**Validação de Dados no Padrão Nacional**

1. **Aula 1 – Validando CPF e Acessando PYPI:**
   1. Para verificar o tamanho de um iterável, ele precisa ser do tipo str. Int não reconhece/não tem len().

****

****

* + 1. Para corrigir, precisamos passar o cpf para o tipo str, mas ao invés de fazer isso direto na variável, fazemos na verificação:





* 1. Site para procurar pacotes pro python: <https://pypi.org>
     1. Sempre existe uma biblioteca para o que você está precisando, basta acessar o site e achar a que melhor se encaixa na sua necessidade.
     2. Instalamos uma biblioteca verificadora de documentos br, link: <https://pypi.org/project/validate-docbr/>.
     3. No meu caso, ocorreu erro de instalação, mas não é difícil resolver, basta seguir esses passos que será possível fazer a instalação facilmente: <https://www.jetbrains.com/help/pycharm/installing-uninstalling-and-upgrading-packages.html>.
     4. Para fazer a verificação, basta importar o método CPF da biblioteca validate-docbr após a instalação, instanciar o CPF e poderá printar a verificação, recebendo um True se for um CPF válido e False se não for:









* + 1. Podemos implementar essa biblioteca na nossa já existente validação de CPF criada:



* + 1. Assim podemos verificar se tem 11 dígitos ou não, se não tiver, sobe um erro de valor, se tiver, ele passa para a segunda verificação para ver se o CPF é válido ou não.
  1. Já possuímos uma máscara que nós mesmos fizemos no nosso código para exibir o CPF já formatado, porém, a biblioteca de validação que estamos utilizando já possuí esse método, sendo ele o .mask(“cpf”). Portanto, ao invés de fazer todo esse código que fizemos:



* + 1. Podemos simplesmente instanciar o objeto como fizemos com a validação e retornar o CPF com a máscara, dessa forma, nosso \_\_str\_\_ continuará exibindo nosso CPF formatado, mas com um código melhor:



* 1. Implementar bibliotecas é facilitar o processo e a vida, não tem porque reinventar a roda se podemos pegar algo que já está pronto e otimizar o nosso processo.
  2. O que aprendemos:
     1. Validar um documento pela quantidade de caracteres;
     2. Encontrar, instalar e importar bibliotecas no PyPI;
     3. Ler documentações de bibliotecas e utilizá-las em seus códigos.

1. **Aula 2 – Validando CNPJ e Construindo uma Factory:**
   1. Para fazer essa validação, continuaremos utilizando a biblioteca de validate-docbr.
      1. Criamos toda a validação do CNPJ, que nada mais é do que a mesma coisa que o CPF, mas com mais dígitos:



* + 1. Alteramos o nome da classe para implementar o CNPJ e colocamos mais um parâmetro nela, pois estaremos fazendo a verificação no construtor inicial assim como o CPF:



* + 1. Fizemos dois tipos de verificação, para saber se o que está sendo passado é CPF ou CNPJ, e fazer a sua respectiva validação. Caso não seja nenhum dos dois, sobe um erro de valor com o documento inválido.
  1. Fizemos a máscara para o CNPJ na mesma classe em que está a verificação e a máscara do CPF, além de implementar uma verificação no \_\_str\_\_ para que o programa saiba qual dos dois deve ser exibido:





* + 1. Exemplo de cada um no console:





* 1. Para melhorar o nosso código que estava muito confuso, fizemos o refatoramento dele, ou seja, criamos uma classe que, a partir dela, outras duas fazem as verificações separadas. Em resumo, uma classe principal reconhece se o que foi passado é um CPF ou CNPJ e a partir disso direciona o código para sua respectiva classe, fazendo a verificação, formatação e exibição do documento passado:
     1. Classe de refatoramento. Estática:



* + 1. Classe do CPF:



* + 1. Classe do CNPJ:



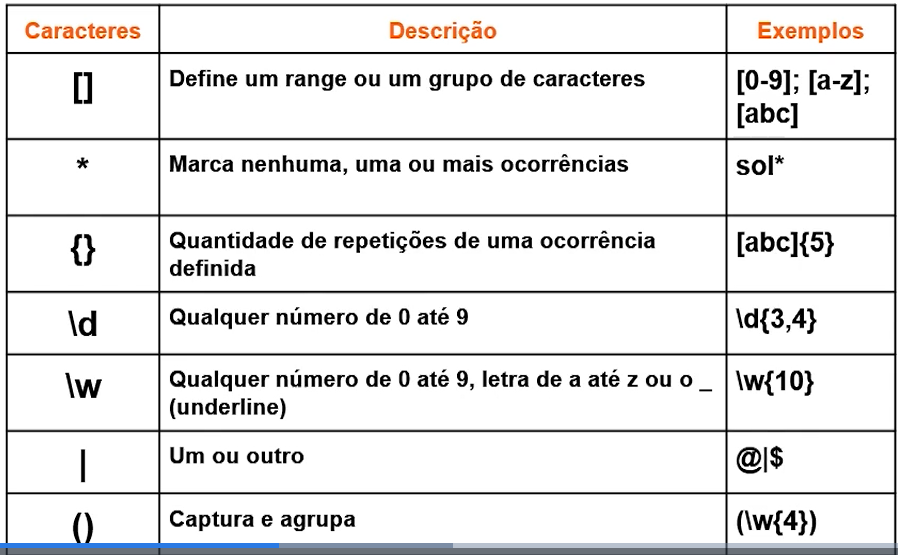
* + 1. Instanciando documentos e exibindo:



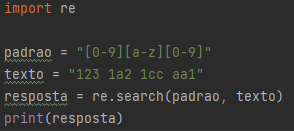


* 1. Artigos sobre refatoramento:
     1. <https://pt.wikipedia.org/wiki/Factory_Method>
     2. <https://www.thiengo.com.br/padrao-de-projeto-factory-method>
     3. <https://sourcemaking.com/design_patterns/factory_method>
     4. <https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Factory.html>
  2. O que aprendemos:
     1. Como utilizar mais uma classe da validate\_docbr;
     2. O que é o padrão de projeto Factory;
     3. Como e quando implementar uma Factory em nosso código.

1. **Aula 3 – Validando Telefone com Expressões ReGex:**
   1. Expressões regulares servem para encontrar padrões bem definidos em str.
   2. Utilizamos uma biblioteca chamada de RE para nos ajudar e esses são os caracteres especiais e o que fazem:



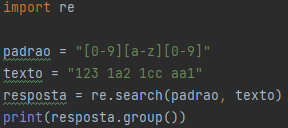
* + 1. Utilizando a biblioteca podemos criar uma variável com o padrão que desejamos procurar em determinada str/texto e pedir para a biblioteca procurar esse padrão no mesmo, assim:



* + 1. A resposta que recebemos no terminal é essa:

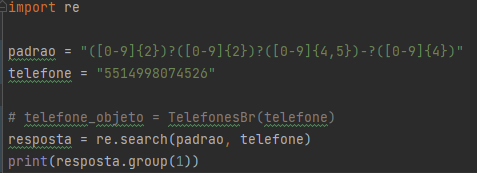


* + 1. Basicamente dizendo que a biblioteca RE encontrou o objeto e ele está na posição 4 até 7 (lembrando que as posições são contadas a partir do 0), e o objeto encontrado é o ‘1a2’.
    2. Caso queiramos receber uma resposta mais enxuta, apenas com o padrão encontrado, sem nenhuma outra informação extra, podemos utilizar a função .group() da biblioteca:



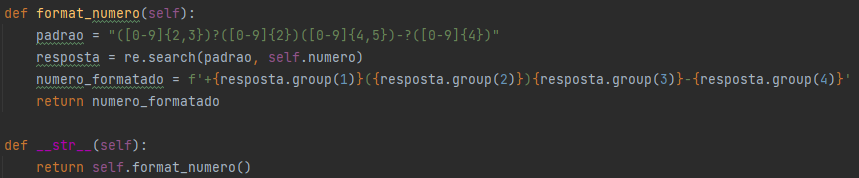


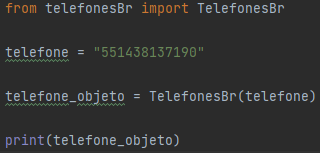
* 1. .findall(): função utilizada para retornar todas as correspondências regex em um str, não parando apenas na primeira.
  2. Fazer a separação da regex por grupos é uma ótima estratégia, pois com ela podemos pedir para o Search nos trazer apenas determinado grupo:





* + 1. Com esse conhecimento podemos criar mascaras de exibição, assim como fizemos com o CPF e CNPJ anteriormente:

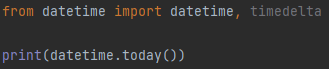






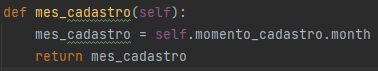
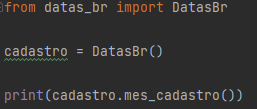
* 1. O que aprendemos:
     1. O que são expressões regulares;
     2. Como construir padrões e encontra-los dentro de textos;
     3. Como validar com expressões regulares;
     4. Como criar máscaras com expressões regulares.

1. **Aula 4 – Manipulando e Formatando Datas:**
   1. Utilizamos uma biblioteca chamada *datetime*.
      1. Com ela conseguimos usar o método/função chamada datetime.today() que trás todas as informações da data completa e hora exata da compilação do códico:



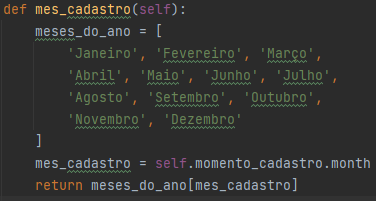


* + 1. Essa biblioteca também nos permite pegar informações separadas através de métodos já implementados nela, como por exemplo o month, que nos trás somente o mês atual:

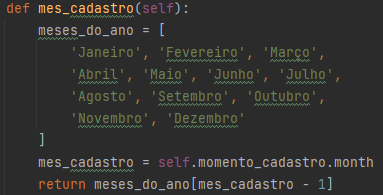


* + 1. Porém, como vimos, ele exibe apenas o número. Se quisermos que ele mostre o nome do mês, podemos criar uma lista com todos os meses e pedir para nossa função retornar o mês do ano com o índice sendo o mês de cadastro:



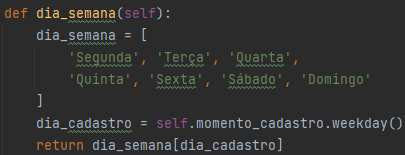


* + 1. Porém note que ele passou o mês errado, considerando dezembro como o mês 11. Isso acontece porque, como já vimos anteriormente, a lista conta a partir do 0, ou seja, o primeiro item é o índice 0. Para consertar isso podemos simplesmente subtrair 1 do índice do retorno:



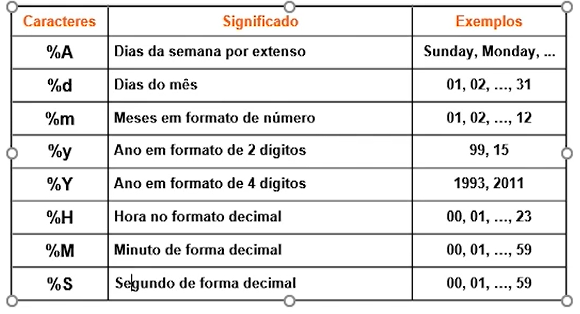


* + 1. Outra funcionalidade muito boa dessa biblioteca é a weekday(), onde ele retorna o dia da semana no momento da compilação do código e podemos utilizar a mesma técnica que para o mês para que ele retorne o nome do dia da semana e não número dela:

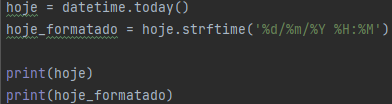




* 1. Strftime(): Função da classe datetime que permite fazer a formatação da data da maneira que desejar. Segue a tabela de caracteres especiais para fazer a formatação, como no regex:



* + 1. Dessa maneira, podemos realizar a formatação das nossas informações:



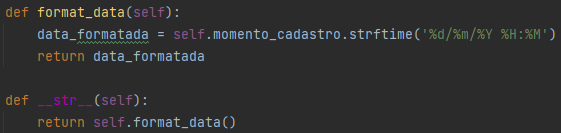


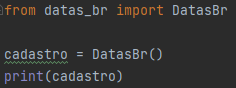
* + 1. Link para outros caracteres especiais existentes: <https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior>.
    2. Artigo que aborda outros métodos além desse: <https://blog.alura.com.br/lidando-com-datas-e-horarios-no-python/>.
    3. Porém, assim que formatamos nossa data e hora ela altera o seu tipo, se tornando uma str:





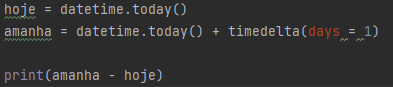
* + 1. Então ele deixa de ser um datetime.datetime e se torna um str, portanto, é bom fazermos isso somente para o usuário final ver, evitando que alteremos seu tipo durante os usos em outras partes do código.
    2. Utilizando esse conhecimento podemos criar uma função que formate nossa data e hora e um \_\_str\_\_ que retorne nossa formatação, para que assim que imprimirmos o cadastro ele retorne automaticamente nossa formatação:





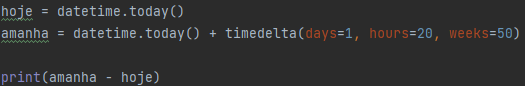


* 1. Timedelta: Responsável por fazer operações aritméticas com datas, podendo somar dias, horas, meses, anos dentre outros em alguma data:



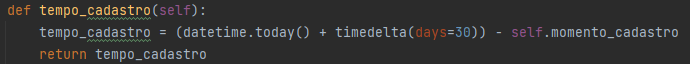


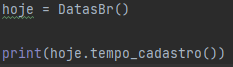
* + 1. Utilizando mais funções:





* + 1. Isso só é possível devido as inúmeras funções existentes e em execução por baixo dos panos na biblioteca datetime tais como: \_\_add\_\_, \_\_sub\_\_ e outras, que permitem adição e subtração e outras funções matemáticas dentro de uma classe.
    2. Qualquer classe que essas funções sejam colocadas, serão capazes de suportar esses tipos de operações aritméticas.
    3. Com esse conhecimento podemos incrementar uma função na nossa classe que permita saber quanto tempo estamos cadastrados em algum lugar:

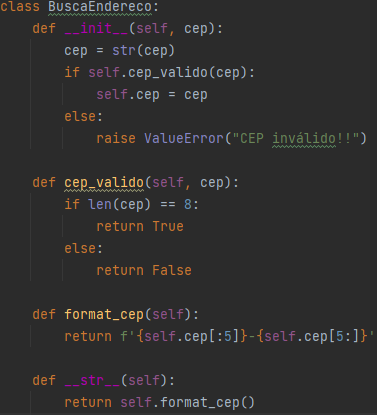




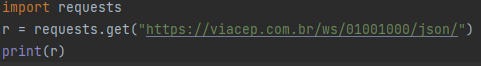


* + 1. Utilizamos o timedelta no nosso exemplo para que desse a diferença, mas no dia a dia essa informação do momento\_cadastro viria de um banco de dados.
  1. O que aprendemos:
     1. Como trabalhar com datas e horas no Python;
     2. Métodos da classe datetime;
     3. Para que serve o timedelta() e como utilizá-lo.

1. **Aula 5 – Trabalhando com CEP e Acassando uma API:**
   1. API’s são basicamente o que faz o intermédio do cliente com um http, requisitando (get), formatando e postando(post) da maneira que você quiser para o cliente final.
      1. Iniciamos a classe fazendo a validação, formatação, exibição e convertendo o cep para str:



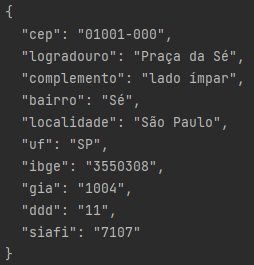
* 1. Podemos utilizar diversos sites como APIs, basta pesquisar API CEP e o primeiro link já pode ser usado: <https://viacep.com.br>.
     1. Todas as informações de como utilizar estão no link.
     2. Para utilizar uma API, precisamos importar uma biblioteca chamada request no python: <https://docs.python-requests.org/en/latest/>.
     3. Nesse site podemos ver todos os verbos que podemos utilizar, como cada um funciona e como utilizá-los.
     4. Essa biblioteca pode vir no python ou pode precisar ser instalada, de qualquer forma é de fácil utilização.
     5. Precisamos importar ela e, a partir disso, podemos começar a utilizar seus métodos e argumentos, tais como o *get*, responsável por requerir informação de uma API:





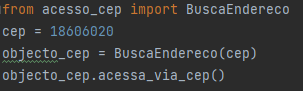
* + 1. Essa resposta é o status que indica se a resposta foi boa, lenta ou se o servidor está off. Resposta 200 quer dizer que nossa requisição foi e voltou sem nenhum problema.
    2. O final da extensão do link *json* é o tipo de resposta que queremos receber, como queremos que ela seja exibida. No link do site da API eles explicam melhor quais outras podemos usar e como usar.
    3. Passando o argumento .text para nossa requisição, recebemos como resposta toda a informação do CEP que foi passado para nossa API, informação completa sobre ele:



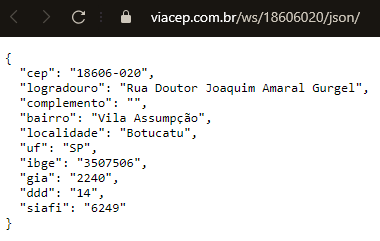


* + 1. Para alterar a url, fazendo a requisição a variar com o CEP passado, podemos simplesmente fazer uma atribuição formatada:

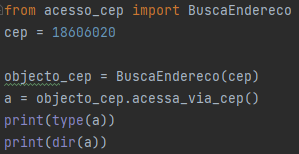


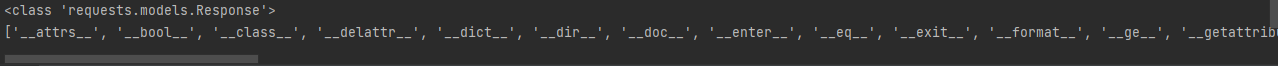
 

* + 1. Ao clicar nesse link recebido no console, somos redirecionados à uma página web no navegador com todas as informações do CEP passado:

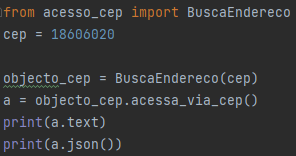


* 1. Dir(objeto): quando printado exibe todos métodos que uma classe possui. Muito útil e utilizado quando não sabemos o que a classe que estamos utilizando faz exatamente:





* + 1. Como podemos observar pela barra de rolagem, os métodos continuam por muito mais tempo. Através desse dir podemos ir testando e utilizando cada um desses métodos.
  1. Diferença entre .text e .json(): a primeira é que um é um atributo e o outro é um método, respectivamente. E a outra diferença é o método de exibição de cada um:





* + 1. Sendo um em linhas de acordo com a informação e o outro totalmente linear.
    2. Outra diferença é o tipo de cada um:





* + 1. Onde o text é um str (string) e o json um dict (dicionário).
    2. E, como já sabemos, é muito mais fácil conseguirmos extrair informações de dentro de um dicionário, uma vez que ao invés de o index ser definido de 0 à N, podemos obter a informação desejada através de chaves, onde ao pedirmos o CEP como índice, ele retorna o valor do CEP, o mesmo é valido para a cidade, logradouro etc. Ao invés de ter que saber qual é a posição de cada um em uma lista:





* + 1. Considerando tudo isso, podemos dar uma modificada no nosso método de acessa cep para que ele pegue todas as informações passadas pela API, converta em dict .json() e a partir dele nos retorne somente as informações que queremos, tal como bairro, cidade e uf:



