**PEP 08**

O PEP - é Python EnhancementProposals (Propostas de aprimoramento do Python). Ou seja, um índice que tem todas as melhoras do Python no website do Python. Lembre-se que o Python é open source. Essa PEP nº08 foi criada para auxiliar de como escrever código bonitos no Python, de maneira clara, simples, elegante.

**DIR E HELP**

O comando “DIR” nos dá uma lista de todas as possibilidades que uma variável pode ter de métodos, propriedades,etc.

O comando help nos ajuda como podemos usar e explicar como usar as funções e métodos, propriedades. Ex: help(tipo de dado.propriedade)

**Variáveis e tipos de dados em Python**

* Tipo numérico:

Int ou float.

* Tipo Float:

O separador de casas decimais em python é o ponto e não a virgula.

* Tipo Booleano:

É True ou False.

* Tipo String:

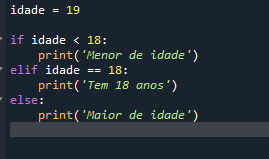
É uma cadeia de caracteres.

O **\n** serve para pular uma linha dentro de uma string.

* Estruturas lógicas em Python   
  (If, else, elif):

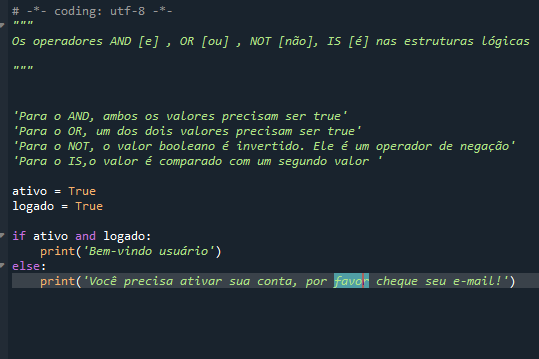
Utilizamos nosso programa mais inteligente com essas estruturas condicionais pois podemos dar caminhos diferentes para resolução de problemas.

Segue uma estrutura condicional de Python onde, neste caso, se idade for menor que 18 escrevemos na tela “Menor de idade”, se a idade for igual a 18 escrevemos na tela “Tem 18 anos” e se a idade for maior que 18 escrevemos na tela “Maior de idade”.



* AND, OR, NOT, IS em Python:

Com esses operadores lógicos podemos usá-los nas estruturas condicionais do Python, como usei no exemplo abaixo:



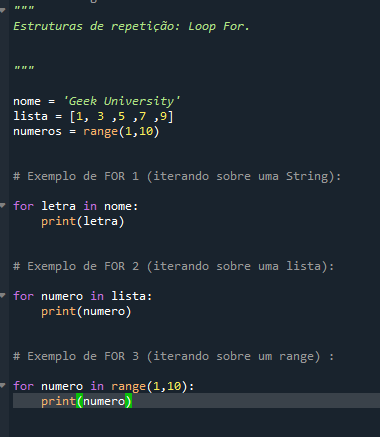
* Estruturas de repetição no Python (Loop, For):

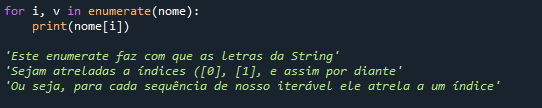
O Loop é uma estrutura de repetição e o For é uma dessas estruturas. Utilizamos o Loop para iterar sobre sequências ou sobre valores iteráveis.

Um valor iterável é um valor que podemos varrer, passar seus elementos. Podemos iterar por cada item de seu objeto. Exemplo de iteráveis:

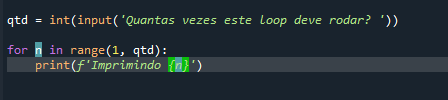
* String – nome = ‘Geek University’
* Lista - lista = [1, 2 , 3, 4, 5]
* Range – números = range(1, 10)

Vejamos um código parta termos uma idéia de como funciona:

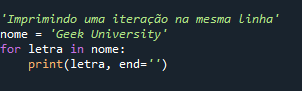




No código abaixo eu atrelo a uma variável um valor imputado por mim e com isso, para cada n dentro da range entre 1 e o valor a mim imputado, imprima a mensagem:



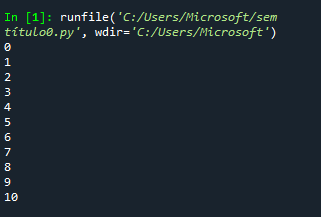
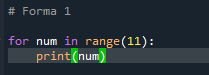
Para imprimir as iterações na mesma linha:

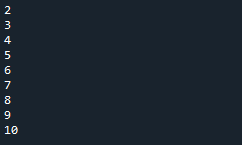
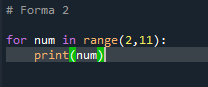


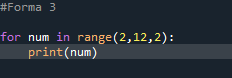
* Conhecendo o range:

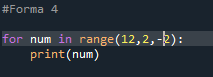
Precisamos conhecer o loop para usar os ranges e precisamos conhecer o range para entender o loop for.

Ranges são utilizados para utilizar sequências numéricas, não de forma aleatória mas sim de maneira especificada. Formas gerais:









Para converter um range para uma lista, temos um macete: list(range(1,10)), por exemplo.

* Conhecendo o Loop While:

Em uma estrutura While é importante que tenhamos um comando de parada no bloco para não causarmos um loop infinito sem sair dele.

* Coleções em Python (listas, tuplas, dicionários, mapas, conjuntos, módulos collections):

Coleções são a parte mais importante do Python para realizar machinelearning, data science, etc.

Agora veremos um pouco mais sobre as **Listas:**

Listas em Python funcionam como vetores ou matrizes, ou seja, arrays em outras linguagens, com a diferença de serem **dinâmicos** e podermos colocar **qualquer tipo de dado. Lembrando que em outras linguagens, como JAVA e C, quando crio um array eles possuem tamanhos e tipos de dados fixos. Ou seja, quando crio nessas linguagens um array com tipo int e com tamanho 5, será sempre do tipo inteiro e sempre no máximo 5 valores, coisa essa que é dinâmico no Python.**

As listas são dinâmicas por que não existe tamanho fixo, eu simplesmente crio a lista e simplesmente vou adicionando elementos. Também não possuem tipo de dado fixo, podemos colocar qualquer tipo de dado.

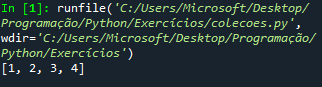
As listas em Python são representadas por colchetes

**“[]”**

Vamos ver um exemplo de listas em python:



Aqui eu crio uma variável com uma lista dentro e printo a lista na tela, tendo o seguinte resultado no console:



Aqui abaixo agora nós convertemos um range de 0 a 10 para uma lista:





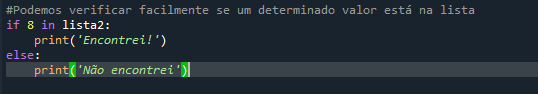
Agora eu transformo uma String em uma lista:



Assim, cada letra da string será separado para um item de lista:

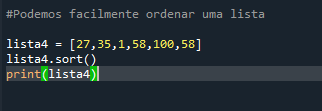


Podemos também identificar facilmente se um determinado valor está contido dentro de uma lista:





Também podemos ordenar de uma maneira fácil uma lista, conforme exemplo abaixo:



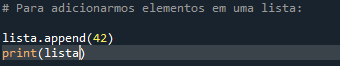


Podemos também facilmente contar o número de ocorrências de um valor em uma lista:





Se quisermos adicionarmos elementos em uma lista, podemos utilizar o seguinte código:





Com o append nós só conseguimos adicionar um elemento por vez, ok?

Com o “extend”, eu consigo fazer a mesma coisa que o append, porém agora eu consigo extender a lista colocando outra lista, ou seja, consigo colocar mais elementos de uma vez:





**O extend não aceito valor único. Ou seja, para adicionar um valor único em uma lista utilize o append, para adicionar vários elementos dentro de uma lista utilize o extend.**

Como vimos, estes valores inseridos com append e extend sempre vai para o final das listas, para adicionarmos em outras posições que não sejam o final, podemos utilizar:



Ou seja, eu utilizo o insert para inserir um valor na lista escolhendo a posição de inclusão. Dentro do insert eu escolho primeiro qual será a posição do meu novo valor e depois qual será o valor a ser incluído.

O insert é mais prático e dinâmico que o append e o extend. Prefira usá-lo para inserir elementos dentro de uma lista.

Agora para podermos facilmente juntar duas listas, podemos fazer da seguinte forma:





Acima incluímos duas listas em uma só.

Podemos facilmente inverter uma lista utilizando reverse:





Podemos também facilmente copiar uma lista:



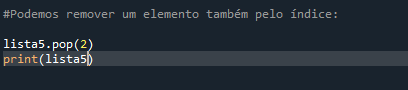
Para sabermos o tamanho de uma lista, podemos utilizar o seguinte comando:



Para removermos facilmente o último elemento de uma lista:



Podemos também remover um elemento pelo índice:

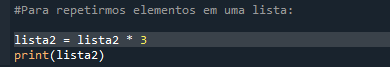


Para removermos todos os elementos da lista, usamos:



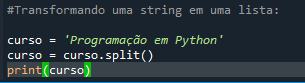


Multiplicando a lista por um número, eu repito a lista por quantas vezes eu quiser:





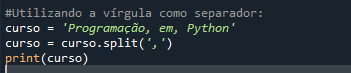
Podemos facilmente também converter uma string para uma lista:





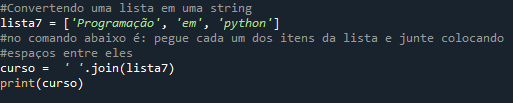
Por padrão o split utiliza o espaço como separador para a lista.

Agora para utilizar um separador específico, como a vírgula:





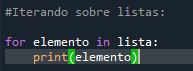
Agora convertendo uma lista em uma String:

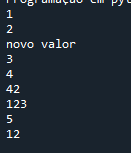




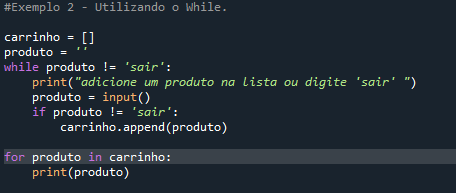
Agora se quisermos iterar sobre listas, como exemplo:

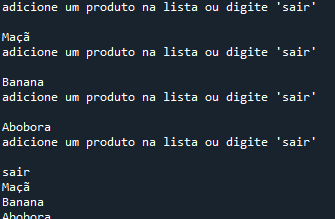
Exemplo1>



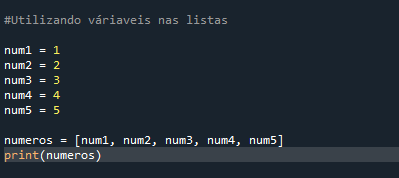


Exemplo2>





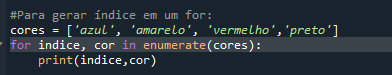
Podemos também criar listas com variáveis, não somente valores fixos:

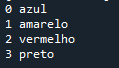


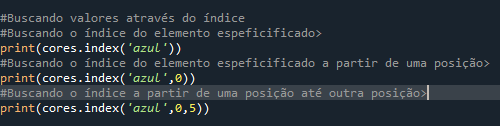


**Lembre-se que em listas, fazemos acessos aos elementos de forma indexada.**

Para gerarmos um par de chave e valor com as listas:





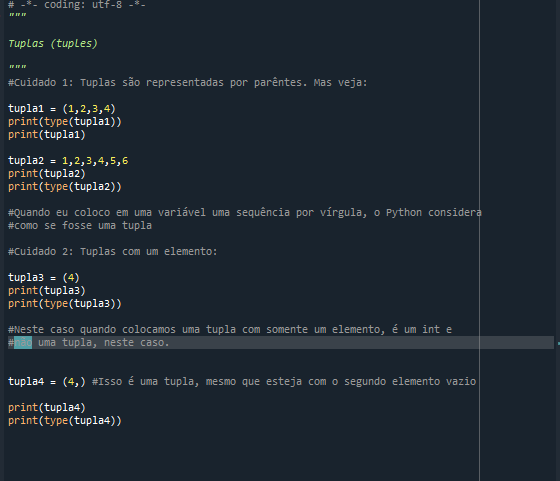
Podemos buscar valores dentro de uma lista de acordo com o índice:

Agora veremos um pouco mais sobre **tuplas:**

Tuplas são bastante parecidas com listas. Existem basicamente duas diferenças básicas entre lãs:

* As tuplas são representadas por parênteses “()”. As listas são representadas por colchetes “[]”.
* **As tuplas são imutáveis, ao contrário das listas. Isso significa que ao se criar uma tupla ela não muda. Toda operação em uma tupla gera uma nova tupla.**

Veja alguns exemplos sobre tuplas:



Podemos definir como conclusão que tuplas são definidas pela vírgula e não pelo uso do parênteses.

Ou seja:

4, > é tupla;

4 > não é tupla;

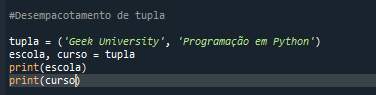
(4,) > é tupla.

Conseguimos também transformar um range em uma tupla com o código abaixo:





Podemos também fazer o desempacotamento de tupla:

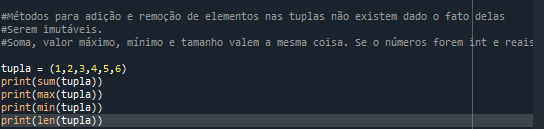




Ou sejo no código acima eu declarei a tupla com dois valores. Depois, usando a mesma ordem dos elementos da tupla, eu atribui dois nomes a estes elementos da tupla: o primeiro elemento “Geek University” recebeu o nome “escola” e o segundo elemento “Programação em Python” recebeu o nome “curso”. Com isso eu posso usar esses nomes para trabalhar com cada elemento da tupla, neste caso eu usei estes nomes para imprimir os elementos na tela do computador.

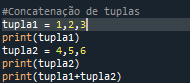
Gerará erro se colocarmos um número de elementos diferentes para desempacotar.

Lembre-se que não há métodos para adição e remoção nas tuplas pois as tuplas são imutáveis. Porém conseguimos realizar cálculos com os valores das tuplas se forem inteiros e reais.





Para fazermos a concatenação de tuplas, fazemos da seguinte forma:

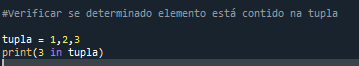




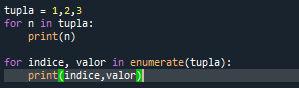
Mesmo após a concatenação, a tupla1 e a tupla2 do exemplo não foram alteradas pois tuplas são imutáveis.

Tuplas são imutáveis porém podemos sobrescrever seus valores.

Podemos também identificar se um valor está contido na tupla, assim como na lista:



Agora veremos como iterar sobre uma tupla:



Usando o **count**podemos contar quantos vezes um elemento aparece na tupla:



Lembre-se sempre que podemos utilizar tuplas sempre que queremos que não mudem os dados dentro de uma coleção, como por exemplo, meses de um ano.

Agora veremos mais sobre os **dicionários:**

Em algumas linguagens de programação os dicionários são conhecidos como mapas, pois os dicionários são coleções por chave e valor, ou seja, um mapeamento por chave e valor.

Dicionários no Python são representados por chaves { }.

Ou seja:

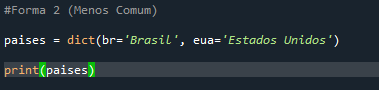
* Tuplas: representados por ();
* Listas: representados por [];
* Dicionários: representados por {};



Chave e valor são separados por dois pontos e cada elemento do dicionário são separados por vírgula.

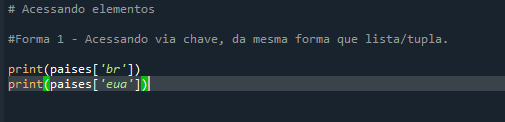
Nos dicionários podemos misturar tipos de dados, elas podem receber tanto na chave quanto no valor qualquer tipo de dado.

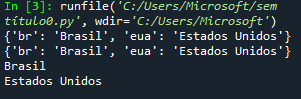
Vejamos agora outra forma para criação de dicionários:



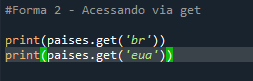
Para acessar elementos dentro de um dicionário:

A primeira forma podemos acessar via chave:

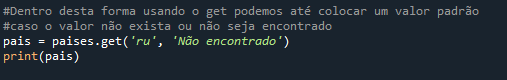




Podemos acessar da forma mais recomendada, via “get”:



Usando o get posso colocar um valor padrão também caso a chave não seja encontrada, como no exemplo abaixo:

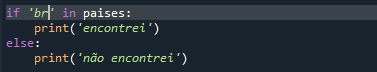


Podemos verificar também se determinado item está dentro do dicionário:

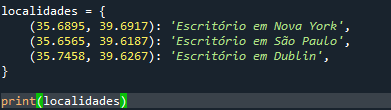




Um exemplo usando isso:

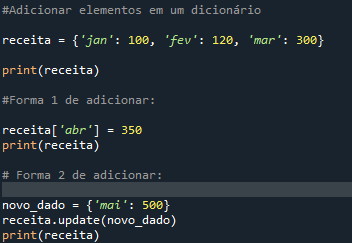


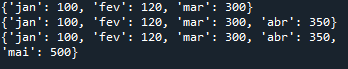
Podemos utilizar qualquer tipo de dado (int, float, string, boolean, etc), inclusive lista, tupla, dicionário, como chaves de dicionários:



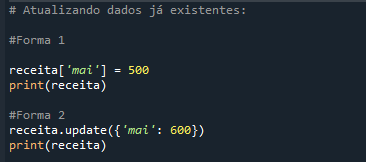
Acima temos um dicionário que possui tuplas como chaves. Neste caso até é seguro usar tupla como chave pois a tupla a gente não modifica como as listas. Dentro estas tuplas temos tipos de dados floats.

