**SEÇÃO 2**

* **Windows (5)**

Comandos Power Shell:

Get-ExecutionPolicy;

Set-ExecutionPolicy AllSigned -Force;

Set-ExecutionPolicy Unrestricted -Force;

Set-ExecutionPolicy Restricted -Force.

* **Configurações Vs Code**

Extensões:

Python;

Code Runner.

**SEÇÃO 3**

* **Função Print (17)**

Argumentos

sep -> alterar o separador;

end -> final de linha.

* **Tipo str (18)**

Escape (\) -> ignora algum caractere;

r -> mostra os caracteres de escape.

* **Exercício de Programação (28)**

Ellipssis (...) -> utilizado para substituir um código não escrito.

* **Uma introdução às f-strings (30)**

Exemplo de formatação de float para moedas -> {variável:,.2f}.

* **Usando a função input (32)**

Como extrair o nome da variável e seu valor na f-strings -> {variável=}.

Obs.: não realizar a conversão de dados no input.

* **Operador lógico “and” (39)**

Valores falsy -> 0 – 0.0 – ‘’ – False.

Tipo None usado para representar um não valor.

Avaliação de curto-circuito (and) -> quando Python avalia uma expressão ‘and’ ele encerra assim que concluí que ela e falsa é retorna o valor onde é falso.

* **Operador lógico “or” (40)**

Avaliação de curto-circuito (or) -> quando Python avalia uma expressão ‘or’ ele encerra assim que encontrar um valor verdadeiro e retorna esse valor.

* **Operadores in e not in (42)**

Iteráveis -> são elementos que é possível percorrer item a item.

* **Interpolação de string com % em Python (43)**

s – String;

d/i – Inteiro;

f – Float;

x/X – Hexadecimal.

Ex.: ‘%s’ % (variável).

* **Formatação de strings com f-strings (44)**

> -> Esquerdo;

< -> Direita;

^ -> centro;

= -> Força o número aparecer antes do sinal;

O sinal de (+) depois do sinal, é usado para dar sinal de positivo e sinal negativo, dependendo do valor do número, como padrão só coloca o sinal de negativo.

* **Parte 2: variáveis, constantes e complexidade de código (51)**

Buscar simplificar as condições dos IFs armazenando em variáveis. Ficando mais descritível o código.

* **ID – A identidade do valor que está na memória (51)**

Função id -> retorna o id do valor da variável na memória do computador.

Obs.: dependendo do valor da variável, se houver duas variáveis com o mesmo valor, Python armazena apenas um valor para as duas variáveis.

* **Flags, is, is not e None (53)**

Flag -> marca um local.

is e is not -> é ou não é (tipo, valor identidade).

* **while / else (recurso peculiar do python) (69)**

O else do while só é executado quando o laço inteiro e executado sem ser interrompido.

* **range + for para usar intervalos de números (73)**

range é um tipo de dado iterável.

* **Como o for funciona por baixo dos panos? (next, iter, iterável e iterador) (74)**

Iterável é todo objeto que pode ser percorrido elemento a elemento. É possui o método \_\_iter\_\_().

\_\_iter\_\_() retorna um objeto iterador.

O objeto iterador é que sabe entregar um valor por vez do iterável. E possui o método \_\_next\_\_().

O método \_\_next\_\_() entrega o próximo valor do iterável.

* **Introdução ao empacotamento e desempacotamento (87)**

Desempacotamento é extrair os valores de uma lista para variáveis simples.

(\_) -> a variável (\_) e usada para representar um valor que não será mais utilizado.

O \* serve para falar que o restante teve ser agrupado em uma lista.

* **enumerate para enumerar valores de iteráveis (89)**

Objetos enumerate são objetos que enumeram um objeto iterável.

Uma vez que eu percorro um enumerate, não consigo percorrê-lo novamente.

* **Imprecisão dos números de ponto flutuante (92)**

Devido a forma com que os números flutuantes são salvos na memoria gerar uma imprecisão neles.

O modulo decimal do Python é responsável por fazer essas correções.

* **split, Join e strip são métodos muito úteis da str (93)**

join uni iteráveis separado por um separador: ‘separador’.join(iterável).

* **Detalhes sobre o interpretador do Python (95)**

python mod.py

python -u

python -m mod (Lib mod como script)

python -c ‘cmd’ (comando)

python -i mod.py (interativo com mod)

* **Desempacotamento em chamadas de funções (96)**

Quanto passar o iterável em uma chamada de função coloque o \*, para o desempacotamento.

**SEÇÃO 4**

* **(parte 1) Escopo de funções e módulos em Python + global (108)**

A palavra global é utilizada dentro de uma função para falar que o valor da variável e o valor do escopo anterior, que será alterado.

* **(parte 2) Escopo de funções e módulos em Python + global (109)**

O valor a ser usado dentro de um escopo é sempre o mais próximo desse escopo.

* **(parte 1) \*args para quantidade de argumentos não nomeados variáveis (111)**

Para receber uma quantidade de valores indeterminados e não nomeados utiliza-se \*args como argumentos na definição.

* **Higher Order Functions – Funções de primeira classe (114)**

Em Python as funções são um tipo de objeto;

Sendo possível retornar funções em outras funções.

* **Closure e funções que retornam outras funções (116)**

Closure, permite que adie o fechamento de uma função;

Também é possível criar uma função para criar funções.

* **(parte 1) Métodos úteis nos dicionários Python (121)**

Se existir chaves iguais Python pega a última;

Método setdefault adiciona valor se a chave não existe.

* **Shallow Copy vs Deep Copy em dados mutáveis Python (122)**

Shallow copy -> copia rasa, não abrange os subníveis;

Deep copy -> copia total, abrange os subníveis, utilizar o modulo copy.

* **(parte 2) Métodos úteis nos dicionários Python (123)**

Método get obtém uma chave e se não encontrar retorna None, se encontrar retorna o valor;

Método popitem remove o último item adicionado;

Método update atualiza um dicionário.

* **Peculiaridades do tipo mutável set em Python (127)**

Não aceitam valores mutáveis;

Seus valores serão sempre únicos;

Não tem indexes;

Não garantem ordem;

São iteráveis;

Obs.: quando criar um set com set(), ele itera o iterável.

* **Métodos úteis do tipo set em Python (128)**

Método add adiciona um elemento ao set;

Métodos update adiciona mais de um elemento, porém ele itera o iterável;

Método clear limpa o set;

Método discard elimina um valor.

* **Operadores importantes para o tipo set (129)**

União (|) -> une os sets;

Intersecção (&) -> itens presentes em ambos;

Diferença (-) -> itens presentes apenas no set da esquerda;

Diferença simétrica (^) -> itens que não estão em ambos.

* **Introdução à função lambda + list.sort e sorted (133)**

Função anônima de uma linha;

Para ordenar um dicionário e preciso indicar como essa ordenação deve ser feita, para isso podemos usar uma função para retornar o elemento do dicionário que devemos colocar em ordem.

Ex.: lambda <parâmetros>: <retorno>, <argumentos>

lista.sort(key=lambda item: item[‘chave’])

* **Empacotamento e desempacotamento de dicionários + \*args e \*\*kwargs (135)**

Para desempacotamento de um dicionário dentro de outro basta colocar \*\* antes do nome do dicionário;

Para empacotar argumentos nomeados em funções basta usar \*\*kwargs.

* **Mapeamento de dados em list comprehension (map) (137)**

Mapeamento é a criação de um iterável a partir de outro, podendo gerar algumas alterações;

Nas list comprehension é possível usar operador ternário;

É possível realizar cálculos e trabalhar com iteráveis.

* **Filtro de dados em list comprehension (Filter) (138)**

Filtrar é excluir alguns valores por determinada condição, para isso usa o if depois do for.

* **List comprehension com mais de um for (139)**

É possível usar 2 for para gerar uma lista de tuplas ou lista;

É possível gerar uma lista de listas.

* **Mais detalhes sobre list comprehension (140)**

Flat, onde é possível pegar uma lista de listas e transformar em uma lista apenas, usando dois laços em exercícios execução ‘simultânea’.

* **Dictionary comprehension e set comprehension (141)**

E possível fazer comprehension com dicionário, no entanto e necessário trabalhar com a chave e valor;

Obs.: uma lista que possua listas com dois elementos, é possível fazer a coerção para dicionário.

* **isinstace() – Para saber se objeto é de determinado tipo (142)**

A função isinstance() serve para determinar de qual instancia é determinado objeto.

* **dir, hasattr e getattr em Python (144)**

dir, para mostrar quais métodos um objeto possui;

hasattr, verifica se o método existe para o objeto;

getattr, pega o método de um objeto.

* **Generator expression, iterables e iterators em Python (146)**

Generator são funções que pausam, sendo executadas somente quando chamar o próximo método next().

* **Introdução às generator functions em Python (147)**

Generator functions são funções que pausam, para isso deve usar o yield como ponto de parada.

* **yield from em generator functions (148)**

Com o yield from é possível executar uma function generator dentro de outra generator function.

* **(parte 1) try e except para tratar exceções (149)**

A classe exception representa todos os erros.

* **(parte 2) try e except para tratar exceções (150)**

Posso extrair um alias de classe de exceção e extrair informações dele.

* **try, except, else e finally + built\_in exception (151)**

Finally e um bloco de código que sempre é executado;

Else só executa quando todo ocorrer bem.

* **raise – lançando exceções (erros) (152)**

raise lança um erro na tela.

* **modularização – entendendo os seus próprios módulos e sys.path no Python (154)**

O primeiro módulo executado chama-se \_\_main\_\_;

É possível configurar para Python buscar módulos em outro pacote diferente do pacote que o \_\_main\_\_ está.

Python reconhece somente os caminhos de módulos para frente do modulo \_\_main\_\_;

* **Recarregando módulos, import lib e singleton (156)**

Módulos Python são singleton, ou seja, eles carregam apenas uma vez;

Para recarregá-lo e necessário utilizar a função reload() do módulo importlib.

* **Introdução aos packages (pacotes) em Python (157)**

Os módulos possuem uma variável chamada \_\_all\_\_, onde está especificado o que deve ser importado do modulo quando ele for importado usando o \*.

* **O ponto de vista do \_\_main\_\_ (158)**

Todas as importações de módulos devem ser feitas a partir do ponto de vista do modulo \_\_main\_\_.

* **\_\_init\_\_.py é um arquivo de inicialização dos packages em Python (159)**

O arquivo \_\_init\_\_.py sempre será executado quando um package for importado.

* **Variáveis livres + nonlocal (locals, globals) (164)**

Variáveis livres são variáveis que estão definidas um escopo acima;

Variáveis livres não podem ser alteradas, para alterar tenho que indicar que são nonlocal.

* **Funções decoradoras em geral (165)**

Funções decoradoras são funções que decoram outras funções.

* **Decoradores em Python (@syntax\_sugar) (166)**

Decoradores são usados para fazer o Python usar as funções em outras funções;

Obs.: o nome da função que recebe o decorador, passa a ser o nome da função interna da função decoradora.

* **Decoradores com parâmetros (167)**

Para poder receber parâmetros no decorador, devo criar uma função que retornara a função decoradora.

* **Ordem de aplicação dos decoradores (168)**

Os decoradores são executados de baixo para cima.

* **Solução do exercício + zip e zip\_longest do itertools (170)**

zip -> é uma função que une duas listas a partir da menor lista;

zip\_longest -> é uma função do modulo itertools que une duas listas a partir da maior, e complementa os valores da menor com um fillvalue, que pode ser passado.

* **Count é um iterador sem fim (itertools) (173)**

count é um iterador sem fim, para gerar um count é necessário importar o modulo itertools e usar a classe count.

* **combinations, permutations e product – itertools (174)**

- Funções do modulo itertools:

- combinations -> realiza a combinação dos itens de uma lista;

- permutations -> realiza a permutação dos itens de uma lista;

- product -> realiza o produto cartesiano de uma lista de listas;

Obs.: É necessário desempacotar na chamada da função.

* **groupby – Agrupando valores (itertools) (175)**

groupby agrupa os valores de um iterável;

Obs.: É necessário ordenar o iterável.

* **map, partial, generatortype e esgotamento de iterators (176)**

No modulo functools tem uma classe chamada partial que gera funções;

A função map faz o mapeamento de um iterável, tem que passar a função e o iterável;

No modulo types temos a classe GeneratorType.

* **Filter é um filtro funcional (177)**

A função filter faz o filtro de um iterável, tem que passar uma função e um iterável.

