
**Universidade Federal do Piauí - UFPI**

**Sistemas de Informação**

# Programação Orientada a Objetos I- POO

## Strings, Listas, Tuplas e Dicionários

*Prof. Flávio Araújo - UFPI - Picos PI*

# Strings

```
primeiro = 'Flavio'
meio = 'Duarte'
ultimo = 'de Araujo'
espaco = ' '

completo = primeiro + espaco + meio + espaco + ultimo

print(completo)
print(len(completo))
```

# Formatação de Strings

```
import math

completo = '{a} {b} {c}.'
print(completo.format(a='Flavio', b='Duarte', c='de Araujo'))

numero = '{:0.2f}'
print(numero.format(math.pi))
```

# Iteração em Strings

```
nome = 'Flavio Henrique'

for letra in nome:
    print(letra)
```

# Métodos de String

Para cada uma delas faça um exemplo:

- split() # transforma um string em lista
- capitalize() # primeira letra maiúscula
- center() #centraliza ver ljust e rjust
- count(substring) # conta a quantidade substring
- endwith(substring) # retornar verdadeiro se o fim for a substring
- find(substring) # procura uma substring
- index() # diferente da Find gera uma exceção ValueError
- join(iterável) # iterável

  lower() # converte para minúsculo
- partition(substring) #divide a string na substring
- replace(procura, valor) # substitui a substring procura por valor
- startwith(substring) # retorna verdadeiro se começa com a substring
- stript() # remove os espaços brancos antes e depois
- swapcase() # transforma letras minúscula em maiúscula e vice-versa
- title() # transforma a letra de cada palavra em maiúscula
- upper() # transforma tudo em maiúsculo

# Principais Sequencias

- Listas []:
  - Conjunto de elementos separados por vírgula.

- Tuplas ():
  - Parecido com Listas mas depois de criado seus valores são imutáveis.

- Dicionários {}:
  - O que diferencia de uma sequencia (Listas e Tuplas) é que seus objetos são indexados.

# Listas

- São representadas por colchetes [];

- Pode ter seus valores alterados e tamanho  também;

```
frutas = ['banana', 'manga', 'abacate', 'goiaba']
print(frutas)
frutas.append('acerola')
print(len(frutas))
print(frutas[0])
print(frutas[0:3])
print(frutas[2:])
```

# Listas

.append(elemento): adiciona no fim da lista

.count(): conta quantas ocorrência de elemento dentro da lista

.sort(): ordena

.pop(): remove o último elemento

.insert(posição, elemento)

.extend(): estende a lista ocm todos os elementos do objeto iterável

.index(): retorna a posição do primeiro elemento na lista, procurando a partir do início

.remove(): remove a primeira ocorrência de um elemento

.reverse(): inverte a lista

# Exercício

Faça um programa que receba números inteiros do teclado, ao final ele deve gerar uma lista que respeite as seguintes regras:

1. Se for par adicione no início;
2. Se for ímpar adicione no fim;
3. Se for 0 adicione no meio;
4. Se for número negativo encerre o programa;
5. Mostre a lista ordenada.

Exemplo: 246000579

# Exercício

Faça um programa para sorteio de números inteiros:

1. O usuário informa os limites dos números sorteados;
2. O usuário informa quantos números serão sorteados;
3. O programa mostra os números sorteados;
4. Não pode mostrar números já sorteados.

Dica:

```
import random
x = random.randint(valor_inicial, valor_final)
```

# Map e Filter

**Map**: aplica uma função a todos os elementos de uma lista e devolve uma nova lista como resultado;

**Filter**: cria uma nova lista com os elementos que passaram no teste implementado pela função fornecida.

```
def metade(n):
    return n/2

lista = [0,1,2,3,4]

lista_metade = map(metade,lista)
for x in lista_metade:
    print(x)
```

```
def isZero(n):
    return n==0

numeros = [0,0,2,3]

zeros = filter(isZero, numeros)
for x in zeros:
    print(x)
```

# Exercício

Refaça o exercício do slide 8 utilizando Map e/ou Filter

# Função Reduce

**Reduce**: executa uma função para cada membro de uma lista, resultando em um único valor de retorno.

```
import functools as ft

def soma(x,y):
    return x+y

numeros = [1,2,3,4]
n = ft.reduce(soma,numeros)

print(n)
```

# Exercício

Faça um programa para somar os números pares de uma lista usando o filter e o reduce.

# Lambda: função anônima

```
import functools as ft

numeros = [1,2,3,4,5,6]

pares = filter(lambda x: x%2==0, numeros)
dobro = map(lambda x: x*2, numeros)
soma = ft.reduce(lambda x,y: x+y, numeros)

for n in pares:
    print(n)

for n in dobro:
    print(n)

print(soma)
```

# Exercício

Refaça os exercícios anteriores de Map, Filter e Reduce (slide 8 e 16) e use o lambda ao invés de declarar funções.

```python
import functools as ft

numeros = [1, 2, 3, 4, 6]
n = ft.reduce(lambda x,y:x+y, filter(lambda x:x%2==0, numeros))

print('n = ', n)
```

# Listas compreensivas

O que se faz com listas compreensivas se pode fazer com map e filter juntos, ela possui a seguinte forma e retorna uma lista:

Lista = [expressão for x in [lista] if condição]

```
pares = [x for x in range(0,10) if x%2==0]
print(pares)
```

# Map, filter e listas compreensivas

Dobrando o valor dos números pares usando listas compreensivas

```
pares = [x*2 for x in range(0,10) if x%2 == 0]
for x in pares:
    print(x)
```

Dobrando o valor dos números pares usando map, filter

```
pares = map(lambda x:x*2, filter(lambda x: not x%2, range(0,10)))
for x in pares:
    print(x)
```

# Exercício

Usando apenas listas compreensivas faça um código para descobrir se um número é primo.

# Tuplas

- Uma tupla é uma lista **imutável**, ou seja, uma tupla é uma sequência que não pode ser alterada depois de criada.
- Uma tupla é definida de forma parecida com uma lista com a diferença do delimitador.
- Enquanto listas utilizam colchetes como delimitadores, as tuplas usam parênteses:
- Como são imutáveis, uma vez criadas não podemos adicionar nem remover elementos de uma tupla.

```
dias = ('domingo', 'segunda', 'terca', 'quarta', 'quinta', 'sexta', 'sabado')
print(dias[1])
```

# Tuplas

- Não é possível atribuir valores aos itens individuais de uma tupla, no entanto, é possível criar tuplas que contenham objetos mutáveis, como listas.

```
lista = [3,4]
tupla = (1, 2, lista)

lista.append(4)
print(tupla)
```

# Conjuntos

- Um conjunto, diferente de uma sequência, é uma coleção **não ordenada** e que **não admite elementos duplicados**.
- Chaves ou a função set() podem ser usados para criar conjuntos.

```
frutas = {'laranja', 'banana', 'uva', 'pera', 'laranja', 'uva', 'abacate'}
frutas
{'uva', 'abacate', 'pera', 'banana', 'laranja'}
```

# Conjuntos

- Operações

```
>>> a =  set('abacate')
>>> b = set('abacaxi')
>>> a
{'a', 'e', 'c', 't', 'b'}
>>> b
{'a', 'x', 'i', 'c', 'b'}
>>> a - b                              # diferença
{'e', 't'}
>>> a | b                              # união
{'c', 'b', 'i', 't', 'x', 'e', 'a'}
>>> a & b                              # interseção
{'a', 'c', 'b'}
>>> a ^ b                              # diferença simétrica
{'i', 't', 'x', 'e'}
```

# Dicionário

- Qualquer chave de um dicionário é associada (ou mapeada) a um valor.
- Os valores podem ser qualquer tipo de dado do Python.
- Portanto, os dicionários são pares de chave-valor não ordenados.

```
>>> pessoa = {'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'}
>>> pessoa
'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'
```

- Também podemos criar dicionários utilizando a função **dict()**:

```
>>> a = dict(um=1, dois=2, três=3)
>>> a
{'três': 3, 'dois': 2, 'um': 1}
```

# Dicionário

- Não é possível acessar um elemento de um dicionário por um índice como na lista.
- Devemos acessar por sua chave:

```
>>> pessoa['nome']
'João'
>>> pessoa['idade']
25
```

# Dicionário

- Como sempre acessamos seus elementos através de chaves, o dicionário possui um método chamado *keys()* que devolve o conjunto de suas chaves:

```
>>> pessoa1.keys()
dict_keys(['nome', 'idade', 'cidade', 'pais'])
```

- Assim como um método chamado values() que retorna seus valores:

```
>>> pessoa1.values()
dict_values(['João', 25, 'São Paulo', 'Brasil'])
```

# Dicionário

.**clear**() = remove todos elementos

. **copy**() = faz uma copia / cuidado dic1 = dic2 não é copia

. **get**(chave) = retorna o valor

.**items**() = retorna a lista de tuplas

.**keys**() = retorna a lista de chaves

.**values**() = retorna a lista de valores

.**pop**(chave) = retorna o valor associado a chave e remove

.**popitem**() = retorna a tupla e remove

.**setdefault**(chave,valor) = similar ao get, mas se não contiver o elemento é inserido

.**update**(dict) = adiciona novos elementos

# Exercício

Crie um dicionário que armazena o nome e idade de uma pessoa. A chave é o cpf.

CPF:(Nome, Idade)

Em seguida, faça uma lista compreensiva para retornar todas as pessoas com a idade maior que 18.

# Número arbitrário de parâmetros (*args)

```
def teste(arg, *args):
    print('primeiro argumento normal: {}'.format(arg))
    for arg in args:
        print('outro argumento: {}'.format(arg))

teste('python', 'é', 'muito', 'legal')
```

Saída

```
primeiro argumento normal:  python
outro argumento:  é
outro argumento:  muito
outro argumento:  legal
```

# Número arbitrário de parâmetros (*args)

```
def minha_funcao(**kwargs):
    for key, value in kwargs.items():
        print('{0} = {1}'.format(key, value))
```

Saída

```
dicionario = {'nome': 'joao', 'idade': 25}
minha_funcao(**dicionario)
idade = 25
nome = joao
```

# Exercício

- Dada a lista=[12,-2,4,8,29,45,78,36,-17,2,12,8,3,3,-52] faça um programa que:
  - imprima o maior elemento
  - imprima o menor elemento
  - imprima os números pares
  - imprima o número de ocorrências do primeiro elemento da lista
  - imprima a média dos elementos
  - imprima a soma dos elementos de valor negativo

# Exercício

- Faça um programa utilizando um dict que leia dados de entrada do usuário. O usuário deve entrar com os dados de uma pessoa como nome, idade e cidade onde mora. Após isso, você deve imprimir os dados como o exemplo abaixo:

```
nome: João
idade: 20
cidade: São Paulo
```

- Utilize o exercício anterior e adicione a pessoa em uma lista. Pergunte ao usuário se ele deseja adicionar uma nova pessoa. Após adicionar dados de algumas pessoas, você deve imprimir todos os dados de cada pessoa de forma organizada.

```
Comparativo de Consumo de Combustível

Veículo 1
Nome: fusca
Km por litro: 7
Veículo 2
Nome: gol
Km por litro: 10
Veículo 3
Nome: uno
Km por litro: 12.5
Veículo 4
Nome: Vectra
Km por litro: 9
Veículo 5
Nome: Peugeout
Km por litro: 14.5

Relatório Final
 1 - fusca           -    7.0 -  142.9 litros - R$ 321.43
 2 - gol             -   10.0 -  100.0 litros - R$ 225.00
 3 - uno             -   12.5 -   80.0 litros - R$ 180.00
 4 - vectra          -    9.0 -  111.1 litros - R$ 250.00
 5 - peugeout        -   14.5 -   69.0 litros - R$ 155.17
O menor consumo é do peugeout.
```

Faça um programa que carregue uma lista com os modelos de cinco carros (exemplo de modelos: FUSCA, GOL, VECTRA etc). Carregue uma outra lista com o consumo desses carros, isto é, quantos quilômetros cada um desses carros faz com um litro de combustível. Calcule e mostre:

- O modelo do carro mais econômico;
- Quantos litros de combustível cada um dos carros cadastrados consome para percorrer uma distância de 1000 quilômetros e quanto isto custará, considerando um que a gasolina custe R$ 2,25 o litro. Abaixo segue uma tela de exemplo. O disposição das informações deve ser o mais próxima possível ao exemplo. Os dados são fictícios e podem mudar a cada execução do programa.

# Exercício

- Faça um programa que simule um lançamento de dados. Lance o dado 100 vezes e armazene os resultados em uma lista. Depois, mostre quantas vezes cada valor foi conseguido. Dica: use uma lista de contadores(1 - 6) e uma função para gerar números aleatórios, simulando os lançamentos dos dados.