Podemos também adicionar elementos em um dicionário:



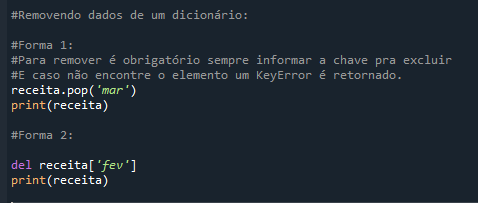


Com o update tanto posso adicionar novos elementos, como acima, no dicionário mas também posso atualizar dados já existentes no dicionário:

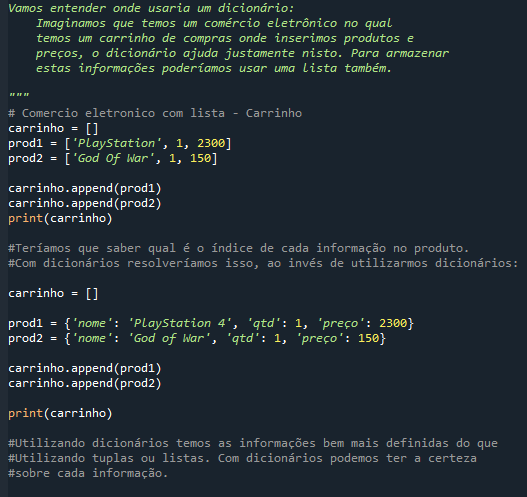


Ou seja, a forma de adicionar novos elementos no dicionário e atualizar elementos já existentes no dicionário é a mesma. Em dicionários, não podemos ter chaves repetidas.

Veremos agora como remover dados de um dicionário:

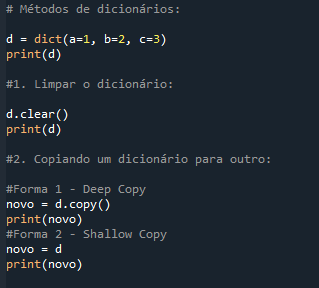


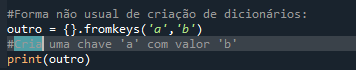
Vamos entender onde usaria um dicionário: Imaginamos que temos um comércio eletrônico no qual temos um carrinho de compras onde inserimos produtos e preços, o dicionário ajuda justamente nisto. Para armazenar estas informações poderíamos usar uma lista também.



As coleções cada um tem suas particularidades (listas, tuplas, dicionários), elas se completam, cada uma oferecendo o que cada uma tem de melhor.

Agora vamos conhecer um pouco mais dos métodos de dicionários:







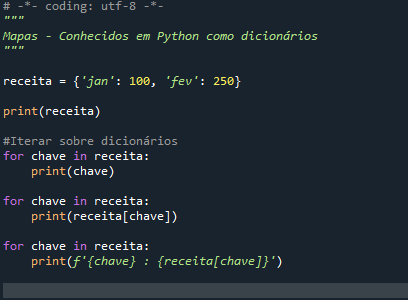


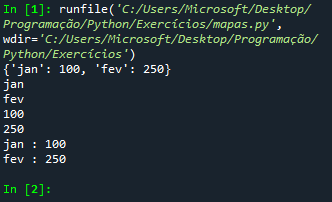


O método fromkeys pede chaves e atribuirá depois valores a estas chaves. No exemplo acima uma lista no primeiro parâmetro significa todas as chaves, no segunda parâmetro “desconhecido”, todos as chaves receberão este valor.

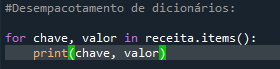
Agora veremos sobre Mapas, conhecidos como dicionários, são representados por {}:

Vejamos agora como iterar sobre os mapas ou dicionários, como quiser chama-lo:





Podemos também desempacotar os itens de um dicionário:





Agora estudaremos um pouco mais sobre os conjuntos em Python:

Quando falamos sobre conjuntos em qualquer linguagem de programação estamos fazendo referência a teoria dos conjuntos que estudamos em matemática.

Aqui no Python os conjuntos são chamados de Sets:

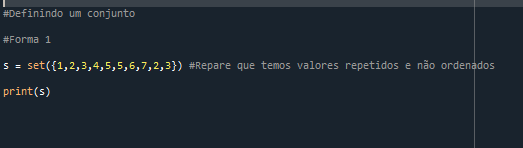
**Estes Sets (Ou conjuntos) não possuem valores duplicados, valores ordenados e elementos não são acessados via índice, ou seja, conjuntos não são indexados ...**

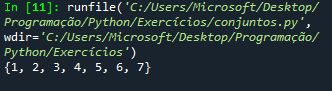
**Conjuntos são bons para se utilizar quando precisamos armazenar elementos mas não nos importamos com a ordenação deles, quando não precisamos se preocupar com chaves, valores itens duplicados...**

**Os conjuntos ou Sets, são referenciados em Python chaves {} também como os mapas...**

**Mas há diferenças entre conjuntos (sets) e mapas (dicionários em Python:**

* **Um dicionário tem chave e valor;**
* **Um conjunto só tem valor, apenas isso;**

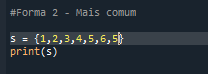




**As repetições acima não foram adicionadas. Não gere erro mas também não são adicionadas.**

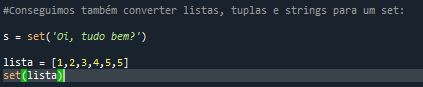
**Caso haja algum valor já existente, o mesmo será ignorado e não fará parte do conjunto.**

Vejamos agora segunda forma:





Conseguimos converter Tuplas, Listas e Strings para Conjunto(set) da seguinte forma:



**#Vale sempre a pena lembrar:**

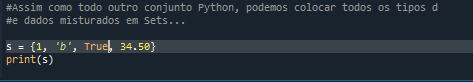
**#Listas aceitam valores duplicados;**

**#Tuplas também aceitam valores duplicados;**

**#Dicionários não aceitam chaves duplicadas;**

**#Conjuntos não aceitam valores duplicados.**

Assim como todo outro conjunto Python, podemos colocar todos os tipos de dados misturados em Sets...



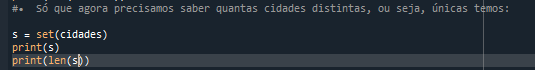
Alguns usos interessantes com Sets:

* Imagine que fizemos um sistema de cadastro de uma feira ou museu e os visitantes informaram manualmente cidade de onde vieram. Nós adicionamos cada cidade em uma lista Python, já que em uma lista podemos adicionar novos elementos e ter repetição.



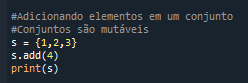


* Só que agora precisamos saber quantas cidades distintas, ou seja, únicas temos:





Vejamos como adicionar elementos em um conjunto agora, conjuntos são sim mutáveis:



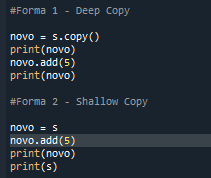


Podemos remover elementos de um conjunto também. Para removermos informamos o valor a ser removido e não o índice, pois os conjuntos não são indexados:

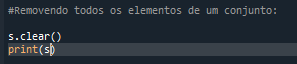




Agora veremos como copiar um conjunto para o outro:

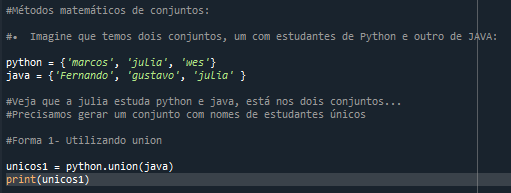


Podemos remover todos os itens de um conjunto:



Temos também alguns métodos matemáticos de conjuntos:

* Imagine que temos dois conjuntos, um com estudantes de Python e outro de JAVA:





Podemos unir de acordo com outra forma também:



Agora gerando um conjunto de estudantes que estão em ambos os cursos:

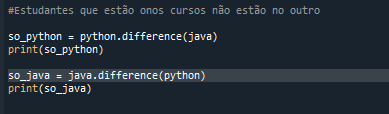




Podemos fazer isso usando outra forma:



Agora vamos gerar um conjunto de estudantes que não se repetem nos dois cursos:



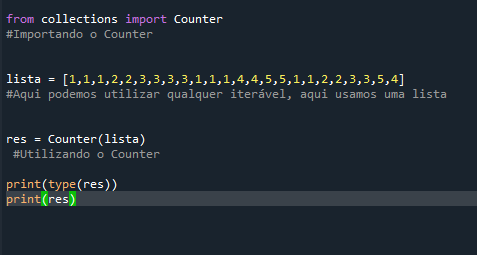


Agora falaremos um pouco mais sobre **Módulo Collections**:

O Python tem um modelo chamado collections que podem ser usados como coleções.

**Vamos conhecer o Counter deste Módulo Collections:**

* **Ele é conhecido por High Performance Container Datatypes, ou seja, tipos de dados de container de alta performance, significa que todas as coleções que estamos estudando são containers, pois dentro deles posso adicionar vários elementos e dentro do módulo collections nós temos alguns tipos extendidos destas coleções que estudamos.**
* **Counter seria algo como um contador, ou seja, ele é bom para contar elementos que atribuímos a ele. Ele recebe um iterável como parâmetro e cria um objeto do tipo CollectionsCounter que é parecido com um Dicionário, contendo como chave o elemento da lista que a gente passou como parâmetro e como valor a quantidade de ocorrências desse elemento na lista.**
* **Vejamos um exemplo:**

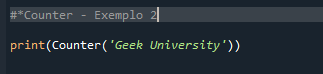




Ele criou acima o tipo Counter contando os valores de cada elemento da lista que eu passei. O tipo impresso é uma classe.

Veja que para elemento da lista, o Counter criou uma chave e colocou como valor a quantidade de ocorrências.

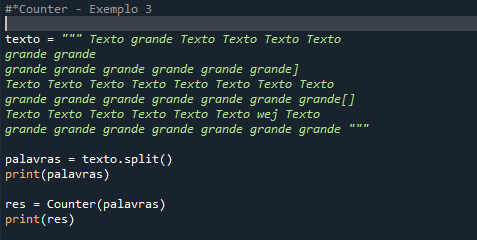
Vejamos agora um segundo exemplo do Counter:

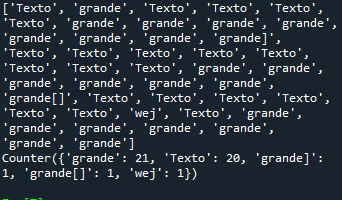




Para cada um dos caracteres ele colocou como uma chave e o número de ocorrências de cada chave.

Vejamos agora o exemplo 3 do Counter:



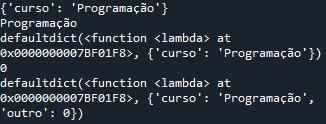
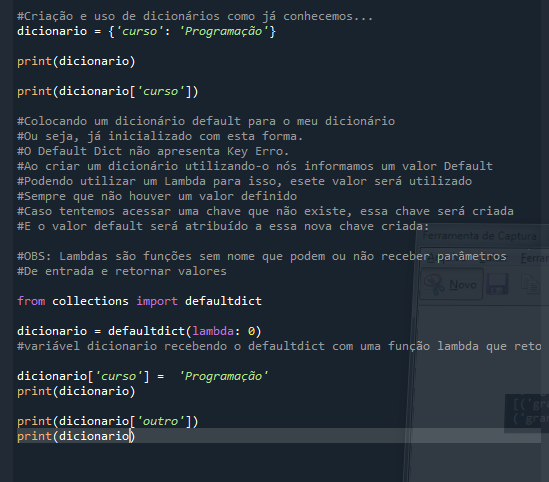


O Counter é muito usado para a linguagem natural, como por exemplo, analisar as cinco palavras que mais se repetem em um texto de um livro:



Agora veremos um pouco mais sobre Módulo Collections: **Default Dict**:

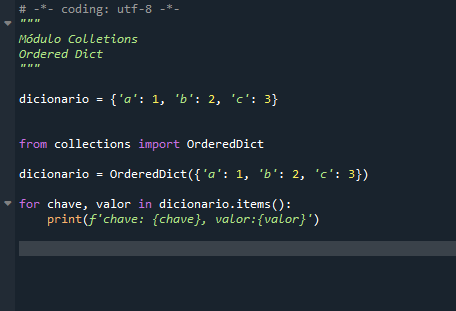
Ele é do módulo collections ainda, relacionado a dicionários. Tudo que vimos em dicionários vale para o Default Dict... Mas existe uma diferença:



No código acima atribuindo uma função lâmbda que retorna 0 ao nosso defaultdict do “dicionário”, sempre que usamos ou atribuímos uma chave inexistente ao dicionário, não retornará mais KeyError e sim criará uma nova chave (esta chave inexistente) e o valor será o default que passamos (0).

Agora veremos sobre o Módulo Collections: **OrderedDict**:

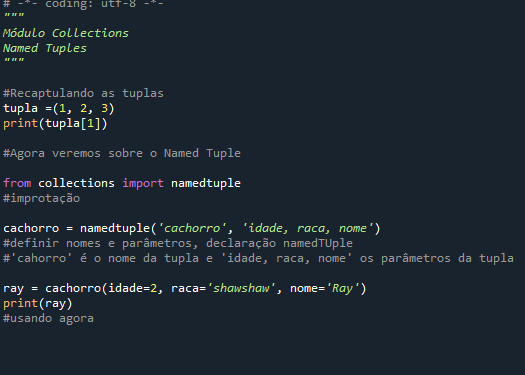
Em um dicionário a ordem de inserção dos elementos não é garantida, com o OrderedDict conseguimos garantir essa ordem:



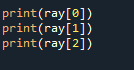
Agora vamos falar sobre Módulo Collections: **NamedTuple**:

Neste caso vamos falar agora algo que tem haver com as Tuplas. Significa uma tupla nomeada, ou seja, imagina que nós colocássemos chave e valor para os elementos da tupla, nomes para os valores da minha tupla. São tuplas diferenciadas onde especificamos um nome para a mesma e também parâmetros.

Por exemplo:

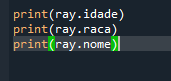




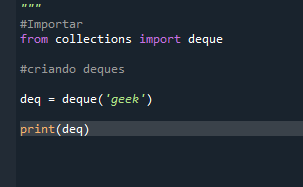


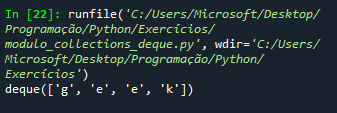


Acima vimos também como acessar os dados. Podemos acessar também pelos nomes dos atributos que dei no namedTuple:

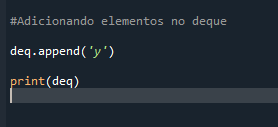


Agora veremos sobre Módulo Collections: **Deque:**

Podemos dizer que o Deque é uma lista de alta performance. 



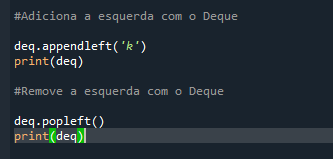
Acima o deque imprimiu cada uma das palavras que passei em uma lista.





Acima adicionamos um elemento a nossa lista deque, assim como fazemos normalmente nas listas usando o “append”.

O que muda de uma lista normal é que podemos remover e adicionar elementos também no início, não somente no final como é na lista normal:

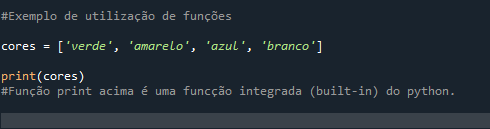


* Funções em Python:

Primeiramente vamos aprender como definir funções em Python...

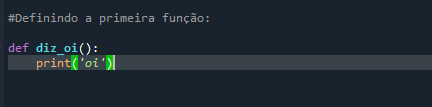
Funções é criar pequenas partes do nosso código para executar uma tarefa específica. Uma função pode ou não receber entradas de dados e retornar uma saída de dados. São muito úteis para realizar procedimentos similares repetidas vezes. Se você escrever uma função que realiza várias tarefas dentro dela, é bom fazer uma verificação para que a função seja simplificada. Procure escrever funções simples e objetivas.

Exemplo de utilização de funções:



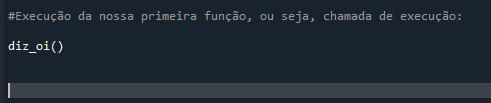
Agora veremos como definimos funções. Em Python, a forma geral de definir uma função é: **defnome da função**(**parâmetros de entrada**): **bloco de comando.**

**Lembre-se que o nome da função sempre será com letras minúsculas, e se nome composto, separado por underline (ex: nome\_da\_funcao). Parâmetros de entrada são opcionais e separados por vírgulas. O bloco de comando é onde o processamento da função acontece, nesse bloco pode ter ou não retorno da função. Veja um exemplo:**

****

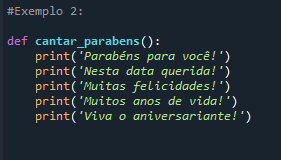
**Veja que dentro das nossas funções podemos utilizar outras funções, como é o caso da minha função acima. Veja também que nossa função só executa uma tarefa, ou seja, a única coisa que ela faz é dizer oi. Veja que esta função acima também não recebe nenhum parâmetro de entrada. Veja também que esta função não retorna nada, somente executa o print dentro dela.**

Agora vamos ver como executamos a função utilizando-a, onde precisamos chamá-la:

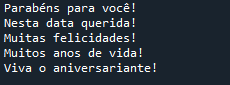




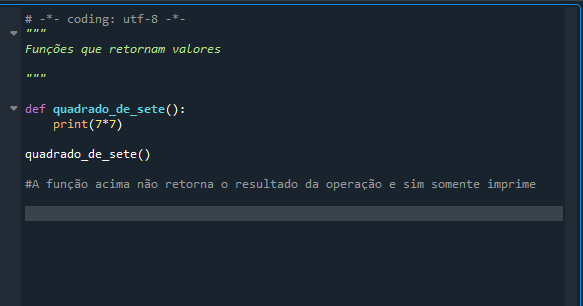
Vejamos agora um segundo exemplo de definição e execução de função:

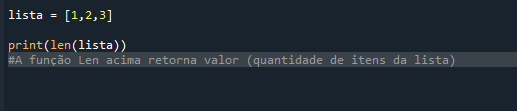




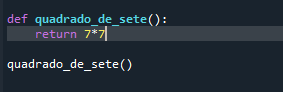


Veremos agora sobre as funções que possuem retorno:



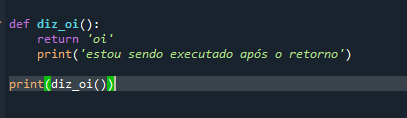


Em python, quando uma função não retorna nenhum valor, o retorno é dado como None. Funções Python que retornam valores, devem retornar estes valores com a palavra reservada “**return**”:

****

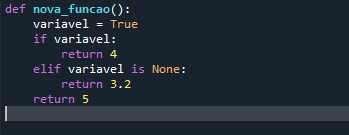
Temos algumas observações sobre a palavra reservada “return”:

* Ela finaliza a função, ou seja, **ela sai da execução da função.**

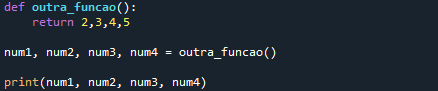




* Podemos ter em uma função com diferentes “returns”, ou seja, em uma função podemos ter mais de um return, porém a função somente retornará um deles.



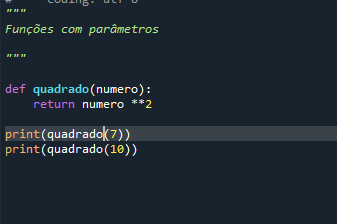
* Podemos, em uma função, retornar qualquer tipo de dado e até mesmo múltiplos valores.





Agora veremos sobre funções com parâmetros:

Geralmente funções recebem dados de entrada, processam esses dados e depois retornam dados de saídas, como uma máquina. Vejamos abaixo um exemplo de função que tem um parâmetro de entrada:

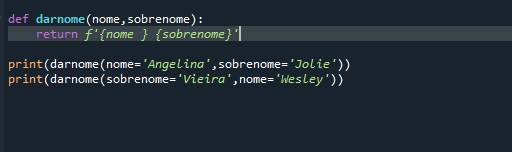


As funções podem ter vários parâmetros de entrada quanto forem necessários para nós. Eles são separados por vírgulas.

Se for informado um número errado de parâmetros ou argumentos da função teremos TypeError.

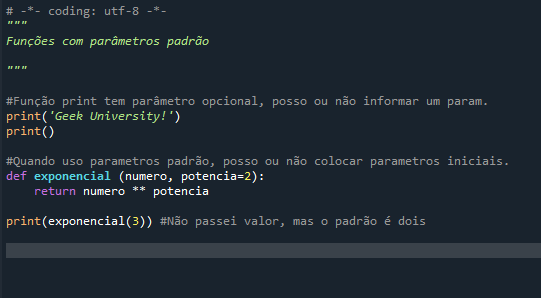
Lembre-se sempre que a ordem dos parâmetros sempre importa!

Argumentos Nomeados (KeyWordsArgs): Caso utilizemos nomes dos parâmetros definidos nos argumentos informados, podemos utilizar qualquer ordem:



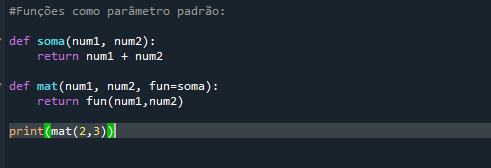


Agora veremos sobre a utilização de parâmetros padrões nas funções:



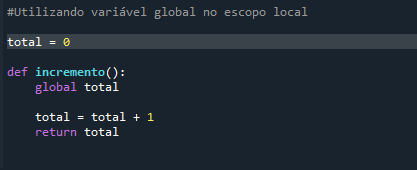


Agora vamos ver quando passamos funções como parâmetro padrão:

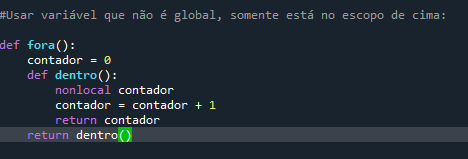


Não passei o parâmetro da função, porém como padrão será executado a soma.

Veja abaixo quando precisamos usar variável global no escopo local:

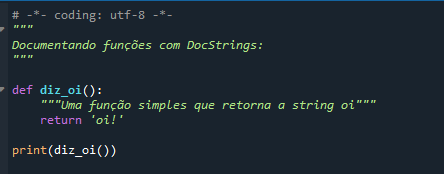


Usar variável que não é global, somente está no escopo de cima:



Vejamos agora como documentamos funções com DocStrings:

A partir do momento que escrevemos nosso código, é importante escrever comentários (DocStrings) no nosso código, o que chamamos de documentação. É para pensarmos futuramente de forma mais fácil, só bater o olho e entender o que queria ser feito.



Consigo acessar a documentação de uma função utilizando o terminal deste jeito, por exemplo: **diz\_oi.\_\_doc\_\_**

Podemos também fazer acesso a documentação de uma função utilizando a função **Help().**

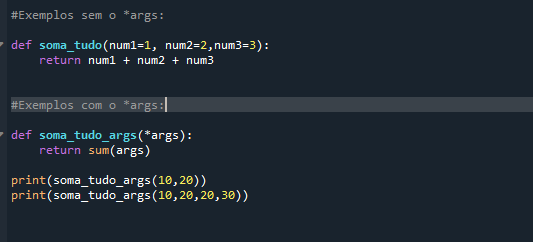
Agora veremos sobre como entender os **\*args**:

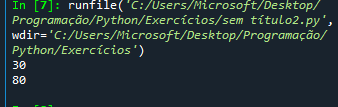
Ele é um parâmetro de entrada de uma função como outro qualquer, eu posso chamá-lo de qualquer coisa, desde que comece com asterisco.

Mas, por convenção, foi adotado o nome args, ficando \*args.

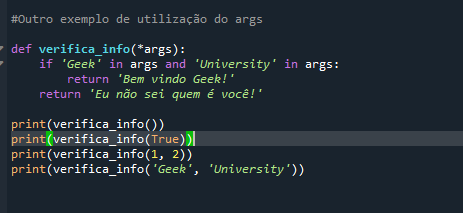
O parâmetro \*args é utilizado em função, coloca os valores extras informados como entrada em uma tupla, então desde já lembre-se que tuplas são imutáveis.

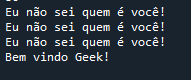
Com os args posso colocar quantos parâmetros de entrada forem necessários, mais ou menos:



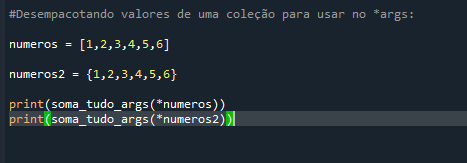


Vejamos outro exemplo de utilização do args:



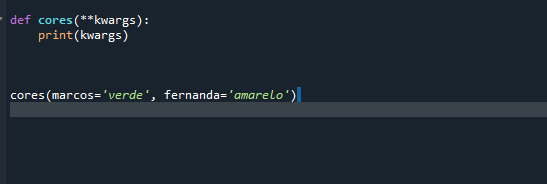


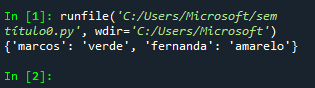
Desempacotando valores de uma coleção para podermos usar como args:

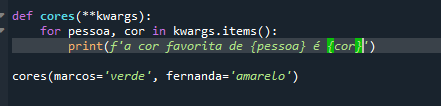


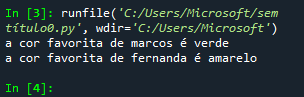
Agora veremos sobre os **\*\*kwargs**:

Ele força que utilizemos parâmetros nomeados e transforma esses parâmetros extras em um dicionário.



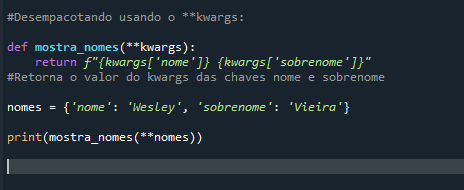






Os parâmetros args e kwargs não são obrigatórios. Então, tanto faz se informarmos nada ou vários elementos. Nas nossas funções podemos ter parâmetros obrigatórios, \*args, parâmetros default (não obrigatórios) e \*\*kwargs e eles devem estar nesta ordem em nossas funções.

Vejamos agora abaixo como desempacotar usando o \*\*kwargs:

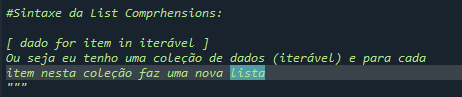




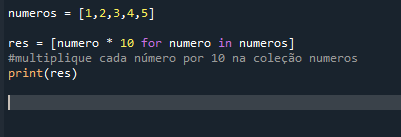
**Os nome das chaves em um dicionário devem ser os mesmos dos parâmetros da função.**

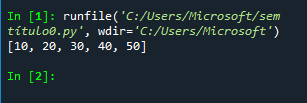
* Comprehensions em Python:

Os **listcomprehensions,** utilizando ela, nós podemos gerar novas listas com dados processados a partir de outro iterável.



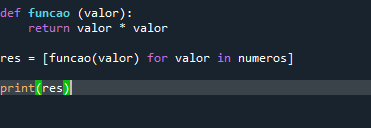
Vejamos um exemplo de como usar:

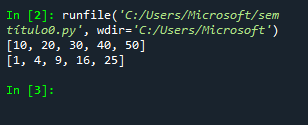




É muito parecido com o map, filter, reduce do JavaScript onde pegamos um iterável, fazemos um processamento e levamos ele para uma nova lista com seus resultados.

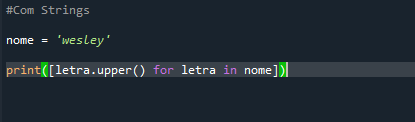
Posso usar funções também dentro do meu listcomprehensions:





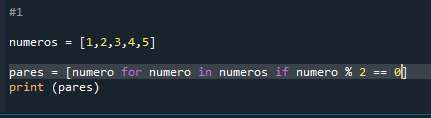
Compreender listcomprehensions te garante que você conhece o Python avançado pois com ele você não precisa fazer o Loop que tem muito mais código e digitação.

Usando o listcomprehension com String:

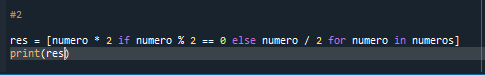




Nós podemos adicionar estruturas condicionais lógicas às nossas ListComprehension:



Explicação do código acima: Mesma estrutura somente adicionando no final a estrutura condicional (IF ou IF NOT).

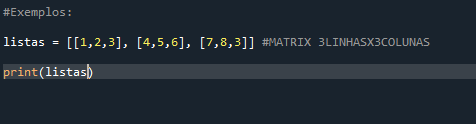


Explicação do código acima: cada numero multiplica por 2 se o número for par, caso o contrário divide o numero por 2 para cada numero na minha lista.

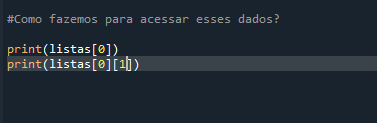
Veremos agora sobre **listas aninhadas:** Algumas linguagens de programação possuem uma estrutura de dados chamadasdeArrays, ou seja arranjos, estes que podem ser unidimensionais (Arrays ou vetores) ou multidimensionais (Matrizes), conforme vimos em Java.

No Python não existe Arrays, qualquer que seja dimensão. No Python nós conhecemos as Listas.

Vamos ver alguns exemplos destas listas aninhadas (NestedLists):

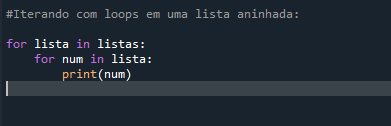


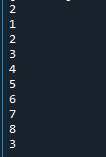
E como fazemos para acessar os dados desta lista aninhada?



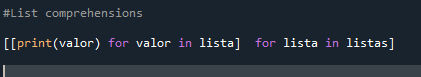


Como iteramos com loops em uma lista aninhada?





E iterando usando listcomprehensions?



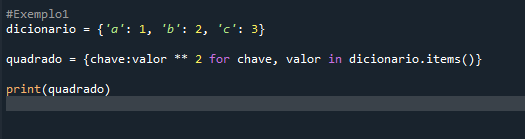
Agora veremos sobre **DictionaryComprehensions:**

Pense no seguinte: se quisermos criar uma lista fazemos, lista = [1,2,3,4], se quisermos criar uma tupla, fazemos: tupla = (1,2,3,4) ou 1,2,3,4, se quisermos criar um set (conjunto): conjunto = {1,2,3,4}, se quisermos um dicionário: dicionário = {‘a’: 1, ‘b’: 2, ‘c’: 3}

Vamos ver agora a sintaxe no dictionarycomprehension:

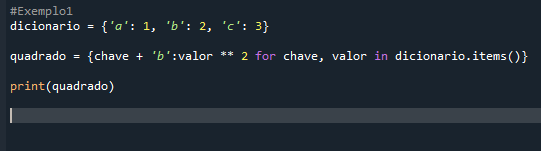
{chave: valor for valor in iterável}

Vamos ver um exemplo:



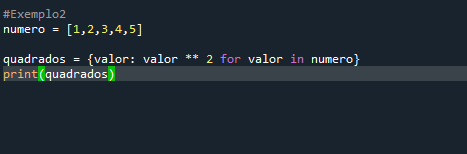


Outro exemplo, não só alterando o valor, mas também alterando a chave:



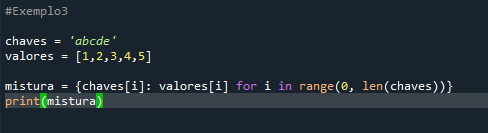


Vejamos mais um exemplo, agora utilizando valores de uma lista:





Vamos ver maaaaais um exemplo:



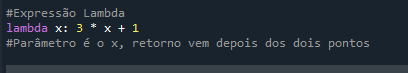


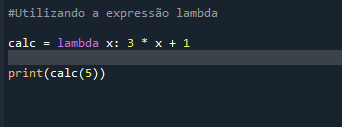
* Funções e Expressões Lambdas:

Vamos aprender agora como usar **lambdas**:

**Elas são funções sem nomes, ou seja, são anônimas.**

Toda expressão lambda em python começa com a palavra reservada “lambda”:

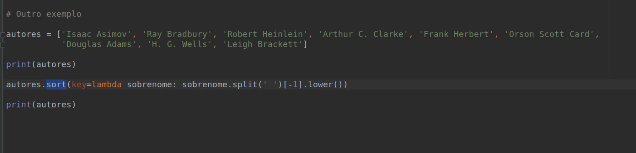




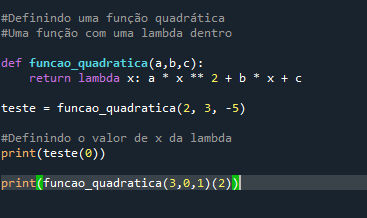
Podemos ter expressões lambdas com múltiplas entradas.

Em funções Python podemos ter nenhuma ou várias entradas, em lambdas também.

Veremos um exemplo abaixo que vai ordenar uma lista de nomes em ordem alfabética com base no sobrenome:



O parâmetro opcional “key” do “sort” é uma forma de ordenarmos do jeito que queremos. Utilizamos a expressão lambda para nossa forma de ordenação, onde a cada sobrenome nós separamos com base nos espaços, pegamos o último nome (“[-1]”) e depois colocamos em letra minúscula.

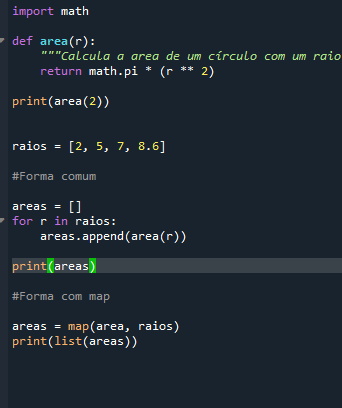


**Aqui acima eu tenho uma função que retorna uma lambda. Para usar, eu coloco primeiro os parâmetros da função e depois coloco o valor dos parâmetros da lambda.**

Veremos agora como usar o **MAP**:

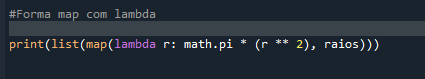
Com map, fazemos mapeamento de valores para função. Vamos ver um exemplo:

Map é uma função que recebe dois parâmetros, o primeiro a função e o segundo o iterável:



O map pega cada um dos iteráveis e coloca como parâmetro na função que passamos.

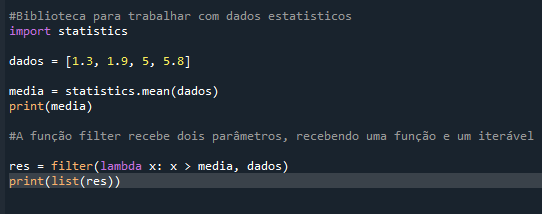
Mas geralmente não utilizamos uma função comum no map, utilizamos uma função lambda:



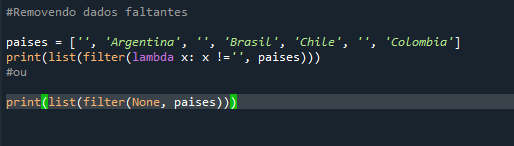
Vejamos agora como usar o **FILTER**:

No filter nós filtramos os dados de uma coleção para outra, ou seja, ela serve para filtrar dados de uma determinada coleção. Imagina que tenho uma coleção, por exemplo, músicas tocadas e existe as listas de músicas e podemos realizar buscas e cada música já foi tocada várias vezes no mundo inteiro e eu tenho uma coleção de dados com o título da música e quantidade de vezes que ela foi tocada e eu quero fazer um top 10 de mais tocadas, pra isso eu preciso filtrar para agrupar somente as dez mais tocadas em uma seleção.

Vejamos na prática um exemplo:



Podemos usar o filter para remoção de dados faltantes:





Agora veremos o **REDUCE**:

Para utilizarmos o reduce devemos, agora, que importar e utilizar a partir do módulo **‘functools’**.

Para entendermos o reduce, **imagine que temos uma coleção de dados e você tem uma função que recebe dois parâmetros (função(x,y)) e retorna um valor.**Assim como map e filter, a reduce recebe dois parâmetros: função e dados.

Com isso, a função reduce() funciona da seguinte maneira:

1. **Res1 = função(dado1, dado2) -> aplica a função nos dois primeiros elementos da coleção e guarda o resultado;**
2. **Res2 = função(res1, dado3) -> resultado do calculo anterior e o próximo dado como parâmetros**

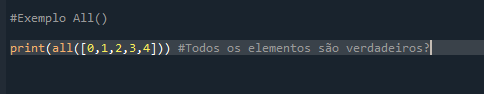
Esses passos são repetidos até o final da coleção, assim por diante, sempre pegando o resultado anterior.

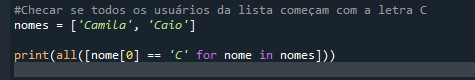
**EM CADA PASSO ELA APLICA A FUNÇÃO PASSANDO COMO PRIMEIRO ARGUMENTO O RESULTADO DA APLICAÇÃO ANTERIOR, NO FINAL REDUZ E IRÁ RETORNAR O RESULTADO FINAL.**

Veremos agora sobre Any e All:

São duas funções integradas do Python, não precisamos nem importa-las.

O All() retorno True se todos os elementos do iterável são verdadeiras ou ainda se o iterável estiver vazio.



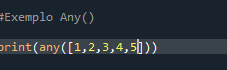




Explicando o código acima: para todos os nomes na lista, em cada nome na primeira posição se começa com a letra ‘C’.

O Any() retorna True se qualquer elemento do iterável for verdadeiro, se o iterável estiver vazio retorna Falso.

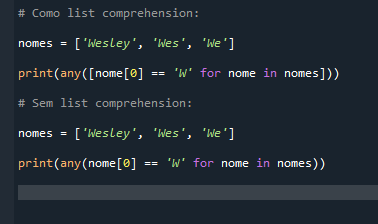
Então o All() retorna verdadeiro se todos do iterável for verdadeiro e o Any() retorna verdadeiro se algum do iterável for verdadeiro.



Veremos agora sobre os Generetors:

Generetors é o que chamamos de TupleGenerators...

Ou seja, é uma listcomprehension sem os colchetes:



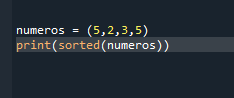
Os dois fazem a mesma coisa mas o Generetor ocupa menos recurso em memória, ele é mais performático.

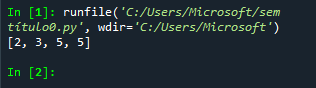
Se precisarmos da lista pra usar algo, usamos o ListComprehension, se formos somente usar para alguma checagem ou dentro de uma função é melhor usar o Generetor.

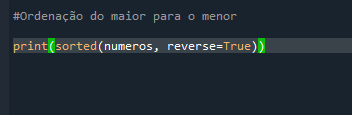
Veremos agora sobre a função **sorted():**

Não confunda a Função Sorted com Sort. A Função Sort só funciona em listas e a função Sorted funciona em qualquer iterável.

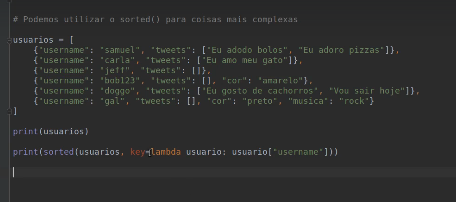
Como o próprio nome diz, Sorted serve para ordenar elementos de um iterável:





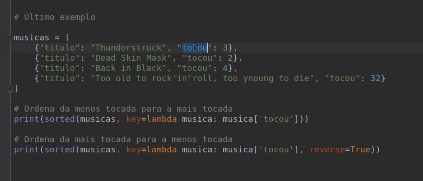


Após ordenar, o sorted retorna uma lista ordenada.

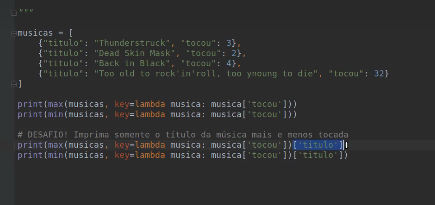


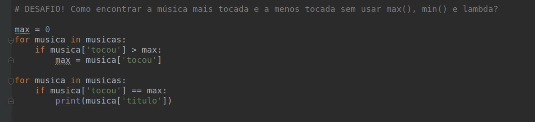
No exemplo acima podemos ordenar nossa lista de dicionários pela nome do usuário, na chave “usurname”.

Agora abaixo vamos ver uma ordenação de uma lista de músicas da menos tocada para mais tocada e vice-versa:



Agora veremos agora sobre o **Max e Min: Podemos usar o Max e o Min que aprendemos das listas em qualquer iterável também.**



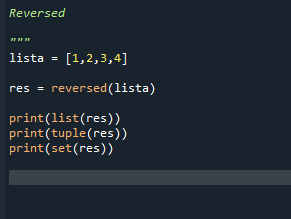


Veremos agora sobre o **Reserved:**

Não confunda com a função reverse que estudamos nas aulas de listas.

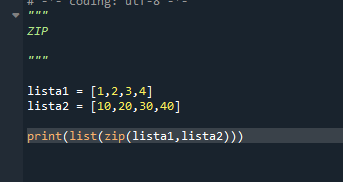
O objetivo é inverter o iterável. Retorna sempre um funçãolist\_reverseiterador.

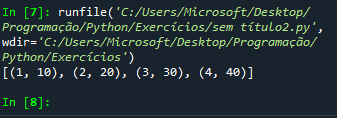
Para pegar o valor podemos reverter para uma lista, tupla ou conjunto:



Veremos agora sobre o **Zip:**

Ele cria um iterável que agrega elementos um dos iteráveis que foram passados como entrada em pares:





Podemos usar diferentes iteráveis com o ZIP.

* Tratamento de erros:

No Python temos os seguintes tipos de erros:

* IndexError: erro de indexação;
* ValueError: erro com os valores que usei;
* NameError: erro do nome dos elementos que usei;
* TypeError: erro nos tipos de dados que usei no meu código;
* SyntaxError: erro de sintaxe do código Python.
* KeyError: erro de chaves quando eu uso de maneira errada o dicionário, por exemplo, acessar um dicionário com uma chave que não existe;
* AttributeError: ocorre quando uma variável não tem um atributo ou função;
* IdentationError: erro de identação no Python.

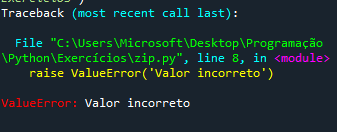
Tem uma lista enorme de erros no Python que podemos ler na documentação oficial da linguagem. E não podemos esquecer que exceções e erros são a mesma coisa em programação.

Agora veremos como vamos levantar nosso próprios erros usando o **Raise**:

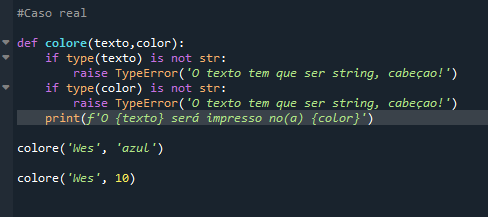
Não é uma função, é uma palavra reservada assim como o “def” o qualquer outra em Python que lança exceções.

Pense no raise como sendo útil para que possamos criar nossas próprias exceções e mensagens de erro.**A forma geral de utilização é: raisetipoErro(‘Mensagem de Erro’)**





Vejamos um caso real agora:



Lembrando que aqui não estamos tratando o erro, estamos somente lançando nossas próprias exceções. Agora vamos ver como tratar um erro:

No Python o bloco de comandos para tratamento de erros é o Try...Except...

Utilizamos este bloco para tratar erros que podem ocorrer no nosso código, prevenindo assim que o programa pare de funcionar e o usuário receba mensagens de erros inesperadas...

A forma geral mais simples é :

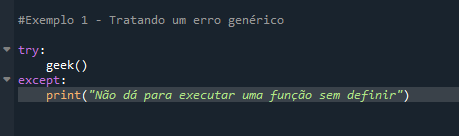
Try:

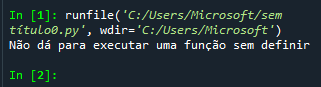
//Execução problemática

Except:

//O que deve ser feito em casos de problemas

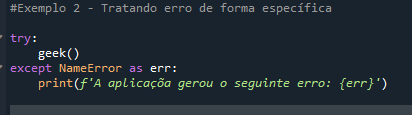
Vamor ver agora um exemplo prático para entendermos melhor:





Neste exemplo estamos tentando executar a função geek(), caso encontre erros, execute o print que coloquei no bloco excpet...

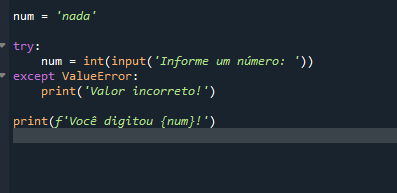
Agora, se quisermos tratar erros especificamente, não somente genéricos, fazemos da seguinte maneira:

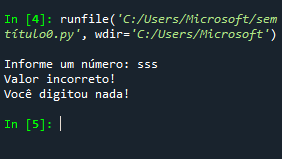




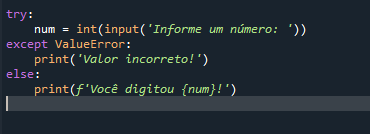
Dica para tratamento de código e seu erro:

* Toda entrada dos usuários devem ser tratadas, ou seja, tem entrada de dados, tem que ser tratada. **Sempre tenha em mente que a função do usuário é destruir seu sistema**:

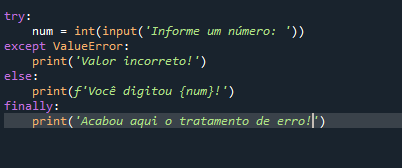




Veremos agora com o uso do Else... Ou seja, Tente (TRY) um código, se der erro (EXCEPT) executa uma função, caso não dê nenhum erro (ELSE) executa um código:

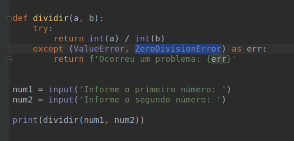


Veremos agora o Finally, que vai no final do tratamento do código... Ele serve para executar um código no final de toda a testagem do erro, meio que pra dizer que o tratamento de erro se encerrou... O bloco Finally é sempre executado independente se houve exceção ou não:



O Finally geralmente é utilizado para fechar ou para desalocar recursos, ou seja, as vezes você se conecta à um banco de dados e geralmente a gente abre uma conexão com banco de dados, utiliza o banco de dados e depois fecha essa conexão com o banco de dados... Com o finally podemos fechar a conexão com banco de dados, fechar um arquivo para leitura ou escrita, etc...

Para tratar erros, o ideal é tratar os erros dentro da própria função.



* Módulos Python:

Em Python Módulos nada mais são do que outros arquivos Python. Ou seja, cada arquivo Pyhton são módulos e importamos eles para dentro de nosso projeto para que possamos utilizar suas funcionalidades e funções.

Eles são úteis para que possamos deixar nossos programas mais simples e para que possamos reutilizar os códigos.

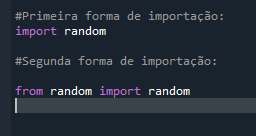
O Módulo Random faz parte dos módulos integrados no Pythons, ou seja, o Built-in. Ele possui várias funções para geração de números pseudo-aleatórios.

Ao realizar o import de todo o módulo, todas as funções, atributos, classes e propriedades que estiverem dentro do módulo ficarão disponíveis (ficarão em memória), o que não é recomendado.

Caso você saiba qual função que vc precisa do módulo, você importa conforme o seguinte item de importação abaixo.

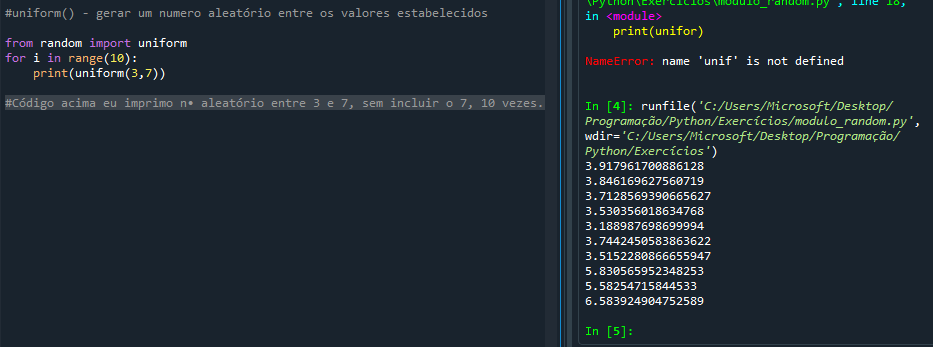
Existem duas formas de se utilizar um módulo ou função deste Random:

* **Importrandom**;
* **fromrandomimportrandom(importando função random do módulo random).**

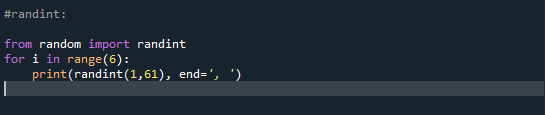


**Veja que para utilizar a função random do pacote random, nós colocamos o nome do pacote e o nome da função logo em seguida, separados por um ponto.**

**Vejamos um exemplo utilizando a função uniform do módulo random:**



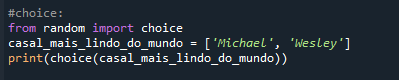
Vejamos agora usando o randint que gera valores inteiros aleatórios entre os valores estabelecidos:





Gera números inteiros de 1 a 60 seis vezes. Este exemplo eu posso gerar números aleatórios para apostar na MegaSena, por exemplo.

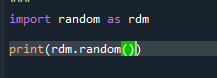
Vamos ver agora o choice que mostrar um valor aleatório dentro de um iterável:

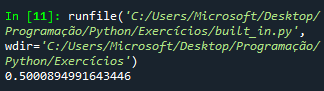




Vamos ver agora como trabalhamos com módulos Built-in:

Mesmo estando já instaladas na nossa máquina, as funções built-in que queremos utilizar devem ser importadas. Podemos utilizar alias, ou seja, apelidos, para utilizar módulos ou funções.









