Coleção de Dados

As coleções permitem armazenar múltiplos itens dentro de uma única unidade, que funciona como um container.

Em Python há basicamente 3 tipos de coleções:
Listas
Tuplas
Dicionários

Coleção de Dados

LISTA

Lista é uma sequência finita de elementos.
A lista uma estrutura de dados amplamente utilizada no desenvolvimento de software.
Estrutura de dados constituída por uma seqüência ordenada e finita de itens (os quais podem, inclusive, ser outras listas, ditas sublistas), e que pode ser modificada com a inserção, exclusão e reordenamento dos itens.

```
lista_inteiros = [12,34,56,67]

lista_frutas = [‘Morango’,’Uva’,’Manga’,’Tomate’,’Laranja’]

lista_mista = [‘Morango',23,’Uva',45,’Tomate’]

print(lista_inteiros)
print(lista_frutas)
print(lista_mista)
```

Coleção de Dados

TUPLA

Tupla é uma Lista imutável. O que diferencia da Lista é que a Lista pode ter elementos adicionados a qualquer momento, enquanto que a Tupla após estrutura definida, não permite a adição ou remoção de elementos.

```
tupla_numeros = (1,2,56,45)
print(type(tupla_numeros))
print(tupla_numeros)
```

```
tupla_carros = “Gol”,”Fusca”,”Opala”,”Marea”
print(type(tupla_carros))
print(tupla_carros)
```

Coleção de Dados

DICIONÁRIOS

Dicionários são um coleção desordenada de objetos representados na forma de chave, valor onde a chave é usada para referenciar um determinado valor. As chaves de um dicionário só podem ser de um tipo imutável como inteiros, floats e strings. Tuplas também podem ser aceitas desde que não contenham direta ou indiretamente um tipo mutável como listas.

Dicionários são listas de associações compostas por:
Uma chave e um valor correspondente

dicionario = { 'chave' :'valor’ }

Exemplo de Dicionários:

estados_siglas = { ’SC' : ’Santa Catarina', ’PR' : ’Paraná' , ’RS' : ’Rio Grande do Sul', ’SP' : ’São Paulo'}
print(estados_siglas)

Funções em Python

Na programação, funções são blocos de código que realizam determinadas tarefas que normalmente precisam ser executadas diversas vezes dentro de uma aplicação. Quando surge essa necessidade, para que várias instruções não precisem ser repetidas, elas são agrupadas em uma função, à qual é dado um nome e que poderá ser chamada/executada em diferentes partes do programa.

```
def oi():
    print("Olá Estou dando OI")

oi()
```

Funções com parâmetros:

```
def soma(a, b):
    return  a + b

c= soma(1,3)
print(c)
```

JCAVI
TREINAMENTOS EM TI

Funções em Python

Argumentos *args e **kwargs

Podemos passar um número arbitrário de parâmetros em uma função. Utilizamos as chamadas variáveis mágicas do Python: ***args** e ****kwargs**.
Não é necessário utilizar exatamente estes nomes: *args e **kwargs. Apenas o asterisco(*), ou dois deles(**), serão necessários. Podemos optar, por exemplo, em escrever *var e **vars. Mas *args e **kwargs é uma convenção entre a comunidade que também seguiremos.

*args e **kwargs permitem passar um número variável de argumentos de uma função. O que a variável significa é que o programador ainda não sabe de antemão quantos argumentos serão passados para sua função, apenas que são muitos. Então, neste caso usamos a palavra chave *args.

Funções em Python

*args retorna vários elementos em uma tupla.

```
def soma(*args):
    print (args)

soma(1,4,5,6)

def soma_total(*args):
    total = 0
    for numero in args:
        total = numero + total
    return total

print(soma_total(4,4,2,25,234,2312))
```

JCAVI
TREINAMENTOS EM TI

Funções em Python

****Kwargs** também permite você passar um número indeterminado de parâmetros para uma função. Mas, diferente do ***args**, aqui os argumentos são passados com um identificador ou chave, similar a um dicionário Python. Isto nos facilita recuperar os argumentos. Veja no script a seguir:

```
def saudacoes(**kwargs):
    print(kwargs)

saudacoes(manha="bom dia",tarde="Boa Tarde",noite="Boa Noite“)

def saudacoes_dia(**kwargs):
    for hora, saudacao in kwargs.items():
        print(f"Durante a {hora} dizemos {saudacao}")

saudacoes_dia(manha="bom dia",tarde="boa tarde")
```

Funções Decoradoras

Um decorador permite que você potencialize, modifique ou substitua completamente a logica de uma função ou método.

Vamos criar a função decoradora

```
def master(msg):
    def imprime():
        print("esse é a função principal")
        msg()
    return imprime
```

Segunda Função para utilizarmos juntamente com a função decoradora

```
def chama_funcao():
    print("Esta chamando a função Verdadeira")

chama_funcao()
```

Funções Decoradoras

Podemos decorar uma função de 2 maneiras:

1) Através de uma variavel

```
## Função decorada
chama_funcao = master(chama_funcao)
```

2) Através do decorador @

```
@master
def chama_funcao():
    print("Esta chamando a função Verdadeira")
```

Funções Decoradoras

Outros exemplos de funções decoradas:

```
def decoradora(valor):
        def imprime(*args):
            print("Soma Executada")
            return valor(*args)
        return imprime

@decoradora
def multiplica(a,b):
    return a * b

print(multiplica(2,3))
```

Funções Decoradoras

```
contador = 0

def contar_acessos(funcao_decorada):
    def nova_func(*args, **kw):
        global contador
        contador += 1
        return funcao_decorada(*args, **kw)
    return nova_func

@contar_acessos
def soma(a, b):
    return a + b

print(contador)
print(soma(2,2))
print(contador)
```

JCAVI
TREINAMENTOS EM TI

Exercícios Funções

Crie funções com argumentos *args e **kwargs e faça teste de saida dos argumentos e execute o iterador for nos elementos.

Crie 2 funções onde 1 será a função decoradora e a outra irá receber a função decorada. Teste pelo método decorado e pelo método através de variáveis.

Função Enumerate

Retorna o índice de uma coleção de dados como uma lista por exemplo.

```
animais = [ "Cachorro","Gato",'Periquito','Elefante’]

print(list(enumerate(animais)))
```

Iterar um uma lista com enumerate

```
for i, valor in enumerate(animais):
    print(i,valor)
```

Iterador e Enumerate com Condicionais

```
for i, valor in enumerate(animais):
    if i > 1:
     break
    else:
        print(valor)
```

JCAVI
TREINAMENTOS EM TI

Função lambda

Funções lambdas ou funções anonimas é igual as funções tradicionais
Porém é escrita de uma forma mais otimizada.

```
funlmb = lambda x,y : x + y

print(funlmb(2,3))

lamb1 = lambda : print("Chama Funcao Lambda")

lamb1()
```

Função MAP

A função Map() tem como objetivo aplicar uma função a todos os elementos de uma sequencia ou coleção de dados, retornando uma nova sequência com o resultado aplicado.

```
lista = [1,4,6,5,8]

def soma(x):
    return x + 2

map(soma,lista)

print(list(map(soma,lista)))

import math

print(list(map(math.sqrt,lista)))
```

Função REDUCE

A função reduce, disponível no módulo built-in functools, serve para reduzir um iterável (como uma lista) a um único valor.

```
from functools import reduce

lista = [2,7,10,3,78]

def mult(x,y):
    return x * y

print(reduce(mult,lista))
```

Testa o maior valor com reduce

```
lista2 = [45,1,56,12,6]

testmaior = lambda x,y: x if (x > y)  else y

print(reduce(testmaior,lista))
```

Função FILTER

Como o próprio nome já deixa claro, filter() filtra os elementos de uma Coleção de dados o processo de filtragem é definido a partir de uma função que passa como primeiro argumento da função. Assim, filter() só “deixa passar” para a sequência resultante aqueles elementos para os quais a chamada da função.

```
listamista = [1,"João",25,"Pedro",45]

def valida(x):
    return x == 'João'

print(list(filter(lambda x: x == "Pedro", listamista)))

print(list(filter(valida, listamista)))
```

JCAVI
TREINAMENTOS EM TI

Função ZIP

A função *zip* retorna uma seqüência cujos elementos são tuplas resultantes de cada um dos elementos de uma ou mais seqüências de entrada.A seqüência resultante é sempre truncada ao tamanho da menor seqüência apresentada.

```
dicverduras = { 1:"Cebola", 2:"Alface", 3:"Repolho", 4:"Beterraba"}
dicfrutas = { 1:"Maça", 2:"Laranja", 3:"Pera"}

junta = list(zip(dicverduras,dicfrutas))

print(junta)

juntavalores = list(zip(dicverduras.values(),dicfrutas.values()))

print(juntavalores)

Iterar o resultado do zip
for p in juntavalores:
    print(p)
```

JCAVI
TREINAMENTOS EM TI

Criando Ambiente Virtuais

Quando estamos desenvolvendo diversos projetos em Python, é comum utilizarmos diferentes versões de uma mesma biblioteca entre este projetos.

O que é uma virtualenv?
Como dito acima, um problema muito comum é quando precisamos utilizar diversas versões de uma mesma biblioteca em diferentes projetos Python. Isso pode acarretar em conflitos entre as versões e muita dor de cabeça para o desenvolvedor. Para resolver este problema, o mais correto é a criação de um ambiente virtual para cada projeto.
Basicamente, um ambiente virtual empacota todas as dependências que um projeto precisa e armazena em um diretório, fazendo com que nenhum pacote seja instalado diretamente no sistema operacional. Sendo assim, cada projeto pode possuir seu próprio ambiente e, consequentemente, suas bibliotecas em versões específicas.

Criando Ambiente Virtuais

Criando o Ambiente Virtual

1) Instalando o Virtualenv com o pip:

pip install virtualenv

2) Criar a estrutura de pastas da virtualenv

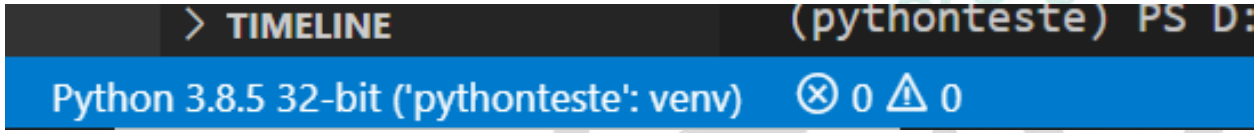
3) Criando a virtualenv

virtualenv nome_da_virtualenv

4) Ativar a virtualenv
Navegar ate pasta scripts e digitar activate

Criando Ambiente Virtuais

Conectar a Ide ao virtualenv



JCAVI
TREINAMENTOS EM TI

Exercício de Fixação

Para esse exercício crie um novo Virtualenv e conecte o Vscode ao novo virtualenv.

Crie uma coleção de dados(lista,tupla,dicionário) e trabalhe com a função enumerate Aplicando condicionais.

Crie 2 funções lambda(anônima) e chame as funções no código.

Efetue um calculo em uma coleção de dados com a função MAP.

Execute a Função REDUCE em uma tupla onde umas das funções a serem utilizadas tem que ser uma função lambda.

Execute um filtro condicional com a função FILTER utilizando uma tupla.

Crie 2 dicionários de dados como valores distintos e execute a função zip nos valores Dos dicionário e imprima os valores.