Boas Práticas com Python

Felipe C. R. dos Santos, ByLearn

1 Boas Práticas com Python

Apenas criar um código e fazê-lo funcionar não é o bastante, ainda mais se você pretende trabalhar em uma grande empresa, onde o trabalho em equipe é essencial.

Seu código, para poder ser melhor entendido e mais facil de dar manutenção/atualização pelos seus colegas (e até mesmo por você), precisa ser **bem escrito**, isto é, ser bem estruturado, organizado e **padronizado**.

Para isso, você sempre deve seguir as *Guidelines* e *Best Practices* da linguagem que está trabalhando.

Neste curso iremos nos atentar justamente a isso:
- Como escrever um código em Python da maneira certa

2 Entendendo Estruturas de Projetos

Quando ouvimos falar de estruturar um código pela primeira vez pode parecer algo difícil e complicado, ou que essa estrutura seja algo bem matemático e complexo, não é mesmo?

Porém, quando dizemos em *estruturar o código* estamos falando apenas sobre decisões a se tomar para, aproveitando todos os recursos da linguagem, criar um código limpo e eficaz.

Basicamente, um código **bem estruturado** é aquele código limpo, fácil de entender, com a lógica clara e evidente em todas as etapas. Além disso, também espera-se ele ser **bem organizado**, tendo seus arquivos e pastas bem divididos e setorizados.

Dessa forma, com o código bem estruturado e organizado, fica fácil qualquer um (tanto nós como nossos colegas de trabalho) entendermos bem **o que** cada parte do código faz, **porquê** ela faz aquilo e **como** ela faz.

Pronto! Dessa maneira o código deixa de ser um monstro de sete cabeças que nem o criador consegue domar e se torna bem mais fácil darmos manutenção no código e criarmos funcionalidades novas, não concorda?

Falando assim pode até continuar parecendo difícil e complicado, mas te garanto que é bem fácil, ainda mais tendo o apoio de toda uma comunidade de desenvolvedores que já passaram por isso antes e decidiram compartilhar seus próprios métodos para resolver estes problemas.

Sendo assim, prepare já o seu café, pois já vamos começar com o aprendizado das Boas Práticas com o Python!

2.1 Organizando seu projeto

Para organizarmos nosso projeto, a maneira mais fácil e recomendada para isso consiste em dar nomes claros e específicos para cada arquivo.

Nunca nomeie um arquivo de *x.py* ou *mat_mul.py*, prefira algo mais fácil de entender ao ler, como **multiplicar.py**.

Além disso, evite deixar seus arquivos de códigos espalhados por aí, cria pastas e coloque-os lá dentro.

Caso você tenha os arquivos *multiplicar.py*, *dividir.py*, *subtrair.py* e *adicionar.py*, porque não criamos a pasta **matematica** e colocarmos todos esses arquivos lá dentro?

Até mesmo coisas básicas como esta ajudam muito na hora de organizarmos e entendermos nosso código.

2.1.1 Organizando Módulos

Na hora de organizar seus módulos, busque sempre nomes curtos, evite precisar fazer uso de underlines e separadores e jamais misture nome com *namespace* ("pasta onde está o arquivo").

Exemplo:
- Ao invés de usar os nomes *bylearn_curso* e *bylearn_ebook*, eu poderia criar a pasta **bylearn** e dentro dela colocar os arquivos **curso** e **ebook.
- *Dessa forma usaremos apenas* ***bylearn.curso*** *e* ***bylearn.ebook****.*
- *Assim,* ***bylearn*** *torna-se o* namespace*.

Além de ficar melhor visivelmente, também faremos mais jus ao uso das pastas para segmentar e organizar nossos arquivos.

2.1.2 Organizando Pacotes

Antes de começarmos a falar sobre essa parte, vamos apenas destacar, embora você provavelmente já saiba, que Módulos são arquivos únicos do Python, então, cada arquivo de código (*.py*) é um **módulo** (*module*), já o conjunto destes módulos, isto é, o conjunto dos seus arquivos do projeto, são considerados o **pacote** (*package*) em Python.

Pronto, sabendo disso, vamos as dicas para uso do pacote. Como sabemos, qualquer pasta que contenha os arquivos **__init__.py** é considerada como um pacote, sendo ele o primeiro a ser executado.

Usando o exemplo anterior do ***bylearn.curso***, o python entenderá que primeiro ele deve ler a pasta **bylearn** e irá procurar pelo arquivo **__init__.py**, rodando-o em seguida (executando todo o código dele) para depois ir até o arquivo **curso.py**.

Sendo assim, caso você crie pacotes, uma boa prática é deixar o arquivo **__init__.py** completamente em branco, ou seja, deixá-lo vazio. Dessa forma, não haverá código nele para ser executado ao fazer alguma chamada para algum modulo dentro deste pacote.

3 Estruturando seu Repositório

O Python vem crescendo cada vez mais e está a apenas uma questão de tempo de se tornar a linguagem mais usada. Isso se deve a diversos fatores, como por exemplo, facilidade de desenvolvimento e alto poder de processamento, porém, muitos consideram que o principal motivo é o fato do Python ser uma **linguagem de código aberto**.

Pode parecer algo simples e banal, mas este é um fator de importância incontestável! Essa transparência permite que qualquer desenvolvedor no mundo todo colabore totalmente com o desenvolvimento e aperfeiçoamento da linguagem e acaba incentivando-os a tornar seus projetos também de código aberto, dessa forma, criando soluções completas que se são de grande ajuda na hora da programação, tornando **totalmente desnecessário recriar a roda**.

Sabendo disso, não tem porque não retribuirmos o favor e colaborarmos com a comunidade do Python!

A melhor forma para isso é criarmos um repositório *GIT*, que basicamente é um local onde podemos armazenar nosso código de forma bem fácil e que possibilita muitas facilidades, como por

exemplo o versionamento de código (controle das versões do projeto).

O principal e mais famoso site para isso é o Github.

Portanto, nada mais justo do que aprendermos a estrutura básica de um repositório e como melhorarmos nossa arquitetura.

3.1 A Estrutura Básica

Como nosso curso é focado em Python, por mais que a boa prática da organização de repositórios seja algo geral para todas as linguagens, vamos ser mais específicos ao Python no momento, abordando os arquivos e arquitetura padronizados para essa tão amada linguagem.

3.1.1 Organização e Ordenação

Os repositórios são comumente organizados na seguinte ordem:

1. Título do Projeto

2. Descrição do Projeto
3. Arquivos do Projeto
4. Conteúdo do *Readme* (Leia-me)

Dessa forma, podemos entender que se um projeto não possui um título claro e uma boa descrição, é mais difícil o usuário descer a tela até ver seus arquivos. Porém, ainda é bem provável ele chegar até seus arquivos, mas aqui que mora o problema...

Imagina você se deparando com um projeto de 2000 arquivos, todos na mesma pasta e com nomes aleatórios. Eu desistiria do projeto, você faria o mesmo, não?

Pois é, por isso é importante organizarmos bem nossos arquivos, dando uma boa estrutura para eles, para que tire o medo do programador que visitar seu repositório e faça com que ele continue descendo a tela, lendo melhor o seu *Readme* e entendendo seu projeto.

3.1.2 Arquivos

README.md
- O Readme é o arquivo principal para descrever e explicar nosso projeto, através do Markdown (por isso a extensão *md*), podemos

escrever de uma forma bem organizada as informações principais do nosso projeto.

