foo', 'bar', 'baz'}
2 >>> x2 = {'baz', 'qux', 'quux'}
3 ##
4 >>> x1 & x2
5 {'bar'}
6 ##
7 >>> x1.intersection(x2)
8 {'bar'}
```



Set Intersection

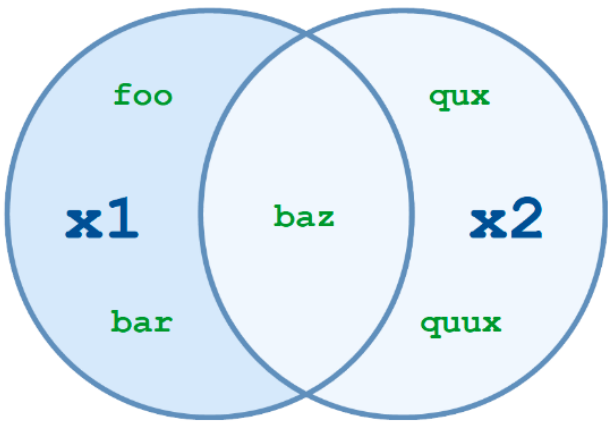
  - Ambos retornam a intersecção dos conjuntos.
  - O método `intersection()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Alguns operadores e métodos:

- Diferença: `-` e `difference()`

```
1 >>> x1 = {'foo', 'bar', 'baz'}
2 >>> x2 = {'baz', 'qux', 'quux'}
3 ##
4 >>> x1 - x2
5 {'foo', 'bar'}
6 ##
7 >>> x1.difference(x2)
8 {'foo', 'bar'}
```



Set Difference

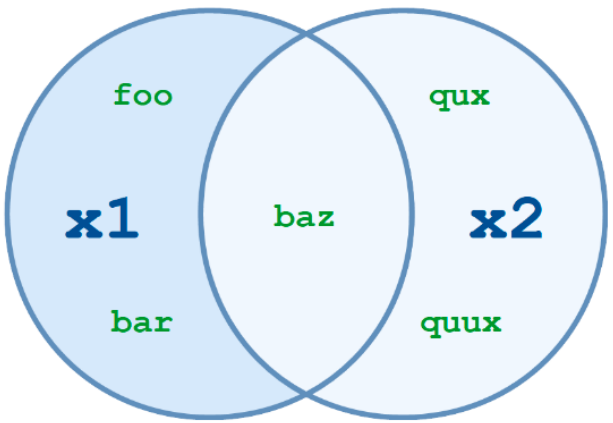
  - Ambos retornam os elementos que estão em `x1`, mas não em `x2`.
  - O método `difference()` aceita outros objetos: tuplas, listas, dicionários.

Sets

Alguns operadores e métodos:

- Diferença: `-` e `difference()`

```
1 >>> x1 = {'foo', 'bar', 'baz'}
2 >>> x2 = {'baz', 'qux', 'quux'}
3 ##
4 >>> x1 - x2
5 {'foo', 'bar'}
6 ##
7 >>> x1.difference(x2)
8 {'foo', 'bar'}
```



Set Difference

  - Ambos retornam os elementos que estão em `x1`, mas não em `x2`.
  - O método `difference()` aceita outros objetos: tuplas, listas, dicionários.

Roteiro

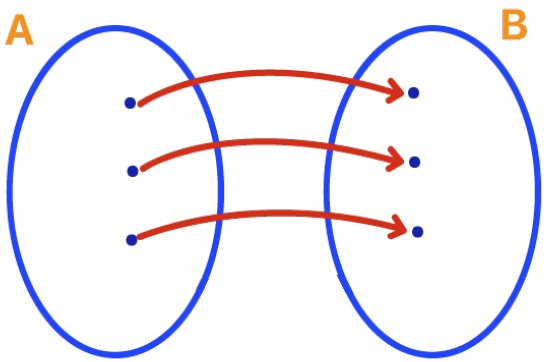
1. Tuplas
2. Sets
3. Dicionários
4. Exemplos
5. Referências

Dicionários

Um **dicionário** é uma coleção associativa de objetos.

- A associação, ou mapeamento, é feita com um par (chave e valor).

```
1 >>> d = {} # dicionário vazio
2 >>> d["um"] = 1
3 >>> d["dois"] = 2
4 >>> d["tres"] = 3
5 >>> d
6 {'um': 1, 'dois': 2, 'tres': 3}
```



  - Os valores podem ser de qualquer tipo, mas as chaves só podem ser de tipos imutáveis.
  - Dicionários são **mutáveis**.
  - As chaves precisam ser **únicas**[2].

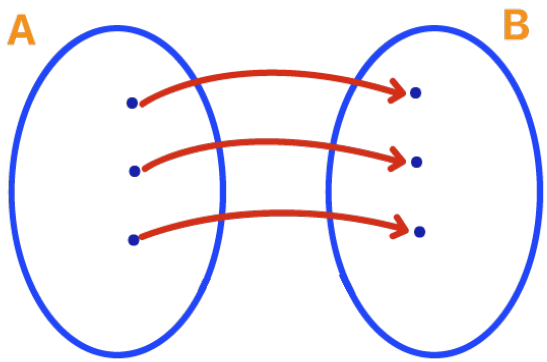
[2] `d["tres"] = "III"` é outra associação.

Dicionários

Um **dicionário** é uma coleção associativa de objetos.

- A associação, ou mapeamento, é feita com um par (chave e valor).

```
1 >>> d = {} # dicionário vazio
2 >>> d["um"] = 1
3 >>> d["dois"] = 2
4 >>> d["tres"] = 3
5 >>> d
6 {'um': 1, 'dois': 2, 'tres': 3}
```



  - Os valores podem ser de qualquer tipo, mas as chaves só podem ser de tipos imutáveis.
  - Dicionários são **mutáveis**.
  - As chaves precisam ser **únicas**[2].

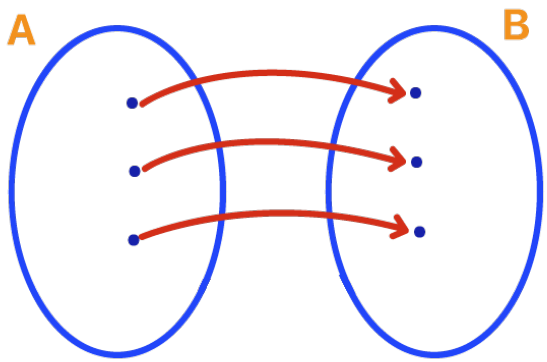
[2] `d["tres"] = "III"` é outra associação.

Dicionários

Um **dicionário** é uma coleção associativa de objetos.

- A associação, ou mapeamento, é feita com um par (chave e valor).

```
1 >>> d = {} # dicionário vazio
2 >>> d["um"] = 1
3 >>> d["dois"] = 2
4 >>> d["tres"] = 3
5 >>> d
6 {'um': 1, 'dois': 2, 'tres': 3}
```



  - Os valores podem ser de qualquer tipo, mas as chaves só podem ser de tipos imutáveis.
  - Dicionários são **mutáveis**.
  - As chaves precisam ser **únicas**[2].

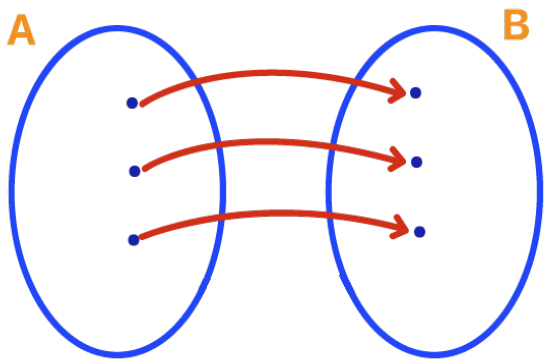
[2] `d["tres"] = "III"` é outra associação.

Dicionários

Um **dicionário** é uma coleção associativa de objetos.

- A associação, ou mapeamento, é feita com um par (chave e valor).

```
1 >>> d = {} # dicionário vazio
2 >>> d["um"] = 1
3 >>> d["dois"] = 2
4 >>> d["tres"] = 3
5 >>> d
6 {'um': 1, 'dois': 2, 'tres': 3}
```



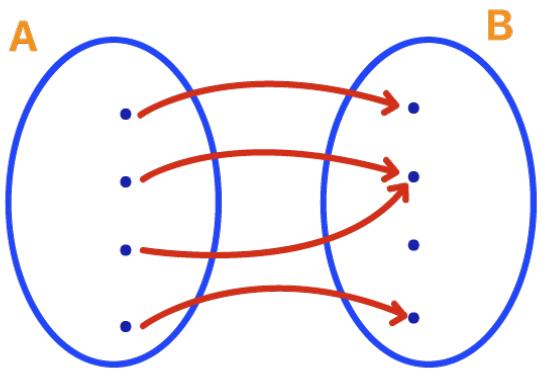
  - Os valores podem ser de qualquer tipo, mas as chaves só podem ser de tipos imutáveis.
  - Dicionários são **mutáveis**.
  - As chaves precisam ser **únicas**[2].

[2] `d["tres"] = "III"` é outra associação.

Dicionários

- Podemos associar mais de uma chave ao mesmo valor:

```
1 >>> d["dos"] = 2
2 >>> d
3 {'um': 1, 'dois': 2, 'tres': 3, \
4  'dos': 2}
```

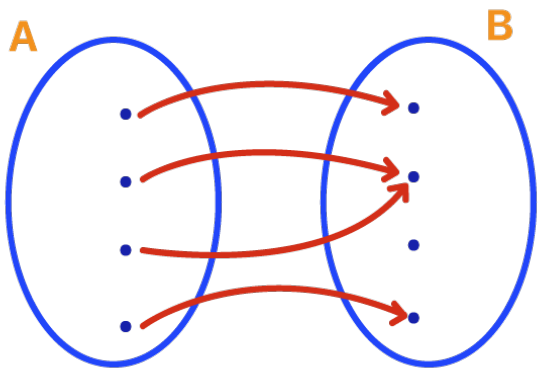


- Não importa em que ordem que escrevemos os pares.
  - O **Python** otimiza a organização dos dados para prover acesso rápido aos pares chave-valor.

Dicionários

- Podemos associar mais de uma chave ao mesmo valor:

```
1 >>> d["dos"] = 2
2 >>> d
3 {'um': 1, 'dois': 2, 'tres': 3, \
4  'dos': 2}
```



- Não importa em que ordem que escrevemos os pares.
  - O **Python** otimiza a organização dos dados para prover acesso rápido aos pares chave-valor.

Dicionários

- Um dicionário pode ser visto como uma lista, em que o acesso é feito pelas chaves:

```
>>> meses = {
              "jan" : 1,
              "fev" : 2,
              "mar" : 3,
              "abr" : 4,
              "mai" : 5,
              "jun" : 6,
              "jul" : 7,
              "ago" : 8,
              "set" : 9,
              "out" : 10,
              "nov" : 11,
              "dez" : 12
             }
>>> d["set"]
9
```

Dicionários

Vamos ver um exemplo em que um dicionário é utilizado como lista telefonica:

```
>>> dd = {
          "Jose" : 12345678,
          "Maria": 78765432,
          "Eduardo" : 12341234
          }
```

- Cada chave (tipo `string`, imutável) → um valor (tipo `inteiro`).

Dicionários

Podemos acessar o valor associado à qualquer chave:

```
>>> dd["Maria"]
78765432
```

- Se a chave não existir, temos um erro:

```
>>> dd["Antonio"]
Traceback (most recent call last):
  File "<pyshell#164>", line 1, in <module>
    dd["Antonio"]
KeyError: 'Antonio'
```

Dicionários

- Podemos verificar se uma chave está ou não no dicionário:

```
>>> dd
{'Jose': 12345678, 'Maria': 78765432, 'Eduardo': 12341234}
>>> "Maria" in dd
True
>>> "Antonio" in dd
False
```

- O método `get()` acessa uma chave, sem lançar uma exceção quando o item não está no dicionário.

```
>>> dd.get("Maria")
78765432
>>> dd.get("Antonio")
>>>
```

Dicionários

- Podemos verificar se uma chave está ou não no dicionário:

```
>>> dd
{'Jose': 12345678, 'Maria': 78765432, 'Eduardo': 12341234}
>>> "Maria" in dd
True
>>> "Antonio" in dd
False
```

- O método `get()` acessa uma chave, sem lançar uma exceção quando o item não está no dicionário.

```
>>> dd.get("Maria")
78765432
>>> dd.get("Antonio")
>>>
```

Dicionários

Podemos alterar o valor associado a uma chave:

```
>>> dd["Maria"] = 1111111
>>> dd["Jose"] = 2222222
>>> dd
{'Maria': 1111111, 'Jose': 2222222, 'Eduardo': 12341234}
```

- E adicionar novos pares ao dicionário:

```
>>> dd["Carlos"] = 45677654
>>> dd
{'Jose': 2222222, 'Maria': 1111111, 'Eduardo': 12341234,\
 'Carlos': 45677654}
```

  - Se a chave existir, o valor é substituído.
  - Outra opção, `dd.update({"Carlos": 45677654})`

Dicionários

Podemos alterar o valor associado a uma chave:

```
>>> dd["Maria"] = 1111111
>>> dd["Jose"] = 2222222
>>> dd
{'Maria': 1111111, 'Jose': 2222222, 'Eduardo': 12341234}
```

- E adicionar novos pares ao dicionário:

```
>>> dd["Carlos"] = 45677654
>>> dd
{'Jose': 2222222, 'Maria': 1111111, 'Eduardo': 12341234,\
 'Carlos': 45677654}
```

  - Se a chave existir, o valor é substituído.
  - Outra opção, `dd.update({"Carlos": 45677654})`

Dicionários

Podemos alterar o valor associado a uma chave:

```
>>> dd["Maria"] = 1111111
>>> dd["Jose"] = 2222222
>>> dd
{'Maria': 1111111, 'Jose': 2222222, 'Eduardo': 12341234}
```

- E adicionar novos pares ao dicionário:

```
>>> dd["Carlos"] = 45677654
>>> dd
{'Jose': 2222222, 'Maria': 1111111, 'Eduardo': 12341234,\
 'Carlos': 45677654}
```

  - Se a chave existir, o valor é substituído.
  - Outra opção, `dd.update({"Carlos": 45677654})`

Dicionários

O método `pop()` remove um item a partir de sua chave:

```
>>> 2222222
{'Jose': 2222222, 'Maria': 1111111, 'Eduardo': 12341234, \
 'Carlos': 45677654}
##
>>> dd.pop("Eduardo")
12341234
>>> dd
{'Jose': 2222222, 'Maria': 1111111, 'Carlos': 45677654}
```

- Tentar remover um item que não está no dicionário causa um erro:

```
>>> dd.pop("Eduardo")
Traceback (most recent call last):
  File "<pyshell#222>", line 1, in <module>
    dd.pop("Eduardo")
KeyError: 'Eduardo'
```

  - `dd.pop("Eduardo", -1)` faz com que nenhuma exceção seja lançada.

Dicionários

O método `pop()` remove um item a partir de sua chave:

```
>>> 2222222
{'Jose': 2222222, 'Maria': 1111111, 'Eduardo': 12341234, \
 'Carlos': 45677654}
##
>>> dd.pop("Eduardo")
12341234
>>> dd
{'Jose': 2222222, 'Maria': 1111111, 'Carlos': 45677654}
```

- Tentar remover um item que não está no dicionário causa um erro:

```
>>> dd.pop("Eduardo")
Traceback (most recent call last):
  File "<pyshell#222>", line 1, in <module>
    dd.pop("Eduardo")
KeyError: 'Eduardo'
```

  - `dd.pop("Eduardo", -1)` faz com que nenhuma exceção seja lançada.

Dicionários

- O método `clear()` remove **todos os itens** de um dicionário:

```
>>> dd.clear()
>>> dd
{}
```

- O operador `del` pode remover um item ou o dicionário inteiro:

```
>>> dd = {'Jose': 12345678, 'Maria': 78765432, 'Carlos': 45677654}
>>> del dd["Jose"]
>>> dd
##
>>> del dd
>>> dd
Traceback (most recent call last):
  File "<pyshell#235>", line 1, in <module>
    dd
NameError: name 'dd' is not defined
```

Dicionários

- O método `clear()` remove **todos os itens** de um dicionário:

```
>>> dd.clear()
>>> dd
{}
```

- O operador `del` pode remover um item ou o dicionário inteiro:

```
>>> dd = {'Jose': 12345678, 'Maria': 78765432, 'Carlos': 45677654}
>>> del dd["Jose"]
>>> dd
##
>>> del dd
>>> dd
Traceback (most recent call last):
  File "<pyshell#235>", line 1, in <module>
    dd
NameError: name 'dd' is not defined
```

Dicionários

- O método `clear()` remove **todos os itens** de um dicionário:

```
>>> dd.clear()
>>> dd
{}
```

- O operador `del` pode remover um item ou o dicionário inteiro:

```
>>> dd = {'Jose': 12345678, 'Maria': 78765432, 'Carlos': 45677654}
>>> del dd["Jose"]
>>> dd
##
>>> del dd
>>> dd
Traceback (most recent call last):
  File "<pyshell#235>", line 1, in <module>
    dd
NameError: name 'dd' is not defined
```

Dicionários

- O método `keys()` retorna uma lista com as chaves:

```
>>> dd.keys()
dict_keys(['Jose', 'Maria', 'Eduardo', 'Carlos'])
```

```
type(dd.keys())
<class 'dict_keys'>
```

```
>>> l = list(dd.keys())
>>> l
['Jose', 'Maria', 'Eduardo', 'Carlos']
```

Dicionários

- O método `keys()` retorna uma lista com as chaves:

```
>>> dd.keys()
dict_keys(['Jose', 'Maria', 'Eduardo', 'Carlos'])
```

```
type(dd.keys())
<class 'dict_keys'>
```

```
>>> l = list(dd.keys())
>>> l
['Jose', 'Maria', 'Eduardo', 'Carlos']
```

Dicionários

- O método `values()` retorna uma lista com os valores:

```
>>> dd.values()
dict_values([12345678, 78765432, 12341234, 45677654])
```

```
>>> type(dd.values())
<class 'dict_values'>
```

```
>>> l = list(dd.values())
>>> l
[12345678, 78765432, 12341234, 45677654]
```

Dicionários

- O método `values()` retorna uma lista com os valores:

```
>>> dd.values()
dict_values([12345678, 78765432, 12341234, 45677654])
```

```
>>> type(dd.values())
<class 'dict_values'>
```

```
>>> l = list(dd.values())
>>> l
[12345678, 78765432, 12341234, 45677654]
```

Dicionários

- Podemos percorrer todos os elementos do dicionário com um **laço**:

```
>>> for item in dd:
  print(item)

Jose
Maria
Eduardo
Carlos
```

  - Acessa apenas as chaves.