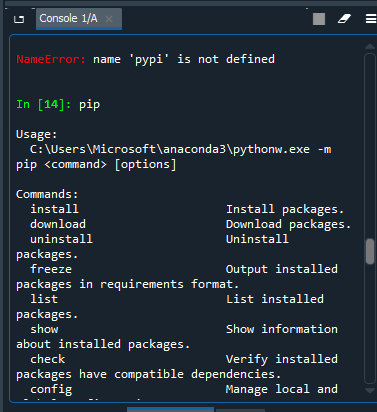
Como todos os arquivos Python que criamos são módulos, podemos importar esses arquivos em outros arquivos para reutilizar suas funções.

Agora vamos ver como instalamos e utilizamos módulos externos:

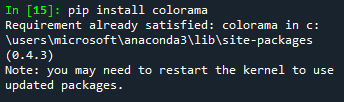
Para isso, utilizamos o gerenciador de pacotes chamado PIP (Python Installer Package), é tipo o NPM (Node Package Manager) que também é um gerenciador de pacotes só que do JavaScript, do Node. 😉

Podemos conhecer todos os pacotes externos e oficiais do Python em <https://pypi.org>

Para utilizar o PIP, digito PIP no terminal Python e navego para instalar, fazer o download, etc:

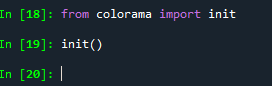


Como exemplo, vamos instalar o módulo externo **Colorama**:



Esse módulo que instalamos, o Colorama, é utilizado para permitir impressão de textos coloridos no terminal Python.

Para iniciarmos o uso do módulo Colorama, usamos o comando de inicializador:



Logo após importamos algumas funções desta lib:



Vamos ver agora sobre o que são os Pacotes:

A diferença entre módulo e pacote é que, módulo é apenas um arquivo Python que pode ter diversas funções para utilizarmos. Já um pacote é um diretório contendo uma coleção de módulos, ou seja, vários arquivos, como uma pasta com vários módulos.

**Nas versões 2.x (menores que 3) do Python, um pacote Python deveria conter dentro dele um arquivo chamado \_\_init\_\_.py**

**Nas versões do Python 3x(maiores que 3), não é mais obrigatório a utilização deste arquivo, mas normalmente ainda é utilizado para manter compatibilidade.**

**Para importar um módulo de um pacote, basta fazer assim:**

**Fromnome\_do\_pacoteimportnome\_do\_modulo**

**Para importar uma função de um módulo de um pacote, basta fazer assim:**

**Fromnome\_do\_pacote.nome\_do\_moduloimportnome\_da\_funcao**

**Vamos ver agora sobre o DunderMain e DunderName:**

**Quando você ouvir Dunder, significa Double Under, são os métodos que possuem o duplo underline, no início e no fim. Quando falamos em DunderName = \_\_name\_\_ e em DunderMains = \_\_main\_\_.**

**A comunidade Python utilizam estes dunders como funções e atributos internos do Python para que não entrem em conflitos com os nomes que damos às nossas funções e atributos que criamos.**

Na linguagem C, em um programa C, temos um programa da seguinte forma:

*Intmain () { return 0 }; Sem essa função não existe um programa C.*

Na linguagem JAVA, em um programa JAVA, temos um programa da seguinte forma:

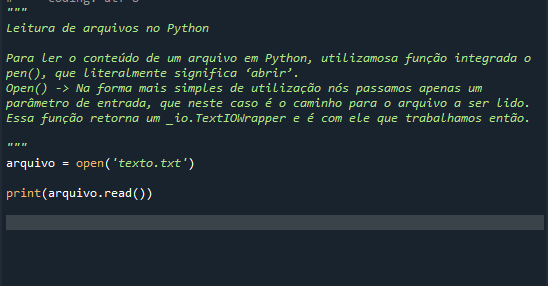
*Publicstaticvoidmain(String[] args { }; Sem essa função não existe um programa JAVA.*

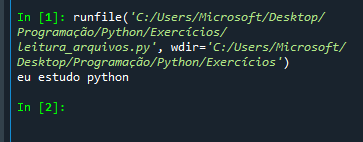
Em python não tem isso... Se executarmos um módulo Python diretamente na linha de comando, internamente o Python atribuirá a variável \_\_name\_\_ o valor \_\_main\_\_, indicando que este módulo é o módulo de execução principal.

* Leitura e Escrita em arquivos:

Para ler o conteúdo de um arquivo em Python, utilizamosa função integrada open(), que literalmente significa ‘abrir’.

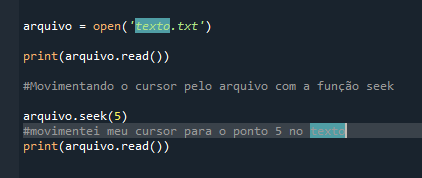
Open() -> Na forma mais simples de utilização nós passamos apenas um parâmetro de entrada, que neste caso é o caminho para o arquivo a ser lido. Essa função retorna um **\_io.TextIOWrapper** e é com ele que trabalhamos então.

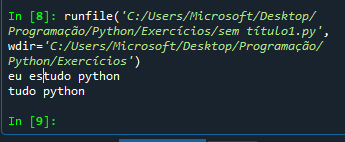




Vamos ver agora sobre **seek** e **cursors**:

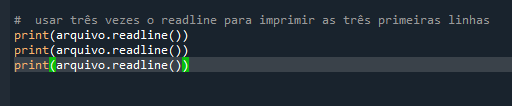
A função seek é utilizada para movimentar o cursor pelo arquivo. Ou seja, movimentamos a barrinha pelo arquivo, indo e voltando.



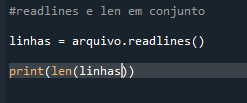


Ou seja, a função seek indica onde queremos colocar o cursor.

Para lermos linha a linha no arquivo, usamos a **readline:**



Agora readlines, no plural, faz com quem nós lemos as linhas, colocando-as em uma lista com o conteúdo de cada linha em cada elemento da lista. Ou seja, cada linha acaba sendo um item da lista. Usando o readlines junto com o len podemos contar a quantidade de linhas de um texto, por exemplo.





***Quando abrimos um arquivo com a função open() é criada uma conexão entre o arquivo no disco do computador e nosso programa. Essa conexão é chamada de streaming. Ao finalizar os trabalhos com o arquivo, devemos fechar esta conexão, utilizando a função close().***

Com a função read, podemos limitar a quantidade de caracteres a serem impressos no arquivo! Tipo, só 50 caracteres: **arquivo.read(50).**

Vamos ver agora o **blocoWith**:

**O bloco With é utilizado para criar um contexto de trabalho com os arquivos que manipulamos, onde os recursos utilizados são fechados após o bloco With.**



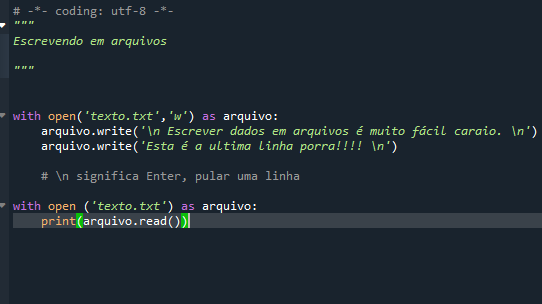
Ou seja, com o comando de abrir um arquivo, chame ele de “arquivo” e trabalhe com ele dentro do bloco. Saiu do bloco With, saiu do contexto. Com isso não precisamos mais fazer o fechamento do arquivo.

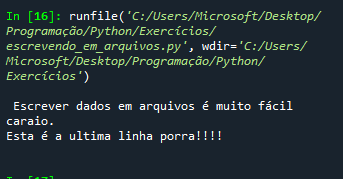
**Agora vamos ver como se escreve em arquivos:**

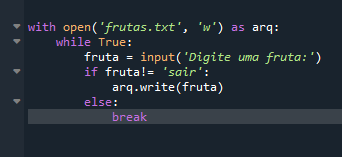
Primeiro de tudo para ler ou escrever em arquivos, eu tenho que abri-lo.

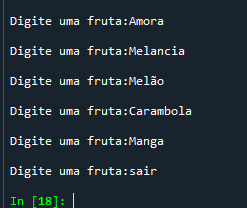
Vamos ver um exemplo de escrita:

Na hora que eu passo para escrita, este arquivo é criado. Nós começamos no open abrindo um modo de abertura ‘w’ que significa ‘Write’, para escrever:









Vamos entender agora outros modos de manipulação de arquivos:

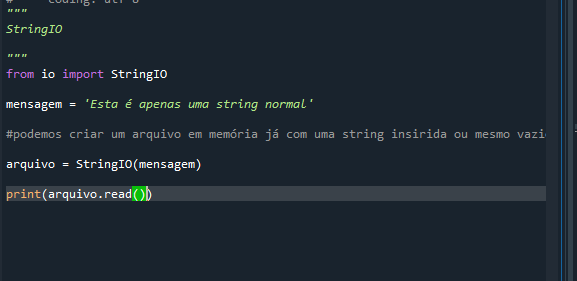
* ‘r’: abre para leitura, é o padrão;
* ‘w’: abre para escrita, ele sobrescreve caso o arquivo já exista e cria o arquivo caso ainda não exista;
* ‘x’: abre para criação exclusiva, falha se já existir. Ou seja, abre para escrita somente se o arquivo não existir.
* ‘a’: abre para para escrita, adicionando o novo conteúdo no final do arquivo caso ele já exista.
* ‘+’: abre o arquivo para leitura e escrita. Nós não colocamos o ‘+’ sozinho, colocamos sempre com alguém. Por ex: ‘r+’, substitui o conteúdo, ‘a+’ adiciona o conteúdo no final. Esse + é para atualização de arquivo.

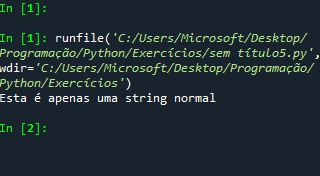
Vamos ver agora sobre **StringIO:**

Para podermos ler ou escrever em arquivos dados do sistema operacional o software que estamos escrevendo precisa de permissão: de leitura para ler o arquivo e permissão de escrita para escrever o arquivo.

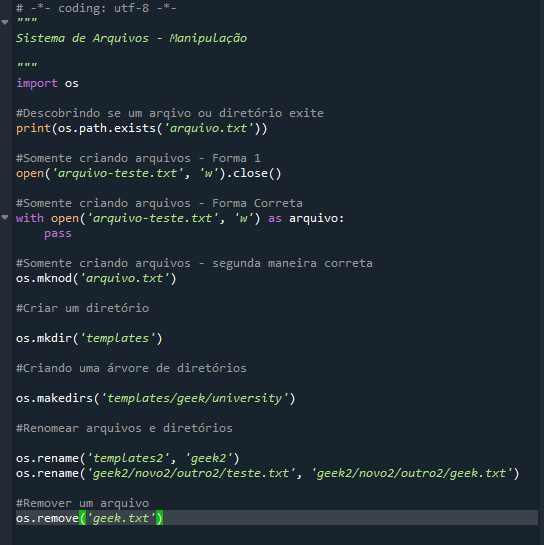
O StringIO é utilizado para ler e criar arquivos em memória do computador, não no disco.

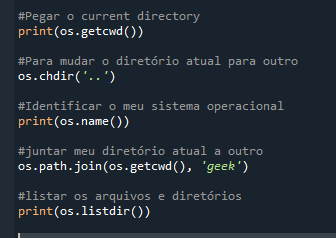
Para utilizar, devemos primeiro importar-lo:

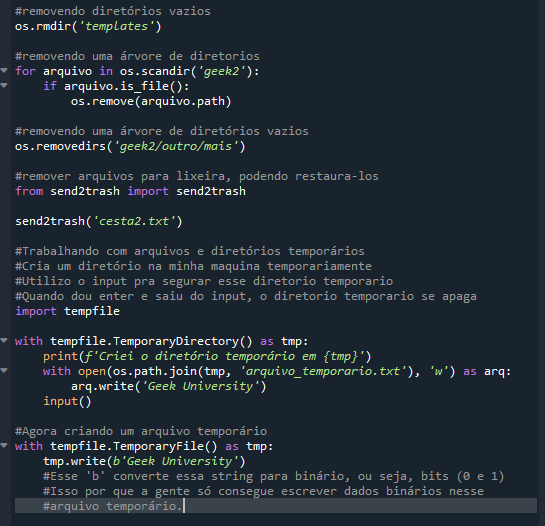




Vamos ver agora alguns comandos de manipulação de arquivos e diretórios do sistema operacional com Python:





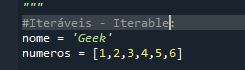


* Iteradores e Geradores:

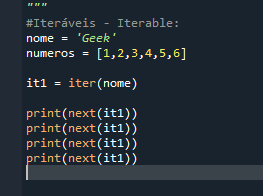
Aqui vamos entender melhor nos recursos mais legais de Python, e vamos ver a diferença entre iteradores e iteráveis, vamos criar nossa própria versão de loop, etc...

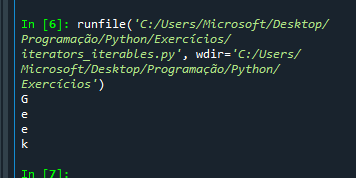
Vamos lá para diferença entre Iterador e Iterável:

* Iteráveis (Iterable): **Um objeto que irá retornar um iterator quando a função iter() for chamada**



Para transformar um iterável em iterador eu uso a função iter():

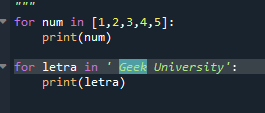




* Iterador (Iterator): **Ele é um objeto de programação, é um elemento da programação, que pode ser iterado. É um objeto que retorna um dado sendo umelemento por vez quando uma função *next é chamada.***

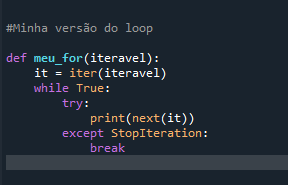
Vamos agora entender como criar nossa própria versão de **loop:**

Normalmente nós criamos nossas próprias versões de loops para ver a fundo de verdade o que por debaixo dos panos o Python faz...

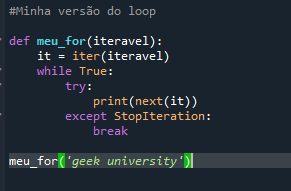


**Por baixo dos panos, o Python vai pegar essa lista ou essa String que são iteráveis e transformar em um iterador com a função iter() e depois usar o next() para iterar e realizar o loop...**

Mas agora vamos ver como criar uma versão nossa do loop:

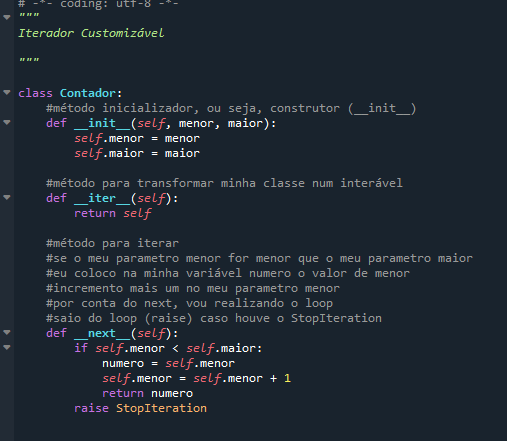


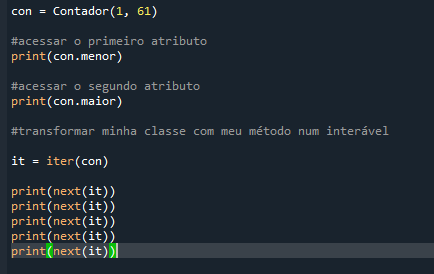
**Ou seja, eu criei meu próprio loop assim: eu passo como parâmetro um iterável, transforme este iterável como iterador, depois, enquanto for verdadeiro, eu vou tentar imprimir na tela, usando o next, os elementos do iterável... Quando acabar meu iterável (StopIteration) eu saio do meu loop (break)...**

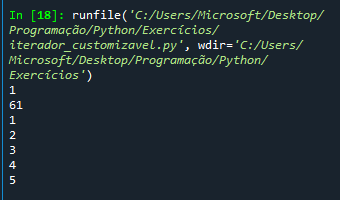


Agora vamos ver como escrever um iterador customizado:

Vamos criar algo parecido com a função range, que gera um range entre um intervalo de números ou letras, por exemplo...







Vamos ver agora sobre os **geradores**:

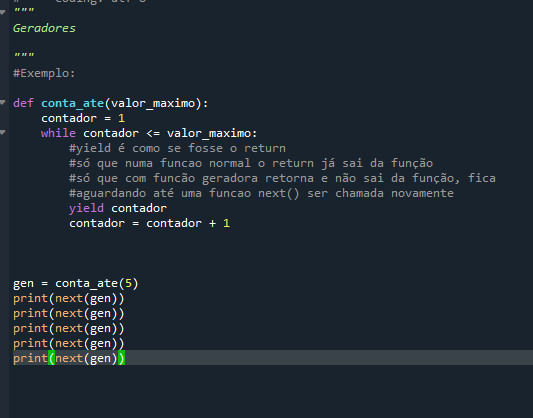
Geradores, conhecidos como Generators, são iterators, ou seja, são iteradores... O contrário não é verdadeiro, ou seja, nem todo iterator é um generator.

