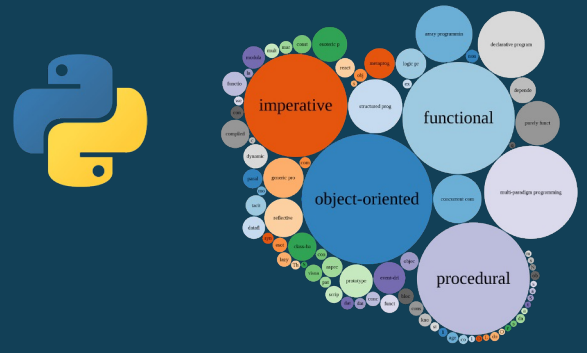
Paradigmas de Linguagem de Programação em Python



Listas, Dicionarios e Tuplas

listas+dicionarios+tuplas.ipynb

Média dos Alunos

```
while True:
      nome = input("Nome: ")
      if nome == '-1':
           break

       n1 = float(input("Nota da primeira prova: "))
       n2 = float(input("Nota da segunda prova: "))

       media = (n1 + n2) / 2
          if media >= 7:
           print("O aluno %s foi aprovado com media %.2f" % (nome, media))
          else:
           print("O aluno %s foi reprovado com media %.2f" % (nome, media))

print("Fim!")
```

Média dos Alunos & Media da Turma

```
soma, qtde = 0, 0
while True:
      nome = input("Nome: ")
      if nome == '-1':
        break
      n1 = float(input("Nota da primeira prova: "))
      n2 = float(input("Nota da segunda prova: "))
      media = (n1 + n2) / 2
      if media >= 7:
            print("O aluno %s foi aprovado com media %.2f" % (nome, media))
      else:
            print("O aluno %s foi reprovado com media %.2f" % (nome, media))
      qtde += 1
      soma += media
print("Media da turma: " % (soma / qtde))
print("Fim!")
```

Mas...

E se quiséssemos calcular o desvio padrão?

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Precisaríamos ter a **média geral** e a **média de cada aluno**

Mas **como**, se as médias dos estudantes são sobrescritas a cada iteração?

Mas...

E se quiséssemos calcular o desvio padrão?

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Precisaríamos ter a **média geral** e a **média de cada aluno**

Mas **como**, se as médias dos estudantes são sobrescritas a cada iteração?

→O ideal seria **armazenar** cada média numa variável que não fosse sobrescrita a cada iteração. ←

Coleções em Python

Python tem 4 tipos de coleções de dados que nos permite armazenar informação:

List (Lista): coleção de dados ordenada e mutável, que permite elementos duplicados.

Tuple (Tupla): coleção de dados ordenada e imutável, que permite elementos duplicados.

Set (Conjunto): coleção de dados não ordenados e não indexados, que não admite repetição de elementos.

Dictionary (Dicionário): coleção de dados não ordenados, mutáveis e indexáveis, que não permite "membros" repetidos.

Voltando...

E se quiséssemos calcular o desvio padrão?

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Precisaríamos ter a **média geral** e a **média de cada aluno**

Mas **como**, se as médias dos estudantes são sobrescritas a cada iteração? → **Coleções!**

Voltando...

E se quiséssemos calcular o desvio padrão?

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Precisaríamos ter a **média geral** e a **média de cada aluno**

Mas **como**, se as médias dos estudantes são sobrescritas a cada iteração? → **Coleções!**

- Mas qual?

Voltando...

E se quiséssemos calcular o desvio padrão?

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Precisaríamos ter a **média geral** e a **média de cada aluno**

Mas **como**, se as médias dos estudantes são sobrescritas a cada iteração? → **Coleções!**

- Mas qual? → **Listas!** (maneira mais simples e prática)

Listas

`List (Lista)` é uma coleção de dados ordenada e mutável, que permite elementos duplicados.

São um tipo de variável que permite o armazenamento de vários valores, acessados por um índice.

Pode conter **zero ou mais elementos** de um mesmo tipo ou de tipos diferentes.

