**Python Collections**

Link para o colab, onde fiz toda a programação desse curso no lugar o pycharm: <https://colab.research.google.com/drive/1NHQhWymKR-BCoCLFM8OtsdJ7UoOKG11V#scrollTo=lI689BYCsnEX>.

**Parte 1: Listas e Tuplas**

1. **Aula 1 – Listas e Operações:**
   1. Coleções são utilizadas quando queremos trabalhar com diversos valores.
   2. *Append(valor)*: Adiciona um valor em uma lista.
   3. *Remove(valor)*: Remove um elemento da lista.
      1. Se tiver mais de um elemento/valor igual, ele removerá o primeiro que aparecer na lista.
   4. *Clear()*: Remove todos os itens da lista.
   5. Precisa começar a ler documentação!
   6. *Valor in lista*: Se queremos saber se um valor existe dentro de uma lista, utilizamos essa estrutura. O python retornará verdadeiro ou falso.
   7. Podemos criar condições com essas funções:



* 1. *Insert(posição, x)*: Adiciona um elemento em uma lista na posição desejada e o elemento (x) que quiser.
  2. Podemos utilizar o *append()*, com uma gambiarra, para acrescentar mais de um elemento em uma lista, mas, para isso, precisamos criar outra lista:



* + 1. O problema de fazer isso é que quando mandamos imprimir usando o for, ele imprime esses dois elementos como um único:



* 1. *Extend([n, n1, n2...])*: aumenta o iterável, fazendo com que essa lista se torne parte da lista já existente:
     1. Código:



* + 1. Resultado:



* 1. Tudo isso e muito mais está na documentação oficial no python.
  2. Podemos mandar o python fazer operações matemáticas com listas sem precisar criar grandes códigos.
     1. Código grande:



* + 1. Encurtado:



* 1. Também podemos criar condições utilizando o for:



* 1. Os “()” são opcionais.
  2. Utilizando *ifs* estamos utilizando filtros.
  3. Coloque sempre *“none”* como parâmetro padrão para listas para evitar problemas.
  4. O que aprendemos:
     1. O que é coleção;
     2. Criar lista;
     3. Verificar o tipo da lista e o tamanho da lista;
     4. Mostrar na tela o valor conforme sua posição na lista;
     5. Alterar valores que estão dentro da lista;
     6. Adicionar valores no final da lista;
     7. Percorrer a lista;
     8. Remover elemento da lista;
     9. Remover todos elementos da lista;
     10. Verificar se o elemento está dentro da lista;
     11. Inserir um elemento na posição que desejamos;
     12. Utilizar um for *comprehension*;
     13. Fazer filtragens;
     14. Criar uma função e deixar um valor padrão;
     15. Quais são os problemas da mutabilidade.

1. **Aula 2 – Tuplas:**
   1. Colocar objetos em listas não instancia eles, apenas referência mais uma vez.
   2. Se em uma lista colocar duplicatas de objetos, isso não fará com que crie outro objeto igual, apenas dá dois valores de referência a um mesmo objeto.
   3. Uma tupla é como uma lista, porém imutável, ou seja, não podemos usar *append()* ou *remove()*.
   4. Podemos colocar valores de vários tipos: *str*, *int*, *float*, dentre outros em uma tupla.
   5. Diferentemente da lista, que é demarcada com [], as tuplas são com ().
   6. A posição dos elementos também é significante, diferentemente das listas.
   7. Podemos criar listas de tuplas, dizendo que uma lista de usuários recebe 2 usuários que são tuplas, onde a ordem é nome, idade e ano de nascimento, imutáveis. Ex.:



* 1. Nesse caso podemos colocar outras tuplas dentro dessa lista usando o *append()*. Ex.:



* 1. O que aprendemos:
     1. Criar uma classe e métodos;
     2. Criar uma lista de referência para objetos;
     3. Lidar com referências;
     4. O que é uma tupla;
     5. Fazer tupla de listas;
     6. Diferença entre programação orientada a objetos e funcional;
     7. Fazer uma lista de tuplas.

1. **Aula 3 – Polimorfismo e Arrays:**
   1. Podemos importar uma biblioteca como alguma outra coisa/nome para referenciar. Ex.:



* + 1. Esse é o Array de verdade, as listas e *arrays* em python são coisas diferentes.
    2. Utilizado para termos uma eficácia maior quando trabalhamos com números.
    3. Todos os elementos tem que ser do mesmo tipo, não podemos começar uma *Array* com números e querer colocar str depois.
    4. *Array* é considerado um tipo no python.
    5. É muito mais comum a utilização de listas e tuplas no dia a dia.
    6. E mesmo quando vamos utilizar *arrays*, não utilizamos a do python, mas sim uma biblioteca chamada *numpy*. Ex.:



* + 1. Para utilizar essa biblioteca precisamos instalar ela com *“pip install numpy”* na linha de comando ou no colab.research.google.com
    2. Utilizando *np*:



* + 1. Podemos atribuir *arrays* em variáveis e somar valores a esses números. Ambos os números de uma Array sofrerão alterações. Ex.:



* + 1. Evitamos usar Array puro, sempre que precisamos utilizar para trabalhos numéricos, vamos de biblioteca *numpy*. Não é regra, apenas um costume.
  1. Em uma classe mãe podemos mandar ele *raise* um erro para métodos utilizados por classes que não sobrescreveram ele.
     1. Caso tenhamos vários tipos de conta bancárias e todas possuem o método *passa\_o\_mês* sobrescritos com taxas e valores diferentes, essas classes podem utilizar esse método tranquilamente, mas, caso uma delas não tenha sobrescrito e tente utilizar esse método, o código exibirá um erro. Ex. de classe mãe e erro para colocar no método:



* + 1. Isso era mais utilizado em versões anteriores do python, hoje podemos implementar o método e colocar uma anotação dizendo que ele é abstrato, mas lembrando sempre de importar a biblioteca abc e colocar a classe mãe como herdeira do *metaclass=ABCMeta*:



* + 1. Isso é muito útil para que você force todas as classes filhas a sobrescrever e utilizar esse método abstrato, uma vez que se não o fizerem, na hora que forem tentar instanciar uma classe filha que não tenha o método abstrato sobrescrito/implementado, irá dar um erro:



* 1. O que aprendemos:
     1. Conceito de herança e polimorfismo;
     2. Herdar classe;
     3. O que é o *duck typing*;
     4. Fazer um *array* no Python;
     5. Fazer anotações.

1. **Aula 4 – Igualdade:**
   1. O ‘==’ é uma representação de igualdade por objeto, ou seja, ele verifica se dois objetos que estão sendo comparados são o mesmo ou não.
   2. Podemos definir nossa própria definição de igualdade em python, sendo esse o *def \_\_eq\_\_* de *equals*.
      1. Podemos, por exemplo, dizer que nesse nosso objeto queremos saber se o código é igual entre 2 objetos. Ex.:



* + 1. Passando a ser representado assim:



* + 1. Valor esse que antes retornava falso, pois ele estava verificando se ambas as contas (1 e 2) eram o mesmo objeto ou não, ou seja, se tinham a mesma localização na memória ou não.
    2. Esse *\_\_eq\_\_* pode ser muito mais polido e complexo, comparando diversas coisas, colocando condições ou não, e assim por diante para deixar o nosso objeto cada vez melhor.
  1. O que aprendemos:
     1. Utilizar o *\_\_eq\_\_*;
     2. Utilizar boas práticas para comparação ;
     3. Usar o *isinstance()* para verificar se uma instância de um objeto;

1. **Aula 5 – Outros Builtins:**
   1. Temos uma sequência de idades e queremos saber a posição de algumas específicas
      1. Podemos utilizar a função *range()* do *len()* de idades para saber de qual posição até qual posição essa lista vai:



* + 1. Lembrando que a última posição é sempre exclusiva, ou seja, possuí 8 elementos do 0 até 7, o 8 não conta.
    2. Levando isso em conta, podemos fazer um for para cada posição, onde ele imprime a posição seguido pela idade encontrada naquela posição:



* + 1. Mas é claro que, como essa é uma coisa muito comum de se fazer, existe um *builtin* chamado *enumerate()* que faz esse tipo de coisa. Encontramos mais sobre ele na documentação oficial: <https://docs.python.org/3/library/functions.html>.
    2. *Enumerate()* faz vários trabalhos em sequência, é um iteravel. É um gerador *lazy*, ou seja, preguiçoso. Ele faz as duplinhas de posição e seu respectivo elemento, mas você precisa pedir para ele fazer isso. Não basta apenas colocar a variável dentro da função:



* + 1. Exatamente como o *range()*, ambos são *lazy*:



* + 1. *List()*: Função que cria listas através de funções iteráveis, assim como o *range()* e o *enumerate()*. Forçando assim a produção de uma lista com esses elementos:



* + 1. O mesmo vale para *enumerate()* e ele já nos devolve tuplas da posição com a idade/elemento respectivo:



* + 1. Contudo, com o *enumerate()* diferentemente do *range()*, quando usamos o for ele já automaticamente cria essas tuplas (na medida do necessário, já que ele é *lazy*) até onde o for mandar:



* + 1. Também podemos desempacotar essas tuplas direto no for, pedindo para ele exibir tanto o índice quanto a idade:



* + 1. Outra funcionalidade é poder desempacotar somente o que você deseja da tupla, não precisando desempacotar tudo. Como no exemplo de contas bancárias onde temos usuário, idade e ano de nascimento:
       1. Exemplo sem desempacotar:



* + - 1. Exemplo desempacotado: Só o nome



* + - 1. Exemplo desempacotado: Só a idade



* + - 1. Exemplo desempacotado: Só o ano de nascimento:



* + - 1. Exemplo desempacotado: Nome e ano de nascimento



* + - 1. Exemplo desempacotado: Ano de nascimento e idade



* + - 1. Exemplo desempacotado: Nome e idade



* + 1. Podemos também desempacotar somente uma variável sem dar nome as outras porque não nos interessa. Para fazer isso, basta colocar “\_” nas variáveis que não quer desempacotar:



* + 1. Sempre precisamos colocar a quantidade de nomes ou “\_” respectivo aos elementos da tupla, caso contrário, recebemos um erro:



* + 1. Ainda assim seria uma boa prática colocar nome para todos os elementos para melhorar a leitura.
  1. O que aprendemos:
     1. O que são *enumerated()*;
     2. Como funciona a função range;
     3. Desempacotar tuplas;
     4. Utilizar a função *len()*.

1. **Aula 6 – Ordem Natural:**
   1. *Sorted(variável)*: Devolve todos os elementos de uma lista ordenado em sequência crescente:



* 1. *Reversed(variável)*: O contrário do *sorted()* devolve a sequência em ordem decrescente:



* + 1. Devolve um iterador, portanto devemos utilizar o *list()* caso queiramos ver o resultado:



* + 1. Porém, se tivéssemos olhado na documentação, veríamos que o próprio *sorted()* possui o reverse como segundo argumento:



* + 1. O mesmo resultado pode ser obtido se utilizarmos o *list()*, *reversed()* e *sorted()* juntos:



* + 1. Lembrando que nada disso acima altera a variável original, apenas seu modo de exibição.
  1. Contudo, se quisermos alterar a variável em si, também possuímos métodos para tal, basta olhar na documentação:
     1. *Sort()*: função que coloca a lista em ordem crescente:



* + 1. *Reverse()*: reverse the elements of the list in place (dentro da variável, assim como o *sort()*).
  1. O que aprendemos:
     1. Utilizar a função *sorted()* para fazer a ordenação sem mudar o conteúdo na lista original;
     2. Usar a função *sort()* para ordenar atribuindo e mudando a lista original;
     3. Utilizar a função *reversed()* que ordena do maior para o menor sem alterar a lista original.

1. **Aula 7 – Ordenação Customizada:**
   1. Existem determinadas situações onde você quer que as informações venham à medida do pedido, e não que seja tudo entregue de uma vez, por isso o *reversed()* vai te entregando aos poucos, já o *sorted()* te entrega tudo de uma vez.
   2. O sorted() também coloca str em ordem crescente, já que toda str possui uma len().
      1. Nomes em letras maiúsculas e minúsculas interferem na ordem, onde as maiúsculas ficam em primeiro lugar e as minúsculas vem depois.
      2. Quando se trata de objetos temos um problema: o python não consegue usar a operação “<” em objetos, portanto, não consegue ordenar do menor para o maior em uma lista com vários objetos.
      3. Para corrigir esse problema, podemos passar uma key para o sorted() dizendo o que queremos que ele compare para fazer a ordenação. No caso de contas bancárias, podemos definir a key=extrai\_saldo:



* + 1. Porém, como podemos ver, precisamos criar um método para fazer isso, onde ele retorna o saldo, caso contrário dará erro. Outra ressalva é o fato de o sorted() retornar a representação e não o valor. Para resolver isso basta colocar um for:



* + 1. Um problema que temos na imagem acima é o fato de termos utilizado um atributo reservado para fazer o retorno. Para corrigir podemos utilizar attrgetter’s,. Link: <https://docs.python.org/3/library/operator.html>. A partir disso, basta importar a função attrgetter da biblioteca operator. Exemplo de como ficaria:



* + 1. O resultado é basicamente a mesma coisa que o anterior, porém sem a necessidade de criar uma função para retornar um atributo. No lugar dela, colocamos attrgetter e passamos como parâmetro o nome do atributo, seja ele privado ou não.
  1. \_\_lt\_\_: lessthan (menor que). Método implementado em classes que possibilita fazer operações de ordenação e verificações de “<”:
     1. Exemplo de aplicação em uma conta salário do nosso código:



* + 1. Após isso, podemos testar a operação e veremos que ao invés desse erro:



* + 1. Teremos esse resultado:



* + 1. Naturalmente, após implementar esse método interno, também podemos fazer operações de “maior que”, uma vez que é o exato oposto de lessthan:



* + 1. A partir disso, podemos simplesmente utilizar o for para sorted() de cada conta que ele irá imprimir normalmente:



* + 1. Naturalmente, também podemos utilizar os parâmetros que o sorted() suporta, como o reverso=True:



* 1. O que aprendemos:
     1. O que é ordem natural;
     2. Ordenar e comparar objetos;
     3. Utilizar o attrgetter
     4. Usar o \_\_lt\_\_: menor que (less than) para comparações.

1. **Aula 8 – Ordenação Total:**
   1. O attrgetter também aceita outros argumentos para poder desempatar quando existem dois números iguais, como no caso de contas bancárias, onde o saldo pode ser igual.
      1. Quando o saldo for igual, podemos pedir para o attrgetter desempatar com relação ao código bancário, deixando o que tenha o código menor em primeiro caso duas ou mais contas tenham o mesmo saldo:



* + 1. Outra opção seria implementar isso no próprio \_\_lt\_\_, onde dizemos que se o saldo de uma conta for diferente que o de outra usa e retorna isso, se não, usa o código:



* + 1. O resultado é esse:



* + 1. Porém, apesar de todas essas mudanças e melhorias que fizemos, se quisermos verificar se uma conta é “menor/maior e igual que”, o nosso código ainda dará um erro:



* + 1. Isso ocorre porque não implementamos isso no nosso objeto.
    2. Para corrigir isso existe o “*total ordering*”, ou, ordenação completa. Link para acessar o functools (ferramentas funcionais) do python onde podemos encontrar essa documentação bem detalhada sobre “*total ordering*”: <https://docs.python.org/3/library/functools.html>.
    3. Basicamente, exemplificando o que foi descrito acima temos que: importar o total\_ordering do functools e colocar o decorator, assim:



* + 1. Após essa implementação, e depois de termos implementado o *\_\_eq\_\_* e o *\_\_lt\_\_*, o total\_ordering se vira para fazer as outras ordenações de <=/>= dentre outras disponíveis:



* + 1. Não somente o \_\_eq\_\_ e o \_\_lt\_\_ precisam ser implementados, na documentação ele nos dá exemplos de outras implementações que podemos fazer no lugar, o importante mesmo é que tenha pelo menos dois métodos implementados.