LICENSE.md

- É o arquivo que contém informações sobre a licença de uso e distribuição do nosso repositório. Por ser código aberto, é aconselhavel buscar licenças mais 'liberais', que não limite o seu uso por parte de quem usar nosso código.

setup.py - É o arquivo de instalação do nosso projeto, basta o programador baixar nosso repositório e rodar esse arquivo.

requirements.txt

- Como falamos anteriormente, não podemos ficar recriando a roda, então provavelmente nosso código também possui códigos, módulos e bibliotecas externas. Neste arquivo detalhamos quais os requisitos externos para utilizar nosso código.

Makefile

- O Make no Python pode não ser tão usual quanto na linguagem C, mas o Makefile é um arquivo muito útil para definirmos tarefas genéricas ao nosso projeto. Não é um arquivo obrigatório, mas bem recomendado caso seja viável ao seu projeto.

3.1.3 Pastas

"root" - Root é como chamamos nossa pasta raíz, isto é, aquela inicial do nosso projeto. É nela onde vamos inserir todos os arquivos ditos acima e todas as pastas abaixo.

docs

- Esta é a pasta responsável pela documentação do nosso projeto, caso você tenha um projeto grande, ficará inviável dar manutenção e aprimorá-lo sem ter uma documentação, e é justamente nessa pasta que você irá colocá-la.

tests

- Não é porque um código está rodando que ele está **funcionando**. Nós precisamos testá-lo para garantir que todas as partes estão executando corretamente. Caso você tenha alguma rotina de testes automatizados, é aqui que você irá inserir os arquivos.

nome_do_pacote - Agora precisamos apenas criar uma pasta com nome do nosso pacote ou projeto e colocar seus arquivos aqui dentro. Lembrando que os arquivos do projeto também devem estar organizados, separados por pastas e com nomes bem definidos, como comentamos anteriormente.

Um bom exemplo de repositório neste modelo pode ser visto em: https://github.com/navdeep-G/samplemod

4 PEP - *Python Enhancement Proposal*

O primeiro tema que vamos abordar será o *Python Enhancement Proposal* (Proposta de Aprimoramento do Python), ou **PEP** para os íntimos.

Como já sabemos, o Python é uma linguagem totalmente de código aberto e que tem uma relação bem saudável com sua comunidade de desenvolvedores.

Muitos desses desenvolvedores pretendem ajudar a comunidade Python fazendo dicas e manuais de como aprimorar a linguagem e tais propostas são submetidas ao PEP.

Quando uma proposta é aceita pela comunidade e pela equipe de manutenção e administração oficial do Python, ela é indexada no **PEP** com algum identificador numérico único.

Por exemplo, o **PEP 8** é responsável pelo *Code Style* (estilo de codificação) do Python e o **PEP 20** pelo *Zen of Python*.

Para consultar todas as propostas aceitas, basta consultar o **PEP 0**: https://www.python.org/dev/peps/

Não foi atoa que dentre todos os **PEPs** nós usamos apenas o **8** e o **20** como exemplo, eles são os mais conhecidos e mais bem falados quando nos referimos a *Boas Práticas* e como estruturar o código da melhor forma possível.

No decorrer do curso, além de algumas dicas de programação, falaremos bem sobre esses dois.

5 Code Styles

Os *Code Styles* são guias de estilização de código, ou sejam, eles vão te auxiliar a como estruturar o seu código.

Já passamos por estruturação de projetos, de repositórios e agora vamos para os códigos.

Para isso, vamos aprender sobre o PEP8, o PEP20 (Zen of Python) e demais Code Styles bem aceitos pela comunidade python

6 PEP8

Assim como dito anteriormente, o PEP8 é o principal PEP sobre Code Styles, nele são abordados diversos tópicos das mais distintas parte do código e dá varias dicas sobre como estruturar seu código para a melhor entendimento e visualização.

Esse Guideline foi escrito em 2001 por Guido van Rossum, Barry Warsaw e Nick Coghlan, sendo que Guido é o **criador do Python**.

6.1 Nomeclatura

A principal informação que temos que ter antes de aprender até mesmo as regrinhas de nomeclatura é saber que:

> Explícito é melhor do que implícito.

Sendo assim, não importa o que for nomear, sempre deixe de uma maneira intuitiva e que expresse a real motivação para a sua criação. Como por exemplo:

- Não nomeie uma variável de **var1**, coloque algo mais intuitivo, como **resultado_multiplicacao**.
- Não chame de **velo_leop** (implícito) quando pode chemar de **velocidade_leopardo** (explícito).

- Não cause ambuiguidade. **desligar()** é diferente de **desligar_pc()** e **desligar_servidor()**.

Além disso, vale lembrar que o simples é melhor que o complexo. Sempre tente utilizar palavras pequenas e fáceis.

Agora que já sabemos disso vamos aprender como nomear as coisas no Python!

6.1.1 Variáveis

- Use apenas letras minúsculas.

- Caso precise de mais de uma palvra, use underlines para melhor a visualização e leitura.

- Evite usar apenas uma letra no nome, com exceções aos casos de variáveis temporárias de iteração.

Exemplos:
- resultado_divisao.
- media_alunos_escola.
- nome.
- for **i** in **items**.
- Embora seja só uma letra, é uma variável temporária e para iteração, tendo sua função explicitamente a mostra.

6.1.2 Funções e métodos

- Use apenas letras minúsculas.
- Caso precise de mais de uma palvra, use underlines para melhor a visualização e leitura.
- Nunca use apenas uma letra.

Exemplos:
- calcular_media().
- renomear_aluno().
- somar().

6.1.3 Módulos

- Use apenas letras minúsculas.
- Evite usar mais de uma palavra, mas caso precise, separe-as por underline.
- Não use o **namespace** no nome do arquivo, mesmo que separado por underline.

Exemplos:
- calculadora.py.
- helpers.py.
- rede_neural.py.
- Não use testes_cadastro e testes_formularios, use testes.cadastro e testes.formularios.
- No caso, testes é um namespace e não deve estar presente.
- Não use meu_formulario.py, use formulario.py.
- No caso, evitar nomes compostos desnecessários, deixe nomes curtos.

6.1.4 Pacotes

- Use apenas letras minusculas.
- Evite precisar de mais de uma palavra, mas caso seja necessário, não as separe.

Exemplos:
- calculadora.
- bpylearn.
- redeneural.

6.1.5 Classes e Exceções

- Comece todas as palavras com letra maiúsculas.
- Não separe as palavras.

Exemplos:
- Veiculo.
- CarroEletrico.
- CarroGasolina.

6.1.6 Constantes

- Use todas as letras em maiúsculo.
- Caso precise de mais de uma palavra, as separe por underline.

Exemplos:
- VERSAO_PHP_SERVIDOR.
- FABRICANTE.
- MODO_DEBUG.

6.2 Linhas em Branco

Muitas vezes uma simples quebra de linha nos ajuda imensamente a entender o real sentido de uma parte do código, além de deixar tudo bem mais fácil de visualizar.

Para isso, o PEP8 nos trás algumas dicas e formas de usar as linhas em branco (quebra de linhas) de forma otimizada.

6.2.1 Classes e Funções (top-level)

- Deixe um espaço de duas linhas em branco entre classes e funções em top-level (nível mais a esquerda possível).

Exemplo:

```
class PrimeiraClasse:
    pass


class SegundaClasse:
    pass
```

```
def funcao_top_level():
    return None
```

6.2.2 Métodos dentro de classes

- Deixe apenas uma linha em branco entre os métodos dentro das classes.
- O primeiro método fica 'colado' na classe.

Exemplo:

```
class Classe:
    def primeiro_metodo(self):
        return None

    def sgundo_metodo(self):
        return None
```

6.2.3 Funcionalidades dentro dos métodos

- Deixe uma linha em branco entre blocos de funcionalidades distintas.
- Em outras palavras, separe as partes da sua lógica para melhor visualização.
- A primeira linha do método sempre fica colado com sua definição.

Exemplo:

```
def calcular_media():
    nome = 'Felipe'
    primeira_prova = 10
    segunda_prova = 9

    soma_notas = primeira_prova + segunda_prova
    quantidade_provas = 2

    return soma / quantidade_provas
```

6.3 Tamanho máximo das linhas

Para melhor visualização do conteúdo de cada linha, gerando maior legibilidade ao código, o PEP8 usa um limite máximo de caracteres por linha, onde nenhuma linha deve conter mais do que 79 caracteres.

Porém, nem sempre é possível uma linha/operação/rotina/função ter menos de 79 caracteres originalmente e nesses casos, é preciso quebrar a linha, 'separando' seu conteúdo.

Como adendo, é legal sabermos que o limite de 79 caracteres é para linhas de código, já para linhas de comentários o limite é de 72, mas veremos isso de forma mais detalhadas nos capitulos seguintes.

Abaixo vemos algumas formas de se concatenar essas linhas e conseguir ficar dentro do limite imposto pelo PEP8:

6.3.1 Separando argumentos por vírgula

```
def minha_funcao(arg_um, arg_dois,
                 arg_tres, arg_quatro):
    return arg_one
```

6.3.2 Separando por barra invertida

```
from meu_pacote import exemplo1, \
    exemplo2, exemplo3
```

6.3.3 Operadores binários

```
total = (primeira_variavel
         + segunda_variavel
         - terceira_variavel)
```

6.3.4 Strings longas

```
string = ("minha "
          "string em "
          "multiplas "
          "linhas")
```

6.4 Comentários

Como já falamos brevemente nos capítulos anteriores, as linhas de comentários possuem tratamentos especiais e também entram no PEP8.

A primeira regra é ter um limite de apenas 72 caracteres. Além disso, também precisamos usar sentenças completas, sem 'comer palavras' ou ocultar informações, começando as frases novas com letras maiúsculas.

Por fim, como boa prática, sempre que atualizar seu código, atualize também seu comentário.

6.4.1 Comentários em Bloco

Sempre que for preciso, você pode usar comentários em bloco, usando o caractere de 'jogo da velha' (**#**) no início de cada linha.

Caso seja preciso mais de um parágrafo, não tem problema, apenas deixe uma linha de comentário em branca (apenas como o caractere de comentário).

Exemplo:

```
def calcular_raiz_dobro(numero):
    # Primeiro iremos dobrar o valor do número, ou seja,
    # faremos número * 2.
    #
    # Em seguida, iremos calcular a raiz quadrada, portanto,
    # usamos a função sqrt do módulo math.
    dobro = numero * 2
    raiz = math.sqrt(dobro)
```

6.4.2 Comentários Inline (mesma linha)

Nada te impede também de usar comentários na mesma linha do código em questão, mas este uso deve ser feito com moderação.

Além dessa moderação, o PEP8 também pede um espaço de 2 caracteres em branco desde o final da linha para o inicio de comentário, como por exemplo:

```
    dobro = numero * 2  # Multiplicamos por dois para termos o dobro.
```

É importante também se atentar se realmente é necessário o comentário ou se apenas uma melhor nomeclatura resolveria. Prefira sempre a nomeclatura.

Errado:

```
x = 'Boas Práticas com Python' # Nome do curso
```

Correto:

```
nome_curso = 'Boas Práticas com Python'
```

6.4.3 Documentação (docstring)

Para as boas práticas na documentação o PEP8 informa apenas poucas coisas, sendo elas:

- Usar três aspas duplas no lugar de três aspas simples;
- Comentar apenas o necessário, mas de forma clara;
- Iniciar na mesma linha que usar as três aspas duplas;
- Finalizar os docstrings em linha extra (sem texto);
- Alinhar o texto com o início da primeira aspas;
- Documentar todos métodos, classes, módulos e funções públicas.

Já para o conteúdo e organização dentro do seu docstring, isso fica livre segundo a PEP8, porém, temos a PEP257 que é bem aceita.

Em resumo, a PEP257 diz para **separarmos por blocos** os parâmetros, atributos, retornos e afins, deixando uma linha de "cabeçalho" seguida de uma linha de traços no início de cada um desse bloco separador.

Por exemplo, seguinto tanto a PEP8 quanto a PEP257 temos:

```
def fazer_pipoca(sabor, tempo):
    """ Essa função faz pipoca no microondas.

    Parametros
    ----------
    sabor : str
        Sabor da sua pipoca.
    tempo : int
        Tempo para fazer a pipoca.

    Retorno
    -------
    Retornará um objeto do tipo Pipoca com o sabor escolhido.
    """
```

```
    pipoca = escolher_pipoca(sabor)
    modo = microondas.modo('pipoca',tempo)

    return microondas(modo,sabor)
```

A principal função das docstrings são nos ajudar a entender melhor a classe/método/função/pacote, sabendo seu objetivo e funcionamento.

Para isso basta usar a função **help()** e passar por parâmetro o que queremos ajuda.

```
help(fazer_pipoca)
```

Com isso temos de retorno o seguinte:

```
Help on function fazer_pipoca in module __main__:

fazer_pipoca(sabor, tempo)
    Essa função faz pipoca no microondas.


    Parametros
    ----------
    sabor : str
        Sabor da sua pipoca.
    tempo : int
        Tempo para fazer a pipoca.

    Retorno
    -------
    Retornará um objeto do tipo Pipoca com o sabor escolhido.
```

Caso não seja preciso realizar muitos comentários na documentação (se uma linha apenas bastar), você pode tanto abrir quanto fechar as aspas duplas na mesma linha, como no exemplo abaixo:

```
def mostrar_nome():
    """ Mostrará o meu nome na tela """
    print(meu_nome)
```

6.4.4 Não alimente o óbvio

Comentar e documentar é não só recomendável como impressindível para que seu código esteja de acordo com as boas

práticas e seja de fácil entendimento e manutenção, porém, não devemos exagerar.

Em vários momentos o código já está com seu funcionamento explícito e um comentário alí apenas serviria para aumentar o número de linhas e caracteres de leitura, não é nem um pouco recomendado você explicar o óbvio.

Veja o exemplo abaixo sabendo como é o uso **incorreto** de se comentar o obvio:

```
nome = 'Felipe'  # Informa o nome
idade = 22  # Informa a Idade
# Iremos printar o nome da pessoa
# seguido da idade dela
print(nome,idade)
```

6.5 Indentação

Também temos algumas regras e dicas para como se usar a indentação no nosso código para facilidar a leitura e organização, para isso, como podemos ver a seguir.

6.5.1 Espaçamentos no inicio

Algo que sempre gera discussão entre os programadores é a guerra entre **Tabs *vs* Espaços**, muitos defendem a facilidade do **Tab** e outros preferem a consistência dos **Espaços**.

Para o PEP8, o mais importânte é a consistência, sendo assim, o vencedor é o *Espaço*. Para ser mais específico, é recomendado usar sempre **4 Espaços** e nunca usar Tab.

6.5.2 Argumentos nas linhas de baixo

Caso a linha ultrapasse os 79 caracteres, como visto acima, temos que quebrar a linha, certo?

No caso de funções e métodos temos que realizar essa operação nos argumentos e o PEP8 diz que o argumento da linha debaixo deve iniciar na mesma posição do primeiro argumento da linha superior, veja o exemplo:

Correto:

```
def minha_funcao(arg_um, arg_dois,
                 arg_tres, arg_quatro):
```

```
    return arg_one
```

Incorreto:

```
def minha_funcao(arg_um, arg_dois,
         arg_tres, arg_quatro):
    return arg_one
```

Ou seja, os argumentos estão alinhados.

6.5.3 Abertura e fechamento de chaves e colchetes

Por fim, o PEP8 também ajuda na hora de identar a abertura e o fechamento de Chaves *{}* e Colchetes *[]*.

Como algumas vezes teremos que quebrar linhas que contemplam listas, será necessário quebrar a abertura e fechamento dos colchetes.

Uma recomendação do PEP8 é abrir a lista *[* na linha em que definimos a variável, mas começar a inserir os dados nas linhas de baixo. Já para o fechamento, também inserí-lo em uma linha extra (após terminar de inserir os dados), alinhada com o primeiro elemento da lista, como no exemplo abaixo:

```
lista_numeros = [
    1, 2, 3,
    4, 5, 6,
    7, 8, 9
    ]
```

6.6 Espaçamento entre caracteres

O PEP8 também irá te auxiliar sobre os espaçamentos entre caracteres, segundo ele você deve usar caracteres nos seguintes casos:

- Operadores de Atribuição:
  - **=, +=, -=**...
- Comparações:
  - **==, !=, >, <=**...

- **is**, **is not**, **in**, **not in**.

- Booleanos:
  - **and**, **not**, **or**.

- Operadores matemáticos:
  - +, -, /, *.

Sendo assim, temos como exemplo os seguintes usos:

```
if x == 0:
    x += y

if y not in z:
    print(y + x)

if cond1 and cond2:
    lista = []
```

Porém, veja como fica estranho essa regra ao tivermos mais de um dos casos listados acima acontecendo na mesma expressão, como duas comparações, dois booleanos, booleanos e operações e etc...

```
if y > 0 and x + 1 > z:
    if y not in z and x is not y:
        print("Versão estranha")
```

Para isso, o PEP8 criou uma solução onde apenas a operação de menor importância/prioridade deve ter espaçamento, as demais ficam sem espaço. Além disso, o uso do bom e velho parênteses não mata ninguem, veja como tudo fica mais claro:

```
if y>0 and x+1>z:
    if (y not in z) and (x is not y):
        print("Versão melhorada")
```

Veja em outros exemplos:
Errado:

```
y = x ** 2 + 5
z = (x + y) * (x - y)
```

Correto:

```
y = x**2 + 5
z = (x+y) * (x-y)
```

6.7 Casos gramaticais

Como todo sabem, o Python é bem simples de se aprender e seu uso se assemelha ao inglês escrito, sendo assim, o PEP8 também utiliza dos princípios gramaticais do inglês para regrar os espaçamentos, veja a seguir como é fácil:

6.7.1 Vírgulas

Apenas há espaçamento após a vírgula

Errado:

```
minha_lista = [1 , 2 , 3 , 4]
minha_lista = [1 ,2 ,3 ,4]
```

Correto:

```
minha_lista = [1, 2, 3, 4]
```

6.7.2 Parênteses, Chaves e Colchetes

Não há espaçamento nem antes nem depois.

Errado:

```
z = ( x+y ) + ( y * x )
minha_tupla = ( 1 , 3 )
minha_lista = [ 1, 2, 3, 4 ]
meu_dicionario = { '1':'a' , '2': 'b' }
```

Correto:

```
z = (x+y) + (y*x)
minha_tupla = (1, 2, 3)
minha_lista = [1 ,2 ,3 ,4]
meu_dicionario = {'1':'a', '2':'b'}
```

6.8 Recomendações

Além das regrinhas acima, específicas para cada caso, o PEP8 também trás algumas observações e recomendações gerais, como podemos observar logo abaixo.

6.8.1 Comparação de *Booleano* com *True* e *False*

Sabendo que um booleano já recebe um valor de verdadeiro e falso e toda comparação por trás dos panos valida apenas a veracidade da condição (se é verdadeira ou falsa), fica redundante checar se algo é ou não true. Sendo assim, devemos evitar o uso do == *True* ou == *False*.

Errado:

```
meu_bool = 10 > 5
if meu_bool == True:
    print("10 é maior do que 5")

meu_segundo_bool = jogador.vencer
if meu_segundo_bool == False:
    print("O jogador perdeu")
```

Correto:

```
meu_bool = 10 > 5
if meu_bool:
    print("10 é maior do que 5")

meu_segundo_bool = jogador.vencer
if not meu_segundo_bool:
    print("O jogador perdeu")
```

6.8.2 Sequências vazias

Sempre que for fazer alguma comparação com sequências vazias, elas devem ser *falsas* na validação do ***if***.

Embora pareça que uma lista vazia seja aquela cuja comparação do ***len()*** seja 0, o Python também já daria diretamente o valor de falso para ela, sendo assim, podemos evitar o uso do len da seguinte forma:

Errado:

```
minha_lista = []
if not len(minha_lista):
    print('A lista está vazia!')
```

Correto:

```
minha_lista = []
if not minha_lista:
    print('A lista está vazia!')
```

6.8.3 Usar *is not* é melhor do que *not ... is*

Para gerar melhor legibilidade no seu código, dê preferência ao *is not*, ele é mais fácil de ler e entender do que o *not ... is*, veja o exemplo abaixo:

Errado:

```
if not x is None:
    return 'x existe!'
```

Correto:

```
if x is not None:
    return 'x existe!'
```

6.8.4 Saiba quando checar o *None*

Muitas vezes, checar se é *True* ou *False* não é bem o que esperamos, como por exemplo, para a seguinte função:

```
def calcular_idade(ano_nascimento,ano_atual=None):
    if not ano_atual:
        print("Você não informou o ano atual, calcule sua idade usand
        return

    idade = ano_atual - ano_nascimento
    print("Você fez/fará", idade, "anos neste ano")
```

Parece que a função está correta, não é mesmo? Veja uns exemplos abaixo:

```
calcular_idade(1996,2019)
```

Output = Você fez/fará 23 anos neste ano

```
calcular_idade(1996)
```

Output = Você não informou o ano atual, calcule sua idade usando o ano atual e subtraindo 1996

Porém, o que acontece se nós voltarmos ao passado e entregar esse programa para algum morador do ano Zero? Ok... Provavelmente ele iria assutar e sair correndo, mas quando ele entendesse a tecnologia e usasse o programa, veja o que aconteceria:

```
calcular_idade(-50,0)
```

Output = Você não informou o ano atual, calcule sua idade usando o ano atual e subtraindo -50

Estranho, não? Pois é, isso ocorre devido ao fato de 0 ser considerado como falso para a lógica booleana, sendo assim, o Python sempre entenderá que o Zero é falso. Ou seja... Nossa função acima está **errada**.

Sendo assim, como apenas queremos saber se o usuário informou ou não um ano, deveremos comparar o valor com *None*, da seguinte forma, sendo ela a **correta**:

```
def calcular_idade(ano_nascimento,ano_atual=None):
    if ano_atual is None:
        print("Você não informou o ano atual, calcule sua idade usand
        return

    idade = ano_atual - ano_nascimento
    print("Você fez/fará", idade, "anos neste ano")
```

Ou até mesmo checar se **não** é *None*

```
def funcao(arg1,arg2=None):
    if arg2 is not None:
        print("A execução pode continuar")
    else:
        print("Você deve informar um valor para arg2")
```

6.8.5 Use *startswith()* e *endswith()* ao invés de fatiar

Sempre que puder use o *startswith()* ou o *endswith()* no lugar de usar o *slicing* (fatiar).

Errado:

```
nome_arquivo = "ola mundo.jpg"

if nome_arquivo[:3] == 'ola':
    print('O arquivo começa com a palavra "olá"')

if nome_arquivo[-3:] == 'jpg':
    print('É um arquivo JPG')
```

Correto:

```
nome_arquivo = "ola mundo.jpg"
```

```
if nome_arquivo.startswith('ola'):
    print('O arquivo começa com a palavra "olá"')

if nome_arquivo.endswith('jpg'):
    print('É um arquivo JPG')
```

6.9 Linha no final do arquivo

Sim, o PEP8 tem informações até mesmo sobre isso. A recomendação é sempre deixar uma linha em branco após o final do arquivo.

Isso se dá pelo caractere de nova linha ser considerado como um "finalizador de linha" e não um delimitador. Sendo assim, essa nova linha funcionará como deliimtador.

Além disso, também melhora a compatibilidade com terminais de sistemas Unix e para facilitar a combinação de scripts (sem que a ultima linha do primeiro arquivo se mescle com a primeira linha do segundo arquivo).

6.10 Bônus: Checadores de PEP8

O PEP8 como você percebeu tem várias regrinhas e no início pode ser um pouco difícil de decorar todas, não é mesmo?

Felizmente, há várias formas de checarmos se nosso código está seguindo as regras do PEP8 de forma fácil, vejamos algumas delas.

6.10.1 Linters

Linter são prgramas que analisam códigos e avisam erros. Eles são bem úteis para checarmos se nosso código está condinzente com o padrão do linter atual.

Por sorte nós temos o **flask8**, um linter justamente feito para checar se seu código Python está seguindo todas as regras do PEP8 e que nos avisa quais linhas estão erradas (e qual o motivo).

Para instalá-lo, basta executar o seguinte comando:

```
pip install flake8
```

Em seguida, podemos rodá-lo da seguinte forma:

```
flake8 meu_script.py
```

E como resposta teremos algo como, por exemplo:

```
meu_script.py:1:17: E231 missing whitespace after ','
meu_script.py:2:21: E231 missing whitespace after ','
meu_script.py:3:17: E999 SyntaxError: invalid syntax
meu_script.py:6:19: E711 comparison to None should be 'if cond is Non
```

6.10.2 Formatadores Automáticos

Os formatadores automáticos, ou autoformatters, fazem exatamente o que o nome diz, eles pegam o seu código e formatam automaticamente para o *code style* em questão, no nosso caso, o PEP8.

O mais famoso deles para o Python é o **black**, que podemos instalar da seguinte forma:

```
pip install black
```

Em seguida, podemos rodá-lo da seguinte forma:

```
black meu_script.py
```

E como resposta teremos algo como, por exemplo:

```
reformatted meu_script.py
All done! [U+2728] [U+FFFD][U+FFFD] [U+2728]
```

Pronto! Nosso código já está seguindo as normas do PEP8. Porém, nem sempre é bom deixar apenas para os Linters e Autoformatters, é sempre bom já irmos aprendendo e usando as dicas acima no nosso dia a dia, isso irá aumentar muito nossa produtividade e rendimento, além, é claro, de nos acostumarmos a programar como bons programadores.

7 PEP20 - Zen Of Python

O PEP20 é sem sombras de dúvidas um dos mais amados pela comunidade Python!

Diferente do PEP8, os ensinamentos do Zen Of Python não são no formato de regras teóricas e formais, eles são feitos através de aforismos, isto é, frases curtas que explicam um preceito moral ou prático.

Ele é tão marcante e aceito pela comunidade Python que ele viroua té mesmo um *easter egg* (segredo escondido) no Python que para acessar basta executar o seguinte código:

```
import this
```

Teremos então como retorno, o texto origial (em inglês) com os 19 aforismos de Tim Peters, veja só que legal:

> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one– and preferably only one –obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.
> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea – let's do more of those!

Agora em tradução livre, temos o seguinte: >The Zen of Python, by Tim Peters
>
> Bonito é melhor que feio.
> Explícito é melhor que implícito.
> Simples é melhor que complexo.
> Complexo é melhor que complicado.
> Linear é melhor do que aninhado.
> Esparso é melhor que denso.
> Legibilidade conta.
> Casos especiais não são especiais o bastante para quebrar as regras.

> Ainda que praticidade vença a pureza.
> Erros nunca devem passar silenciosamente.
> A menos que sejam explicitamente silenciados.
> Diante da ambiguidade, recuse a tentação de adivinhar.
> Deveria haver um — e preferencialmente apenas um — modo óbvio para fazer algo.
> Embora esse modo possa não ser óbvio a princípio a menos que você seja holandês.
> Agora é melhor que nunca.
> Embora nunca freqüentemente seja melhor que já.
> Se a implementação é difícil de explicar, é uma má idéia.
> Se a implementação é fácil de explicar, pode ser uma boa ideia.
> Namespaces são uma grande ideia — vamos usar mais desses!

Ah, ficou curioso sobre o porquê do *holandês* décimo quarto aforismo?
Simples, isso foi apenas uma homenagem ao Guido van Rossum, o holandês criador do Python e BDFL (Benevolent Dictator for Life, ou Benevolente Ditador Vitalício) da linguagem, ou seja, quem sempre ajuda a manter e possui a palavra final.

7.1 Bonito é melhor que feio

Esse nós já vimos bastante, basicamente os *code styles* servem para isso.
Não faça do seu código uma bagunça, prefira sempre deixar ele bonito e organizado.

Errado:

```
exp = 2+3
res = 5
cond1 = exp == res
if cond:
    print("Verdade")
```

Correto:

```
if 2+3 == 5:
    print("Verdade")
```

7.2 Explícito é melhor que Implícito

Nunca deixe seu código implícito, fazendo com que o próximo programador (ou as vezes até mesmo você, no futuro) tenha que tentar adivinhar o que está acontecendo alí.

Errado:

```
x = 'F'
y = 22
z = b_u(f,22)
```

Correto:

```
inicial = 'F'
idade = 22
usuario = buscar_usuario(inicial,idade)
```

7.3 Simples é melhor que complexo

Provavelmente em algum momento você terá como fazer o que deseja usando diferentes tecnologias e metodologias, como por exemplo, salvar o dado do usuário em um banco de dados local (e apenas local, nunca usando um servidor) ou serializar para um JSON.

Uma dessas opções é bem simples e prática, enquanto a complexidade da outra é bem maior e desnecessária. Vá pelo caminho mais simples!

Em alguns casos o complexo até terá um desempenho melhor, mas caso não seja nada significativo ou realmente necessário, não complique o seu código.

Errado:

```
def store(measurements):
    import sqlalchemy
    import sqlalchemy.types as sqltypes

    db = sqlalchemy.create_engine('sqlite:///measurements.db')
    db.echo = False
    metadata = sqlalchemy.MetaData(db)
    table = sqlalchemy.Table('measurements', metadata,
        sqlalchemy.Column('id', sqltypes.Integer, primary_key=True),
        sqlalchemy.Column('weight', sqltypes.Float),
        sqlalchemy.Column('temperature', sqltypes.Float),
        sqlalchemy.Column('color', sqltypes.String(32)),
        )
    table.create(checkfirst=True)

    for measurement in measurements:
```

```
        i = table.insert()
        i.execute(**measurement)
```

Correto:

```
import json

def store(measurements):
    with open('measurements.json', 'w') as f:
        f.write(json.dumps(measurements))
```

7.4 Complexo é melhor que complicado

Embora o simples seja melhor que o complicado, muitas vezes não podemos fazer da maneira simples, é triste, mas é verdade.

Seguindo o exemplo passado, vamos supor que agora os dados do usuário são inseridos no servidor e não na máquina local. Será que seria seguro e eficaz ficar criando milhares ou até milhões de json no servidor? Será que não deveriamos usar o banco de dados mesmo dessa vez?

Porém, como veremos em breve, é muito fácil complicar o simples, mas é ainda mais fácil complicar algo que já é complexo!

No caso, imagine pegarmos o exemplo do banco de dados no tópico anterior, que é complexo, porém até simples de entender e usar, e usarmos a maneira abaixo, veja como ficará bem mais complicado:

Errado:

```
def store(measurements):
    import MySQLdb
    db = MySQLdb.connect(user='user', passwd="password", host='localh

    c = db.cursor()
    c.execute("""
        CREATE TABLE IF NOT EXISTS measurements
          id int(11) NOT NULL auto_increment,
          weight float,
          temperature float,
          color varchar(32)
          PRIMARY KEY id
          ENGINE=InnoDB CHARSET=utf8
          """)
```

```
    insert_sql = (
        "INSERT INTO measurements (weight, temperature, color) "
        "VALUES (%s, %s, %s)")

    for measurement in measurements:
        c.execute(insert_sql,
            (measurement['weight'], measurement['temperature'], measu
            )
```

Correto:

```
Usar da maneira 'apenas complexa' mostrada no tópico anterior
```

7.5 Linear é melhor do que aninhado

Não aninhe códigos a menos que seja realmente necessário, deixar o código linear (alinhado, um abaixo do outro) é bem melhor e mais legível

Errado:

```
if time.venceu_jogo:
    if campeonato.final:
        print("Você venceu o campeonato!")
    else:
        if time.posicao == 1:
            print("Você está em primeiro lugar, continue assim")
        else:
            if time.possibilidade_vencer > 0:
                print("Você ainda pode ser o vencedor do campeonato")
            else:
                print("Você venceu o jogo, mas perderá o campeonato")
else:
    print("Você perdeu o jogo")
```

Correto:

```
if not time.venceu_jogo:
    print("Você perdeu o jogo")
    return

if campeonato.final:
    print("Você venceu o campeonato!")
    return
```

```
if time.posicao == 1:
    print("Você está em primeiro lugar, continue assim")

if time.possibilidade_vencer > 0:
    print("Você ainda pode ser o vencedor do campeonato")
    return

print("Você venceu o jogo, mas perderá o campeonato")
```

7.6 Esparso é melhor que denso

Assim como nós já vimos no PEP8, temos de dar alguns espaços internos (esparsar) dentro de um bloco de código para melhorar a legibilidade daquela parte, "dividindo" as funcionalidades e responsabilidades deste trecho de código.

Vamos usar como exemplo o código feito no tópico anterior (de forma linear) sem deixar esparsado, veja como ficaria bem pior:

Errado:

```
if not time.venceu_jogo:
    print("Você perdeu o jogo")
    return
if campeonato.final:
    print("Você venceu o campeonato!")
    return
if time.posicao == 1:
    print("Você está em primeiro lugar, continue assim")
if time.possibilidade_vencer > 0:
    print("Você ainda pode ser o vencedor do campeonato")
    return
print("Você venceu o jogo, mas perderá o campeonato")
```

Correto:

```
Fazer da maneira esparsada (dividida) igual no tópico anterior
```

7.7 Legibilidade conta

Este tópico nada mais é do que um aglomerado dos anteriores, você deve sempre focar na legibilidade do seu código.

Foque em fazer o simples, foque em fazer apenas o necessário, faça de uma maneira estruturada e bonita, deixe seu código sempre o mais fácil de ler e entender apenas *batendo o olho*.

7.8 Casos especiais não são especiais o bastante para quebrar as regras, ainda que praticidade vença a pureza

Este é um dos mais filosóficos, se não o mais filosófico de todo o *Zen of Python*.

Por mais que as vezes pareça que um certo caso especial que você está trabalhando seja melhor fazendo de qualquer jeito do que fazendo usando o PEP8 ou PEP20, acredite, ele não é tão especial assim.

Um farol vermelho continua sendo um sinal vermelho independente do horário ou local, podendo sofrer multas caso algum motorista ultrapasse este sinal ainda em vermelho, certo?

Com códigos são a mesma coisa, se você começar agora a ir pelo mais rápido e pelo mais prático, você começará a pegar mal costume e irá cometer um *crime de desenvolvimento de software*, você fará um código totalmente desestruturado e isso só tenderá a ir piorando com o tempo.

A dica aqui é se educar para sempre fazer o certo, mesmo que pareça mais difícil.

7.9 Erros nunca devem passar silenciosamente, a menos que sejam explicitamente silenciados

Nesta parte o mais importante de se saber é que erros **devem** ser mostrados, você não pode apenas ignorar ou dar um *pass* sem ter nenhuma informação disponível.

Também não é recomendável deixar apenas um *except* genérico. Veja abaixo os exemplso:

Errado 100%:

```
try:
    print(1/0)
except:
    pass
```

Errado 50%:

```
try:
    print(1/0)
except:
    print("Ops...Aconteceu um problema, tente novamente")
```

Correto:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("Por favor, não divida por zero!")
```

Viu só?
- Nós começamos com um código que não está nem ai para erros, apenas tenta fazer tudo e ser der errado deu; - Fomo para um outro código que apenas fala "Olha, ocorreu um erro aqui" e deixa o usuário adivinhar o que aconteceu; - Finalmente, fomos para um que realmente sabe os possíveis erros enfrentados, que vai te ajudar a entender os problemas do seu código e ao mesmo tempo informar ao usuário como usar corretamente o seu software.

Agora um exemplo ainda melhor de como fazer corretamente essa nossa divisão:

```
try:
    n = int(input("Numero: "))
    m = int(input("Numero: "))
    x = n / m
except ZeroDivisionError:
    print("Não divida por zero")
except ValueError:
    print("Insira apenas números inteiros")
```

7.10 Diante da ambiguidade, recuse a tentação de adivinhar

Caso você tenha alguma dúvida, se o seu código está ambíguo, não tente apenas adivinhar, se o código pode dar errado, provavelmente em algum momento ele dará errado.

A melhor forma para esse caso é prevenir e não apenas buscar entender melhor futuramente e resolver a ambiguidade.

Por exemplo, sempre que for fazer seu código, deixe bem claro o que você pretendia fazer e o que você fez.

Deixe documentado e comentado o que você fez para que outros (ou até mesmo você) entenda futuramente o que aquela parte do código está fazendo.

Errado:

```
def processar(response):
    db.store(url, response.body)
```

Neste caso, não conseguimos saber exatamente como virá o conteúdo do *body*, qual será o *charset* dele. Portanto, ou explicamos como é o uso normal ou evitamos essa ambiguidade.

Veja uma maneira melhor de fazer isso, que funciona de maneira geral e evita ambiguidade:

Correto:

```
def process(response):
    charset = detect_charset(response)
    db.store(url, response.body.decode(charset))
```

Dessa forma, estamos detectando automaticamente o *charset* e em seguida decodificando ele.

7.11 Deveria haver um — e preferencialmente apenas um — modo óbvio para fazer algo

Há uma 'piada' inserida aqui ao mesmo tempo que se passa um ensinamento. Muitos programadores dizem que Perl, por exemplo, é uma linguagem que varias coisas fazem o mesmo resultado, de maneiras complicadas e diferentes, já no Python, a ideia desde sua criação é ser uma linguagem com uma solução, o que faz com que a linguagem seja bem mais simples do que suas alternativas.

Porém, ao mesmo tempo ele passa um ensinamento mais genérico e útil, se você de cabeça já pensa numa solução para um problema, se for uma solução óbvia, não perca tempo tentando encontrar uma nova.

Imagine o seguinte caso, você quer elevar um número fixo (10) a uma determinada potência que é escolhida pelo usuário, você pode usar a forma normal, que é usando ** (dois asteriscos) ou pode pensar em uma alternativa, como vemos abaixo:

Errado:

```
numero = 10
potencia = 5  # Usuário escolheu

resultado = 0
for i in range(potencia-1):
    resultado *= numero
```

Correto:

```
numero = 10
potencia = 5  # Usuário escolheu

resultado = numero ** potencia
```

Embora o exemplo acima seja fácil, acredite, muita gente consegue complicar muito um código simples por achar que *"É tão simples que só pode ser pegadinha"*

7.12 Embora esse modo possa não ser óbvio a princípio a menos que você seja holandês

Essa é outra parte engraçada, serve apenas para continuar a parte anterior de forma humorada, fazendo referência ao holandês Guido van Rossum, o criado do Python.

Dessa forma, obviamente que ele vai saber a maneira *Pythonica* de se resolver aquilo, tendo sido ele o inventor.

7.13 Agora é melhor que nunca. Embora nunca freqüentemente seja melhor que já

Este é outro ensinamento mais filosófico do que prático, basicamente, o que ele diz é que você não pode ficar remediando algo para sempre, não deixe tudo para depois quando estiver melhor preparado e mais treinado, se não esse adiamento será eterno.

Quer resolver algum problema? Vá lá e resolva, **pesquise e coloque a mão na massa**. E é aqui que está a segunda parte do ensinamento. Não faça tudo de imediatismo, sem pensar antes, sem ter certeza do que está fazendo.

Embora fazer seja melhor do que adiar, precisamos pelo menos de uma pesquisa antes, precisamos pensar antes de agir.

7.14 Se a implementação é difícil de explicar, é uma má idéia

Se nem você que é o criador da funcionalidade sabe descrever como ela funciona, então certamente essa implementação foi uma péssima ideia!

De nada adianta algo funcional se ninguém jamais conseguirá fazer uma manutenção ou melhoria naquele código.

O próprio Guido (criador do Python) diz para fazer códigos para outros programadores e não para o computador, pois

querendo ou não, o público alvo do seu **código** (quem vai ler) é outro programador.

7.15 Se a implementação é fácil de explicar, pode ser uma boa ideia

Porém, não basta a ideia ser fácil de explicar, isso só quer dizer que ela não é dispensada pela *'regra'* anterior.

Além de ter uma idéia fácil de explicar, você deve analisar a complexidade de implementação, a complexidade na notação Big-O, o quanto de recursos do hardware esse método irá precisar, se ele condiz com o restante do seu projeto e por aí vai...

Apenas o tempo e prática irão te ensinar quando uma ideia é boa ou não.

7.16 Namespaces são uma grande ideia - vamos usar mais desses

O último ensinamento baseia-se nos *namespaces*, onde nos gera a necessidade de pensar sobre a utilidade e importância dos namespaces, que ajudam na organização não só dentro do código (script) quanto no projeto (organização dos arquivos).

Por exemplo, podemos pegar uma pasta com 30 ou 50 arquivos com várias funcionalidades e responsabilidades e podemos dividir esses arquivos em conjuntos menores, mais organizados e estrutados. Dessa forma, fica bem mais fácil sabermos com o que estamos lidando naquela parte do nosso pacote.

Veja o exemplo abaixo:

Errado:

```
import animais

def animar_animal():
    cachorro = animais.modelos(cachorro)
    gato = animais.modelos(gato)

    animador = animais.controladores.animacao

    animador.animar(gato,"miando") # Pega o modelo 3D e executa uma a
    animador.animar(cachorro,"latindo")
```

Correto:

```
import animais.modelos.gato as gato
import animais.modelos.cachorro as cachorro
import animais.controladores.animacao as animador

def animal_animal():
    animador.animar(gato,"miando")
    animador.animar(cachorro,"latindo")
```

8 Extras

Vejamos a seguir outras boas práticas, que não são essencialmente do Python nem de algum PEP em específico, mas todo bom programador deveria conhecer e fazer uso.

8.1 Não recrie a roda

Como dito no início, o Python é uma linguagem aberta com uma comunidade de programadores imensa, todos compartilhando códigos e soluções.

Dessa forma, provavelmente sempre terá algo pronto que possa te ajudar a atingir o resultado final do seu projeto, seja ele qual for.

Todos esses (ou a maioria deles) ficam armazenados no indexador de pacotes do Python, ou Pypi, para os íntimos, e pode ser acessado através do link: https://pypi.org/.

É de forte recomendação não tentar recriar a roda, muito menos se for apenas para mostrar que você é capaz disso!
Não se sinta fraco ou um programador pior, vá sem medo e aproveite todas as facilidades que os pacotes do Python podem te oferecer. Isto apenas fará de você um programador mais eficaz e produtivo!

8.2 Corrija o código imediatamente

O clichê "Nunca deixe para amanhã o que você pode fazer hoje" não seria clichê se não fosse verdade.
Deixar para depois apenas vai criar uma bola de neve de erros no seu código e aumentar o número de "gambiarras" para fazer seu monstrengo funcionar.

Achou algum erro? Corrija-o imediatamente!

8.3 Use ambientes virtuais

Os ambientes virtuais, ou virtual environments (ou até mesmo virtualenv/venv), são ótimos para melhorar a confiabilidade e segurança do seu código.

Um ambiente virtual basicamente é uma emulação/virtualização de um sistema com o Python instalado, mas apenas isso, sem nenhuma outra dependência.

Caso precise de algum módulo, biblioteca ou script para o seu projeto, você pode instalá-lo diretamente no seu ambiente virtual.

Sendo assim, vamos supor que um projeto use 30 dependências na versão mais atualizada possível, outro projeto usa apenas 5 dependências, mas todas em versões mais antigas, onde 2 delas são conflitantes com as dependências do primeiro projeto.

Se você usar tudo no seu ambiente normal, certamente dará problemas e dores de cabeça, mas caso esteja usando ambientes virtuais, isso não será problema algum! Basta criar um novo *virtualenv* para cada projeto e instalar apenas o necessário.

Caso nunca tenha usado um ambiente virtual, você primeiro precisa instalar o *virtualenv*, para isso:

```
pip install virtualenv
```

Agora basta ir na pasta do seu projeto pelo terminal e digitar:

```
virtualenv env
```

Onde *env* é o nome do seu ambiente virtual.

Agora basta ativarmos esse ambiente para o Python entender com qual *virtualenv* estamos trabalhando:

```
Linux: env\Scripts\activate
Windows: env\Scripts\activate.bat
```

Para desativarmos esse ambiente é mais fácil, basta executarmos:

```
deactivate
```

8.4 Criando logs

Muitas vezes precisamos de informações do que estamos fazendo no nosso código para serem usadas futuramente, para isso, precisamos de um registro de eventos, ou seja, um **log**.

Para isso python possui o módulo **logging** por padrão e pode ser usado da seguinte forma:

```
import logging

# Configurações principais do nosso registrador (logger)
logging.basicConfig(filename='meus_registros.log',
                    filemode='w',
                    format='%(name)s - %(levelname)s - %(message)s',
                    level=logging.DEBUG)

# Definiremos os nomes dos nossos logs
logger = logging.getLogger('log_principal')
logger2 = logging.getLogger('log_secundario')

# Abaixo temos as mensagens que queremos registrar,
# junto ao seu tipo de mensagem (de acordo com a função chamada)
logger2.debug('Eu sou uma mensagem de Depuração (Debug)')
logger2.info('Eu sou uma mensagem de Informação')
logger.warning('Eu sou uma mensagem de Aviso (Warning)')
logger.error('Eu sou uma mensagem de Erro')
logger.critical('Eu sou uma mensagem Crítica')
```

Onde no **basicConfig** podemos fazer as configurações principais do nosso logger, como o nome do arquivo, a forma a se trabalhar com ele (como apenas continuar escrevendo ou deletar todo o conteúdo e recomeçar), o formato de output das mensagens e o nível mínimo da mensagem para se logar.

Vejamos agora o resultado que tivemos ao executar o código acima e abrir o arquivo **meus_registros.log**:

```
log_secundario - DEBUG - Eu sou uma mensagem de Depuração (Debug)
log_secundario - INFO - Eu sou uma mensagem de Informação
log_principal - WARNING - Eu sou uma mensagem de Aviso (Warning)
log_principal - ERROR - Eu sou uma mensagem de Erro
log_principal - CRITICAL - Eu sou uma mensagem Crítica
```

8.4.1 Hierarquia das mensagens

Como dito acima, podemos definir qual o nível mínimo de mensagem para logarmos. Para uma melhor configuração, precisamos saber exatamente qual a hierarquia correta, afinal, imagina logarmos o que não precisa ou ficar sem informações que queriamos.

Então, a hierarquia da menos importante até a mais importante é a seguinte:

1. Debug;

2. Info;
3. Warning;
4. Error;
5. Critical.

8.5 Testes

Embora poucos realmente façam isso, todos sabem que testes são partes fundamentais do desenvolvimento de software e arquiteturas como o TDD (Test Driven Development) ajudam muito na hora de organizar o seu código de forma eficiente, voltada para testes unitários, por exemplo.

Usando testes automatizados, nós conseguimos garantir uma maior confiabilidade para o nosso código, sabendo se nossas funcionalidades estão sendo executadas como gostariamos ou se há algum imprevisto acontecendo, como por exemplo, não tratar uma divisão por zero e retornar um erro.

Atualmente os três principais tipos de testes de software são: 1. Testes unitários; 2. Testes de integração; 3. Testes funcionais.

Aproveitando o assunto dos testes unitários, no python nós podemos usá-los através do **unittest**.

Para isso, basta primeiro importarmos o módulo **unittest**, criar uma classe aceitando como parâmetro um **unittest.TestCase** e em seguida criar um método de teste. Pode até parecer complicado, mas é bem fácil, veja o exemplo a seguir:

```
import unittest

class MyTest(unittest.TestCase):
    def teste(self):
        self.assertEqual(proximo(3), 4)
```

No caso, estamos testando se a função *proximo* está funcionando e para isso, iremos nos garantir que se enviarmos o valor 3, o retorno será 4 (***assertEqual***). Para isso, trataremos a função *proximo* como sendo a seguinte:

```
def proximo(x):
    return x + 1
```

Pronto, agora é só executar o **unittest**, para isso, podemos fazer o seguinte:

```
if __name__ == '__main__':
    unittest.main()
```

Ou caso estejamos executando o código no **IPython**, **Jupyter** e derivados:

```
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

Dessa forma, o código completo fica da seguinte forma:

```
import unittest

def proximo(x):
    return x + 1

class MyTest(unittest.TestCase):
    def test(self):
        self.assertEqual(proximo(3), 4)

if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

8.6 Orientação a Objetos e Design Patterns/Principles

Outra boa prática que podemos ter é estruturar nosso código com Orientação a Objetos, criando classes e objetos, principalmente quando temos um programa com elementos que possuem um tempo de vida útil longo, como uma janela de interface gráfica, por exemplo, ou estruturas de dados complexas que são constantemente usadas.

Além disso, é bem interessante seguir os padrões de projeto de software (Design Patters) como os disponíveis no Gang of Four (GoF) e princípios de projeto de software (Design Principles) como o SOLID, por exemplo.

Embora o uso de Design Patters/Principles seja mais complicado para um programador iniciante, não é nenhum monstro de sete cabeças e rapidamento você irá se acostumar a empregar tais Padrões e Princípios, fazendo com que seu código fique imensamente mais padronizado, robusto, prático e fácil de aprimorar e dar manutenção.

Deixo, como orientação, pesquisar sobre o assunto, começando por Orientação a Objetos, caso não saiba, em seguida ir para Design Principles, como o SOLID e por fim aprender os padrões disponíveis no GoF.

Seu código certamente ficará muito melhor!

8.7 Pesquise... Pesquise SEMPRE

Esse tópico não foi deixado para o final por acaso, ele pode ser considerado até mesmo o mais importante!

Ningúem nunca saberá tudo e nem terá todas as respostas, além disso, sempre surgirão problemas e necessidades novas que teremos que aprender.

Muito provavelmente, você terá duvidas que não foram vistas nesse livro, afinal, se abordassemos todas as possibilidades para todos os códigos, isso nunca terminaria. Um exemplo disso pode ser o uso de um dicionário, onde a boa prática é **não** usar o has_key(), como no caso abaixo:

Não Recomendado:

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print d['hello']    # printa 'world'
else:
    print 'hello não encontrado'
```

Recomendado:

```
d = {'hello': 'world'}

print d.get('hello', 'valor não encontrado') # printa 'world'
print d.get('thingy', 'valor não encontrado') # printa 'valor não enc

# Ou até mesmo:
if 'hello' in d:
    print d['hello']
```

Tudo isso você só aprenderá de uma forma **pesquisando**! Precisa trabalhar com arquivos? Uma rápida procura no Google pode te ensinar que a seguir **não é recomendada:**

```
f = open('file.txt')
a = f.read()
```

```
print a
f.close()
```

O correto para trabalhar com textos seria algo como:

```
with open('file.txt') as f:
    for line in f:
        print line
```

Veja como fica bem melhor sem precisar se preocupar com o close() no final e lendo de uma maneira mais *Pythonica*!

Em resumo, minha mensagem final para você é nunca parar de aprender, nunca parar de pesquisar e sempre estar disposto a entender coisas novas e seguir o caminho mais correto ou recomendado, mesmo que não seja como você está acostumado.

9 Considerações Finais

Muito obrigado por ter checado até aqui! Espero que o que você aprendeu com este livro tenha te ajudado.

As boas práticas são muitas e várias vezes nos causam uma dificuldade para nos acostumarmos, mas isso é totalmente normal e rapidamente pegamos o jeito! O importante é nunca desistir e sempre tentar melhorar e evoluir como programador.

Por ter lido até aqui, tenho certeza que você é esforçado e certamente irá dominar todas essas boas práticas e muitas outras em pouquíssimo tempo!

Caso tenha qualquer dificuldade e precise de ajuda, pode entrar em contato conosco sempre que quiser.
Abaixo deixo alguns meios de contato, caso precise:

Facebook: @bylearn
Instagram: @bylearn
Youtube: @bylearn
Site: https://www.bylearn.com.br
Dojô: https://dojo.bylearn.com.br
E-mail: contato@bylearn.com.br

10 Curso de Python Completo

Ah, e se você quiser um curso completo de Python, que te ensine do Júnior ao Sênior, te faço um convite para participar do nosso super treinamento Python.

Lá nós ensinaremos tudo o que você precisa para se destacar no mercado de trabalho Python.

Além da versão em vídeo deste livro, você também encontrará muitos outros temas legais e interessantes para sua carreira, e tudo sem precisar de nenhum conhecimento prévio.

Ah, e o melhor, todo o conteúdo é vitalício e você terá nosso suporte sempre preparado para solucionar suas dúvidas.

Para saber mais sobre todos os nossos benefícios, acesse: https://bylearn.com.br/cursos/python

Te espero lá,

Obrigado pela companhia e leitura até aqui,

Felipe.