Algoritmos e Lógica de Programação

80 horas // 4 h/semana

Outras Estruturas de Dados

Aula 11

Prof. Piva

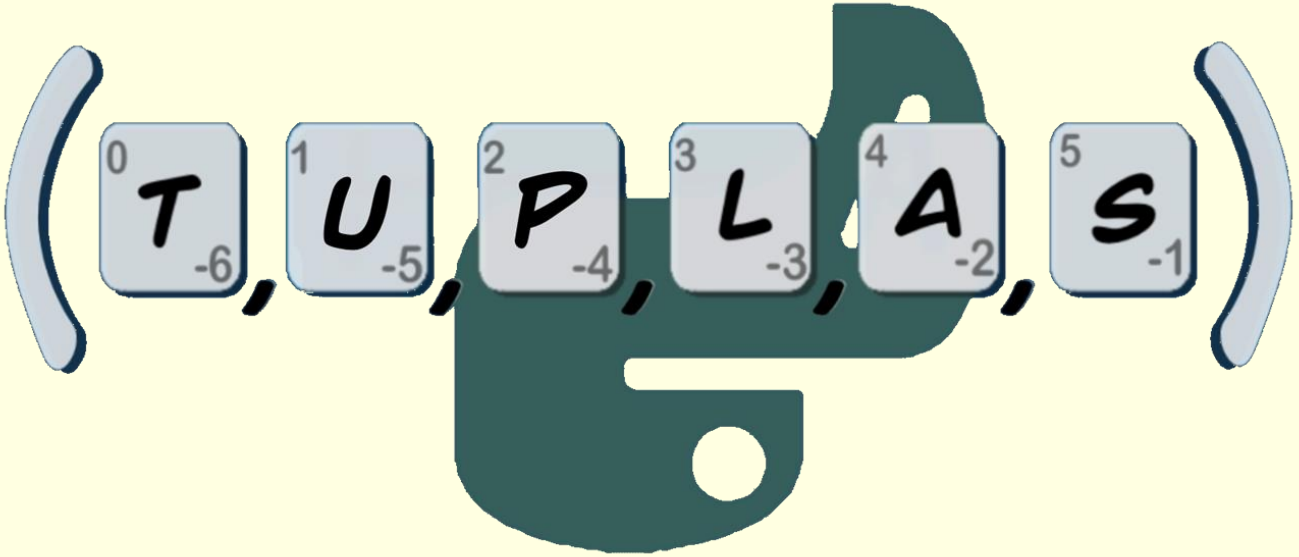
Para começar...

- Aprendemos a importância e a aplicabilidade dos vetores (ou Listas). Com uma ou mais dimensões.
- Existem algumas variações de uma lista...
  - Uma lista que não permite alteração: TUPLAS
  - Uma lista, onde os elementos estão dispostos por uma relação entre <chave>:<valor>: DICIONÁRIOS
  - Uma lista, onde os elementos não se repetem: CONJUNTOS
- Com essas variações, a forma de utilização se amplia e se consolida, tornando determinadas tarefas mais fáceis de serem realizadas.

VAMOS PARA A PRÁTICA ?!!!

Tuplas

Python:
Tuplas



Tuplas

- São estruturas de dados parecidas com listas, mas com a particularidade de serem *imutáveis*
- Tuplas são sequências e, assim como listas, podem ser indexadas e fatiadas, mas não é possível modificá-las
- Um valor do tipo tupla é uma série de valores separados por vírgulas e entre parênteses

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x [0]
1
>>> x [0]=1
...
TypeError: object does not support item assignment
```

Tuplas

- Uma tupla vazia se escreve ()
- Os parênteses são opcionais se não provocarem ambiguidade
- Uma tupla contendo apenas um elemento deve ser escrita com uma vírgula ao final
  - Um valor entre parênteses sem vírgula no final é meramente uma expressão:

```
>>> (10)
10
>>> 10,
(10,)
>>> (10,)
(10,)
>>> 3*(10+3)
39
>>> 3*(10+3,)
(13,13,13)
```

A função *tuple*

- Assim como a função **list** constrói uma lista a partir de uma sequência qualquer, a função **tuple** constrói uma tupla a partir de uma sequência qualquer

```
>>> list("abcd”)
['a', 'b', 'c', 'd']
>>> tuple("abcd”)
('a', 'b', 'c', 'd’))
>>> tuple([1,2,3])
(1, 2, 3)
>>> list((1,2,3))
[1, 2, 3]
```

Quando usar tuplas

- Em geral, tuplas podem ser substituídas com vantagem por listas
- Entretanto, algumas construções em Python requerem tuplas ou sequências imutáveis, por exemplo:
  - Tuplas (ao contrário de listas) podem ser usadas como chaves de dicionários
  - Funções com número variável de argumentos acessam os argumentos por meio de tuplas
  - O operador de formatação aceita tuplas, mas não listas

O operador de formatação

- Strings suportam o operador % que, dada uma string especial (template) e um valor, produz uma string *formatada*
- O formato geral é
  - *template % valor*
- O template é uma string entremeada por códigos de formatação
  - Um código de formatação é em geral composto do caractere % seguido de uma letra descritiva do tipo do valor a formatar (s para string, f para float, d para inteiro, etc)
- Exemplo:

```
>>> '====%d====' % 100
'====100===='
>>> '====%f====' % 1
'====1.000000===='
```

Formatando tuplas

- Um template pode ser aplicado aos diversos valores de uma tupla para construir uma string formatada
- Ex.:

```
>>> template = "%s tem %d anos"
>>> tupla = ('Pedro', 10)
>>> template % tupla
'Pedro tem 10 anos'
```

- Obs: mais tarde veremos que o operador de formatação também pode ser aplicado a dicionários

Anatomia das especificações de formato

- Caractere %
- Flags de conversão (opcionais):
  - - indica alinhamento à esquerda
  - + indica que um sinal deve preceder o valor convertido
  - “ ” (um branco) indica que um espaço deve preceder números positivos
  - 0 indica preenchimento à esquerda com zeros
- Comprimento mínimo do campo (opcional)
  - O valor formatado terá este comprimento no mínimo
  - Se igual a * (asterisco), o comprimento será lido da tupla
- Um “.” (ponto) seguido pela precisão (opcional)
  - Usado para converter as casas decimais de floats
  - Se aplicado para strings, indica o comprimento máximo
  - Se igual a *, o valor será lido da tupla
- Caractere indicador do tipo de formato

Tipos de formato

- d, i Número inteiro escrito em decimal
- o Número inteiro sem sinal escrito em octal
- u Número inteiro sem sinal escrito em decimal
- x Número inteiro sem sinal escrito em hexadecimal (minúsculas)
- X Número inteiro sem sinal escrito em hexadecimal (maiúsculas)
- e Número de ponto flutuante escrito em notação científica ('e' minúsculo)
- E Número de ponto flutuante escrito em notação científica ('E' maiúsculo)
- f, F Número de ponto flutuante escrito em notação convencional
- g Mesmo que e se expoente é maior que -4. Caso contrario, igual a f
- G Mesmo que E se expoente é maior que -4. Caso contrario, igual a F
- C Caractere único (usado com inteiro ou string de tamanho 1)
- r String (entrada é qualquer objeto Python que é convertido usando a função repr)

Exemplos

```
>>> "Numero inteiro: %d" % 55
'Numero inteiro: 55'
>>> "Numero inteiro com 3 casas: %3d" % 55
'Numero inteiro com 3 casas:  55'
>>> "Inteiro com 3 casas e zeros a esquerda: %03d" % 55
'Inteiro com 3 casas e zeros a esquerda: 055'
>>> "Inteiro escrito em hexadecimal: %x" % 55
'Inteiro escrito em hexadecimal: 37'
>>> from math import pi
>>> "Ponto flutuante: %f" % pi
'Ponto flutuante: 3.141593'
>>> "Ponto flutuante com 12 decimais: %.12f" % pi
'Ponto flutuante com 12 decimais: 3.141592653590'
>>> "Ponto flutuante com 10 caracteres: %10f" % pi
'Ponto flutuante com 10 caracteres:   3.141593'
>>> "Ponto flutuante em notacao cientifica: %10e" % pi
'Ponto flutuante em notacao cientifica: 3.141593e+00'
>>> "String com tamanho maximo definido: %.3s" % "Pedro"
'String com tamanho maximo definido: Ped'
```

Exemplo: Imprimindo uma tabela

```
Itens    = ["Abacate", "Limão", "Tangerina", "Melancia”, "Laranja da China"]
precos = [2.13, 0.19, 1.95, 0.87, 12.00]

len_precos = 10 # Coluna de precos tem 10 caracteres

# Achar a largura da coluna de itens
len_itens = len(itens[0])
for it in itens : len_itens = max(len_itens,len(it))

# Imprimir tabela de precos
print ("-"*(len_itens+len_precos))
print ("%-*s%*s" % (len_itens, "Item", len_precos, "Preço”))
print ("-"*(len_itens+len_precos))
for i in range(len(itens)):
        print ("%-*s%*.2f" % (len_itens, itens[i],len_precos, precos[i]))
```

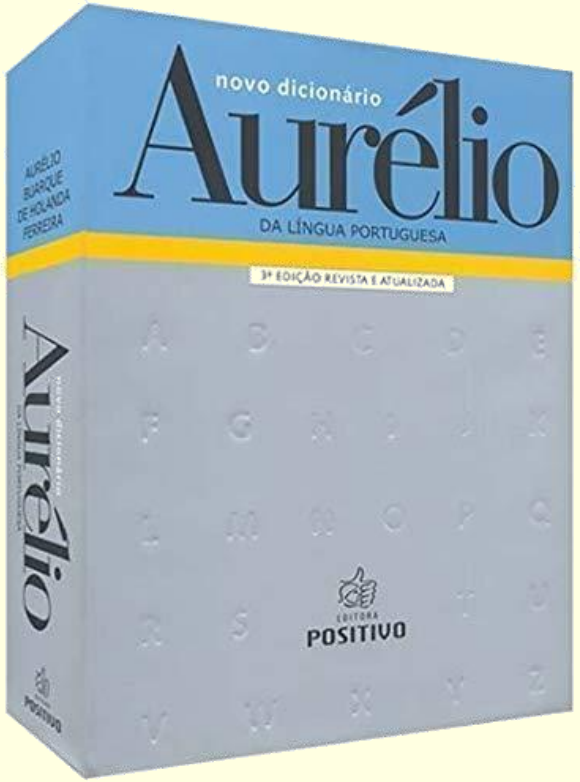
Exemplo: resultados

Item	Preço
Abacate	2.13
Limão	0.19
Tangerina	1.95
Melancia	0.87
Laranja da China	12.00

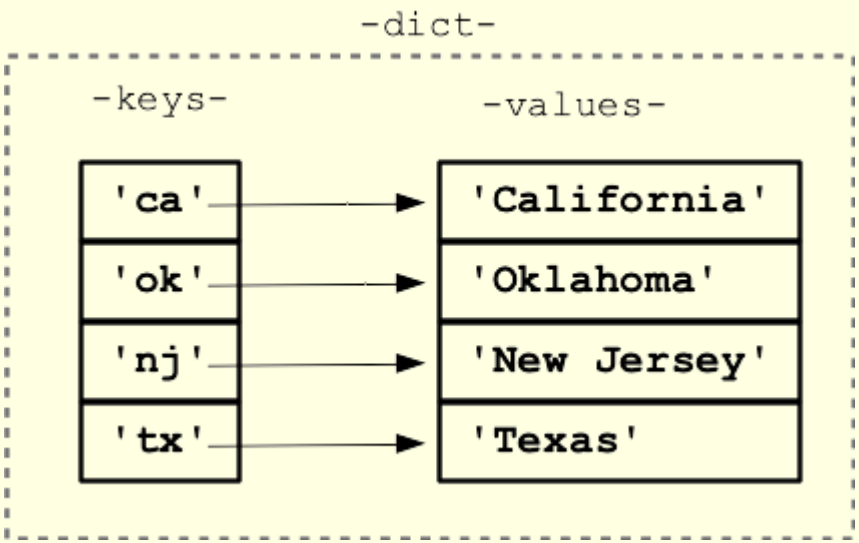
Métodos do tipo: TUPLAS

Função	Sintaxe	Descrição
`count()`	`Tupla.count(item)`	Retorna a quantidade de vezes que um item ocorre em uma tupla
`index()`	`Tupla.index(item)`	Busca um item em uma tupla e retorna a sua posição. Se tiver mais de uma ocorrência do item, retorna a primeira. Se não encontrar o item, retorna Erro para o método.

Dicionários



Python: Dicionários



Dicionários

- São estruturas de dados que implementam *mapeamentos*
- Um mapeamento é uma coleção de associações entre pares de valores
  - O primeiro elemento do par é chamado de *chave* e o outro de *conteúdo*
- De certa forma, um mapeamento é uma generalização da ideia de acessar dados por índices, exceto que num mapeamento os índices (ou chaves) podem ser de qualquer tipo *imutável*

Chaves vs. Índices

- Considere que queiramos representar um caderno de telefones
  - Uma solução é ter uma lista de nomes e outra de telefones
    - Telefone de nome[i] armazenado em telefone[i]
    - Acrescentar “Joao” com telefone “20122232”:

      nome+= “Joao” telefone+=“20122232”
    - Para encontrar o telefone de “Joao”:

      Tel = telefone[nome.index[“Joao”]]
  - Dicionários tornam isso mais fácil e *eficiente*

    telefone[“Joao”] = “20122232”

    Tel = telefone[“Joao”]

Criando dicionários

- Uma constante do tipo dicionário é escrita

  { *chave1*:*conteúdo1*, ... *chaveN*:*conteúdoN*}
- Uma variável do tipo dicionário pode ser “indexada” da maneira habitual, isto é, usando colchetes
- O conteúdo associado a uma chave pode ser alterado atribuindo-se àquela posição do dicionário
- Novos valores podem ser acrescentados a um dicionário fazendo atribuição a uma chave ainda não definida
- Não há ordem definida entre os pares chave/conteúdo de um dicionário

Exemplo

```
>>> dic = {"joao":100,"maria":150}
>>> dic["joao"]
100
>>> dic["maria"]
150
>>> dic["pedro"] = 10
>>> dic
{'pedro': 10, 'joao': 100, 'maria': 150}
>>> dic = {'joao': 100, 'maria': 150, 'pedro': 10}
>>> dic
{'pedro': 10, 'joao': 100, 'maria': 150}
```

Dicionários não têm ordem

- As chaves dos dicionários não são armazenadas em qualquer ordem específica
  - Na verdade, dicionários são implementados por tabelas de espalhamento (*Hash Tables*)
  - A falta de ordem é proposital
- Diferentemente de listas, atribuir a um elemento de um dicionário não requer que a posição exista previamente

```
X = []
X [10] = 5  # ERRO!
   . . .
Y = {}
Y [10] = 5  # OK!
```

A função *dict*

- A função dict é usada para construir dicionários e requer como parâmetros:
  - Uma lista de tuplas, cada uma com um par chave/conteúdo, ou
  - Uma sequência de itens no formato *chave*=*valor*
    - Nesse caso, as chaves têm que ser strings, mas são escritas sem aspas

Exemplo

```
>>> d = dict([(1,2),('chave','conteudo')])
>>> d[1]
2
>>> d['chave']
'conteudo'
>>> d = dict(x=1,y=2)
>>> d['x']
1
>>> d = dict(1=2,3=4)
SyntaxError: keyword can't be an expression
```

Formatando com Dicionários

- O operador de formatação quando aplicado a dicionários requer que os valores das chaves apareçam entre parênteses antes do código de formatação
  - O conteúdo armazenado no dicionário sob aquela chave é substituído na string de formatação
  - Ex:

```
>>> dic = { "Joao":"a", "Maria":"b" }
>>> s = "%(Joao)s e %(Maria)s"
>>> s % dic
'a e b'
```

Métodos do tipo: DICIONÁRIO

Função	Sintaxe	Descrição
`clear()`	`Dic.clear()`	Remove todos os pares chave:valor de um dicionário.
`fromkeys()`	`Dic.fromkeys(seq[,valor])`	Cria um novo dicionário a partir da sequencia dos elementos com um valor informado
`get()`	`Dic.get(chave[,padrão])`	Retorna o valor associado à chave especificada no dicionário. Se a chave não estiver presente, ela retornará o valor padrão. Se padrão não for fornecido, o padrão será *None*, para que esse método nunca gere um KeyError.
`items()`	`Dic.items()`	Retorna uma nova visão dos pares de chave e valor do dicionário como tuplas.
`keys()`	`Dic.keys()`	Retorna uma nova view que consiste em todas as chaves do dicionário.
`pop()`	`Dic.pop(chave[,padrão])`	Remove a chave do dicionário e retorna seu valor. Se a chave não estiver presente, ela retornará o valor padrão. Se o padrão não for fornecido e a chave não estiver no dicionário, isso resultará em KeyError.
`popitem()`	`Dic.popitem()`	Remove e retorna um par de tuplas arbitrário (chave, valor) do dicionário. Se o dicionário estiver vazio, chamar popitem() resultará em KeyError.
`setdefault()`	`Dic.setdefault(chave[,padrão])`	Retorna um valor para a chave presente no dicionário. Se a chave não estiver presente, insere a chave no dicionário com um valor padrão e retorna o valor padrão. Se a chave estiver presente, o padrão é None, para que esse método nunca gere um KeyError.
`update()`	`Dic.update([outro])`	Atualiza o dicionário com os pares chave:valor de outro objeto de dicionário e retorna None.
`values()`	`Dic.values()`	Retorna uma nova view que consiste em todos os valores do dicionário.

Método *clear*

- clear()
  - Remove todos os elementos do dicionário
  - Ex.:

```
>>> x = { "Joao":"a", "Maria":"b" }
>>> y = x
>>> x.clear()
>>> print (x,y)
{} {}
```

  - Diferente de atribuir {} à variável:

```
>>> x = { "Joao":"a", "Maria":"b" }
>>> y = x
>>> x = {}
>>> print (x,y)
{} {'Joao': 'a', 'Maria': 'b'}
```

Método *copy*

- copy()
  - Retorna um outro dicionário com os mesmos pares chave/conteúdo
  - Observe que os conteúdos não são cópias, mas apenas referências para os mesmos valores

```
>>> x = {"Joao":[1,2], "Maria":[3,4]}
>>> y = x.copy()
>>> y ["Pedro"]=[5,6]
>>> x ["Joao"] += [3]
>>> print (x)
{'Joao': [1, 2, 3], 'Maria': [3, 4]}
>>> print (y)
{'Pedro': [5, 6], 'Joao': [1, 2, 3], 'Maria': [3, 4]}
```

Método *fromkeys*

- fromkeys(*lista*,*valor*)
  - Retorna um novo dicionário cujas chaves são os elementos de *lista* e cujos valores são todos iguais a *valor*
  - Se valor não for especificado, o default é None

```
>>> {}.fromkeys([2,3])
{2: None, 3: None}

# Podemos usar o nome da classe ao invés
# de um objeto:
>>> dict.fromkeys(["Joao","Maria"],0)
{'Joao': 0, 'Maria': 0}
```

Método *get*

- get(*chave*,*valor*)
  - Obtém o conteúdo de *chave*
  - Não causa erro caso chave não exista: retorna *valor*
  - Se *valor* não for especificado chaves inexistentes retornam None
  - Ex.:

```
>>> dic = { "Joao":"a", "Maria":"b" }
>>> dic.get("Pedro”)
>>> print (dic.get("Pedro”))
None
>>> print (dic.get("Joao”))
a
>>> print (dic.get("Carlos","N/A”))
N/A
```

Método *has_key*

- has_key(*chave*)
  - dic.has_key(*chave*) é o mesmo que *chave* in dic
  - Ex.:

```
>>> dic = { "Joao":"a", "Maria":"b" }
>>> dic.has_key("Joao”)
True
>>> dic.has_key("Pedro”)
False
```

Métodos *items*, *keys* e *values*

- items() retorna uma lista com todos os pares chave/conteúdo do dicionário
- keys() retorna uma lista com todas as chaves do dicionário
- values() retorna uma lista com todos os valores do dicionário
- Ex.:

```
>>> dic.items()
[('Joao', 'a'), ('Maria', 'b')]
>>> dic.keys()
['Joao', 'Maria']
>>> dic.values()
['a', 'b']
```

Método *pop*

- pop (chave)
  - Obtém o valor correspondente a chave e remove o par chave/valor do dicionário
  - Ex.:

```
>>> d = {'x': 1, 'y': 2}
>>> d.pop('x’)
1
>>> d
{'y': 2}
```

Método *popitem*

- popitem()
  - Retorna e remove um par chave/valor aleatório do dicionário
  - Pode ser usado para iterar sobre todos os elementos do dicionário
  - Ex:

```
>>> d
{'url': 'http://www.python.org', 'spam': 0, 'title': 'Python Web Site'}
>>> d.popitem()
('url', 'http://www.python.org')
>>> d
{'spam': 0, 'title': 'Python Web Site'}
```

Método *update*

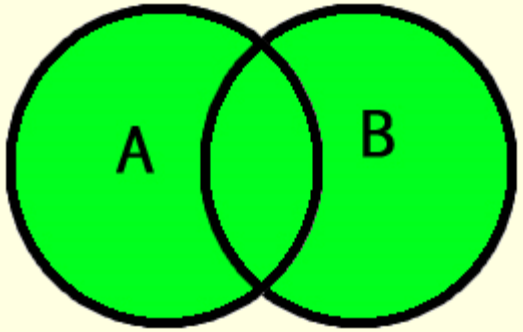
- update(*dic*)
  - Atualiza um dicionário com os elementos de outro
  - Os itens em *dic* são adicionados um a um ao dicionário original
  - É possível usar a mesma sintaxe da função dict para especificar *dic*
  - Ex.:

```
>>> x = {"a":1,"b":2,"c":3}
>>> y = {"z":9,"b":7}
>>> x.update(y)
>>> x
{'a': 1, 'c': 3, 'b': 7, 'z': 9}
>>> x.update(a=7,c="xxx”)
>>> x
{'a': 7, 'c': 'xxx', 'b': 7, 'z': 9}
```

Conjuntos

Python: Conjuntos

A = {1,2,3}
B = {3,4,5}
AUB = {1,2,3,4,5}



Conjuntos

- Um conjunto é uma coleção de valores distintos
- Pode-se implementar conjuntos de diversas formas
  - Uma lista de valores
    - Têm-se que tomar o cuidado de evitar valores duplicados
  - Um dicionário
    - As chaves de um dicionário são necessariamente únicas
    - O valor associado a cada chave pode ser qualquer um
- Python suporta um tipo primitivo chamado **set** que implementa conjuntos
  - Mais apropriado do que o uso de listas ou dicionários

O tipo *set*

- Pode-se construir um set usando a construção set(*sequência*)
  - Onde *sequência* é uma sequência qualquer, como uma lista, uma tupla ou uma string
  - Caso use-se uma lista, os elementos devem ser imutáveis
- Exemplos:

```
>>> set((1,2,3))
set([1, 2, 3])
>>> set ("xxabc")
set(['a', 'x', 'c', 'b'])
>>> set ([1,(1,2),3,1])
set([(1, 2), 1, 3])
>>> set ([1,[1,2],3,1])
ERROR...
```

Trabalhando com sets

- *x* in *s* → True se o elemento *x* pertence a *s*
- *s*.add(*x*) → Inclui o elemento *x* em *s*
- *s*.copy() → Retorna uma cópia de *s*
- *s*.union(*r*) → Retorna a união entre *s* e *r*
- *s*.intersection(*r*) → Retorna a interseção entre *s* e *r*
- *s*.difference(*r*) → Retorna a diferença entre *s* e *r*
- list*(s)* → Retorna os elementos de *s* numa lista
- tuple*(s)* → Retorna os elementos de *s* numa tupla

Exemplos

```
>>> s = set([1,2,3])
>>> r = set([2,5,9,1])
>>> 1 in s
True
>>> 1 in r
True
>>> s.union(r)
set([1, 2, 3, 5, 9])
>>> s.intersection(r)
set([1, 2])
>>> s.difference(r)
set([3])
>>> r.difference(s)
set([9, 5])
>>> s.add(5)
>>> s.intersection(r)
set([1, 2, 5])
```

Iterando sobre sets

- Pode-se também usar o comando for com sets
- Observe-se que a iteração não necessariamente visita os elementos na mesma ordem em que eles foram inseridos no conjunto
- Exemplo:

```
>>> s = set([1,2,9,100,"a"])
>>> for x in s:
                print x,


a 1 2 100 9
```

Outros métodos

- *s*.discard(*x*) → Exclui o elemento *x* de *s* (se existir)
- *s*.issubset(*r*) → True sse *s* contido em *r*
- *s*.issuperset(*r*) → True sse *s* contém *r*
- *s*.symmetric_difference(*r*) → Retorna a diferença simétrica entre *s* e *r*, isto é, a união entre *s* e *r* menos a interseção de *s* e *r*
- *s*.update(*r*) → mesmo que *s* = *s*.union(*r*)
- *s*.intersection_update(*r*) → mesmo que *s* = *s*.intersection(*r*)
- *s*.difference_update(*r*) → mesmo que *s* = *s*.difference(*r*)

Exemplos

```
>>> s = set([1,2,3])
>>> r = set([2,5,9])
>>> s.update(r)
>>> s
set([1, 2, 3, 5, 9])
>>> s.issuperset(r)
True
>>> r.issubset(s)
True
>>> s.discard(5)
>>> s
set([1, 2, 3, 9])
>>> s.symmetric_difference(r)
set([3, 5, 1])
```

EXERCÍCIOS

Algoritmos ...
Tuplas, Dicionários e Conjuntos

EXERCÍCIO 1

Faça um algoritmo que carregue uma tupla de 10 elementos numéricos inteiros. Após a finalização da entrada, o algoritmo deve escrever a mesma tupla, na ordem inversa de entrada.

EXERCÍCIO 2

Faça um algorimto que carregue um dicionário de 10 elementos onde a chave é o sobrenome da pessoa e o valor a sua idade. Após a finalização da entrada, o algoritmo deve escrever o sobrenome da pessoa com maior idade.

EXERCÍCIO 3

Faça algoritmo que carregue duas listas de dez elementos numéricos inteiros cada um. A partir dessas duas listas, crie um conjunto da união entre essas duas listas.