Dicionários e Tuplas em linguagem Python,

TRABALHO DE PESQUISA E COMPOSIÇÃO DE ARTIGO

Python

UFRA – UNIVERSIDADE FEDERAL RURAL DA AMAZONIA

Bacharelado em Sistema de informação

Docente Instrutor: Edivar Oliveira

Discente: Maria Cristina Sousa

Dicionário e Tupla no Python:

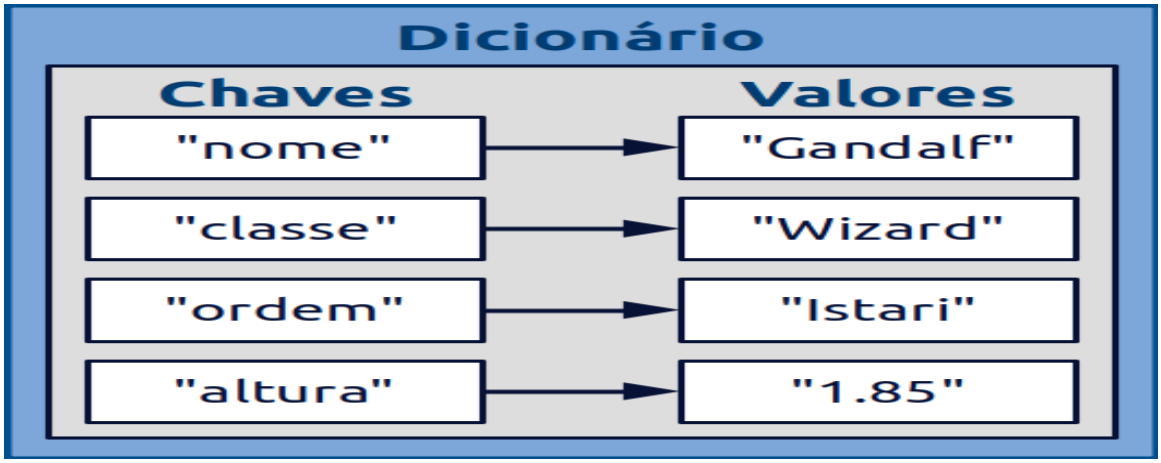
Introdução:

Python é uma linguagem de programação interpretada, de alto nível, que pode ser aplicada em diversas soluções. Famosa por sua sintaxe amigável, ela vem ganhando uma expressiva popularidade nos últimos anos. Foi criada por Guido Van Rossum no período de 1985-1990, seu código fonte está disponível sob o GNU General Public License (GPL). Python conta com uma tipagem dinâmica e um sistema de gerenciamento de memória automático, sendo capaz de suportar múltiplos paradigmas de programação.

Python é uma linguagem poderosa e de fácil aprendizado. Possui estruturas de dados eficientes e de alto nível e uma simples e efetiva abordagem em relação à programação orientada a objetos. A sintaxe elegante de Python e sua tipagem dinâmica, juntamente com sua natureza interpretada, faz do Python uma linguagem ideal para *scripting* e desenvolvimento rápido de aplicações em muitas áreas e para diversas plataformas.

O interpretador Python e a extensiva standard library estão disponíveis gratuitamente na forma *source* ou *binary* para todas as maiores plataformas através do site oficial Python.

Dicionários:



Os dicionários são encontrados em outros linguagens de programação como "memórias associativas" ou "matrizes associativas". Ao contrário das sequências, que são indexadas por um intervalo de números, os dicionários são indexados por chaves, que podem ser de qualquer tipo imutável.

Eles são uma coleção ***não-ordenada*** de pares de ***chave*** & ***valor***, são normalmente usados quando temos uma grande quantidade de dados**.**

Dicionários são definidos entre chaves ***{}*** onde cada item é um par na forma de chave:valor, separados ***por*** ,, sendo a chave e o valor podendo ser de qualquer tipo.

Chaves:

- Devem ser únicas
- Tipo imutável ***(int, float, string, tuple, bool)*** - Na verdade precisa de um objeto que seja **hashable,** mas imagine como imutável uma vez que todos os tipos imutáveis são **hashable** - Tenha cuidado com o tipo *float* como chave
- Nenhuma ordem para chaves ou valores

Valores:

- Qualquer tipo ***(imutável e mutável)***
- Pode ser ***duplicado***
- Valores de dicionários podem ser listas, até mesmo outros dicionários

Compreendermos melhor o funcionamento dos dicionários

```
album = {'nome':'A Night at the Opera','artista':'Blind Guardian','lançamento':2002}

print(type(album)) # <class 'dict'>

print(album) # {'nome': 'A Night at the Opera', 'artista': 'Blind Guardian', 'lançamento': 2002}

print(album['nome']) # A Night at the Opera

print(album['artista']) # Blind Guardian

print(album['lançamento']) # 2002
```

Acessando e Manipulando Dados

Para acessarmos os itens do dicionário, precisamos nos referir a sua chave, por exemplo:

```
print(elemento["nome"]) # Ouro
```

Utilizar o método ***get()*** para acessarmos:

```
print(elemento.get("nome")) # Ouro
```

Alterar os valores de um dicionário:

```
elemento["nome"] = "Prata"

elemento["símbolo"] = "Ag"

elemento["número atômico"] = 47

print(elemento) # {'nome': 'Prata', 'símbolo': 'Ag', 'número atômico': 47}
```

Adicionando Itens ao Dicionário

Para adicionarmos um novo item em nosso dicionário usamos uma nova chave e atribuímos um valor a ela

```
personagem['altura'] = 1.85

print(personagem)  # {'nome': 'Gandalf', 'classe': 'Wizard', 'Ordem': 'Istari', 'altura': 1.85}
```

Removendo Itens do Dicionário

É possível também, de várias formas, remover os itens de um dicionário.

Usando o método ***pop(),*** nos será retornado o item removido:

```
personagem.pop('altura') # 1.85

print(personagem)  # {'nome': 'Gandalf', 'classe': 'Wizard', 'Ordem': 'Istari'}
```

Acessando Itens de uma Tupla

O acesso aos itens de uma tupla é idêntico ao de uma lista:

```
print(linguagens[0:2]) # ('Python', 'Ruby')

print (linguagens [-1]) # Haskell

print (linguagens [:-2]) # ('Python', 'Ruby', 'Javascript')

print (linguagens [:]) # ('Python', 'Ruby', 'Javascript', 'Perl', 'Haskell')
```

Construtor dict ()

Através do construtor ***dict ()*** também é possível criarmos dicionários:

```
pessoa = dict(nome="Jesus", idade=33)

print(pessoa) # {'nome': 'Jesus', 'idade': 33}
```

O Uso do Método **update ()**:

```
pessoa.update({'nome':'Immanuel'})

print(pessoa) # {'nome': 'Immanuel', 'idade': 33}
```

Dicionários Aninhados

Os dicionários são uma estrutura de dados muito flexível e nos permitem até mesmo guardar dicionários dentro de um dicionário, de forma que possamos acessá-los através de uma chave.

```
jogos = {
'1':{'nome':'castlevania','genero':'aventura'},
'2': {'nome':'super mario','genero':'aventura'},
'3': {'nome':'world of warcraft','genero':'MMORPG'}
}
print (jogos ['2']) # {'nome': 'super mario', 'genero': 'aventura'}
print (jogos.get ('1')) # {'nome': 'castlevania', 'genero': 'aventura'}
print (jogos ['3'] ['nome']) # world of warcraft
```

Tuplas

Tuplas são uma sequência de ***objetos imutáveis***, em outras palavras, uma vez criadas, tuplas não podem ser modificadas, normalmente são usadas para guardar ***dados protegidos.***

As tuplas são escritas entre parênteses ().

Uma tupla em Python é semelhante a uma lista. A diferença entre os dois é que não podemos alterar os elementos de uma tupla depois de atribuída, enquanto podemos alterar os elementos de uma lista.

Criando uma Tupla

Podemos definir uma tupla da seguinte maneira:

```
linguagens = ("Python","Ruby","Javascript","Perl","Haskell")
print(linguagens) # ('Python', 'Ruby', 'Javascript', 'Perl', 'Haskell')
```

Método *type ()* podemos confirmar que se trata de uma *'tuple'*

```
type(linguagens) # <class 'tuple'>
```

Imprimindo Elementos de uma Tupla

As tuplas são úteis para representar o que outras linguagens costumam chamar de registros, algumas informações relacionadas que pertencem entre si, como o registro de

um **estudante**. Não há descrição do que significa cada um desses campos, mas podemos intuir. Uma tupla nos permite "agrupar" informações relacionadas e usá-las em uma única estrutura.

```
estudante = ('Miguel', 29, 1990, 'Brasil')
```

Agora podemos utilizar um ***for loop*** para imprimir todos os elementos da tupla ***estudante:***

```
for e in estudante:  # Percorre os valores da tupla estudante
    print(e) # Imprime os valores
```

Checando por Valores em uma Tupla

Assim como nas listas, também podemos checar se determinado item está presente em uma tupla:

```
print('Gabriel' in estudante) # False
```

```
print(1990 in estudante) # True
```

Verificando o Tamanho de uma Tupla

Para obtermos o número de itens em uma tupla, podemos usar o método ***len():***

```
print(len(estudante)) # 4
```

Deletando a Tupla

Embora seja impossível remover itens da tupla, é possível deletá-la por completo com o uso da palavra-chave ***del.*** Observe que se tentarmos imprimir ela, ocorrerá um erro do tipo ***NameError***, uma vez que a tupla não existe mais.

```
del estudante
print(estudante)
# NameError: name 'estudante' is not defined
```

Construtor tuple ()

Somos capazes também de construir uma tupla com o uso do construtor ***tuple ():***

Diferença entre listas e dicionários em Python

```
numeros = tuple (x for x in range(1,20,3))
print(numeros) # (1, 4, 7, 10, 13, 16, 19)
```

Método count()

O método ***count ()*** nos retorna a quantidade de vezes que determinado valor ocorre em uma tupla

```
print (numeros. Count (7)) # 1
```

Método index ()

O método ***index ()*** nos permite buscar por um determinado elemento e nos retorna o índice dele:

```
numeros. Index (4) # 1
```

Criando uma Função que Retorna uma Tupla

Funções normalmente retornam apenas um ***único valor***, porém ao tornarmos esse valor uma ***Tupla***, nós podemos efetivamente agrupar a quantidade de valores que desejarmos e retorná-los juntos. Esta funcionalidade pode nos ser muito útil em casos que queiramos por exemplo encontrar a ***média*** e o ***desvio padrão*** ou talvez obter o ano, mês e dia.

Vejamos um exemplo para ilustrar esta ideia, em que vamos definir uma função de nome **f** que irá calcular a ***circunferência*** e a ***área*** de um círculo de raio **r** (o raio será informado via argumento por nós):

```
import math
def f(r):
    """ Retorna (circunferência, área) de um círculo de raio r """
    c = 2 * math.pi * r
    a = math.pi * r * r
    return (c, a)
print (f (5)) # (31.41592653589793, 78.53981633974483)
print (f (8)) # (50.26548245743669, 201.06192982974676)
```

Tuplas vs Listas

Por que devemos usar uma Tupla ao invés de uma Lista?

- A execução do programa é mais rápida quando manipulamos uma tupla do que uma equivalente lista.
- As vezes desejamos que os dados não sejam modificados, se determinados que valores em uma coleção devem ser constantes no programa, utilizar uma Tupla nos protege contra acidentes de modificação.

Exemplo de Lista:

```
numeros_primos = [2, 3, 5, 7, 11, 13, 17]
```

Métodos das Listas e Tuplas

Podemos usar o método ***dir ()*** para visualizarmos todos os atributos e métodos disponíveis nas listas e tuplas:

```
print ('Métodos Lista')
print(dir(numeros_primos))
print ('Métodos Tupla')
print(dir(quadrados_perfeitos))
```

Principais métodos de dicionário Python

Os dicionários Python possuem vários métodos úteis que podem ser usados para acessar e manipular os dados armazenados. Abaixo estão alguns dos métodos mais comuns que você pode encontrar ao trabalhar com dicionários:

• **keys (): retorna uma sequência contendo todas as chaves do dicionário.**
• **values (): retorna uma sequência contendo todos os valores do dicionário.**
• **items (): retorna uma sequência de tuplas, em que cada tupla contém os pares de chave-valor do dicionário.**
• **clear (): remove todos os itens do dicionário.**
• **copy (): retorna uma cópia do dicionário.**
• **get (chave, valor_padrao): retorna o valor associado à chave especificada, ou o valor padrão se a chave não existir. Isso evita o KeyError do qual falamos anteriormente, caso a chave não exista.**
• **pop (chave, valor_padrao): remove e retorna o valor associado à chave especificada, ou o valor padrão se a chave não existir.**
• **popitem (): remove e retorna o último par de chave-valor inserido no dicionário (em versões de Python anteriores a 3.7, o par retornado é aleatório).**
• **update(outro_dicionario): atualiza o dicionário com os pares de chave-valor de outro dicionário, isto é, chaves novas são criadas e chaves existentes têm seu valor atualizado**.

Sintaxe:

Python foi originalmente desenvolvido como uma linguagem de ensino, mas sua facilidade de uso e sintaxe limpa levaram-no a ser adotado por iniciantes e especialistas.

A sintaxe da linguagem de programação Python é o conjunto de regras que define como um programa Python será escrito e interpretado (tanto pelo sistema de execução quanto por leitores humanos).

Palavras-chave em Python

As palavras-chave são as palavras reservadas pela linguagem Python, nós não podemos utilizar essas palavras para nomear nossas ***variáveis***, ***funções*** ou qualquer outro identificador, elas são usadas para definir a sintaxe e a estrutura da linguagem Python, vale lembrar que as palavras-chave são case sensitive e devem ser escritas dessa maneira.

A seguir mostramos a lista de todas as palavras-chave:

Nome	Descrição
and.	operador lógico "e"
as	capaz de criar um alias
assert	usado para debugging
async	usado para escrever aplicações asyncio
await	usado para escrever aplicações asyncio
break	para sair de um loop
class	define uma classe
continue	continua para a nova iteração do loop
def	define uma função
del	deleta um objeto
elif	usado em comandos condicionais, como else e if
else	usado em comandos condicionais
except	usado com exceções, para tratar possíveis erros
False	Valor booleano, resulta de operações de comparação
finally	utilizado com exceções, um bloco de código que executará independente de ter uma exceção ou não
for	usado para criar um loop
from	para importar partes específicas de um módulo
global	declara uma variável global
if	usado para comandos condicionais
import	usado para importar módulos
in	capaz de checar se um valor está presente em uma lista, tupla etc.
is	testa se duas variáveis são iguais
lambda	cria uma função anônima
None	representa um valor null
nonlocal	declara uma variável não-local
not	operador lógico de negação
or	operador lógico "ou"
pass	comando null, um comando que não faz nada
raise	dispara uma exceção
return	para sair de uma função e retornar um valor
True	Valor booleano, resulta de operações de comparação
try	Comando de try, usado em conjunto com except
while	Cria um loop while
with	usado para simplificar a lida com exceções
yield	finaliza uma função, retorna um gerador

Veja que nos é retornado uma lista (Estrutura de Dados que veremos com mais detalhes em breve) com todas as palavras-chave da linguagem Python.

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if',
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try',
'while', 'with', 'yield']
```

Identificadores

Os identificadores são nomes dados às entidades como **variáveis**, **funções**, **classes**, etc, eles nos ajudam a diferenciar uma entidade da outra.

Regras

Identificadores podem ser escritos com uma combinação de **letras em lowercase (a até z) ou uppercase (A até Z)** ou dígitos **(0 até 9)** ou um **underline (_).** Nomes como **minhaClasse, variavel_1 e minha variável** são exemplos válidos de identificadores.

Variáveis são **case sensitive** (idade, Idade e IDADE são três variáveis diferentes) Identificadores **não podem começar com dígitos:** 13variavel é inválido, porém variavel13 é aceito!

Palavras-chave jamais podem ser usadas como identificadores!

Indentação

Enquanto em outras linguagens de programação a **indentação** é usada apenas para tornar o código mais legível, em **Python** ela é importantíssima, Python usa a indentação para indicar blocos de código, por exemplo:

```
vida = 100

if vida > 0:

   print ("Você está vivo")
```

Algumas regras de indentação:

- Use dois pontos **:** para iniciar um bloco e pressione ***[Enter].***
- Todas as linhas em um bloco devem usar a mesma indentação, seja com espaços ou ***[tab].***
- Python recomenda quatro espaços como indentação para tornar o código mais legível. Não misture espaço e ***[tab]*** no mesmo bloco.

*Você pode configurar seu editor de texto para a tecla **[tab]** indentar uma quantidade x de espaço.*

Comentários

Python tem a capacidade de comentários para que seja mais fácil de lermos os códigos de outros programadores, melhora muito a comunicação!

Comentários começam com ***#***, por exemplo:

```
# Este é um comentário


print ("Códigos comentados são muito mais fáceis de serem compreendidos")
```

Python também suporta ***docstrings,*** que são comentários que podem ***extender*** até mais linhas, veja:

```
"""
Este é um comentário
que abrange várias
linhas do programa
"""
print ("Procure sempre comentar o seu código"
```

Statements

As instruções em Python geralmente terminam com uma nova linha. Python, entretanto, permite o uso do caractere de continuação de linha ***(\ )*** para indicar que a linha deve continuar. Por exemplo:

```
total = 3 + \
    5 + \
    7
print(total) # 15
```

O ponto e vírgula ***(;)*** permite várias instruções em uma única linha, visto que nenhuma instrução inicia um novo bloco de código. Aqui está uma amostra que ilustra esta ideia:

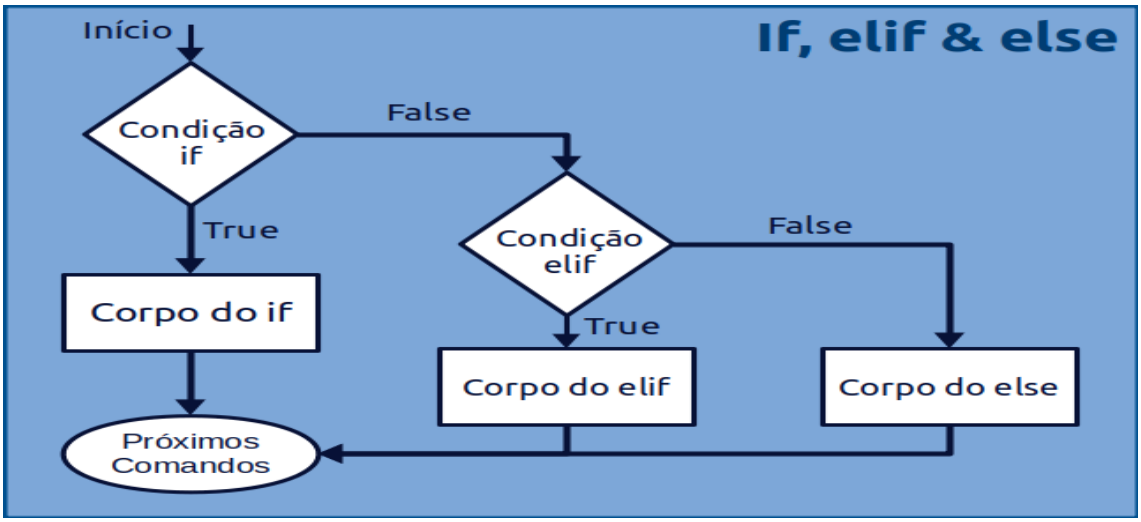
```
x, y = 9, 3; z = x * y; print(f'{x} x {y} = {z}')


# 9 x 3 = 2
```

Para saber mais detalhes específicos e técnicos sobre boas práticas de estilo de programação em Python, recomendamos que você visite e leia PEP 8 -- Style Guide for Python Code

If... Else

A tomada de decisão é necessária quando queremos executar um código apenas se uma determinada ***condição*** for satisfeita.

As instruções ***if***, ***elif*** e ***else*** são usadas em Python para nos auxiliar na tomada de decisões.



A tomada de decisão é um conceito muito importante da programação e representa a capacidade de executarmos determinados comandos apenas se condições especificadas forem satisfeitas.

Lembrando que Python é capaz de suportar as condições lógicas tradicionais da matemática:

• Igualdade: x == y
• Diferente de: x! = y
• Menor que: x < y
• Menor que ou igual a: x <= y
• Maior que: x > y
• Maior que ou igual a: x >= y

Essas condições podem ser usadas de várias maneiras, e são comumente utilizadas em instruções if e loops.

A sintaxe para construirmos uma estrutura de tomada de decisão funciona então da seguinte forma:

```
if <expressão>:
    <comandos>
elif <expressão>:
    <comandos>
else:
    <comandos>
```

Neste exemplo, o programa executa a primeira linha ***if <expressão>,*** se a ***<expressão>*** for avaliada como True, os ***<comandos>*** dentro do bloco **if** serão executados.

Caso a ***<expressão>*** da primeira linha ***if <expressão>*** seja avaliada ***como False***, nosso programa irá pular para a linha elif ***<expressão>*** e irá testar a nova ***<expressão>:***

- Se ela for avaliada como ***True*** os comandos do bloco **elif** serão executados.
- Se ela for avaliada como ***False*** os comandos do bloco **else** serão executados.

Exemplos práticos para compreender melhor a ideia de tomada de decisão em nossos códigos.

```
num = 5
if num > 0:
    print("O número é positivo")
print("Esse valor é sempre impresso, pois está fora do bloco if")
```

Nesse caso podemos perceber que a expressão dentro do bloco ***if*** será executa, uma vez que o valor de ***num*** é maior que ***0***, sendo assim, positivo.

Usar a palavra-chave ***elif*** para testarmos outras possibilidades

```
x = 12
y = 12
if x > y:
    print("x é maior do que y")
elif x == y:
    print("x e y são iguais")
```

Conclusão:

Os dicionários são uma estrutura de dados poderosa em Python que permite armazenar pares de chave-valor de forma eficiente e flexível. Eles são mutáveis, o que significa que você pode adicionar, remover e atualizar valores depois de criá-los.

Esperamos que este post tenha sido útil para ajudá-lo a entender o que são dicionários em Python e como utilizá-los em seus programas. Lembre-se de que a prática é a melhor forma de aprender a programar, então experimente criar seus próprios dicionários e utilizar os métodos apresentados aqui para explorar ainda mais essa estrutura de dados.

Referências:

A Internet é um oceano infinito de informações e Python é uma linguagem riquíssima em conteúdo online e conta com uma comunidade muito forte.

A seguir temos um levantamento de referências essenciais que podem ser utilizadas para maximizar o nosso conhecimento perante a linguagem Python e suas bibliotecas.

https://www.alura.com.br/artigos/conhecendo-as-tuplas-no-python#o-que-e-tupla?-como-manipular?

https://asimov.academy/dicionarios-em-python-o-que-sao-e-como-utilizar/

https://pythoniluminado.netlify.app/sintaxe