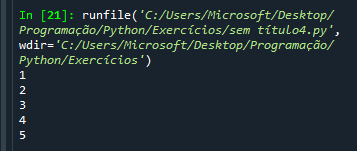
Geradores podem ser criados com funções geradores. Funções geradores utilizam a palavra reservada **yield.** Geradores podem ser criados com expressões geradoras.

Vamos ver agora algumas diferenças entre funções e funções geradores:

|  |  |
| --- | --- |
| **FUNÇÕES** | **FUNÇÕES GERADORES** |
| Utilizam “return” | Utilizam “yield” |
| Retorna uma só vez | Podem utilizar yield múltiplas vezes |
| Quando executada retorna um valor | Quando executada, retorna um generator |

Exemplo de generatorfunction:







Vale lembrar que uma operação feita com função normal é muitooooo maior de uso de memória que irá usar do que se usasse com função geradores se fosse possível.

* Decoradores em Python:

Para entendermos o que são decoradores, precisamos entender o que são funções de maior grandeza... Em inglês chamamos de Higher Order Functions (HOF)... Funções de maior grandeza significa que quando uma linguagem de programação suporta HOF, esta linguagem consegue suportar que uma função possa retornar outra função.

Em Python as funções são Cidadões de Primeira Classe, ou em inglês são First Class Citizens (FCC).

Temos também em Python as Funções Aninhadas, ou em inglês Nested Functions... Ou seja, em Python podemos ter funções também dentro de funções, que são conhecidas como Listed Functions ou também Inner Functions que são funções internas.

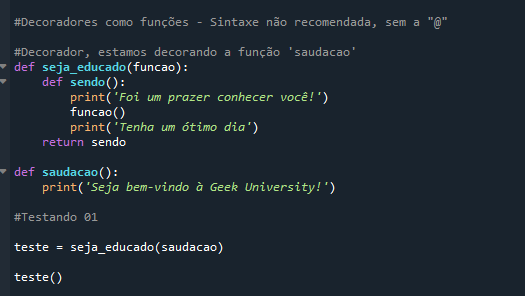
Temos também como retornar funções de outras funções.

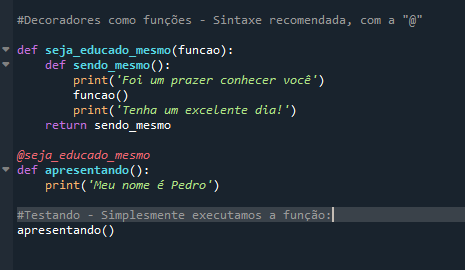
Agora vamos, de fato, entendermos que são decoradores:

Decoradores são funções, pra começar tudo. Elas também envolvem outras funções e aprimoram seus comportamentos, ou seja, imagine que você tem uma função que você quer decorar e a gente coloca externamente a função decorada e internamente temos a função normal. Ou seja, quando temos uma função decorada é quando pegamos uma função normal e aprimoramos ela, ou seja, decoramos ela. Pegando no mundo real é quando temos um apartamento e decoramos nosso apartamento, por exemplo.

Decoradores também são exemplos de HOF, ou seja, Higher Order Functions. Eles tem uma sintaxe própria usando “@”, que não obrigatório, mas é considerado um padrão visual.

Vamos ver um exemplo de decorador:



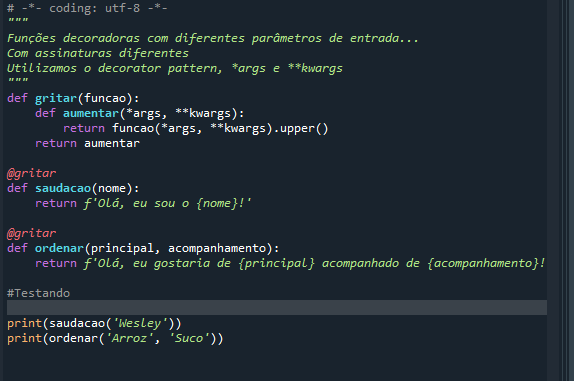


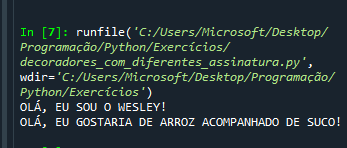


Ou seja, tenho uma função “apresentando”, aí eu altero o seu comportamento para ficar com o comportamento da função decoradora “seja\_educado\_mesmo”, com isso, minha função normal fica agora com os comportamentos da função “seja\_educado\_mesmo”.

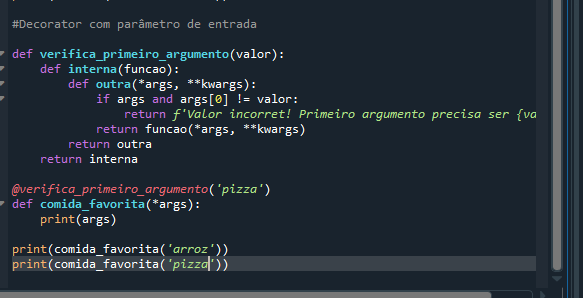
Vamos ver agora decoradores com diferentes assinaturas:

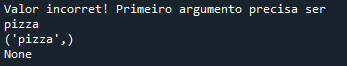
Quando queremos utilizar um decorador com uma função de dois parâmetros ou mais, utilizamos um padrão de projeto que se chama decorator pattern, que nada mais são do que os \*args e \*\*kwargs:





Nós podemos ter assinaturas de decorators com parâmetros de entradas:



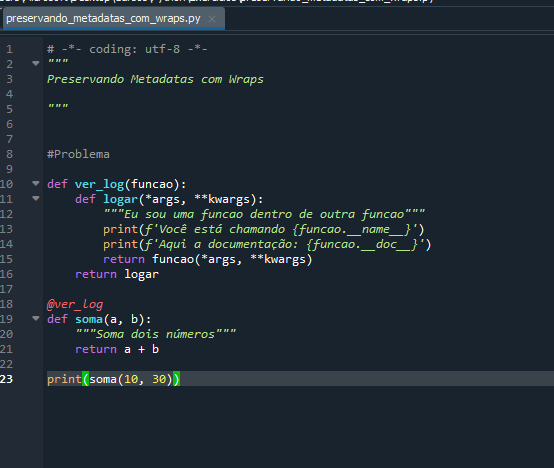


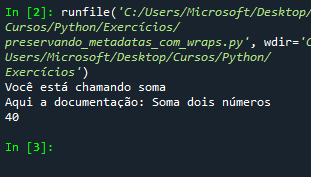
Agora vamos ver como preservar Metadatas com Wraps:

Lembrando que Metadatas são dados intrínsecos, ou seja, metadatas são dados de dados, ou seja ainda, dados de um arquivo, imagem, documento, etc.

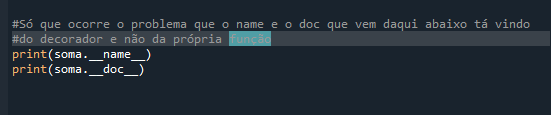
**Agora Wraps são funções que em envolvem elementos com diversas funcionalidades.**

Precisamos preservar metadatas com Wraps , certo? Para entender vamos ver o problema abaixo:





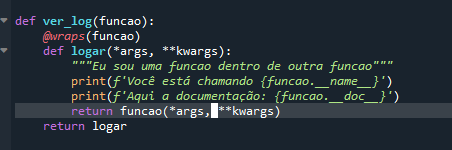
Mas temos um problema com nossos metadados:





Para resolver isso, precisamos realizar a seguinte ação:

* Importe de functools o wraps: 
* E dentro da função decorator colocamos outro decorator Wraps:



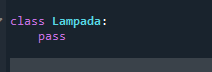


* Orientação a Objetos em Python:

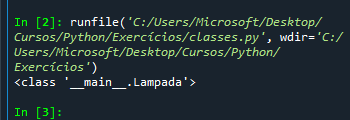
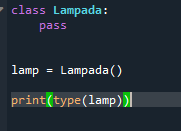
Vamos ver primeiramente sobre as classes do Python...

As classes são um dos elementos dentro da orientação a objetos dentro do Python. **Em POO Classes nada mais são do que modelos dos objetos do mundo real sendo representadas computacionalmente.**

Imagine que você queira fazer um sistema para automatizar o controle das lâmpadas da sua casa. Se você criasse um sistema em POO possivelmente você criaria uma Classe chamada Lâmpada. Você tá representando ela no seus sistema computacionalmente. Para isso criamos uma classe. Por que na verdade não existe o tipo de dado Lâmpada em nenhuma linguagem de programação, por isso criamos uma Classe para representa-la.

 Uma classe Lampada que faz nada.

Quando crio uma classe eu crio um tipo de dado do tipo Classe:



Classes podem conter atributos que representam as características do objeto, ou seja, pelos atributos conseguimos representar computacionalmente os estados de um objeto. No caso da Lâmpada, por exemplo, possivelmente iríamos querer saber se a lâmpada é 110 ou 220V, se ela é branca, amarela, vermelha, ou seja lá que cor, qual é luminosidade dela e etc.

Além de atributos nós temos métodos nas classes, que podemos olhar como funções. Representam os comportamentos do objeto, ou seja, as ações que este objeto pode realizar no seu sistema. No caso da lâmpada, por exemplo, um comportamento comum que muito provavelmente iríamos querer representar no nosso sistema é o de ligar e desligar a lâmpada.

Em Python para definirmos uma classe nós utilizamos uma palavra reservada chamada “**class**”, simples assim. O “**pass**” dentro da classe é utilizada em Python quando temos um bloco de código que ainda não está implementado. Ou seja, ainda não coloquei os itens dentro da classe, mas já quero definir a minha classe, ou seja, **passe carai!!!!**

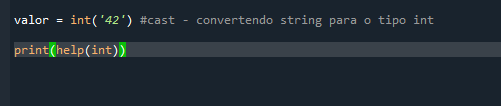
Quando nomeamos uma classe em Python, utilizamos por convenção o nome com inicial em maiúsculo. Se o nome for composto, utilizamos as iniciais de ambas as palavras em maiúsculo, ou seja, todas juntas. Ou seja, nomes de classe, por convenção, devem ser em Camel Case.



Uma dica é que em computação não utilizamos acentuação, caracteres especiais, espaços ou similares para nomes de classes, atributos, métodos, arquivos, diretórios e etc. Não existe acentuação, não existe nada para esses casos em computação.

**A diferença entre a criação de classes em Java e Python é que o nome da minha classe principal tem que ser o nome da minha classe em Java, se não for assim dá erro. Agora aqui em Python não: posso ter um nome de arquivo e criar quantas classes eu quiser com nomes diferentes.**

Quando vamos fazer uma operação de casting, por exemplo, trazer do string para int e usamos a “**int()**”, “**int()**” é uma classe Python que vem na linguagens:



O Python criou essa classe por padrão com letra minúscula pois as classes do próprio Python são minúsculos, as classes internas. Já as nossas classes, que nós criamos é com maiúsculo, pois simples fato que dessa forma fica fácil identificar o que é uma classe interna do Python e o que não é.

Agora vamos ver os atributos, que são as características de um objeto. Agora com atributos vamos conseguir representar os estados dos objetos computacionalmente.

Em Python dividimos os atributos em três grupos:

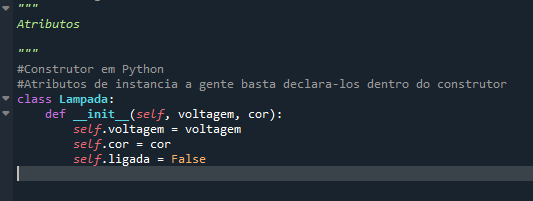
* **Atributos de instância;**
* **Atributos de classe;**
* **Atributos dinâmicos.**

Primeiro vamos ver o atributo de instância:

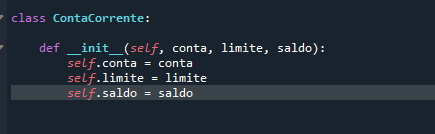
Quando falamos de atributo de instância estamos falando do método construtor, aquele construtor que vimos também na instanciação de objetos no Java.

Só que no Python é algo mais simples. Ou seja atributos de instância são atributos declarados dentro dos métodos construtores e método construtor é um método especial utilizado para a construção do objeto.

Vamos ver um construtor em Python:



**Quando uso o self no Python eu to dizendo que o atributo é privado.**



O **Self** que utilizamos nestes exemplos, para entender, imagina que você já comeu no restaurante self-service você faz seu prato, você se serve. A ideia aqui é a mesma, o Self nada mais é que é o próprio objeto que está executando o método.

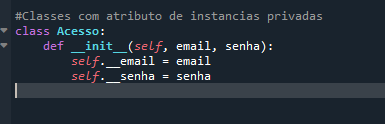
Quando estamos utilizando atributos de instância eles podem ser públicos ou privados e quando estamos falando de públicos ou privados estamos falando de visibilidade. Ou seja quando eu declaro um atributo privado ele só poderá ser usado dentro da própria classe onde ele foi declarado. Agora um atributo público pode ser acessado por elementos de fora da sua própria classe.

Vamos ver atributos públicos e privados aqui no Python:

**Em Python nós não temos as palavras “Public” e “Private”, como temos no Java, para definir o escopo de um atributo e sim por convenção, então por convenção ficou estabelecido que todo atributo de uma classe é público em Python.**

**Ou seja, todo atributo de uma classe em Python pode ser acessado por qualquer elemento do projeto. Caso queiramos demonstrar que determinado atributo deve ser tratado como privado, ou seja, que deve ser acessado/utilizado somente dentro da própria classe onde está declarado utiliza-se duplo underscore no ínicio de seu nome \_\_. Isso é conhecido também como Name Mangling.**

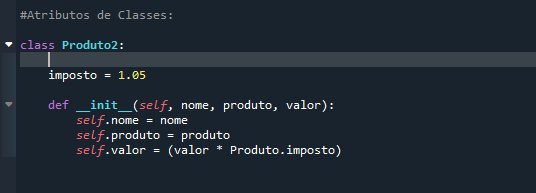
Exemplo:



Mas temos uma observação e lembre-se que isso é apenas uma convenção, a linguagem Python não vai impedir que façamos acessos aos atributos sinalizados como privados fora da classe.

Agora vamos ver sobre atributos de Classes:

**Atributos de Classe são atributos que são declarados diretamente na classe, ou seja, fora do construtor, geralmente já inicializamos um valor e este valor é compartilhado entre todas as instâncias da classe, ou seja, ao invés de cada instância da classe ter seus próprios valores como é o caso dos atributos de instâncias.**

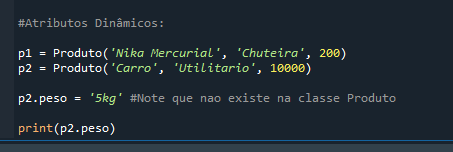


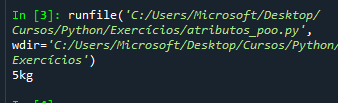
**Neste exemplo o imposto é o mesmo para seja qual for o produto em qualquer instância, por isso declaramos o atributo imposto como atributo de classe.**

Em outras linguagens como Java os atributos de classes são chamados de atributos estáticos, justamente por ter essa característica de permanecer o mesmo valor seja qual for a instância.

Agora vamos ver sobre os atributos dinâmicos:

Nada mais é que um atributo de instância que pode ser criado em tempo de execução. O atributo dinâmico será exclusivo da instância que o criou.





Para visualizar em JSON, ou dicionário os atributos e seus valores de minha classe eu utilize o \_\_dict\_\_. Exemplo: print(p2.\_\_dict\_\_).

Para deletar um atributo de uma classe eu faço **del** p2.valor.

**Assim em Python assim como conseguimos adicionar atributos dinamicamente eu consigo deletar atributos dinamicamente também.**

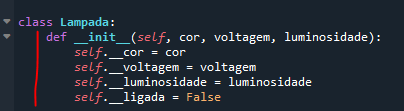
Agora vamos ver sobre os métodos. Tudo que já sabemos sobre funções se aplicam aos métodos. Os métodos representam o comportamento do objeto, ou seja, as ações que o objeto pode realizar em seu sistema. No caso de uma lâmpada, o comportamento comum é de ligar e desligar a lâmpada. Para desligar e ligar ela criamos um método.

Em Python dividimos os métodos em dois grupos:

* Métodos de instância;
* Métodos de classe.