```
>>> for item in dd:
  print(item, dd[item])

Jose 12345678
Maria 78765432
Eduardo 12341234
Carlos 45677654
```

Dicionários

- Podemos percorrer todos os elementos do dicionário com um **laço**:

```
>>> for item in dd:
  print(item)

Jose
Maria
Eduardo
Carlos
```

  - Acessa apenas as chaves.

```
>>> for item in dd:
  print(item, dd[item])

Jose 12345678
Maria 78765432
Eduardo 12341234
Carlos 45677654
```

Dicionários

- O método `items()` retorna uma lista de `tuplas` com os pares chave/valor:

```
>>> dd.items()
dict_items([('Jose', 12345678), ('Maria', 78765432), \
            ('Eduardo', 12341234), ('Carlos', 45677654)])
```

```
>>> type(dd.items())
<class 'dict_items'>
```

- Podemos iterar sobre essa `lista` de `tuplas`:

```
>>> for k, v in dd.items():
...   print(k, v)
```

Dicionários

- O método `items()` retorna uma lista de `tuplas` com os pares chave/valor:

```
>>> dd.items()
dict_items([('Jose', 12345678), ('Maria', 78765432), \
            ('Eduardo', 12341234), ('Carlos', 45677654)])
```

```
>>> type(dd.items())
<class 'dict_items'>
```

- Podemos iterar sobre essa `lista` de `tuplas`:

```
>>> for k, v in dd.items():
  print(k, v)
```

Roteiro

1. Tuplas
2. Sets
3. Dicionários
4. Exemplos
5. Referências

Exemplo 1: Extraindo textos

Escreva a função `extraiTexto()`, que recebe uma `tupla` e retorna outra `tupla` apenas com valores de texto.

```
>>> t = ("Carlos", "Daniel", 1993, 11, 30, "Belo Horizonte", \
         "MG", "Brazil")
>>> extraiTexto(t)
("Carlos", "Daniel", "Belo Horizonte", "MG", "Brazil")
```

Exemplo 1: Extraindo textos

- Tuplas são **imutáveis**, vamos criar uma `lista` primeiro:

```
1 def extraiTexto(t):
2   x = []
3   for item in t:
4     if(type(item) == str):
5       x.append(item)
6   ##
7   t = tuple(x)
8   return t
```

Exemplo 1: Extraindo textos

- Tuplas são **imutáveis**, vamos criar uma `lista` primeiro:

```
1 def extraiTexto(t):
2   x = []
3   for item in t:
4     if(type(item) == str):
5       x.append(item)
6   ##
7   t = tuple(x)
8   return t
```

Exemplo 1: Extraindo textos

- Tuplas são **imutáveis**, vamos criar uma `lista` primeiro:

```
1 def extraiTexto(t):
2   x = []
3   for item in t:
4     if(type(item) == str):
5       x.append(item)
6   ##
7   t = tuple(x)
8   return t
```

Exemplo 2: Lei de De Morgan

Escreva a função `interseccao()`, que recebe dois `conjuntos` A e B com números em $[0, 10)$ e retorna a $A \cap B$.

- Utilize a Lei de *De Morgan*:

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

```
>>> interseccao({1, 3, 5, 7}, {0, 2, 3, 7, 9})
{3, 7}
```

Exemplo 2: Lei de De Morgan

- Vamos criar $\overline{A}$ e $\overline{B}$:

```
1 def complemento(C, ini, fim):
2   R = set() #conjunto vazio
3   for i in range(ini, fim):
4     if(i not in C):
5       R.add(i)
6   return R
```

- Depois, $A \cap B = \overline{\overline{A} \cup \overline{B}}$

```
1 def interseccao(A, B):
2   notA = complemento(A, 0, 10)
3   notB = complemento(B, 0, 10)
4   ##
5   C = notA.union(notB)
6   D = complemento(C, 0, 10)
7   ##
8   return D
```

Exemplo 2: Lei de De Morgan

- Vamos criar $\overline{A}$ e $\overline{B}$:

```
1 def complemento(C, ini, fim):
2   R = set() #conjunto vazio
3   for i in range(ini, fim):
4     if(i not in C):
5       R.add(i)
6   return R
```

- Depois, $A \cap B = \overline{\overline{A} \cup \overline{B}}$

```
1 def interseccao(A, B):
2   notA = complemento(A, 0, 10)
3   notB = complemento(B, 0, 10)
4   ##
5   C = notA.union(notB)
6   D = complemento(C, 0, 10)
7   ##
8   return D
```

Exemplo 2: Lei de De Morgan

- Vamos criar $\overline{A}$ e $\overline{B}$:

```
1 def complemento(C, ini, fim):
2   R = set() #conjunto vazio
3   for i in range(ini, fim):
4     if(i not in C):
5       R.add(i)
6   return R
```

- Depois, $A \cap B = \overline{\overline{A} \cup \overline{B}}$

```
1 def interseccao(A, B):
2   notA = complemento(A, 0, 10)
3   notB = complemento(B, 0, 10)
4   ##
5   C = notA.union(notB)
6   D = complemento(C, 0, 10)
7   ##
8   return D
```

Exemplo 2: Lei de De Morgan

- Vamos criar $\overline{A}$ e $\overline{B}$:

```
1 def complemento(C, ini, fim):
2   R = set() #conjunto vazio
3   for i in range(ini, fim):
4     if(i not in C):
5       R.add(i)
6   return R
```

- Depois, $A \cap B = \overline{\overline{A} \cup \overline{B}}$

```
1 def interseccao(A, B):
2   notA = complemento(A, 0, 10)
3   notB = complemento(B, 0, 10)
4   ##
5   C = notA.union(notB)
6   D = complemento(C, 0, 10)
7   ##
8   return D
```

Exemplo 2: Lei de De Morgan

- Podemos comparar com `intersection()`

```
1 def main():
2   A = {1, 3, 5, 7}
3   B = {0, 2, 3, 7, 9}
4   C = interseccao(A, B)
5   ##
6   if(C != A.intersection(B)):
7     print("Erro")
8   else:
9     print(C)
```

Exemplo 3: Contando Letras

Escreva a função `contaLetras()`, que recebe uma `string` e retorna as frequências de cada letra da string, e a letra mais frequente.

```
>>> contaLetras("ana banana")
({'a': 5, 'n': 3, ' ': 1, 'b': 1}, 'a')
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Vamos usar um dicionário para contar cada letra:

```
1 def contaLetrasVersao1(s):
2   F = {} # dicionário vazio
3   ##
4   for letra in s:
5     if letra in F:
6       F[letra] += 1
7     else:
8       F[letra] = 1 # adiciona letra no dicionário
```

- Depois, percorremos as frequências e encontramos a maior:

```
1   letraMais = ""
2   for chave in F:
3     if letraMais == '': #primeira iteração
4       letraMais = chave
5     elif F[chave] > F[letraMais]:
6       letraMais = chave
7   ##
8   return F, letraMais
```

Exemplo 3: Contando Letras

- Podemos usar `sets` e o método `count()`:

```
1  def contaLetrasVersao2(s):
2      C = set(s)
3      F = {}
4      maior = 0
5      letraMais = ""
6      for item in C:
7          occ = s.count(item)
8          if(occ > maior):
9              maior = occ
10             letraMais = item
11         F.update({item : s.count(item)})
12
13     return F, letraMais
```

Fim

Dúvidas?

Leitura complementar

Leitura complementar:

1. https://realpython.com/python-lists-tuples/#python-tuples
2. https://realpython.com/python-sets/
3. https://realpython.com/python-dicts/

Roteiro

1 Tuplas

2 Sets

3 Dicionários

4 Exemplos

5 Referências

Referências

1. Materiais adaptados dos slides do Prof. Eduardo C. Xavier, da Universidade Estadual de Campinas.