

# Listas

**Guilherme Arthur de Carvalho**

Analista de sistemas

**@decarvalhogui**

# Objetivo Geral

Entender o funcionamento da estrutura de dados lista.

# Pré-requisitos

- Python 3
- VSCode

# Percurso

## **Etapa 1**

**Criação e acesso aos dados**

## **Etapa 2**

Métodos da classe list

## Etapa 1

# Criação e acesso aos dados

# Criando listas

Listas em Python podem armazenar de maneira sequencial qualquer tipo de objeto. Podemos criar listas utilizando o construtor **list**, a função `range` ou colocando valores separados por vírgula dentro de colchetes. Listas são objetos mutáveis, portanto podemos alterar seus valores após a criação.

# Exemplo

```
frutas = ["laranja", "maca", "uva"]
```

```
frutas = []
```

```
letras = list("python")
```

```
numeros = list(range(10))
```

```
carro = ["Ferrari", "F8", 4200000, 2020, 2900, "São Paulo", True]
```

# Acesso direto

A lista é uma sequência, portanto podemos acessar seus dados utilizando índices. Contamos o índice de determinada sequência a partir do zero.



# Exemplo

```
frutas = ["maçã", "laranja", "uva", "pera"]  
frutas[0] # maçã  
frutas[2] # uva
```

# Índices negativos

Sequências suportam indexação negativa. A contagem começa em -1.

# Exemplo

```
frutas = ["maçã", "laranja", "uva", "pera"]  
frutas[-1] # pera  
frutas[-3] # laranja
```

# Listas aninhadas

Listas podem armazenar todos os tipos de objetos Python, portanto podemos ter listas que armazenam outras listas. Com isso podemos criar estruturas bidimensionais (tabelas), e acessar informando os índices de linha e coluna.

# Exemplo

```
matriz = [  
    [1, "a", 2],  
    ["b", 3, 4],  
    [6, 5, "c"]  
]  
  
matriz[0]    # [1, "a", 2]  
matriz[0][0] # 1  
matriz[0][-1] # 2  
matriz[-1][-1] # "c"
```

# Fatiamento

Além de acessar elementos diretamente, podemos extrair um conjunto de valores de uma sequência. Para isso basta passar o índice inicial e/ou final para acessar o conjunto. Podemos ainda informar quantas posições o cursor deve "pular" no acesso.

# Exemplo

```
lista = ["p", "y", "t", "h", "o", "n"]  
  
lista[2:] # ["t", "h", "o", "n"]  
lista[:2] # ["p", "y"]  
lista[1:3] # ["y", "t"]  
lista[0:3:2] # ["p", "t"]  
lista[:] # ["p", "y", "t", "h", "o", "n"]  
lista[::-1] # ["n", "o", "h", "t", "y", "p"]
```

# Iterar listas

A forma mais comum para percorrer os dados de uma lista é utilizando o comando **for**.



# Exemplo

```
carros = ["gol", "celta", "palio"]  
  
for carro in carros:  
    print(carro)
```

# Função enumerate

Às vezes é necessário saber qual o índice do objeto dentro do laço **for**. Para isso podemos usar a função **enumerate**.

# Exemplo

```
carros = ["gol", "celta", "palio"]  
  
for indice, carro in enumerate(carros):  
    print(f"{indice}: {carro}")
```

# Compreensão de listas

A compreensão de lista oferece uma sintaxe mais curta quando você deseja: criar uma nova lista com base nos valores de uma lista existente (filtro) ou gerar uma nova lista aplicando alguma modificação nos elementos de uma lista existente.

# Filtro versão 1

```
numeros = [1, 30, 21, 2, 9, 65, 34]
pares = []

for numero in numeros:
    if numero % 2 == 0:
        pares.append(numero)
```

# Filtro versão 2

```
numeros = [1, 30, 21, 2, 9, 65, 34]  
pares = [numero for numero in numeros if numero % 2 == 0]
```

# Modificando valores versão 1

```
numeros = [1, 30, 21, 2, 9, 65, 34]
quadrado = []

for numero in numeros:
    quadrado.append(numero ** 2)
```

# Modificando valores versão 2

```
numeros = [1, 30, 21, 2, 9, 65, 34]  
quadrado = [numero ** 2 for numero in numeros]
```



# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

**Etapa 2**

**Métodos da classe list**

## Etapa 2

# Métodos da classe list

# [] .append

```
lista = []  
  
lista.append(1)  
lista.append("Python")  
lista.append([40, 30, 20])  
  
print(lista)  # [1, "Python", [40, 30, 20]]
```

# `[]`.clear

```
lista = [1, "Python", [40, 30, 20]]  
  
print(lista)  # [1, "Python", [40, 30, 20]]  
  
lista.clear()  
  
print(lista)  # []
```

# `[]`.copy

```
lista = [1, "Python", [40, 30, 20]]  
  
lista.copy()  
  
print(lista)  # [1, "Python", [40, 30, 20]]
```

# {}.count

```
cores = ["vermelho", "azul", "verde", "azul"]
```

```
cores.count("vermelho") # 1
```

```
cores.count("azul") # 2
```

```
cores.count("verde") # 1
```

# [] .extend

```
linguagens = ["python", "js", "c"]  
  
print(linguagens) # ["python", "js", "c"]  
  
linguagens.extend(["java", "csharp"])  
  
print(linguagens) # ["python", "js", "c", "java", "csharp"]
```

# [] .index

```
linguagens = ["python", "js", "c", "java", "csharp"]
```

```
linguagens.index("java") # 3
```

```
linguagens.index("python") # 0
```



# [] .pop

```
linguagens = ["python", "js", "c", "java", "csharp"]  
  
linguagens.pop() # csharp  
linguagens.pop() # java  
linguagens.pop() # c  
linguagens.pop(0) # python
```

# [] .remove

```
linguagens = ["python", "js", "c", "java", "csharp"]  
  
linguagens.remove("c")  
  
print(linguagens) # ["python", "js", "java", "csharp"]
```

# [] .reverse

```
linguagens = ["python", "js", "c", "java", "csharp"]  
  
linguagens.reverse()  
  
print(linguagens) # ["csharp", "java", "c", "js", "python"]
```

# list.sort

```
linguagens = ["python", "js", "c", "java", "csharp"]
linguagens.sort() # ["c", "csharp", "java", "js", "python"]

linguagens = ["python", "js", "c", "java", "csharp"]
linguagens.sort(reverse=True) # ["python", "js", "java", "csharp", "c"]

linguagens = ["python", "js", "c", "java", "csharp"]
linguagens.sort(key=lambda x: len(x)) # ["c", "js", "java", "python", "csharp"]

linguagens = ["python", "js", "c", "java", "csharp"]
linguagens.sort(key=lambda x: len(x), reverse=True) # ["python", "csharp", "java", "js", "c"]
```

# len

```
linguagens = ["python", "js", "c", "java", "csharp"]  
  
len(linguagens) # 5
```

# sorted

```
linguagens = ["python", "js", "c", "java", "csharp"]  
  
sorted(linguagens, key=lambda x: len(x)) # ["c", "js", "java", "python",  
"csharp"]  
  
sorted(linguagens, key=lambda x: len(x), reverse=True) # ["python", "csharp",  
"java", "js", "c"]
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

~~Etapa 2~~

~~Métodos da classe list~~

# Links Úteis

- <https://github.com/digitalinnovationone/trilha-python-dio>



# Dúvidas?

- > Fórum/Artigos
- > Comunidade Online (Discord)



# Tuplas

**Guilherme Arthur de Carvalho**

Analista de sistemas

**@decarvalhogui**

# Objetivo Geral

Entender o funcionamento da estrutura de dados tupla.

# Pré-requisitos

- Python 3
- VSCode

# Percurso

## **Etapa 1**

**Criação e acesso aos dados**

## **Etapa 2**

**Métodos da classe tuple**

## Etapa 1

# Criação e acesso aos dados

# Criando tuplas

Tuplas são estruturas de dados muito parecidas com as listas, a principal diferença é que tuplas são imutáveis enquanto listas são mutáveis. Podemos criar tuplas através da classe **tuple**, ou colocando valores separados por vírgula de parenteses.

# Exemplo

```
frutas = ("laranja", "pera", "uva",)
```

```
letras = tuple("python")
```

```
numeros = tuple([1, 2, 3, 4])
```

```
pais = ("Brasil",)
```



# Acesso direto

A tupla é uma sequência, portanto podemos acessar seus dados utilizando índices. Contamos o índice de determinada sequência a partir do zero.

# Exemplo

```
frutas = ("maçã", "laranja", "uva", "pera",)  
frutas[0] # maçã  
frutas[2] # uva
```

# Índices negativos

Sequências suportam indexação negativa. A contagem começa em -1.

# Exemplo

```
frutas = ("maçã", "laranja", "uva", "pera",)  
frutas[-1] # pera  
frutas[-3] # laranja
```

# Tuplas aninhadas

Tuplas podem armazenar todos os tipos de objetos Python, portanto podemos ter tuplas que armazenam outras tuplas. Com isso podemos criar estruturas bidimensionais (tabelas), e acessar informando os índices de linha e coluna.

# Exemplo

```
matriz = (  
    (1, "a", 2),  
    ("b", 3, 4),  
    (6, 5, "c"),  
)  
  
matriz[0]    # (1, "a", 2)  
matriz[0][0] # 1  
matriz[0][-1] # 2  
matriz[-1][-1] # "c"
```

# Fatiamento

Além de acessar elementos diretamente, podemos extrair um conjunto de valores de uma sequência. Para isso basta passar o índice inicial e/ou final para acessar o conjunto. Podemos ainda informar quantas posições o cursor deve "pular" no acesso.

# Exemplo

```
tupla = ("p", "y", "t", "h", "o", "n",)  
  
tupla[2:] # ("t", "h", "o", "n")  
tupla[:2] # ("p", "y")  
tupla[1:3] # ("y", "t")  
tupla[0:3:2] # ("p", "t")  
tupla[:] # ("p", "y", "t", "h", "o", "n")  
tupla[::-1] # ("n", "o", "h", "t", "y", "p")
```



# Iterar tuplas

A forma mais comum para percorrer os dados de uma tupla é utilizando o comando **for**.

# Exemplo

```
carros = ("gol", "celta", "palio",)  
  
for carro in carros:  
    print(carro)
```

# Função enumerate

Às vezes é necessário saber qual o índice do objeto dentro do laço **for**. Para isso podemos usar a função **enumerate**.

# Exemplo

```
carros = ("gol", "celta", "palio",)

for indice, carro in enumerate(carros):
    print(f"{indice}: {carro}")
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

**Etapa 2**

**Métodos da classe tuple**

## Etapa 2

# Métodos da classe tuple

# ()count

```
cores = ("vermelho", "azul", "verde", "azul",)
```

```
cores.count("vermelho") # 1
```

```
cores.count("azul") # 2
```

```
cores.count("verde") # 1
```

# ()index

```
linguagens = ("python", "js", "c", "java", "csharp",)
```

```
linguagens.index("java") # 3
```

```
linguagens.index("python") # 0
```



# len

```
linguagens = ("python", "js", "c", "java", "csharp",)  
len(linguagens) # 5
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

~~Etapa 2~~

~~Métodos da classe tuple~~

# Links Úteis

- <https://github.com/digitalinnovationone/trilha-python-dio>

# Dúvidas?

- > Fórum/Artigos
- > Comunidade Online (Discord)



# Conjuntos

**Guilherme Arthur de Carvalho**

Analista de sistemas

**@decarvalhogui**

# Objetivo Geral

Entender o funcionamento da estrutura de dados set.

# Pré-requisitos

- Python 3
- VSCode

# Percurso

## Etapa 1

**Como criar conjuntos**

## Etapa 2

**Métodos da classe set**



## Etapa 1

# Como criar conjuntos

# Criando sets

Um set é uma coleção que não possui objetos repetidos, usamos sets para representar conjuntos matemáticos ou eliminar itens duplicados de um iterável.

# Exemplo

```
set([1, 2, 3, 1, 3, 4]) # {1, 2, 3, 4}
```

```
set("abacaxi") # {"b", "a", "c", "x", "i"}
```

```
set(("palio", "gol", "celta", "palio")) # {"gol", "celta", "palio"}
```

# Acessando os dados

Conjuntos em Python não suportam indexação e nem fatiamento, caso queira acessar os seus valores é necessário converter o conjunto para lista.

# Exemplo

```
numeros = {1, 2, 3, 2}
```

```
numeros = list(numeros)
```

```
numeros[0]
```

# Iterar conjuntos

A forma mais comum para percorrer os dados de um conjunto é utilizando o comando **for**.

# Exemplo

```
carros = {"gol", "celta", "palio"}  
  
for carro in carros:  
    print(carro)
```

# Função enumerate

Às vezes é necessário saber qual o índice do objeto dentro do laço **for**. Para isso podemos usar a função **enumerate**.



# Exemplo

```
carros = {"gol", "celta", "palio"}

for indice, carro in enumerate(carros):
    print(f"{indice}: {carro}")
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

**Etapa 2**

**Métodos da classe set**

## Etapa 2

# Métodos da classe set

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

~~Etapa 2~~

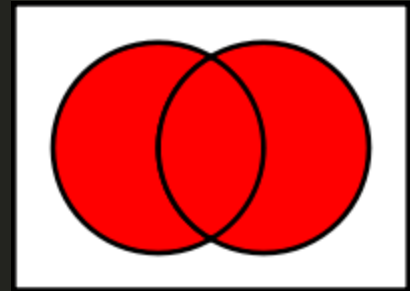
~~Métodos da classe set~~

# {}.union

```
conjunto_a = {1, 2}
```

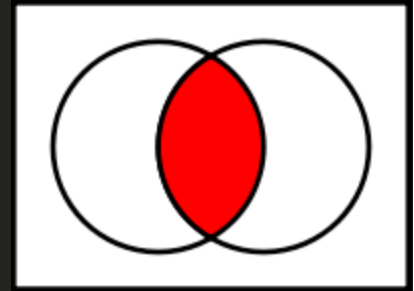
```
conjunto_b = {3, 4}
```

```
conjunto_a.union(conjunto_b)  # {1, 2, 3, 4}
```



# {}.intersection

```
conjunto_a = {1, 2, 3}  
conjunto_b = {2, 3, 4}  
  
conjunto_a.intersection(conjunto_b)  # {2, 3}
```



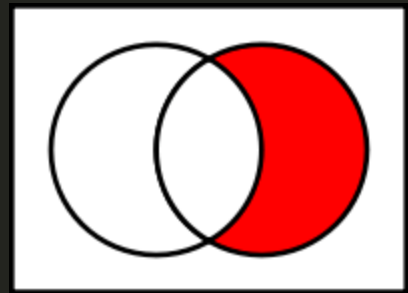
# `{}.difference`

```
conjunto_a = {1, 2, 3}
```

```
conjunto_b = {2, 3, 4}
```

```
conjunto_a.difference(conjunto_b) # {1}
```

```
conjunto_b.difference(conjunto_a) # {4}
```

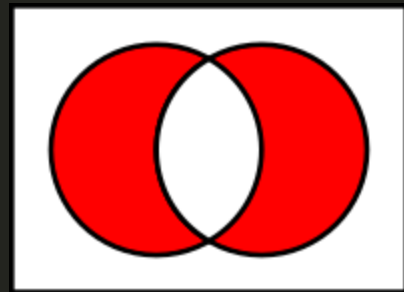


# `{}.symmetric_difference`

```
conjunto_a = {1, 2, 3}
```

```
conjunto_b = {2, 3, 4}
```

```
conjunto_a.symmetric_difference(conjunto_b)  # {1, 4}
```

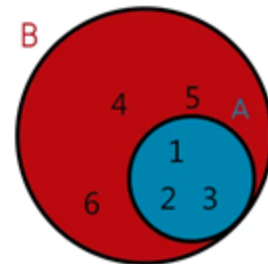




# `{}.issubset`

```
conjunto_a = {1, 2, 3}
conjunto_b = {4, 1, 2, 5, 6, 3}

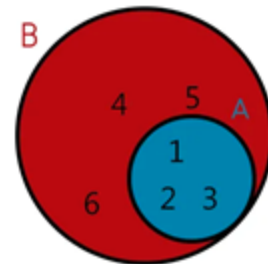
conjunto_a.issubset(conjunto_b)  # True
conjunto_b.issubset(conjunto_a)  # False
```



# {}.issuperset

```
conjunto_a = {1, 2, 3}
conjunto_b = {4, 1, 2, 5, 6, 3}

conjunto_a.issuperset(conjunto_b) # False
conjunto_b.issuperset(conjunto_a) # True
```



# {}.isdisjoint

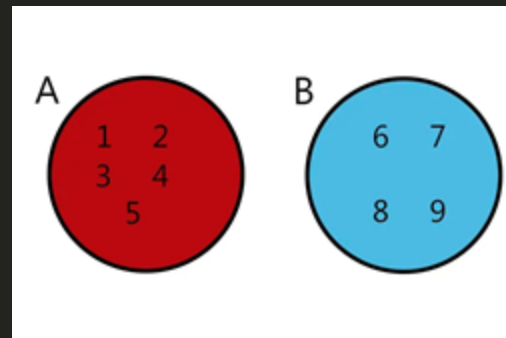
```
conjunto_a = {1, 2, 3, 4, 5}
```

```
conjunto_b = {6, 7, 8, 9}
```

```
conjunto_c = {1, 0}
```

```
conjunto_a.isdisjoint(conjunto_b) # True
```

```
conjunto_a.isdisjoint(conjunto_c) # False
```



# `{}.add`

```
sorteio = {1, 23}
```

```
sorteio.add(25) # {1, 23, 25}
```

```
sorteio.add(42) # {1, 23, 25, 42}
```

```
sorteio.add(25) # {1, 23, 25, 42}
```

# `{}.clear`

```
sorteio = {1, 23}
```

```
sorteio # {1,23}
```

```
sorteio.clear()
```

```
sorteio # {}
```

# `{}.copy`

```
sorteio = {1, 23}
```

```
sorteio # {1, 23}
```

```
sorteio.copy()
```

```
sorteio # {1, 23}
```

# `{}.discard`

```
numeros = {1, 2, 3, 1, 2, 4, 5, 5, 6, 7, 8, 9, 0}
```

```
numeros # {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
```

```
numeros.discard(1)
```

```
numeros.discard(45)
```

```
numeros # {2, 3, 4, 5, 6, 7, 8, 9, 0}
```

# `{}.pop`

```
numeros = {1, 2, 3, 1, 2, 4, 5, 5, 6, 7, 8, 9, 0}
```

```
numeros # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
numeros.pop() # 0
```

```
numeros.pop() # 1
```

```
numeros # {2, 3, 4, 5, 6, 7, 8, 9}
```



# `{}.remove`

```
numeros = {1, 2, 3, 1, 2, 4, 5, 5, 6, 7, 8, 9, 0}
```

```
numeros  # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
numeros.remove(0)  # 0
```

```
numeros  # {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

# len

```
numeros = {1, 2, 3, 1, 2, 4, 5, 5, 6, 7, 8, 9, 0}
```

```
len(numeros)  # 10
```

# in

```
numeros = {1, 2, 3, 1, 2, 4, 5, 5, 6, 7, 8, 9, 0}
```

```
1 in numeros # True
```

```
10 in numeros # False
```

# Links Úteis

- <https://github.com/digitalinnovationone/trilha-python-dio>

# Dúvidas?

- > Fórum/Artigos
- > Comunidade Online (Discord)



# Dicionários

**Guilherme Arthur de Carvalho**

Analista de sistemas

**@decarvalhogui**

# Objetivo Geral

Entender o funcionamento da estrutura de dados dicionário.

# Pré-requisitos

- Python 3
- VSCode



# Percurso

## **Etapa 1**

**Criação e acesso aos dados**

## **Etapa 2**

Métodos da classe dict

## Etapa 1

# Criação e acesso aos dados

# Criando dicionários

Um dicionário é um conjunto não-ordenado de pares chave:valor, onde as chaves são únicas em uma dada instância do dicionário. Dicionários são delimitados por chaves: {}, e contém uma lista de pares chave:valor separada por vírgulas.

# Exemplo

```
peessoa = {"nome": "Guilherme", "idade": 28}
```

```
peessoa = dict(nome="Guilherme", idade=28)
```

```
peessoa["telefone"] = "3333-1234"  # {"nome": "Guilherme", "idade": 28,  
"telefone": "3333-1234"}
```

# Acesso aos dados

Os dados são acessados e modificados através da chave.

# Exemplo

```
dados = {"nome": "Guilherme", "idade": 28, "telefone": "3333-1234"}
```

```
dados["nome"] # "Guilherme"
```

```
dados["idade"] # 28
```

```
dados["telefone"] # "3333-1234"
```

```
dados["nome"] = "Maria"
```

```
dados["idade"] = 18
```

```
dados["telefone"] = "9988-1781"
```

```
dados # {"nome": "Maria", "idade": 18, "telefone": "9988-1781"}
```

# Dicionários aninhados

Dicionários podem armazenar qualquer tipo de objeto Python como valor, desde que a chave para esse valor seja um objeto imutável como (strings e números).

# Exemplo

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"},  
    "giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3443-2121"},  
    "chappie@gmail.com": {"nome": "Chappie", "telefone": "3344-9871"},  
    "melaine@gmail.com": {"nome": "Melaine", "telefone": "3333-7766"},  
}  
  
contatos["giovanna@gmail.com"]["telefone"] # "3443-2121"
```



# Iterar dicionários

A forma mais comum para percorrer os dados de um dicionário é utilizando o comando **for**.

# Exemplo

```
for chave in contatos:  
    print(chave, contatos[chave])
```

```
for chave, valor in contatos.items():  
    print(chave, valor)
```

```
# guilherme@gmail.com {'nome': 'Guilherme', 'telefone': '3333-2221'}  
# giovanna@gmail.com {'nome': 'Giovanna', 'telefone': '3443-2121'}  
# chappie@gmail.com {'nome': 'Chappie', 'telefone': '3344-9871'}  
# melaine@gmail.com {'nome': 'Melaine', 'telefone': '3333-7766'}
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

**Etapa 2**

**Métodos da classe dict**

## Etapa 2

# Métodos da classe dict

# `{}.clear`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"},  
    "giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3443-2121"},  
    "chappie@gmail.com": {"nome": "Chappie", "telefone": "3344-9871"},  
    "melaine@gmail.com": {"nome": "Melaine", "telefone": "3333-7766"},  
}  
  
contatos.clear()  
contatos  # {}
```

# `{}.copy`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
copia = contatos.copy()  
copia["guilherme@gmail.com"] = {"nome": "Gui"}  
  
contatos["guilherme@gmail.com"] # {"nome": "Guilherme", "telefone": "3333-  
2221"}  
copia["guilherme@gmail.com"]   # {"nome": "Gui"}
```

# `{}.fromkeys`

```
dict.fromkeys(["nome", "telefono"]) # {"nome": None, "telefono": None}
```

```
dict.fromkeys(["nome", "telefono"], "vazio") # {"nome": "vazio", "telefono":  
"vazio"}
```

# `{}.get`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos["chave"]  # KeyError  
  
contatos.get("chave")  # None  
contatos.get("chave", {})  # {}  
contatos.get("guilherme@gmail.com", {})  # {"guilherme@gmail.com": {"nome":  
"Guilherme", "telefone": "3333-2221"}}
```



# `{}.items`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos.items() # dict_items([('guilherme@gmail.com', {'nome': 'Guilherme',  
'telefone': '3333-2221'})])
```

# `{}.keys`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos.keys() # dict_keys(['guilherme@gmail.com'])
```

# `}`.pop

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos.pop("guilherme@gmail.com") # {'nome': 'Guilherme', 'telefone': '3333-  
2221'}  
contatos.pop("guilherme@gmail.com", {}) # {}
```

# `{}.popitem`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos.popitem() # ('guilherme@gmail.com', {'nome': 'Guilherme', 'telefone':  
'3333-2221'})  
contatos.popitem() # KeyError
```

# `{}.setdefault`

```
contato = {'nome': 'Guilherme', 'telefone': '3333-2221'}

contato.setdefault("nome", "Giovanna") # "Guilherme"
contato # {'nome': 'Guilherme', 'telefone': '3333-2221'}

contato.setdefault("idade", 28) # 28
contato # {'nome': 'Guilherme', 'telefone': '3333-2221', 'idade': 28}
```

# `{}.update`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"}  
}  
  
contatos.update({"guilherme@gmail.com": {"nome": "Gui"}})  
contatos  # {'guilherme@gmail.com': {'nome': 'Gui'}}  
  
contatos.update({"giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3322-8181"}})  
contatos  # {'guilherme@gmail.com': {'nome': 'Gui'}, 'giovanna@gmail.com': {'nome': 'Giovanna', 'telefone': '3322-8181'}}
```

# `{}.values`

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"},  
    "giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3443-2121"},  
    "chappie@gmail.com": {"nome": "Chappie", "telefone": "3344-9871"},  
    "melaine@gmail.com": {"nome": "Melaine", "telefone": "3333-7766"},  
}
```

```
contatos.values() # dict_values([{'nome': 'Guilherme', 'telefone': '3333-  
2221'}, {'nome': 'Giovanna', 'telefone': '3443-2121'}, {'nome': 'Chappie',  
'telefone': '3344-9871'}, {'nome': 'Melaine', 'telefone': '3333-7766'}])
```

in

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"},  
    "giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3443-2121"},  
    "chappie@gmail.com": {"nome": "Chappie", "telefone": "3344-9871"},  
    "melaine@gmail.com": {"nome": "Melaine", "telefone": "3333-7766"},  
}  
  
"guilherme@gmail.com" in contatos # True  
"megui@gmail.com" in contatos # False  
"idade" in contatos["guilherme@gmail.com"] # False  
"telefone" in contatos["giovanna@gmail.com"] # True
```



# del

```
contatos = {  
    "guilherme@gmail.com": {"nome": "Guilherme", "telefone": "3333-2221"},  
    "giovanna@gmail.com": {"nome": "Giovanna", "telefone": "3443-2121"},  
    "chappie@gmail.com": {"nome": "Chappie", "telefone": "3344-9871"},  
    "melaine@gmail.com": {"nome": "Melaine", "telefone": "3333-7766"},  
}  
  
del contatos["guilherme@gmail.com"]["telefone"]  
del contatos["chappie@gmail.com"]  
  
contatos # {'guilherme@gmail.com': {'nome': 'Guilherme'}, 'giovanna@gmail.com':  
{'nome': 'Giovanna', 'telefone': '3443-2121'}, 'melaine@gmail.com': {'nome':  
'Melaine', 'telefone': '3333-7766'}}
```

# Percurso

~~Etapa 1~~

~~Criação e acesso aos dados~~

~~Etapa 2~~

~~Métodos da classe tuple~~

# Links Úteis

- <https://github.com/digitalinnovationone/trilha-python-dio>

# Dúvidas?

- > Fórum/Artigos
- > Comunidade Online (Discord)



# Funções

**Guilherme Arthur de Carvalho**

Analista de sistemas

**@decarvalhogui**

# Objetivo Geral

Entender como funcionam as funções em Python.

# Pré-requisitos

- Python 3
- VSCode

# Percurso

## **Etapa 1**

**Estudo aprofundado sobre funções**



## Etapa 1

# Estudo aprofundado sobre funções

# O que são funções?

Função é um bloco de código identificado por um nome e pode receber uma lista de parâmetros, esses parâmetros podem ou não ter valores padrões. Usar funções torna o código mais legível e possibilita o reaproveitamento de código. Programar baseado em funções, é o mesmo que dizer que estamos programando de maneira estruturada.

# Exemplo

```
def exibir_mensagem():  
    print("Olá mundo!")  
  
def exibir_mensagem_2(nome):  
    print(f"Seja bem vindo {nome}!")  
  
def exibir_mensagem_3(nome="Anônimo"):  
    print(f"Seja bem vindo {nome}!")  
  
exibir_mensagem()  
exibir_mensagem_2(nome="Guilherme")  
exibir_mensagem_3()  
exibir_mensagem_3(nome="Chappie")
```

# Retornando valores

Para retornar um valor, utilizamos a palavra reservada **return**. Toda função Python retorna **None** por padrão. Diferente de outras linguagens de programação, em Python uma função pode retornar mais de um valor.

# Exemplo

```
def calcular_total(numeros):  
    return sum(numeros)  
  
def retorna_antecessor_e_sucessor(numero):  
    antecessor = numero - 1  
    sucessor = numero + 1  
  
    return antecessor, sucessor  
  
calcular_total([10, 20, 34]) # 64  
retorna_antecessor_e_sucessor(10) # (9, 11)
```

# Argumentos nomeados

Funções também podem ser chamadas usando argumentos nomeados da forma chave=valor.

# Exemplo

```
def salvar_carro(marca, modelo, ano, placa):  
    # salva carro no banco de dados...  
    print(f"Carro inserido com sucesso! {marca}/{modelo}/{ano}/{placa}")  
  
salvar_carro("Fiat", "Palio", 1999, "ABC-1234")  
salvar_carro(marca="Fiat", modelo="Palio", ano=1999, placa="ABC-1234")  
salvar_carro(**{"marca": "Fiat", "modelo": "Palio", "ano": 1999, "placa": "ABC-  
1234"})  
  
# Carro inserido com sucesso! Fiat/Palio/1999/ABC-1234
```

# Args e kwargs

Podemos combinar parâmetros obrigatórios com args e kwargs. Quando esses são definidos (\*args e \*\*kwargs), o método recebe os valores como tupla e dicionário respectivamente.



# Exemplo

```
def exhibir_poema(data_extenso, *args, **kwargs):
    texto = "\n".join(args)
    meta_dados = "\n".join([f"{chave.title()}: {valor}" for chave, valor in
kwargs.items()])
    mensagem = f"{data_extenso}\n\n{texto}\n\n{meta_dados}"
    print(mensagem)

exibir_poema("Zen of Python", "Beautiful is better than ugly.", autor="Tim
Peters", ano=1999)
```

# Parâmetros especiais

Por padrão, argumentos podem ser passados para uma função Python tanto por posição quanto explicitamente pelo nome. Para uma melhor legibilidade e desempenho, faz sentido restringir a maneira pelo qual argumentos possam ser passados, assim um desenvolvedor precisa apenas olhar para a definição da função para determinar se os itens são passados **por posição, por posição e nome, ou por nome.**

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):  
    -----  
    |           |           |  
    |           | Positional or keyword |  
    |           |           |  
    |           | - Keyword only  
    |  
    -- Positional only
```

# Positional only

```
def criar_carro(modelo, ano, placa, /, marca, motor, combustivel):  
    print(modelo, ano, placa, marca, motor, combustivel)  
  
criar_carro("Palio", 1999, "ABC-1234", marca="Fiat", motor="1.0",  
combustivel="Gasolina") # válido  
  
criar_carro(modelo="Palio", ano=1999, placa="ABC-1234", marca="Fiat",  
motor="1.0", combustivel="Gasolina") # inválido
```

# Keyword only

```
def criar_carro(*, modelo, ano, placa, marca, motor, combustivel):  
    print(modelo, ano, placa, marca, motor, combustivel)  
  
criar_carro(modelo="Palio", ano=1999, placa="ABC-1234", marca="Fiat",  
motor="1.0", combustivel="Gasolina") # válido  
  
criar_carro("Palio", 1999, "ABC-1234", marca="Fiat", motor="1.0",  
combustivel="Gasolina") # inválido
```

# Keyword and positional only

```
def criar_carro(modelo, ano, placa, /, *, marca, motor, combustivel):  
    print(modelo, ano, placa, marca, motor, combustivel)
```

```
criar_carro("Palio", 1999, "ABC-1234", marca="Fiat", motor="1.0",  
combustivel="Gasolina") # válido
```

```
criar_carro(modelo="Palio", ano=1999, placa="ABC-1234", marca="Fiat",  
motor="1.0", combustivel="Gasolina") # inválido
```

# Objetos de primeira classe

Em Python tudo é objeto, dessa forma **funções também são objetos** o que as tornam objetos de primeira classe. Com isso podemos **atribuir funções a variáveis, passá-las como parâmetro para funções, usá-las como valores em estruturas de dados** (listas, tuplas, dicionários, etc) e usar como valor de retorno para uma função (closures).

# Exemplo

```
def somar(a, b):  
    return a + b  
  
def exibir_resultado(a, b, funcao):  
    resultado = funcao(a, b)  
    print(f"O resultado da operação {a} + {b} = {resultado}")  
  
exibir_resultado(10, 10, somar)  # O resultado da operação 10 + 10 = 20
```



# Escopo local e escopo global

Python trabalha com escopo local e global, dentro do bloco da função o escopo é local. Portanto alterações ali feitas em objetos imutáveis serão perdidas quando o método terminar de ser executado. Para usar objetos globais utilizamos a palavra-chave **global**, que informa ao interpretador que a variável que está sendo manipulada no escopo local é global. Essa **NÃO** é uma boa prática e deve ser evitada.

# Exemplo

```
salario = 2000

def salario_bonus(bonus):
    global salario
    salario += bonus
    return salario

salario_bonus(500)  # 2500
```

# Percurso

~~Etapa 1~~

~~Estudo aprofundado sobre funções~~

# Links Úteis

- <https://github.com/digitalinnovationone/trilha-python-dio>

# Dúvidas?

- > Fórum/Artigos
- > Comunidade Online (Discord)