Vamos ver primeiro sobre os métodos de instância:

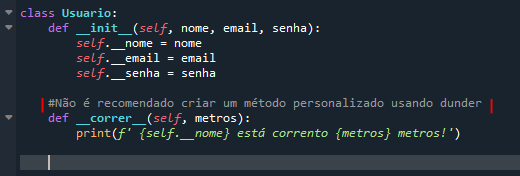
**O método de instância (dunder init -> \_\_init\_\_) é um método especial chamado de construtor e sua função é construir o objeto a partir da classe:**



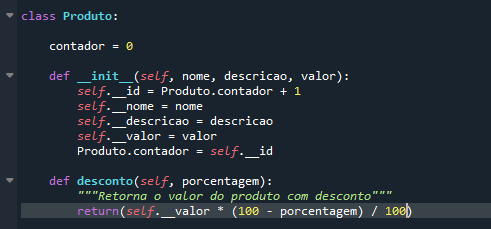
OBS: Todo elemento em Python que inicia e finaliza com duplo underline (dunder) é chamado de dunder (Double Underline).

**Os métodos/funções dunder em Python são chamados de métodos mágicos. Por exemplo: \_\_init\_\_ é um método mágico.**

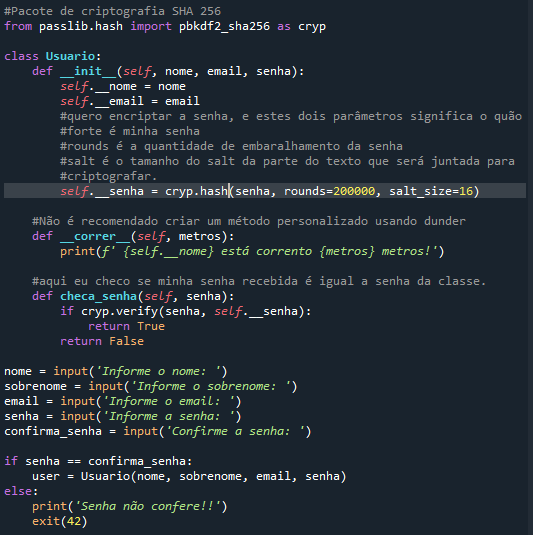
Atenção: por mais que possamos criar nossas próprias funções, utilizando dunder, isto é altamente não recomendado, conforme exemplo abaixo:

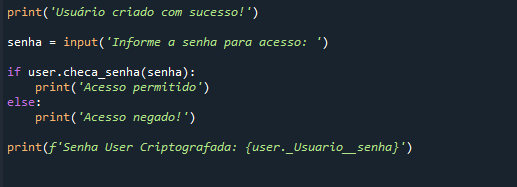


Vejamos abaixo a maneira correta deter um método personalizado dentro de uma classe:



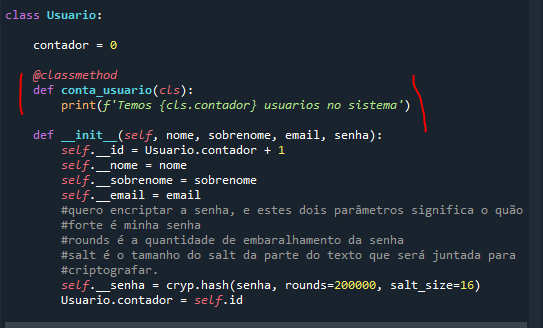
Abaixo criamos um sistema de login com senha criptografada com o auxílio da lib passlib:



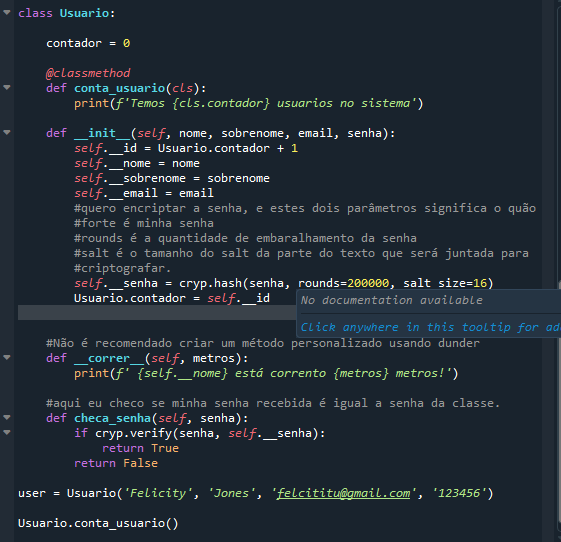


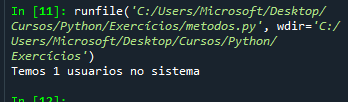
Agora vamos ver os métodos de classe:

Para a gente identificar que um método é de classe e não de instância nós usamos um decorator.



Para criarmos um atributo de classe precisamos criar um @classmethod. Quando criamos o decorator @classmethod o parâmetro que recebemos nele é a própria classe, por isso coloquei cls, que é uma convenção utilizada no Python.





**Usamos métodos de classe para acessar atributos de classe e usamos métodos de instância para acessar atributos de instância. Métodos de classes são conhecidos como métodos estáticos em outras linguagens.**

**Agora vamos ver abaixo os métodos privados, que só podem ser acessados dentro da classe. Criamos eles com duplo underline somente no começo:**



Agora vamos ver um método estático de verdade em Python que é um método que não tem acesso a nem classe e nem instância, a nada, só retorna um valor:



Agora vamos ver sobre os objetos:

Objetos são instâncias de classes, ou seja, após o mapeamento de objetos do mundo real para sua representação para o mundo computacional, podemos criar quantos objetos forem necessários. Podemos pensar nos objetos/instâncias de uma classe como variáveis do tipo definido na classe.

Vamos ver agora sobre Abstração e Encapsulamento:

**O grande objetivo da POO é encapsular nosso código dentro de um grupo lógico e hierárquico utilizando classes.**

Mas o que é encapsular? É uma cápsula. Um encapsulamento é o ato de encapsular os elementos de uma classe. Imagine que temos uma classe e queremos proteger os elementos da minha classe de elementos externos. Eu estou tomando uma ação de encapsular. Podemos definir encapsulamento também como um agrupamento de atributos e métodos.

Mas o que é Abstração? A gente tá falando em expor somente dados (atributos e métodos) relevantes para o usuário. Ou seja, é o ato de expor somente apenas dados relevantes de uma classe, escondendo atributos e métodos privados para o usuário. Vamos pensar assim: temos um Smartphone, só fica acessível para gente que tá usando ele as funções e atributos necessários para meu uso, tendo suas funcionalidades externas escondidas de mim, não entra na minha user experience. Os passos que o usuário não precisas saber, que são irrelevantes, são abstraídas, são removidas da experiência visível do cliente.

* Herança e Polimorfismo:

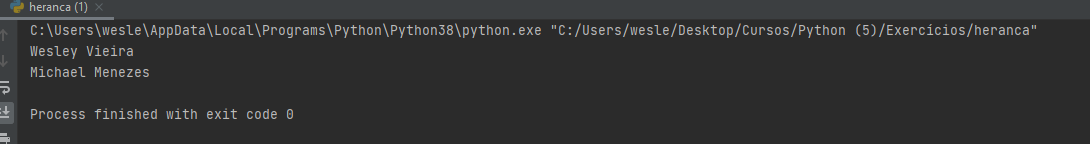
A ideia de herança é a de reaproveitar código. Mas também extender nossas classes. Então quando falamos de herança pensa em reaproveitamento de código e extensão de classes. Isso por que se uma classe for genérica suficiente nós conseguimos utilizar outras classes reutilizando atributos e métodos dessa classe genérica. Com herança, a partir de uma classe existente, nós estendemos outra classe que passa a herdar atributos e métodos da classe genérica herdada.

Vamos imaginar que queremos criar instâncias para clientes e funcionários de um Banco:

class Cliente:  
 def \_\_init\_\_(self, nome, sobrenome, cpf, renda):  
 self.\_\_nome = nome  
 self.\_\_sobrenome = sobrenome  
 self.\_\_cpf = cpf  
 self.\_\_renda = renda  
  
 def nome\_completo(self):  
 return f'{self.\_\_nome} {self.\_\_sobrenome}'  
  
class Funcionario:  
 def \_\_init\_\_(self, nome, sobrenome, cpf, matricula):  
 self.\_\_nome = nome  
 self.\_\_sobrenome = sobrenome  
 self.\_\_cpf = cpf  
 self.\_\_matricula = matricula  
  
 def nome\_completo(self):  
 return f'{self.\_\_nome} {self.\_\_sobrenome}'  
  
cliente1 = Cliente('Wesley', 'Vieira', 39129375865, 3500)  
  
funcionario1 = Funcionario('Michael', 'Menezes', 5486987895758, 10000)  
  
print(Cliente.nome\_completo(cliente1))  
print(Funcionario.nome\_completo(funcionario1))

Mas veja que temos atributos e métodos que se repetem nas duas classes. Podemos resolver essa repetição com o uso de herança e polimorfismo:

class Pessoa:  
 def \_\_init\_\_(self, nome, sobrenome, cpf):  
 self.\_\_nome = nome  
 self.\_\_sobrenome = sobrenome  
 self.\_\_cpf = cpf  
  
 def nome\_completo(self):  
 return f'{self.\_\_nome} {self.\_\_sobrenome}'  
  
  
#Cliente herda a classe Pessoa  
class Cliente(Pessoa):  
 def \_\_init\_\_(self, nome, sobrenome, cpf, renda):  
 super().\_\_init\_\_(nome, sobrenome, cpf)  
 self.\_\_renda = renda  
  
#Funcionário herda a classe Pessoa  
class Funcionario(Pessoa):  
 def \_\_init\_\_(self, nome, sobrenome, cpf, matricula):  
 super().\_\_init\_\_(nome, sobrenome, cpf)  
 self.\_\_matricula = matricula  
  
  
cliente1 = Cliente('Wesley', 'Vieira', 39129375865, 3500)  
  
funcionario1 = Funcionario('Michael', 'Menezes', 5486987895758, 10000)  
  
print(Cliente.nome\_completo(cliente1))  
print(Funcionario.nome\_completo(funcionario1))



Quando uma classe herda de outra classe ela herda todos os atributos e métodos da classe herdada.

**Ou seja, a gente fala qual classe está sendo a Super Classe da minha classe. Depois eu, dentro do meu construtor, passo todos os parâmetros dentro de \_\_init\_\_ (os da Super Classe e o da minha classe atual) e dentro do meu \_\_init\_\_ eu uso o método super() para alcançar o construtor da minha Super Classe.**

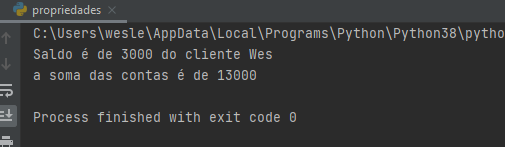
Quando trabalhamos com herança, em determinadas filhas, pode ser que precisamos sobrescrever os métodos da super classe, ou do inglês, override:

#Funcionário herda a classe Pessoa  
class Funcionario(Pessoa):  
 def \_\_init\_\_(self, nome, sobrenome, cpf, matricula):  
 super().\_\_init\_\_(nome, sobrenome, cpf)  
 self.\_\_matricula = matricula  
  
 def nome\_completo(self):  
 return f'{self.\_\_matricula} {self.\_Pessoa\_\_nome}'

Agora vamos ver sobre Propriedades:

Aqui vamos ver os getters e setters para manipularmos os atributos das classes para trabalharmos com eles.

class Conta:  
  
 contador = 0  
  
 def \_\_init\_\_(self, titular, saldo, limite):  
 self.\_\_numero = Conta.contador + 1  
 self.\_\_titular = titular  
 self.\_\_saldo = saldo  
 self.\_\_limite = limite  
 Conta.contador += 1  
  
 def extrato(self):  
 return f'Saldo é de {self.\_\_saldo} do cliente {self.\_\_titular}'  
  
 def depositar(self, valor):  
 self.\_\_saldo += valor  
  
 def sacar(self, valor):  
 self.\_\_saldo -= valor  
  
 def transferir(self, valor, destino):  
 self.\_\_saldo += valor  
 destino.\_\_saldo += valor  
  
 def get\_numero(self):  
 return self.\_\_numero  
  
 def get\_titular(self):  
 return self.\_\_titular  
  
 def get\_saldo(self):  
 return self.\_\_saldo  
  
 def get\_limite(self):  
 return self.\_\_limite  
  
 def set\_numero(self, numero):  
 self.\_\_numero = numero  
  
 def set\_titular(self, titular):  
 self.\_\_titular = titular  
  
 def set\_saldo(self, saldo):  
 self.\_\_saldo = saldo  
  
 def set\_limite(self, limite):  
 self.\_\_limite = limite  
  
conta1 = Conta('Wes', 3000, 5000)  
conta2 = Conta('Mi', 10000, 25000)  
print(conta1.extrato())  
print(f'a soma das contas é de {conta1.get\_saldo() + conta2.get\_saldo()}')



Mas utilizar Getters e Setter é uma forma muito Java de se trabalhar. Para trabalharmos com Propriedades de uma forma mais Pythonica, utilizamos da seguinte forma:

#Criando as propriedades  
 @property  
 def numero(self):  
 return self.\_\_numero  
  
 @property  
 def titular(self):  
 return self.\_\_titular  
  
 @property  
 def saldo(self):  
 return self.\_\_saldo  
  
 @property  
 def limite(self):  
 return self.\_\_limite  
  
  
  
conta1 = Conta('Wes', 3000, 5000)  
conta2 = Conta('Mi', 10000, 25000)  
print(conta1.extrato())  
print(f'O saldo do {conta2.titular} é {conta2.saldo}')

Agora para criar um setter criamos um decorator especial para isso:

@limite.setter  
def limite(self, novo\_limite):  
 self.\_\_limite = novo\_limite

conta1.limite = 5000

**Então, ou seja, com @property eu faço os meus getters no Python**. **Com @nomedapropriedade.setter eu faço os setters da minha propriedade**. Assim quando eu for utilizar essas propriedades, posso pegar e alterar os valores das minhas propriedades diretamente e mesmo assim meus atributos estarão de forma privada.

Agora veremos sobre Heranças Múltiplas:

Herança Múltipla nada mais é que a possibilidade de uma classe herdar de múltiplas classes fazendo com que a classe filha herde todos os atributos e métodos de todas as classes herdadas.

A herança múltipla pode ser feita de duas maneiras: por multi derivação direta ou por multi derivação indireta.

Nós falamos que uma classe deriva da outra quando uma classe herda de outra classe. E quando falamos de multi derivação estamos falando de herança múltipla.

* Multi derivação direta:
* # Multiderivação direta  
    
  class Base1:  
   pass  
    
  class Base2:  
   pass  
    
    
  class Multiderivado(Base1, Base2):  
   pass
* Multi derivação indireta:
* # Multiderivação indireta  
    
  class Base1:  
   pass  
    
  class Base2(Base1):  
   pass  
    
  class Multiderivado(Base2):  
   pass

Veremos agora sobre **MRO – Method Resolution Order**:

Em português significa Resolução de Ordem de Métodos, ou seja, é a ordem de execução dos métodos (quem será executado primeiro).

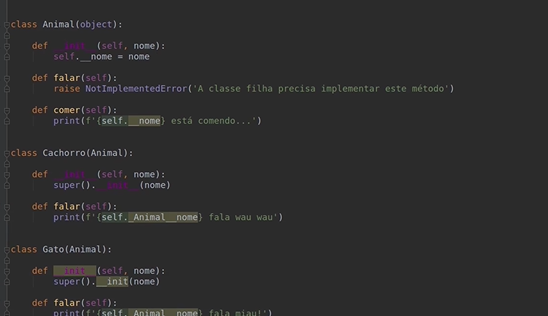
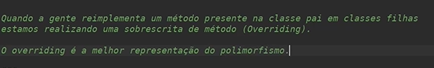
Em Python, a gente pode conferir a ordem de execução dos métodos (MRO) de 3 formas:

* Via propriedade da classe \_\_mro\_\_;
* Via método mro()
* Via help.

Agora vamos ver sobre **Polimorfismos:**

É múltiplas formas.

Para isso vamos criar um método pai e vamos deixar um método “falar” somente para que as classes filhas possam implementar... Utilizei o raise abaixo somente para que qualquer classe filha que não implementar esse método, lance um exceção:

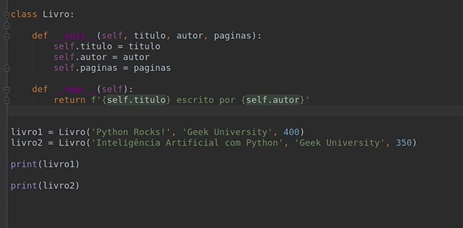
]

Agora vamos ver sobre os **métodos mágicos**:

Já conhecemos um método mágico do Python... Lembra o que são os métodos mágicos? Já falamos sobre eles... Métodos mágicos são todos os métodos que utilizam dunder (double underline: “\_\_”)... Por exemplo já conhecemos o dunder init, que é o nosso construtor no Python.

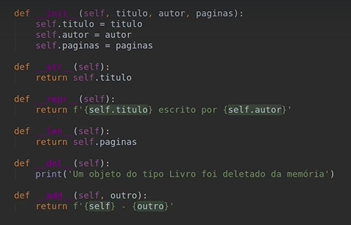
Ou seja, chamamos métodos dunder quando utilizamos double underline, duplo underline...