* **Reduce – Faz a redução de um iterável em um valor (178)**

A função reduce do modulo functools, reduz um iterável a apenas um valor, tem que passar uma função, um iterável e um valor inicial.

* **Limite de recursão e cuidados com funções recursivas (180)**

No Python há um limite de recursão, para alterar teve configurar com sys.setrecursionlimit().

* **O que são ambientes virtuais? (venv) (181)**

Ambiente virtual carrega toda a instalação Python para uma pasta;

Ao ativar o ambiente virtual, a instalação do ambiente virtual que será utilizada;

venv é o modulo que utilizamos para criar ambientes virtuais.

* **Como criar o seu ambiente virtual com venv (182)**

- Comando para criar o ambiente virtual

python -m venv venv

- Comando para ver o caminho onde Python está instalado

gcm python -syntax

* **Ativando e desativando o meu ambiente virtual venv (183)**

Comando para ativar o ambiente virtual

venv/Scripts/activate

Comando para desativar o ambiente virtual

deactivate

* **pip – instalando pacotes e bibliotecas (184)**

Comando para instalar lib

pip install <lib>==version

Comando para atualizar lib

pip install <lib> --upgrade

Comando para desinstalar lib

pip uninstall <lib>

Comando para listar as lib instaladas

pip freeze

Comando para listar as versões de uma lib

pip index versions <lib>

* **Criando e usando um requirements.txt (185)**

Comando para criar o arquivo requirements.txt

pip freeze > requirements.txt

Comando para instalar as libs do arquivo requirements.txt

pip install -r .\requirements.txt

* **Criando arquivos com Python + context manager with (186)**

Context manager -> with (abre e fecha).

* **with open (context manager) e métodos úteis do TextIOWrapper (187)**

- Métodos para trabalhar com arquivos:

write -> escrever;

writelines -> escreve várias linhas;

read -> ler;

readline -> lê uma linha;

readlines -> lê várias linhas;

seek -> move o cursor.

* **Modos de abertura de arquivos e enconding com with open (188)**

Modos do arquivo:

r -> leitura;

w -> escrita;

a -> concatenação;

b -> binário;

t -> texto;

+ -> adicionar leitura ou escrita.

O parâmetro enconding para abrir o arquivo, serve para especificar o padrão de codificação.

* **os.remove, os.unlink e os.rename – Apagando, renomeando e movendo arquivos (189)**

os.remove e os.unlink -> apagam um arquivo;

os.rename -> renomeia ou move um arquivo.

* **Salvando dados Python em json com módulo json (190)**

Arquivos json são usados para armazenar e transportar os dados pelo projeto.

Para salvar dados em um arquivo json, teve usar a função dump() do modulo json, como parâmetro deve-se passar o dado a ser armazenado e o arquivo;

Obs.: tem também o parâmetro indent, para passar uma indentação para a formatação do arquivo.

Para pegar os dados de um arquivo json, teve usar a função load() do modulo json, como parâmetro deve-se passar o nome do arquivo;

Obs.: as tuplas são convertidas em listas.

* **Problemas dos parâmetros mutáveis em funções Python (191)**

Quando passa valor padrão para um tipo mutável na criação de uma função, quando chamar a função sem o parâmetro especificado ele vai usar a mesma e não vai criar outra;

Para resolver o problema, coloca o parâmetro como None e verifica para criar dentro do escopo da função.

* **Evitando uso de condicionais + Guard Clause (194)**

Evitar o uso do else em funções utilizando apenas o return (Guard Clause);

Dicionário com métodos e utilizado get para pegar seu valor através da chave.

* **Positional-Only Parameters (/) e Keyword-Only Arguments (\*) (196)**

- (/) -> utilizo para determinar que os parâmetros que vem antes devem ser argumentos posicionais;

- (\*) -> utilizo para determinar que os parâmetros que vem depois devem ser argumentos nomeais.

**SEÇÃO 5**

* **Atributos de classe (204)**

Os atributos de classe são atributos que pertencem a toda a classe e todas as instâncias.

* **\_\_dict\_\_ e vars para atributos de instâncias (205)**

\_\_dict\_\_ é o dicionário que armazena as informações do objeto;

vars é uma função que retorna o \_\_dict\_\_ do objeto.

* **Métodos de classe (@classmathod) + factories methods (métodos fábrica) (209)**

São métodos que são executados sem a necessidade de uma instância. Portanto ao invés de recebe self será cls, e recebem o decorador @classmethod.

* **@staticmethod (métodos estáticos) (210)**

Métodos estáticos, são métodos que se comportam como uma função dentro da classe.

* **@property – um getter no modo Pythônico (212)**

O decorador @property ele possibilita que um método possa ser chamado como atributo no código cliente.

* **@property + @setter – getter e setter no modo Pythônico (213)**

Para criar o método setter é só passar o decorador @<nome do método property>.setter;

Obs.: nomes de atributos com um ou dois underlines não devem ser usados fora da classe.

* **Encapsulamento (modificadores de acesso: public, \_protected, \_\_private) (214)**

Atributos public -> sem underlines (todo lugar);

Atributos protected -> um underlines (classe e subclasses);

Atributos private -> dois underlines (classe);

Obs.: os atributos com um underlines não são importados quando importamos tudo.

* **Relações entre classes: associação, agregação e composição (215)**

Associação é um tipo de relação entre os objetos dentro do sistema. É possui os subconjuntos como agregação e composição;

Na associação temos um objeto que possui um atributo que referencia outro objeto controla o outro objeto.

* **Agregação – Python orientado a objeto (216)**

Agregação é possível fazer a associação entre dois ou mais objetos. Cada objeto é independente, mas pode se tratar de uma relação em que um objeto precisa de outro para realizar determina tarefa. Geralmente é uma relação de um para muitos, onde um objeto tem um ou muitos objetos.

* **Composição – Python orientado a objeto (217)**

Composição é uma especificação da agregação, mas nela se o objeto “pai” for apagado, todas as referencias dos objetos “filhos” também são apagados.

* **Herança simples – Python orientado a objeto (220)**

Method resolution order -> é a ordem em que os métodos ou atributos serão buscados;

Função help() traz um relatório sobre a classe.

* **(parte 2) Super e a sobreposição de membros (222)**

Para referir a classe superior deve usar super(class, self), ou o nome da classe;

Para visualizar a method resolution ordem pode usar o método de classe rmo().

* **Teoria – Herança múltiplas – Python orientado a objeto (223)**

Herança múltipla é quando uma classe herda de mais de uma classe;

Mixins é uma classe que não faz parte da árvore de uma classe, mas é herdada por alguns métodos que serão utilizados pela nova classe.

* **Classe abstratas – Abstract base class (ABC) (229)**

Classe abstratas são usadas para definir contratos para novas classes. Elas forçam a criação de métodos concretos em outras classes. Podem também ter métodos concretos;

Os métodos abstratos são métodos sem corpo e possuem o decorador @abstractmethods. Esses devem ser implementados nas subclasses;

A classe abstrata possui uma metaclasse sendo ABCmeta. É não podem ser instanciadas diretamente.

* **Na prática: polimorfismo, assinatura de métodos e liskov substitution principle (232)**

Polimorfismo permite que classes derivadas de uma mesma superclasse tenham métodos com mesma assinatura, mas comportamentos diferentes;

Assinatura de métodos -> nome e parâmetros (retorno);

Princípio da substituição de Liskov -> objetos de uma superclasse podem substitutos por objetos das suas subclasses.

* **Criando exceptions em Python orientado a objeto (233)**

Para criar uma exception em Python basta criar uma classe que herde de Exception;

O nome da classe deve terminar com error.

* **Python dunder methods \_\_repr\_\_ e \_\_str\_\_ (237)**

O método \_\_repr\_\_ cria uma representação do objeto para outro desenvolvedor;

Obs.: Para chamar o \_\_repr -> repr() ou !r;

O método \_\_str\_\_ cria uma representação na forma de string do objeto.

* **\_\_new\_\_ e \_\_init\_\_ em classes Python (239)**

O método \_\_new\_\_ cria o objeto e retorna, ele recebe cls(classe);

O método \_\_init\_\_ e responsável por inicializar uma instancia.

* **Context manager com classes – criando e usando gerenciadores de context (240)**

Obs.: Você pode implementar seus protocolos apenas usando os métodos dunder. Esse conceito se chama duck typing, onde Python verifica a existência de alguns métodos para que objetos funcione de forma adequada, não se importando com o tipo;

Para implementar o context manager e necessário ter os métodos \_\_enter\_\_ e \_\_exit\_\_;

O método \_\_exit\_\_ recebe a classe da exceção, a exceção e o traceback e quando ela retorna True a exceção é suprimida no with.

* **Context manager com contextlib.contextmanager (242)**

Pode-se criar também um context manager usando um generator function com o decorador @contextmanager do modulo contexlib.

* **Funções decoradoras e decoradores com classes (243)**

É possível usar funções decoradoras para alterar uma classe.

* **Funções decoradoras e decoradores com métodos (244)**

É possível usar funções decoradoras em um método;

A função decoradora deve receber o método. E a função interna recebe self e os argumentos do método, a função interna deve retornar o método que vem da função decoradora.

* **Métodos especial \_\_call\_\_ (245)**

O método \_\_call\_\_ em classes normais faz a instância de uma classe ser “callable”, callable é algo executável quando colocado os parênteses.

* **Classes decoradoras (decorator classes) (246)**

Se eu colocar o decorador com parênteses a função(func) passa a ser recebida no método \_\_call\_\_, ao invés de no método \_\_init\_\_.

* **Teoria: metaclasses são tipo das classes (247)**

Todas as classes em Python possuem o tipo type, sendo type uma metaclass;

É possível criar uma classe usando type()

type(‘name’, (bases), \_\_dict\_\_)

Ao criar uma classe os métodos \_\_new\_\_() e \_\_call\_\_() da metaclass são chamados;

O método \_\_new\_\_ da metaclass cria a classe. \_\_new\_\_(mcs, name, bases, dct);

O \_\_call\_\_ da metaclass é chamado com os argumentos e chama \_\_new\_\_ e \_\_init\_\_ da classe e depois termina a execução \_\_call\_\_(cls, \*args).

* **Teoria: enum.Enum (enumerações) (254)**

Enums -> são um conjunto de membros ligados a valores únicos;

Podem ser iterados para retornar seus membros canônicas na ordem de definição;

Enums são usados onde temos um determinado número de coisas para escolher, os valores dos membros são constantes;

enum.Enum é a superclasse para criação dos enums, mas a metaclass dos enums é enum.EnumType;

O seu enum podem ser usados para qualquer coisa relacionado com tipo.

* **Código: enum.Enum (enumerações) (255)**

Para obter os dados:

Membro -> classe(valor), classe[‘chave’]

Chave -> classe.chave.name

Valor -> classe.chave.value

enum.auto() -> gera um valor automático a chave.

* **dataclasses – o que são dataclasses? (265)**

dataclass é um decorador do modulo dataclasses que permite a criação de classes de forma muito mais simples.

* **dataclasses com métodos, property e setter (266)**

Pode criar métodos nas dataclasses.

* **\_\_init\_\_ e \_\_post\_init\_\_ em dataclasses (267)**

Para usar o método \_\_init\_\_ e necessário passar init=False como argumento no decorador;

O método \_\_post\_init\_\_ é executado após o init, mas se especificar o init ele não é executado;

* **Configurações do decorator dataclass (268)**

É possível passar argumentos para o decorador da dataclasses para ativar ou desativar métodos.

* **asdict e astuple em dataclasses (269)**

A função asdict gera um dicionário com os dados da metaclass;

A função astuple gera uma tupla com os dados da metaclass.

* **Valores padrão, field e fields em dataclasses (270)**

Não é possível passar valores padrão a atributos com tipo mutável;

field é uma função para configurar os atributos;

fields é uma função que retorna informações sobre os atributos.

* **namedtuple – tuplas imutáveis com nomes para valores (271)**

name tupla -> são tuplas com nomes para valores;

São usadas para criar classes de objetos que são apenas um agrupamento de atributos, como classes normais sem métodos;

As classes devem herdar de namedTuple do modulo typing, mas sua estrutura no corpo é igual da dataclasses.

* **Criando sua própria lista com Iterable, Iterator e Sequence (collections.abc) (272)**

O modulo collections.abc, possui centenas de classes para servir de base de protocolos Python.