O tamanho de uma lista é dado pela quantidade de elementos que ela contém

Listas
Na prática

```
medias = []
soma = 0
while True:
      nome = input("Nome: ")
            if nome == '-1':
                  break
            n1 = float(input("Nota da primeira prova: "))
            n2 = float(input("Nota da segunda prova: "))
            media = (n1 + n2) / 2
            if media >= 7:
                  print("O aluno %s foi aprovado com media %.2f" % (nome, media))
            else:
                  print("O aluno %s foi reprovado com media %.2f" % (nome, media))
            soma += media
            medias.append(media)
print("Todas as medias: %s" % medias)
print("Media da turma: %f" % (soma / len(medias)))
print("Fim!")
```

Listas

Uma lista vazia é iniciada com `[]`

Uma lista com elementos de mesmo tipo pode ser inicializada desta forma: `z = [1, 15, 25, 35]`

Uma lista com elementos de tipos diferentes pode ser inicializada desta forma: `a = [1, 'k', [3, 4]]`

A função `sum()` retorna a soma dos elementos de uma lista

A função `len()` retorna a quantidade de elementos

Listas
Inserção

Para adicionar elementos ao final de uma lista, usa-se o método **append()**

```
a = ["b", "c", "d"]
a.append("e")
```

Para adicionar elementos especificando a posição, usa-se o método **insert()**

insert(1, "b") insere a **"b"** na posição (índice) 1 da lista

```
a = ["a", "c", "d"]
a.insert(1, "b")
```

Listas
Acesso

Os elementos de uma lista podem ser acessados pelo seu índice: **`b = a[0]`**

Os índices começam em zero (primeiro elemento)

```
lista = [3, 8, 'b']
print("Lista inteira: %s" % lista)


i = 0
while i < len(lista):
 print(lista[i])
 i += 1
```

Listas
Acesso

Podem ser percorridas através do uso de um laço **for**

```
lista = [3, 8, 'b']
print("Lista inteira: %s" % lista)
for x in lista:
    print(x)
```

Listas
Acesso [Indexação Negativa]

Indexação negativa significa que, começando pelo final, **-1** se refere ao último item, **-2** penúltimo item, etc.

```
lista = ["laranja", "azul", "verde"]
print(lista[-1])
```

Listas
Acesso [Range de Índices]

É possível especificar o **range** de índices de elementos a serem exibidos **[início, fim)**.

```
lista = ["laranja", "azul", "verde", "preto"]
print(lista[1:3])
```

Listas
Acesso [Range de Índices Negativos]

É possível especificar o **range negativo** de índices de elementos a serem exibidos **`[início, fim)`**.

```
lista = ["laranja", "azul", "verde", "preto"]
print(lista[-3:-1])
```

Listas
Acesso [Mudando Valores]

Para alterar o valor de um item específico, indique o seu índice.

```
lista = ["laranja", "azul", "verde", "preto"]
Lista[1] = "vermelho"
print(lista)
```

Listas
Acesso [Mudando Valores]

Para alterar o valor de um item específico, indique o seu índice.

```
lista = ["laranja", "azul", "verde", "preto"]
for i in range(len(lista)):
    lista[i] = 1
print(lista)
```

Listas
Cópia e Fatiamento

Assim como fazemos com **strings** podemos fatiar as listas:

b = a[2:-3]

Embora seja um recurso poderoso, um dos "problemas" é quando tentamos fazer cópias. → Cuidado!

```
a = [3, 4, 1]
b = a
b[0] = 9
print("B: %s" % b)
print("A: %s" % a)
```

Listas
Cópia e Fatiamento

No código anterior, ao alterar **`b[0]`**, o conteúdo de **`a[0]`** será igualmente alterado.

Isso ocorre porque a e b são apenas "apelidos" para a mesma lista, armazenada num mesmo endereço de memória

Isso se chama cópia rasa (*shallow copy*)

```
a = [3, 4, 1]
b = a
b[0] = 9
print("B: %s" % b)
print("A: %s" % a)
```

Listas
Cópia e Fatiamento

Cópia rasa (*shallow copy*) → Solução:

Fazer uma **cópia profunda** (*deep copy*), i.e., a lista deverá ser copiada para outro endereço de memória

- `b = a[:]` ou `b = a.copy()`
- `copy()` é um método (função) de lista que faz uma cópia profunda da lista

Listas
Concatenação

É possível concatenar listas utilizando o operador **+**

```
a = [3, 5, 7]
b = [9, 11, 13]
c = a + b
print("C: %s" % c)
```

Listas
Remoção

É possível remover elementos pelos seus índices com o comando **del**

del a[2:4] remove de **a** os 3º e 4º elementos (índices 2 e 3).

Após a remoção, a lista é reorganizada: o 5º elemento torna-se o 3º, e assim por diante.

```
a = [1, 2, 3, 4, 5]
del a[2:4]
print("A: %s" % a)
```

Listas
Remoção

É possível remover elementos especificando o elemento com o comando **`remove()`**

`remove("azul")` remove de **`lista`** o elemento que tem conteúdo **`"azul"`**

Após a remoção, a lista é reorganizada

```
lista = ["laranja", "azul", "verde", "preto"]
lista.remove("azul")
print(lista)
```

Listas
Remoção

É possível remover elementos especificando o índice (ou o último elemento se não especificado) com o comando **pop()**

pop() remove de **lista** o seu último elemento

```
lista = ["laranja", "azul", "verde", "preto"]
lista.pop()
print(lista)
```

Listas
Checar existência

Para saber se um determinado elemento está presente em uma lista, usa-se o comando **in**

```
L = ["triangulo", "quadrado", "circulo", "retangulo"]
if "triangulo" in L:
    print("Sim, 'triangulo' esta presente na lista")
```

Listas
Remoção Total

É possível remover todos os elementos de uma lista com o comando **clear()**

```
lista = ["laranja", "azul", "verde", "preto"]
lista.clear()
print(lista)
```

O construtor list()

Também é possível utilizar o construtor **list()** para criar uma lista :

```
minha_lista = list(("carro", "casa", "moto")) # note os duplos parenteses
print(minha_lista)
```

Também transformar um gerador de números, como a função **range**, em uma lista:

```
L = list(range(100, 1100, 50))
print(L)
```

Método	Descrição
lista.append(`c`)	Adiciona `c` ao fim da lista
lista.extend(`l`)	Prolonga a lista, adicionado no fim todos os elementos de `l`
lista.insert(`i`, `c`)	Insere `c` na posição `i`, deslocando os demais elementos
lista.remove(`c`)	Remove o primeiro elemento encontrado cujo valor é igual a `c`
lista.pop()	Remove o ultimo elemento (ou um na posição especificada) e o retorna
lista.index(`c`)	Retorna o índice do primeiro item cujo valor é igual a `c`
lista.count(`c`)	Retorna o número de vezes que `c` aparece na lista
lista.sort()	Ordena os itens da lista
lista.reverse()	Inverte a ordem dos elementos
lista.clear()	Limpa a lista
lista.copy()	Faz uma cópia da lista e a retorna

Média dos Alunos Relembrando...

```
while True:
  nome = input("Nome: ")
  if nome == '-1':
    break
  n1 = float(input("Nota da primeira prova: "))
  n2 = float(input("Nota da segunda prova: "))

  media = (n1 + n2) / 2
  if media >= 7:
    print("O aluno %s foi aprovado com media %.2f" % (nome, media))
  else:
    print("O aluno %s foi reprovado com media %.2f" % (nome, media))

print("Fim!")
```

E se...

E se agora quiséssemos obter a média de determinado aluno, fornecendo o nome como entrada?

```
alunos = ["maciely", "mariana", "alisson", "jose adriano"]
medias = [7, 8, 8, 9]
nome = input("Informe o nome do aluno: ")
i = 0
for n in alunos:
    if nome == n:
        print("Media = %d" % media[i])
        break
    i += 1
```

E se...

Acontece que outra coleção do Python lida muito bem com essa situação!

```
medias = {"maciely": 7,
          "mariana": 8,
          "alisson": 8,
          "jose adriano": 9}
nome = input("Informe o nome do aluno: ")
if nome in medias:
    print("Media = %d" % medias[nome])
```

E se...

Acontece que outra coleção do Python lida muito bem com essa situação! → Dicionários

```
medias = {"maciely": 7,
          "mariana": 8,
          "alisson": 8,
          "jose adriano": 9}
nome = input("Informe o nome do aluno: ")
if nome in medias:
    print("Media = %d" % media[nome])
```

Dicionários

`Dictionary (Dicionário)` é uma coleção de dados não ordenados, mutáveis e indexáveis, que não permite "membros" repetidos.

Composto de um conjunto de chaves (*keys*) e valores (*values*)

Consiste em relacionar uma chave a um valor específico

No exemplo anterior, **`medias`** é um dicionário

Em Python, criamos um dicionário utilizando chaves **`{}`**

Dicionários

```
# adicionando novo par chave/valor
medias["reinaldo"] = 8
# mostrando todas as chaves
medias.keys()
# mostrando todos os valores
medias.values()
# percorrendo com for
for nome, nota in medias.items():
 if nota < 7:
     print("%s reprovado!" % nome)
```

Dicionários

Note que, no **for**, o dicionário foi desempacotado nos seus membros constituintes: chave e valor

O método **items()** do dicionários permite que isso aconteça

Na 1ª iteração, **nome** assume "maciely", **nota** assume 7, e assim por diante

```
# percorrendo com for
for nome, nota in medias.items():
 if nota < 7:
     print("%s reprovado!" % nome)
```

Dicionários

Podemos ter dicionários nos quais as chaves são associadas a listas ou mesmo a outros dicionários como valores.

No trecho abaixo, cada chave tem uma lista como valor: o primeiro elemento é a quantidade, e o segundo, o preço

```
estoque = {"tomate": [1000, 2.30],
           "alface": [500, 0.45],
           "batata": [2001, 1.20],
           "feijao": [100, 1.50]}
print(estoque["batata"])
```

Dicionários
Acesso

```
carros = {"marca": "Ford",
          "modelo": "Mustang",
          "ano": 1964}
```

Passando a chave (*key*) direto no dicionário

```
m = carros["modelo"]
```

Ou, utilizando o método **`get()`** e passando a chave (*key*)

```
a = carros.get("ano")
```

Dicionários
Mudando valores

```
carros = {"marca": "Ford",
          "modelo": "Mustang",
          "ano": 1964}

carros["ano"] = 1984
```

Dicionários
Remoção

```
carros = {"marca": "Ford",
          "modelo": "Mustang",
          "ano": 1964}
```

Utilizando o método **pop()**, e passando a chave (*key*)

```
carros.pop("modelo")
```

Ou, utilizando o **del**, passando a chave (*key*)

```
del carros["ano"]
```

O construtor dict()

```
carros = dict(marca="Ford", modelo="Mustang", ano=1964)
# note que as chaves não são strings
# note o uso da atribuição e não dos "dois pontos"
print(carros)
```

Método	Descrição
d.fromkeys(`k`, `v`)	Retorna um dicionário a key `k` e valor `v` especificados
d.get(`k`)	Retorna o valor do elemento com a key `k` especificada
d.items()	Retorna uma lista com todos as tuplas chave-valor do dicionário
d.keys()	Retorna uma lista com todos as chaves do dicionário
d.pop(`k`)	Remove o elemento com a key `k` especificada
d.popitem()	Remove o último par chave-valor inserido no dicionário
d.setdefault(`k`, `v`)	Retorna o número de vezes que `c` aparece na lista
d.update(`k`, `v`)	Atualiza o dicionário de acordo com um par chave-valor (`k` e `v`)
d.values()	Retorna uma lista com todos os valores do dicionário
d.clear()	Limpa o dicionário
d.copy()	Faz uma cópia do dicionário e o retorna

E mais...

Python ainda permite a criação de listas que **não permite** a **alteração** de seu conteúdo

Essas estruturas são ideais para representar listas de valores constantes e para realizar operações de empacotamento e desempacotamento de valores.

E mais...

Python ainda permite a criação de listas que **não permite** a **alteração** de seu conteúdo

Essas estruturas são ideais para representar listas de valores constantes e para realizar operações de empacotamento e desempacotamento de valores.

→ são as chamadas Tuplas

Tuplas

Tuple (Tupla) é uma coleção de dados ordenada e imutável, que permite elementos duplicados.

Podem ser vistas como listas

São criadas de forma semelhantes às listas, mas utilizamos parênteses **()** em vez de colchetes.

```
tupla = ("a", "b", "c")
print(tupla)
```

Tuplas

```
# inicializando uma tupla
tup = (3, "bom", 0.5)
# outra forma de inicializar
tep = 2, [9, 1], "bum"
# retorna "o"
print(tup[1][1])
# resulta em erro! nao pode alterar uma tupla!
tep[0] = 9
# mas isto pode! a lista foi alterada!
tep[1][0] = 3
```

Tuplas

Note que não podemos alterar a tupla em si, mas, se determinado elemento **mutável** (e.g., lista) fizer parte da tupla, o elemento pode ser alterado!

Importância das "listas imutáveis" inclui:

segurança: se quisermos uma variável **constante**, e, acidentalmente, ela teve seu valor alterado, será emitido um erro (em vez de o programa continuar a executar com a variável assumindo outro valor).

eficiência: variáveis imutáveis são mais facilmente processadas pelo SO, permitindo otimizações “invisíveis” ao usuário.

O construtor tuple()

```
x = ["banana", "kiwi", "morango"]
x[1] = "uva"
y = tuple(x)

print(y)
```

Resumindo...

- **`Lista:`** inicializada com **`[]`**.
- **`Tupla:`** inicializada com **`()`**.
- **`Dicionário:`** inicializada com **`{}`**.

- Possuem uma série de operações equivalentes para acesso, checagem de tamanho, etc

Obrigado!

Alguma dúvida?