* **\_\_repr\_\_:** esse método mágico faz a representação do nosso objetivo, por exemplo, em vez de aparecer aqueles números estranhos quando imprimimos nossa classe, vamos imprimir o que queremos, por exemplo:



* **\_\_str\_\_:** Serve para dar uma representação de nossa classe para o usuário final.
* **\_\_len\_\_:** Definimos o tamanho, retornado usando o método len(), de nossa classe.
* **\_\_del\_\_:** Aqui mudamos o comportamento do nosso código quando deletemos nossa classe da memória, por exemplo, retornar uma mensagem no console após a exclusão da classe na memória.
* **\_\_add\_\_:** representação da nossa adição na classe. Alteramos o comportamento de nossa classe no momento de concatenar ou adicionar a minha classe atual com outra classe qualquer que eu queira adicionar ou concatenar.

Veja um exemplo de uso desses métodos mágicos:



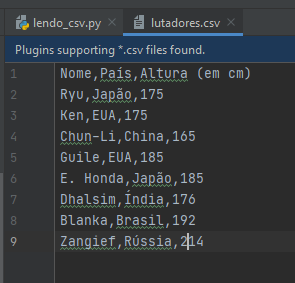
Se utilizarmos o help podemos ver que tem vários outros métodos mágicos que podemos usar para nossas classes e nossos projetos.

* Manipulação de Arquivos CSV e JSON no Python:

Vamos aprender nessa seção como manipularmos dados com JSON e CSV, para trabalharmos com ciência de dados, ou ainda podemos usar para trabalharmos com inteligência artificial, machine learning e, com isso, sempre em qualquer foco temos que trabalhar com esses arquivos de dados, ou CSV ou JSON. Então aqui vamos aprender a trabalhar com estes tipos de arquivos...

Vamos ver agora como fazer leitura de arquivos CSV:

Para trabalharmos com arquivos CSV no Python, to usando um arquivo de dados CSV abaixo:



Para trabalharmos com este arquivo, vamos fazer o que já normalmente fazíamos para trabalhar com um arquivo qualquer:

with open('mutators.csv') as archive:  
 dados = archive.read()  
 dados = dados.split(',')  
 print(dados)

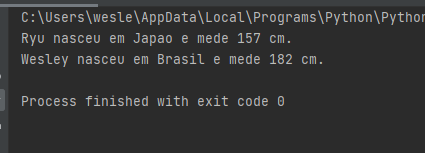
A linguagem Python possui duas formas diferentes para ler dados em arquivos CSV:

Reader: permite que itremos sobre as linhas do arquivo CSV como listas;

DictReader: permite que iteremos sobre as linhas do arquivo CSV como OrderedDict;

Vamos ver o exemplo de cada um deles para ficar fácil de entender:

#Reader - Trabalhar com CSV como listas  
  
from csv import reader  
with open('mutators.csv') as arquivo:  
 leitor\_csv = reader(arquivo)  
 next(leitor\_csv) #Pular uma linha do CSV, serve para pular o cabeçalho  
 for linha in leitor\_csv:  
 print(f'{linha[0]} nasceu em {linha[1]} e mede {linha[2]} cm.')  
#Como temos uma lista para trabalhar, cada linha do arquivo é uma lista para trabalharmos.



#DictReader - Trabalhar CSV como dicionário:  
  
from csv import DictReader  
  
with open('mutators.csv') as arq:  
 leitor\_csv = DictReader(arq)  
 for linha in leitor\_csv:  
 #Cada linha é um OrderedDict  
 print(f"{linha['Nome']} nasceu em {linha['Pais']} e tem {linha['Altura']}")  
 #Não precisamos dar next pois o cabecalho ja é entendido como chave  
 #Devemos usar aqui os parâmetros com o mesmo nome das chaves do CSV

Podemos também indicar o delimitador:



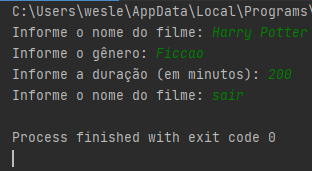
Mas o delimitador padrão é a vírgula.

Vamos ver agora como escrever em **arquivos CSV**:

Como reader() é leitura, write() é para criar um objeto em arquivos CSV:

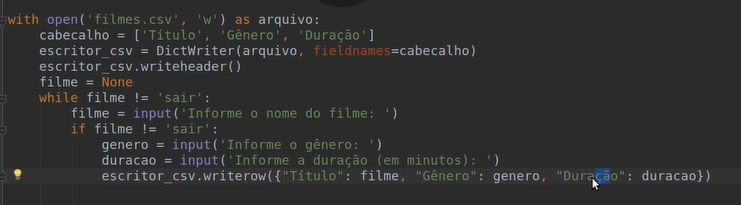
Utilizamos writerow() para escrever uma linha:

from csv import writer  
  
#Criando um arquivo 'filmes.csv'  
with open('filmes.csv', 'w') as arq:  
 #abrindo o escritor csv  
 escritor\_csv = writer(arq)  
 #variavel inicializada com None  
 filme = None  
 #escrever meu cabecalho - o método writerow utiliza lista para escrever a linha  
 escritor\_csv.writerow(['Título', 'Gênero', 'Duração'])  
 while filme != 'sair':  
 filme = input('Informe o nome do filme: ')  
 if filme != 'sair':  
 genero = input('Informe o gênero: ')  
 duracao = input('Informe a duração (em minutos): ')  
 escritor\_csv.writerow([filme, genero, duracao])



Agora vamos aprender a trabalhar com DictWriter:

O DictWrite cria um objeto para que possamos escrever, mas ao invés de trabalhar com lista, trabalhamos com dicionários. Temos um parâmetro aqui kwargs para gente especifica o campo field names onde falamos qual é o cabeçalho do arquivo. Então aqui ele separa um pouco as coisas, como cabeçalho, linhas etc:



**Então primeiro eu fiz um dicionário para o meu cabeçalho, uso o parâmetro fieldnames para falar qual é o meu cabeçalho e depois escrevo meu cabeçalho especificado no arquivo usando o comando “writeheader()”.**

Depois eu uso “writerow” para escrever os dados que quero no restante do meu arquivo.

Vamos conhecer agora o **Pickle**:

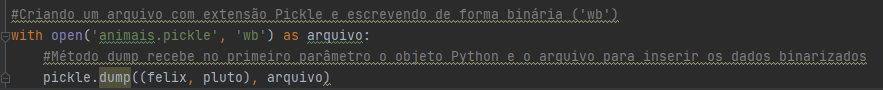
**Nós aprendemos a trabalhar com arquivos CSV e com arquivos de texto, mas note que são arquivos textuais. Por mais que possam ter dados e podemos manipular estes dados, são ainda arquivos textuais, em formato de texto. Mas as vezes precisamos salvar dados que são sensíveis, que outras pessoas não podem ter acesso de forma fácil... Aí que entra o Pickle. Com o Pickle nós salvamos dados como se fosse banco de dados, mas não de forma textual, mas salvamos o dado de forma serializada. Ou seja, os dados são salvos no formato binário, hexadecimal, e a função do Pickle é realizar o seguinte processo:**

**A gente tem um objeto Python, qualquer coisa que nós temos, um objeto instanciado em uma classe e binarizamos ele...**

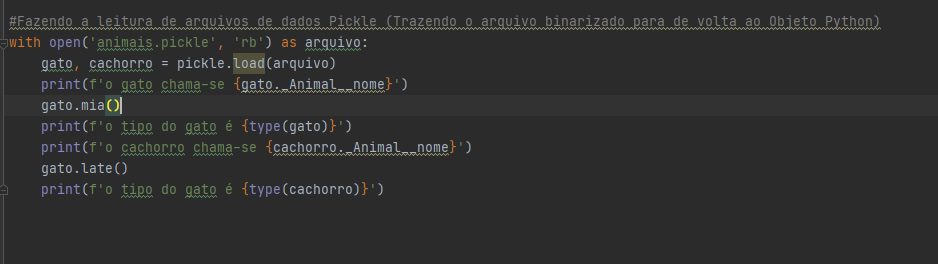
**E se a gente tem algo binarizado podemos voltar para um objeto Python.**

**Este processo é chamado de serialização/deserialização.**

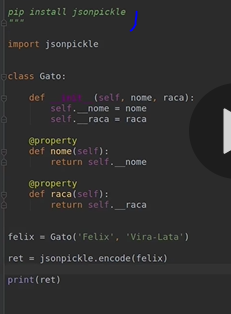
**O MÓDULO PICKLE NÃO É SEGURO CONTRA DADOS MALICIOSOS E DESTA FORMA NÃO É RECOMENDADO TRABALHAR COM ARQUIVOS PICKLE VINDOS DE OUTRAS PESSOAS QUE VOCÊ NÃO CONHEÇA OU DE FONTES DESCONHECIDAS.**



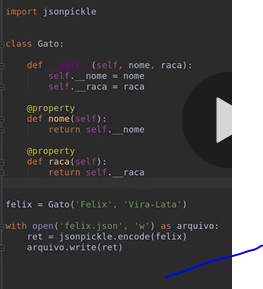
**Agora vamos ver como fazer a leitura de arquivos Pickle:**



**Agora vamos ver como trabalhar com arquivos JSON usando o Pickle:**

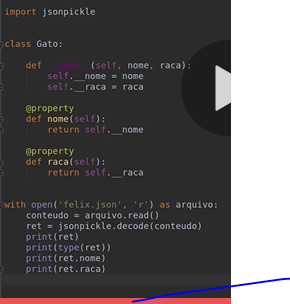


Então com isso eu posso escrever em um arquivo JSON o meu objeto Python utilizando o jsonpickle:



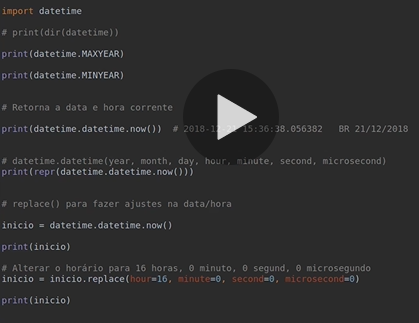


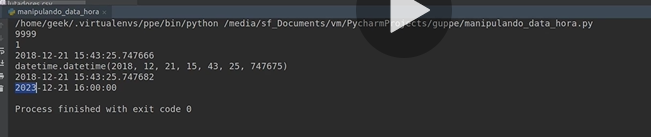
E depois posso transformar um arquivo JSON em um objeto Python... É a chave “py/object” que permite transformar o arquivo JSON em um objeto Python novamente usando jsonpickle:



* Manipulando Data e Hora no Pyton:

Python tem um módulo built-in (integrado) para se trabalhar com data e hora chamado **datetime**.

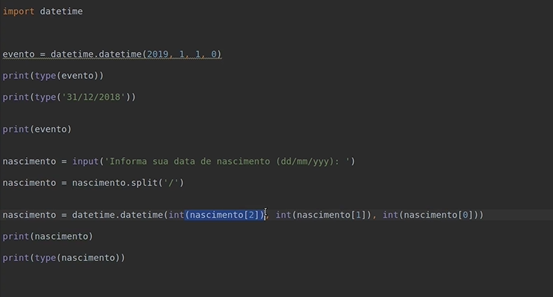




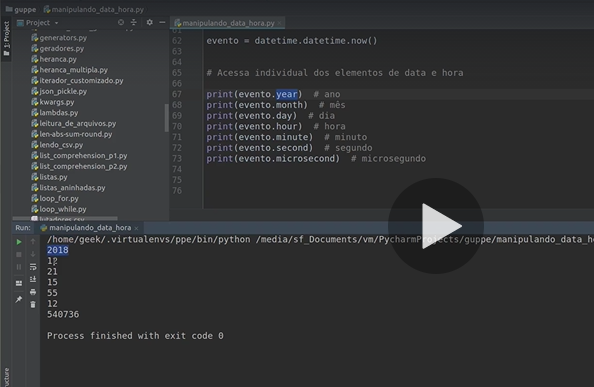
No programa abaixo eu crio uma data inicial instanciada em “evento” como ano 2019, mês 1, dia 1, 0 horas.

Depois eu peço pro usuário informar sua data de nascimento e depois pego essa data de nascimento e transformo em uma lista usando o separador de barras.

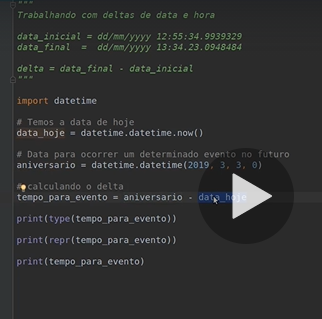
E depois uso cada posição desta lista para setar a nova data do nascimento com base no datetime.

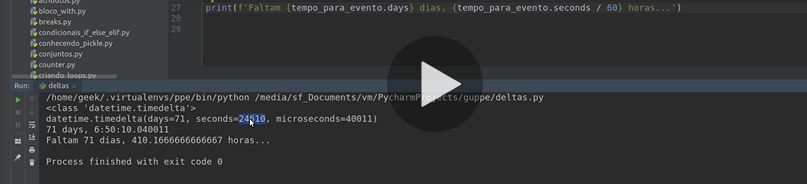


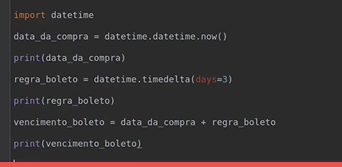
Abaixo também podemos ver como acessar os elementos do datetime de forma individual:



Agora vamos trabalhar com Deltas de data e hora no Python:

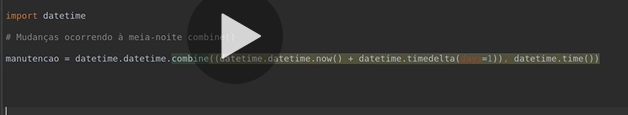


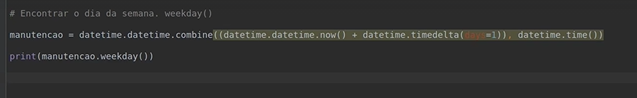




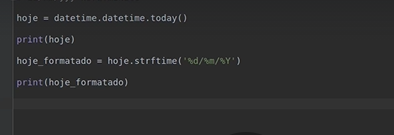
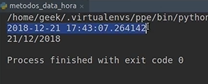
Aqui acima eu fiz o calculo para que some 3 dias a frente da data de meu boleto, para assim eu possa saber a data de vencimento do meu boleto, que vencerá três dias após a geração do meu boleto.

Agora vamos ver sobre métodos de datas e horas em Python:

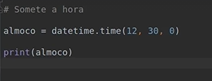


No método acima usando o combine eu combino uma data com outra data... nesse caso eu combinei a data de hoje mais um dia com o horário de 0 horas, que é o que datetime.time() retorna... 

O método weekday() retorna o dia da semana.

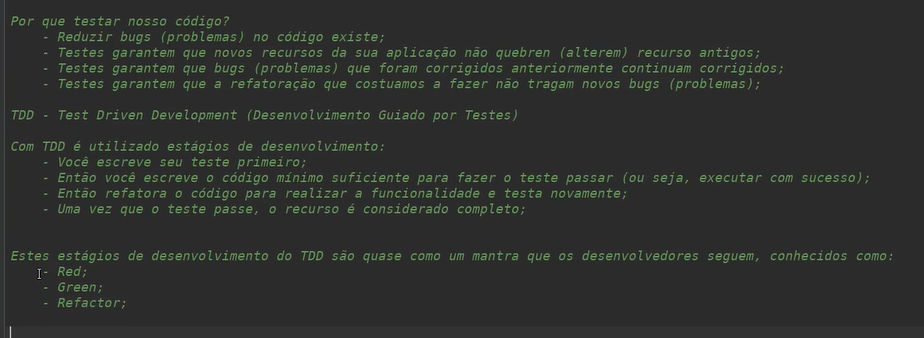
No método acima eu consigo usando o strftime() retornar a data no formato que eu quero... Nesse caso retornei no formato dia, mês e ano separados por barras.

Para retornar somente a hora eu posso utilizar o método acima... sendo o primeiro parâmetro a hora, o segundo parâmetro os minutos e o último parâmetro os segundos.

* Testes com Python:

Aqui veremos como faremos testes de código de formas automatizadas...



Agora vamos ver sobre os Assertions em Python...

Assertions são questionamentos, checagens... Ou seja, em Python utilizamos a palavra reservada ASSERT para tirar simples questionamentos de nosso código nos testes.

UTILIZAMOS O ASSERT EM UMA EXPRESSÃO QUE QUEREMOS CHECAR SE É VÁLIDA OU NÃO, OU SEJA, SE O CÓDIGO ESTÁ CERTO OU NÃO.

SE A EXPRESSÃO FOR VÁLIDA RETORNA NONE, E CASO TENHA ALGUM ERRO RETORNA ASSERTIONERROR.

Nós podemos especificar adicionalmente um segundo argumento ou mesmo uma mensagem de erro personalizada. 

O exemplo acima exemplifica que usamos o assert para checar se A é maior que 0 e B é maior que 0...



* Novas funcionalidades:

