2

TEXTO BASE

PROGRAMAÇÃO ORIENTADA A OBJETOS

Faculdade IMPACTA

Texto base

2

Revisão de Python básico

Prof. MSc. Rafael Maximo Carreira Ribeiro

Resumo

Neste capítulo vamos fazer uma breve revisão de Python (criação de variáveis, tipos de dados, operadores e funções), explorar o funcionamento do modo de depuração (debug) do VSCode, aprender sobre as recomendações para nomear variáveis e funções e ver como configurar o VSCode para nos ajudar na tarefa de padronizar nosso código seguindo as recomendações da PEP 8.

2.1. Revisão de Python básico

Como dito anteriormente, a Programação Orientada a Objetos é construída em cima dos conceitos visto em Linguagem de Programação, e portanto iremos começar com uma revisão geral de Python básico, vendo neste capítulo os seguintes conceitos:

- Tipos de dados: int, float, bool, str;
- Variáveis;
- Operadores; e
- Funções.

2.1.1. Tipos de dados

Os tipos básicos de dados que veremos neste capítulo são o números inteiros, números em ponto flutuante (representando os números reais), valores booleanos e sequências de caracteres (texto). A descrição completa dos tipos integrados à linguagem pode ser acessada em PSF (2021c).

Para saber o tipo de um valor em Python podemos usar a função integrada `type` na *Shell* do Python:

```
>>> type(3)
<class 'int'>
>>> type("3")
<class 'str'>
```

Nesta disciplina vamos aprender em breve sobre o que são classes em POO, e veremos que toda classe que criamos automaticamente define um tipo de objeto, por isso a função `type` retorna essa representação dizendo qual é a "classe" do objeto que passamos para ela como argumento.

2.1.1.1. int

O tipo *int* representa os números inteiros. Em Python não há um limite para o tamanho máximo de um número inteiro, estando apenas limitado pela quantidade de memória disponível para sua alocação.

Os números inteiros podem ser representados nas bases binária, octal e hexadecimal, sendo nesse caso escritos com um prefixo indicativo da base, como mostra a Tabela 2.1. Para a base decimal não há necessidade de nenhum prefixo e é a base adotada por padrão, e podemos separar os dígitos por um sinal de sublinhado para facilitar sua leitura (PSF, 2021a).

Tabela 2.1: Lista de prefixos para bases dos números inteiros

Base	Prefixo	Exemplo
Binária (2)	0b	0b_1110_0101
Octal (8)	0o	0o_345
Decimal (10)	não possui	229
Hexadecimal (16)	0x	0x_e5

Fonte: do autor, 2021

Note que o número é sempre o mesmo (faça o teste digitando os exemplos na *Shell*), e será guardado na memória física sempre em binário, pois essa é a única linguagem que nossos computadores tradicionais entendem, a única coisa que muda é a sua representação. Isso é análogo a medir a distância entre duas cidades em milhas ou quilômetros, o valor representado muda de acordo com a unidade que escolhemos, mas a distância real entre as cidades é sempre a mesma.

2.1.1.2. float

O tipo *float* representa os números reais e ao contrário do tipo *int*, não é ilimitado. Sua precisão e os limites de variação permitidos aos valores variam de acordo com a implementação do interpretador do Python. Isso ocorre devido a representação de um número que em binário não é exato, da mesma forma que ⅓ não é exato em decimal. Essa impossibilidade de representação exata pode levar a problemas de aproximação ou representação dos quais precisamos estar cientes, pois não é um "bug" do Python, mas

sim um problema comum da computação. Esse assunto é tratado em maior detalhe em PSF (2021b).

Veja alguns exemplos de números representados em ponto flutuante no Python:

```
3.14        10.         .001        1e10        5.3e-4
```

Lembrando que os dois últimos exemplos são números representados em notação científica.

2.1.1.3. bool

Há apenas dois objetos constantes que são do tipo *bool*, e representam os valores de verdadeiro e falso, escritos, respectivamente, como `True` e `False` (PSF, 2021d). Observe que não há aspas, pois não são *strings*.

A função integrada `bool()` pode ser usada para converter valores em Python para um booleano. Para entender melhor como funciona a conversão de valores para booleano, confira a documentação do Python (PSF, 2021c).

2.1.1.4. str

O tipo *str* é a abreviação do termo *string*, que representa dados de texto no Python. A partir da versão 3 do Python, toda *string* segue a codificação Unicode.

Para definir uma string podemos usar tanto aspas simples quanto aspas duplas, com o mesmo resultado:

```
>>> 'Olá mundo!'
'Olá mundo!'
>>> "Olá mundo!"
'Olá mundo!'
```

Podemos também construir strings com mais de uma linha utilizando três aspas seguidas (tanto simples quanto duplas):

```
>>> texto = '''Olá mundo!
... escrito em
... várias linhas'''
>>> texto
Olá mundo!\nescrito em\nvárias linhas
>>> print(texto)
Olá mundo!
escrito em
várias linhas
```

Observe que as quebras de linha são mantidas e representadas pelo caractere `\n`. Dizemos que a contra barra `\` é usada para escapar a letra `n`, dando a ela um significado especial, que representa uma quebra de linha.

Ao compararmos duas *strings*, elas serão iguais apenas se possuírem exatamente os mesmos caracteres e na mesma ordem. Quando comparadas com os operadores de maior e menor, o Python faz a comparação colocando-as em ordem alfabética, ou seja, não importa o tamanho das *strings*, mas sim qual delas tem letras menores no alfabeto, sendo a posição no alfabeto definida com base no código Unicode do caractere. Para aprender mais veja a documentação do Python (PSF, 2021g).

VAMOS LER!

Unicode é uma padronização que permite que computadores representem e manipulem caracteres de quase todos os sistemas de escrita existentes. Na tabela Unicode existem códigos associados a símbolos, sendo que cada código está mapeado para apenas um símbolo. Há milhares de códigos, basta consultá-los em: https://unicode-table.com/pt/

Podemos concatenar *strings*, usando o operador + e também criar *strings* formatadas de diferentes maneiras. O método recomendado para formatação de *strings* em novos projetos (Bader, 2018) é chamado de *"strings literais formatadas"* ou *f-strings*, e consiste em uma *string* comum prefixada com a letra `f` ou `F` (antes da abertura das aspas) e com os valores ou expressões que serão formatados inseridos entre pares de chaves. Essa forma de formatação é compatível apenas com versões do Python superiores à 3.6 (PSF, 2021e).

Veja o seguinte exemplo:

```
>>> texto = f'4 + 7 = {4 + 7}, certo?'
>>> texto
'4 + 7 = 11, entendeu?'
```

Note que a expressão entre o par de chaves é avaliada e o resultado inserido no mesmo ponto em que está escrita na *string*. A expressão é avaliada em tempo de execução, o que permite o uso de *f-strings* com variáveis e expressões, como no próximo exemplo:

```
>>> salario = 3500.0
>>> f'20% a mais em R$ {salario} dará R$ {salario*1.2}'
'20% a mais em R$ 3500.0 dará R$ 4200.0'
```

No exemplo acima, seria interessante exibir os valores monetários com duas casas decimais. Para especificar como queremos que um valor seja formatado em uma *f-string*, adicionamos `:` após o valor e complementamos com alguns especificadores.

Para formatar um dado do tipo *float*, podemos usar o seguinte padrão de formatação: `f'{<valor>:<colunas>.<decimais>f}'`. Onde:

- *colunas*: é a quantidade mínima de colunas reservadas para o valor na *string* formatada, note que cada caractere do valor ocupa uma coluna, inclusive o ponto. Pode ser omitido;
- *decimais*: é o número total de casas decimais que serão representadas na *string*, com o valor sendo arredondado caso necessário;

- `f`: indica que a formatação será feita para o tipo *float*, podendo haver uma conversão entre tipos compatíveis, como o *int*.

Veja o código abaixo e, em seguida, tente reescrever o exemplo inicial do salário, porém formatando os valores em reais com duas casas decimais.

```
>>> pi = 3.14159265
>>> f'{pi:f}'  # sem especificar, o padrão é de 6 casas decimais
'3.141593'
>>> f'{pi:.3f}'  # note o arredondamento na última casa decimal
'3.142'
>>> f'{pi:7.3f}'  # 7 colunas totais, 3 casas decimais
'  3.142'
```

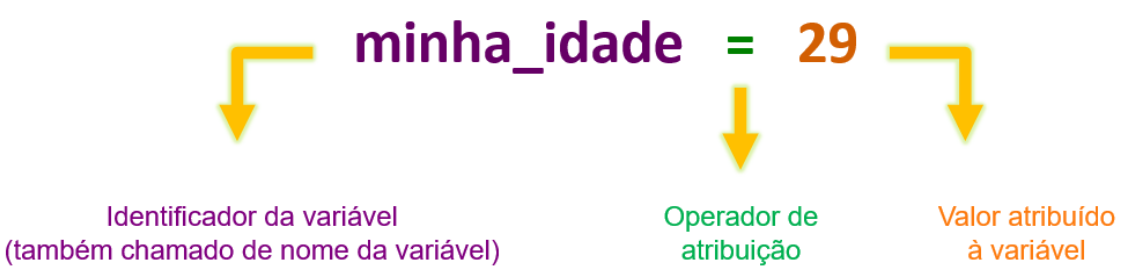
Existem outros especificadores para formatação de *float*, como exibição em notação científica, e também para outros tipos, como *int* e *string*, além de opções para alterar o alinhamento do texto e o caractere padrão de preenchimento dos espaços vazios. A lista completa com as explicações e exemplos de uso pode ser vista na documentação oficial do Python (PSF, 2021f).

2.1.2. Variáveis

Uma variável é um espaço de memória associado a um identificador, ou seja, um nome, e serve para guardar valores que o programa poderá acessar e modificar. Toda variável possui um identificador único, que poderá ser referenciado no código sem ambiguidade.

Em Python, uma variável é criada no momento em que um valor é atribuído a um identificador válido. A atribuição é feita através do operador de atribuição, o sinal de igual, colocando-se um identificador à esquerda e um valor à direita deste operador, conforme a Figura 2.1.

Figura 2.1: Atribuição de um valor a uma variável



Fonte: do autor, 2021

Assim como a nomenclatura evidencia, o conteúdo de uma variável pode "variar", ou seja, uma mesma variável pode guardar valores diferentes em momentos diferentes de um programa em Python. Lembre-se: uma variável só pode guardar um valor por vez, portanto a cada nova atribuição o valor atual será sobrescrito pelo novo.

Execute os comandos a seguir na *Shell* e verifique os resultados:

```
>>> a = True
>>> type(a)
>>> a = 123
>>> a = 'linda casa amarela'
>>> a = 4.40
>>> a
>>> type(a)
```

Um destaque importante da linguagem Python é que o tipo do dado está relacionado ao valor atribuído e não a variável que recebeu esse valor, diferentemente de outras linguagens de programação como C, C++ e Java, para citar alguns exemplos.

Um nome ou identificador de uma variável é formado por uma sequência de um ou mais caracteres, de acordo com as seguintes regras:

- Pode conter apenas letras, números e o símbolo de sublinhado (nenhum outro caractere especial é aceito);
- Não pode começar com um número;
- Não pode ser uma palavra reservada.

Ao criar uma variável, recomenda-se utilizar identificadores que sejam concisos, porém descritivos:

- `idade` é melhor que `i`;
- `tamanho_nome` é melhor que `tamanho_do_nome_da_pessoa`.

No entanto, evite abreviar nomes, escrevendo-os por extenso para melhorar a legibilidade do código, por exemplo:

- `sobrenome` é melhor que `sbrnome`;
- `litros` é melhor que `ltrs`;
- `data_criacao` é melhor que `dt_cri`.

A PEP 8 recomenda usarmos apenas letras minúsculas e sem acentuação para criar identificadores de variáveis e funções, separando as palavras com um símbolo de sublinhado para melhorar a legibilidade.

Vale lembrar que o Python é uma linguagem *case-sensitive*, diferenciando letras maiúsculas de minúsculas, portanto é preciso prestar atenção na grafia exata dos identificadores, pois `meu_nome` não é o mesmo que `Meu_Nome` ou `MEU_NOME`.

O Python possui um conjunto de palavras reservadas, chamadas em inglês de *keywords* (palavras-chave). Essas palavras não podem ser usadas como identificadores, pois possuem um papel especial para o interpretador. O Python 3.9.1 possui 36 palavras reservadas, porém esse número pode variar entre versões diferentes, para saber quais são as da versão que está usando, execute a seguinte instrução na *Shell* do Python:

```
>>> help('keywords')
```

2.1.3. Operadores

Vamos relembrar três tipos de operadores aqui:

1) *Aritméticos*: usados para realizar operações matemáticas entre os operandos;
2) *Relacionais*: usados para comparar dois objetos em Python;
3) *Lógicos*: usados para realizar operações lógicas com valores booleanos.

A Tabela 2.2 mostra a ordem em que os operadores são resolvidos pelo Python, de acordo com sua prioridade, sendo o operador de maior prioridade resolvido primeiro.

Tabela 2.2: Prioridade dos operadores aritméticos, relacionais e lógicos.

Ordem de resolução	Operador	Descrição	Associatividade
1°	`**`	Exponenciação.	à direita
2°	`+`, `-` (unários)	Identidade e negação.	à esquerda
3°	`*`, `/`, `//`, `%`	Multiplicação, divisão real, divisão inteira e resto da divisão.	
4°	`+`, `-` (binários)	Adição e subtração.	
5°	`==`, `!=`, `>`, `>=`, `<`, `<=`	Operadores relacionais.	Não associativos
6°	`not`	Negação lógica.	à esquerda
7°	`and`	E lógico.	
8°	`or`	OU lógico.	

Fonte: do autor, 2021

Dentre os operadores, a atribuição, simples ou composta, tem a menor prioridade independentemente da instrução. Isso é necessário para que a expressão à direita do operador de atribuição possa ser completamente avaliada e o resultado atribuído à variável à esquerda.

2.1.4. Funções

Para criarmos uma nova função em Python, utilizamos a palavra chave `def` seguida do nome da função e um par de parênteses, dentro dos quais listamos os eventuais *parâmetros*. Em seguida colocamos `:` para indicar o fim do cabeçalho[1] da

[1] Em Python, o nome da função mais a quantidade e ordem dos seus parâmetros é também chamado de assinatura da função. Em outras linguagens, esse conceito pode incluir também o tipo dos parâmetros e o tipo do retorno da função, mas tudo isso depende de como a linguagem trata a tipagem de dados.

função e início de seu bloco de código, que deve começar na linha seguinte com indentação de 4 espaços em relação à coluna inicial do cabeçalho.

```
def <nome da função>([<parâmetros>]):
    <bloco de código da função>
```

No bloco da função, podemos utilizar qualquer código Python válido, inclusive podemos chamar outras funções conforme necessário, sejam elas integradas, importadas ou definidas no próprio código. A definição de funções dentro de outras funções é válida e serve a propósitos específicos (por exemplo para a criação de decoradores em Python, um assunto que será visto mais adiante no curso), porém, na maioria das situações, devemos criar funções apenas na raiz do código.

O par de colchetes nos parâmetros indica que eles são opcionais, uma função pode ter uma quantidade qualquer de parâmetros ou não ter nenhum, situação em que o par de parênteses, que é obrigatório, deve ficar vazio. Isso é decidido no momento de sua criação e deve ser respeitado no momento em que a função é chamada.

O nome da função deve seguir as mesmas regras que vimos para a criação dos nomes de variáveis: deve conter apenas letras, dígitos e sublinhados; não pode começar com um dígito e não pode ser uma palavra chave (reservada).

Em Python, as funções e variáveis compartilham o mesmo espaço de nomes, portanto evite criar funções que tenham o mesmo nome de variáveis, ou vice-versa, pois isso entrará em conflito e o valor mais antigo será sobrescrito e apagado.

Por fim, procure criar nomes de funções que, assim como nas variáveis, sejam indicativos daquilo que a função é responsável por executar, pois isso facilita seu uso ao longo e melhora a legibilidade do código. Vejamos um exemplo:

```
>>> def soma_2(x):
...     return x + 2
...
>>> soma_2(5)
7
```

Ao definirmos uma função, o interpretador do Python executa o comando `def`, que irá criar um objeto do tipo *function* na memória contendo o nome da função, quais são os parâmetros e também uma cópia do código que estava no bloco da função. Para ver mais sobre funções, confira a documentação do Python (PSF, 2021h) e o capítulo 3 do livro Pense em Python[2] (DOWNEY, 2016).

2.2. VSCode - modo de depuração

Ao executar um arquivo Python no terminal, ele é executado do início ao fim, de modo que todas as variáveis criadas são perdidas e temos acesso apenas àquilo que foi exibido na tela. Um método de depuração básico, que pode nos ajudar a resolver muitos problemas, consiste em colocar diversos "prints" no código para acompanhar o seu fluxo de execução e conferir o valor das variáveis de interesse.

[2] Versão online gratuita disponível em: https://penseallen.github.io/PensePython2e/03-funcoes.html

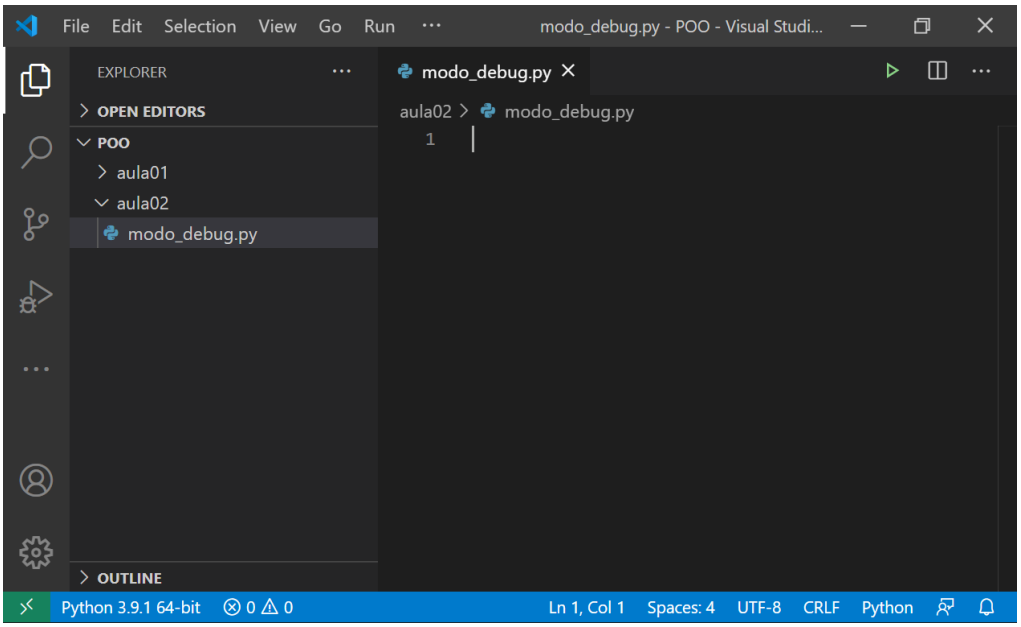
Porém, além de não ser um método interativo (não é possível testar em tempo real, pois precisamos sempre editar o código e executá-lo novamente), esses "prints" não são necessários para a aplicação final, então conforme nosso código cresce em complexidade, isso deixa de ser uma abordagem viável, pois precisamos constantemente colocar e tirar (ou comentar) tais comandos, o que não só reduz nossa produtividade mas também é mais propenso a introdução de erros no código.

Para resolver esse problema, podemos executar o código no modo de depuração, um recurso que a maioria das IDEs possui. Neste modo podemos selecionar pontos de parada ao longo do código e em seguida controlar a execução das instruções passo a passo, com acesso às variáveis em tempo de execução.

Vamos criar o seguinte arquivo no VSCode, mas antes iremos trocar a pasta do projeto para a pasta raiz da disciplina, assim podemos gerenciar os arquivos de todas as aulas diretamente pela interface da IDE. No menu superior, clique em "*File*" e em seguida em "*Open Folder...*", selecione a pasta da disciplina (POO) e confirme.

Agora, na aba de diretórios no menu lateral esquerdo, crie uma nova pasta "aula02" e em seguida crie um novo arquivo dentro de "aula02" com o nome "modo_debug.py". O resultado deve ficar semelhante ao da Figura 2.2.

Figura 2.2: Visualização da IDE após a criação do arquivo "modo_debug.py"



Fonte: do autor, 2021

Agora digite[3] o seguinte trecho de código no editor e salve o arquivo.

[3] Evite copiar e colar o texto pois o ato de digitar ajuda na memorização do conteúdo, e ao fazermos uma cópia do texto, muitas vezes acabamos copiando e introduzindo caracteres "invisíveis" no começo ou final do texto, e esses caracteres não são interpretados corretamente pelo editor de texto, gerando um erro de sintaxe em um código aparentemente 100% correto.

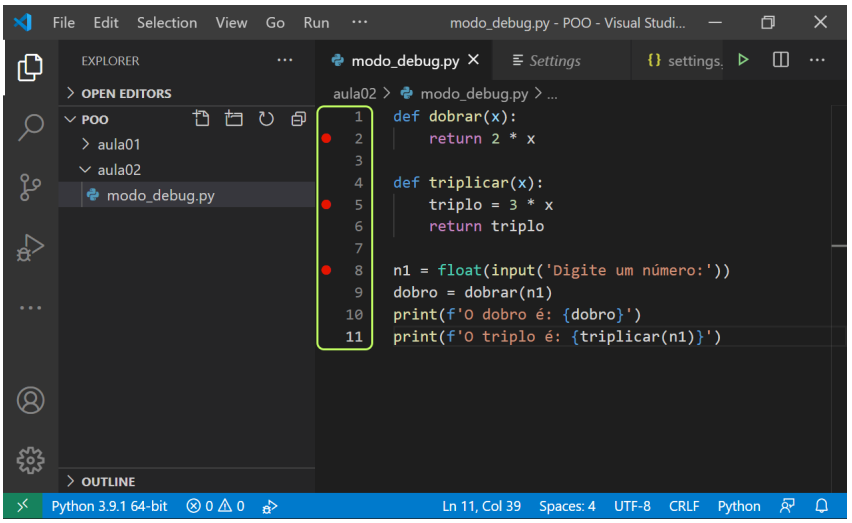
```
def dobrar(x):
    return 2 * x

def triplicar(x):
    triplo = 3 * x
    return triplo

n1 = float(input('Digite um número:'))
dobro = dobrar(n1)
print(f'O dobro é: {dobro}')
print(f'O triplo é: {triplicar(n1)}')
```

Antes de executar um arquivo no modo de depuração no VSCode, precisamos escolher os pontos de parada, em inglês *breakpoints*. Isso é feito clicando à esquerda do número da linha na qual queremos introduzir um ponto de parada, e deve aparecer um pequeno círculo vermelho, indicando que o ponto de parada foi marcado. Insira um ponto de parada na primeira instrução do bloco de cada função e outro na linha que cria a variável `n1`, como mostra a Figura 2.3.

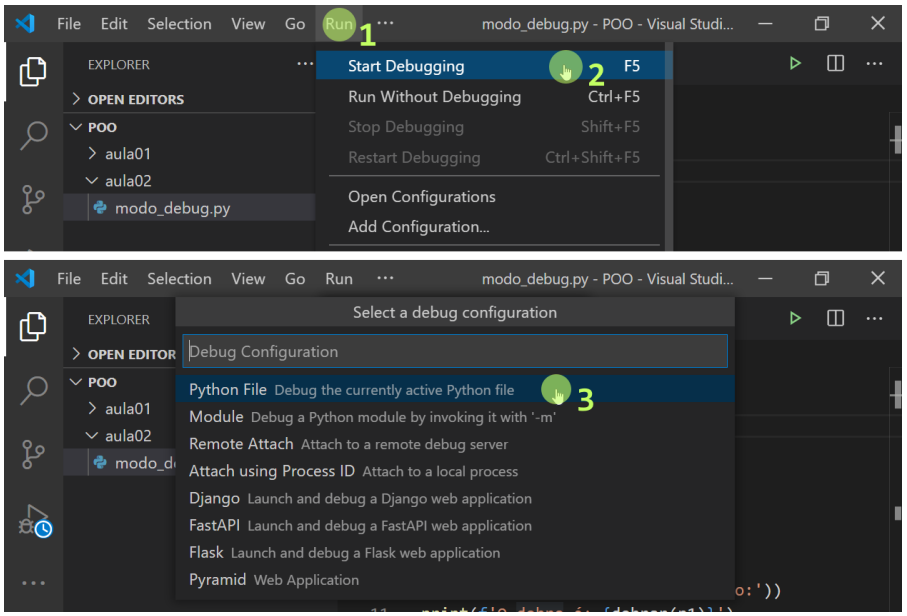
Figura 2.3: Visualização dos pontos de parada (*breakpoints*) no código



Fonte: do autor, 2021

Agora, para iniciar a execução no modo de depuração, podemos clicar em "*Run*" no menu superior e em seguida em "*Start debugging*", ou pressionar a tecla F5. Ao fazer isso, seremos apresentados a diversas opções sobre o que queremos depurar, então devemos escolher a primeira opção "*Python File: Debug the currently active Python file*", para depurar o arquivo atualmente ativo no editor do VSCode, como mostra a Figura 2.4.

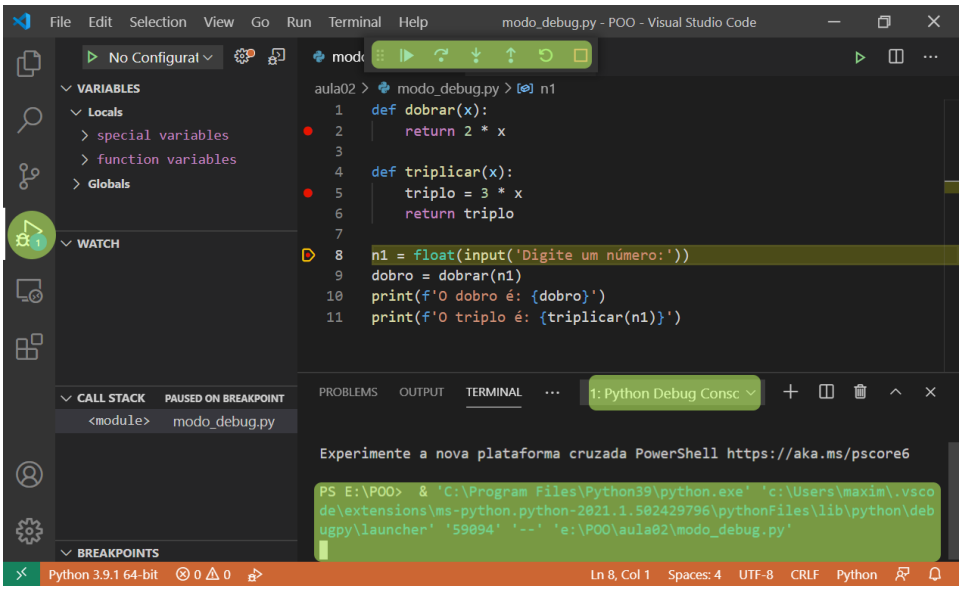
Figura 2.4: Executando o modo de depuração no VSCode



Fonte: do autor, 2021

Após fazer o procedimento descrito acima, o Python irá executar todo código anterior ao primeiro ponto de parada, sem executar a linha na qual colocamos o ponto de parada. Observe na Figura 2.5 o resultado e veja que a variável `n1` ainda não foi criada, pois esta linha ainda não foi executada.

Figura 2.5: Primeiro passo no modo de depuração



Fonte: do autor, 2021

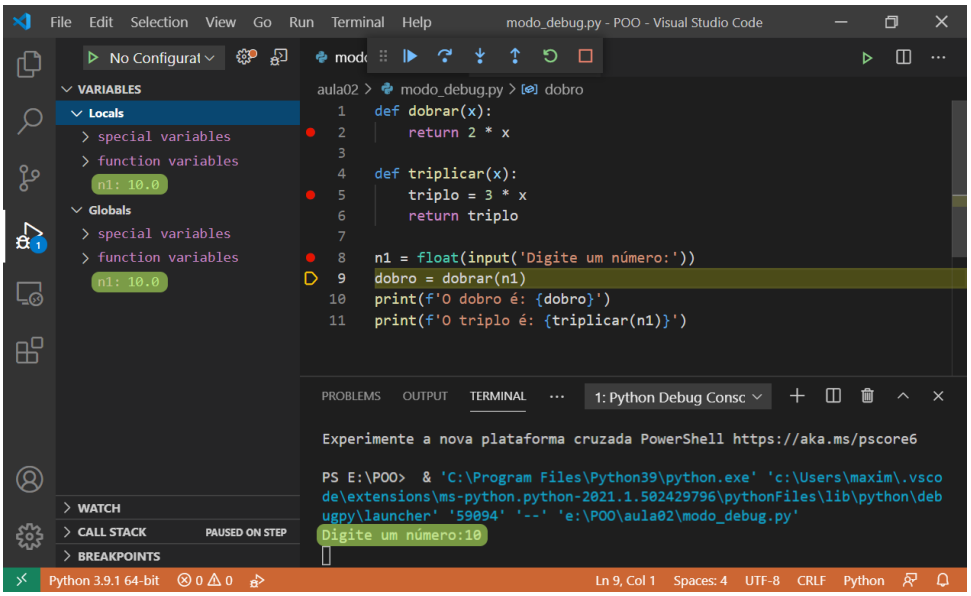
Observe que estamos agora na aba lateral esquerda referente ao modo de depuração, na qual temos algumas janelas, vamos focar agora na de variáveis, portanto podemos minimizar as outras clicando nos símbolos de seta ao lado dos nomes de cada uma. Além disso, surgiu no topo da janela um menu extra com alguns botões relativos ao modo de depuração e há um terminal aberto com um processo do Python sendo executado, e em espera.

Nos botões que surgiram temos as seguintes ações[4]:

- *Continue* (F5): Continua a execução do código até o próximo ponto de parada.
- *Step Over* (F10): Executa a instrução atual em tempo normal e para antes de executar a instrução seguinte.
- *Step Into* (F11): Executa a instrução atual passo a passo, isto é, se for uma função por exemplo, irá parar na primeira linha de código desta função e podemos controlar sua execução linha a linha.
- *Step Out* (Shift + F11): Finaliza a execução da função atual e retorna para o código que a chamou, parando antes de executar a próxima instrução.
- *Restart* (Ctrl + Shift + F5): Reinicia a execução do arquivo no modo de depuração.
- *Stop* (Shift + F5): Encerra a execução do modo de depuração no ponto atual, sem executar mais nenhuma instrução do código.

As ações mais comumente utilizadas são as três primeiras, agora clique em *Step Over*, ou pressione F10, digite um valor numérico na entrada e tecle *Enter*. O resultado deve ser parecido com o da Figura 2.6.

Figura 2.6: Segundo passo no modo de depuração



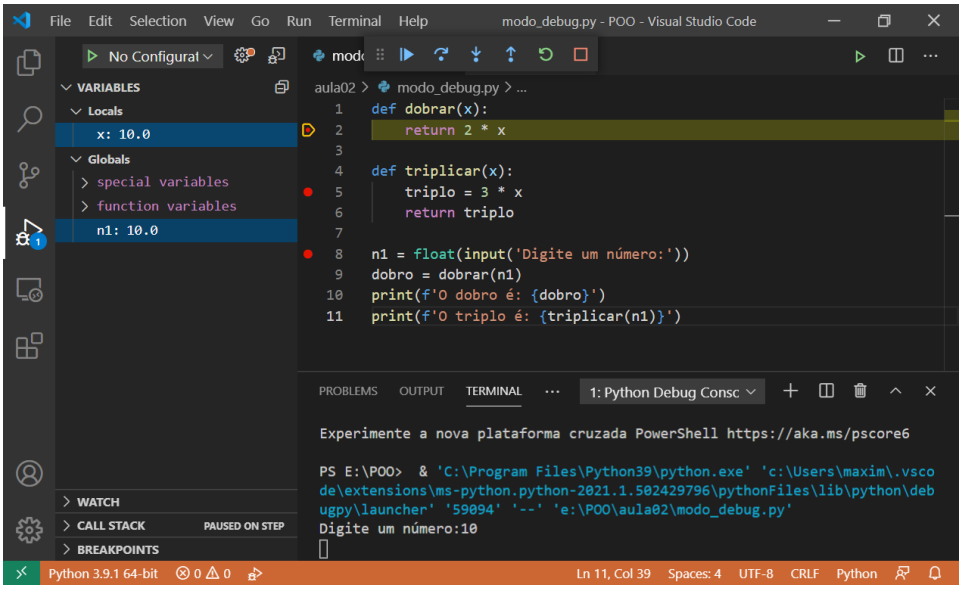
Fonte: do autor, 2021

Observe que a interação (entrada e saída de dados) é feita através do terminal e que há na janela de variáveis a nova variável `n1` criada pela instrução anterior. Quando

[4] Os atalhos podem variar em função do sistema operacional.

estamos executando as instruções no escopo global, ou seja, não estamos dentro de nenhuma função, o escopo local e global são iguais, mas isso não ocorre quando estamos em uma função. Para ver a diferença, clique em *Continue* ou pressione a tecla F5 para continuar a execução até o próximo ponto de parada. O resultado é mostrado na Figura 2.7.

Figura 2.7: Terceiro passo no modo de depuração



Fonte: do autor, 2021

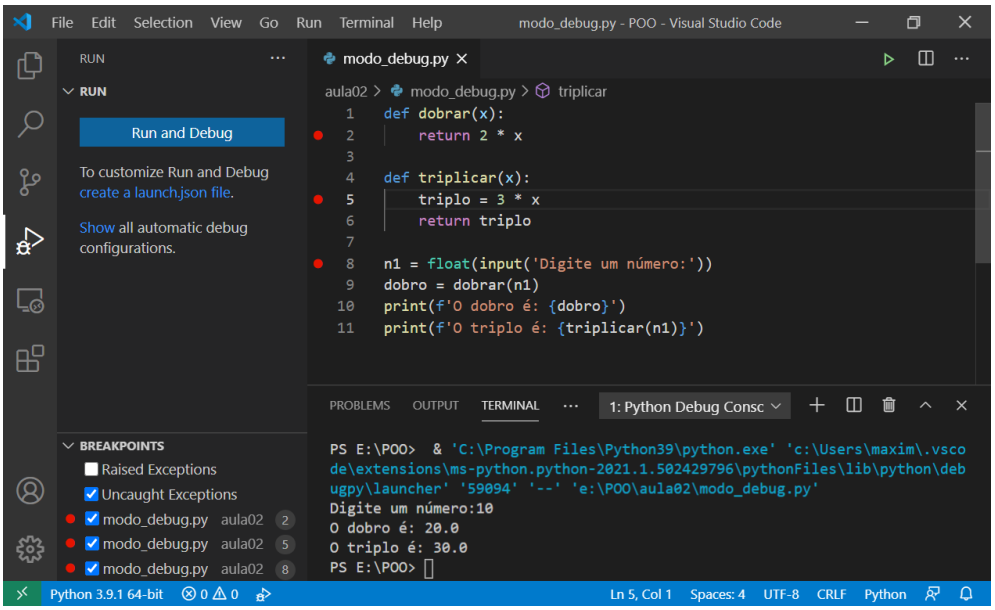
Observe que as variáveis locais agora contém apenas informações referentes ao espaço de variáveis da função `dobrar`, onde foi criada a variável `x` com o valor passado ao chamar a função. Finalize a execução do código usando os botões de Step Over e Continue e acompanhe o fluxo de execução do código e o comportamento dos espaços de variáveis locais e globais. Faça alterações no código e explore seus efeitos nas variáveis criadas e no resultado exibido no terminal. A Figura 2.8 mostra a IDE após a finalização do modo de depuração.

VAMOS PRATICAR!

1) Você consegue responder por que o primeiro ponto de parada foi o da instrução que define a variável n1 e não os pontos de parada colocados nas funções definidas no começo do arquivo?

2) Retire os pontos de parada das funções e execute novamente o arquivo no modo de depuração. Após entrar o valor de `n1`, use o botão de *Step Into* ao invés de continuar a execução como fizemos no exemplo. Você consegue explicar o que aconteceu?

Figura 2.8: Execução no modo de depuração finalizada



Fonte: do autor, 2021

2.3. Recomendações PEP 8

Vamos ver agora as recomendações que deixamos de seguir no arquivo que escrevemos para explorar o modo de depuração.

- Deixar duas linhas em branco entre as definições de função;
- Separar trechos lógicos no código com 1 linha em branco, se necessário;
- Não deixar espaços vazios após o final do texto na linha;
- Não deixar espaços vazios em uma linha sem texto;
- Terminar o arquivo sempre com uma linha em branco.

Mas lembrar e aplicar todas as recomendações pode ser uma tarefa chata e difícil de cumprir por conta própria, então mais uma vez, vamos usar a IDE para nos auxiliar na organização do código segundo as regras definidas na PEP 8. Para isso faça as configurações extras listadas a seguir no seu ambiente de desenvolvimento.

2.3.1. Configurações extras no ambiente de desenvolvimento

A primeira configuração é a instalação dos pacotes para verificação das recomendações definidas pela PEP 8. Recomendamos o uso dos pacotes `flake8`[5] e `pep8-naming`[6]. Para instalar tais pacotes, vá no terminal do VSCode e digite:

```
> py -m pip install flake8          # Windows
$ python3 -m pip install flake8     # Linux e MacOS
```

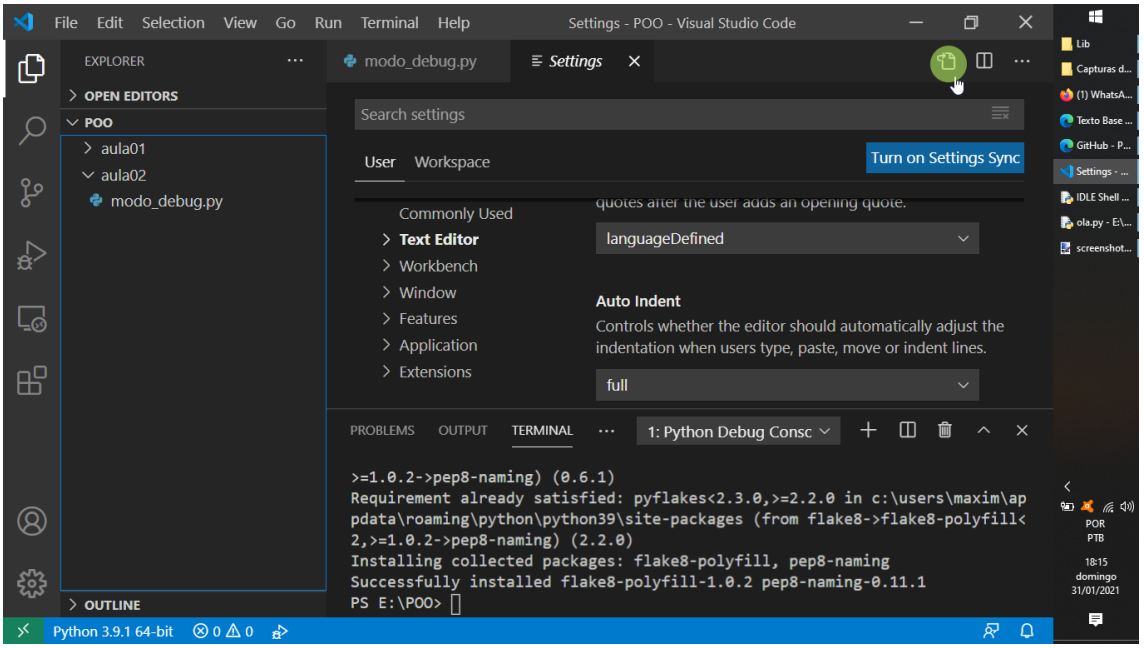
[5] Flake8: Your Tool For Style Guide Enforcement — flake8 3.8.4 documentation (pycqa.org)
[6] GitHub - PyCQA/pep8-naming: Naming Convention checker for Python

Após a instalação finalizar, digite:

```
> py -m pip install pep8-naming            # Windows
> python3 -m pip install pep8-naming       # Linux e MacOS
```

Agora, precisamos configurar o VSCode para trabalhar com tais pacotes. Para isso, abra as configurações no menu "*File* > *Preferences* > *Settings*" ou pressionando "Ctrl + ,". Em seguida clique no ícone "*Open settings*" no canto superior direito, como mostra a Figura 2.9.

Figura 2.9: Abertura do arquivo JSON de configuração do VSCode



Fonte: do autor, 2021

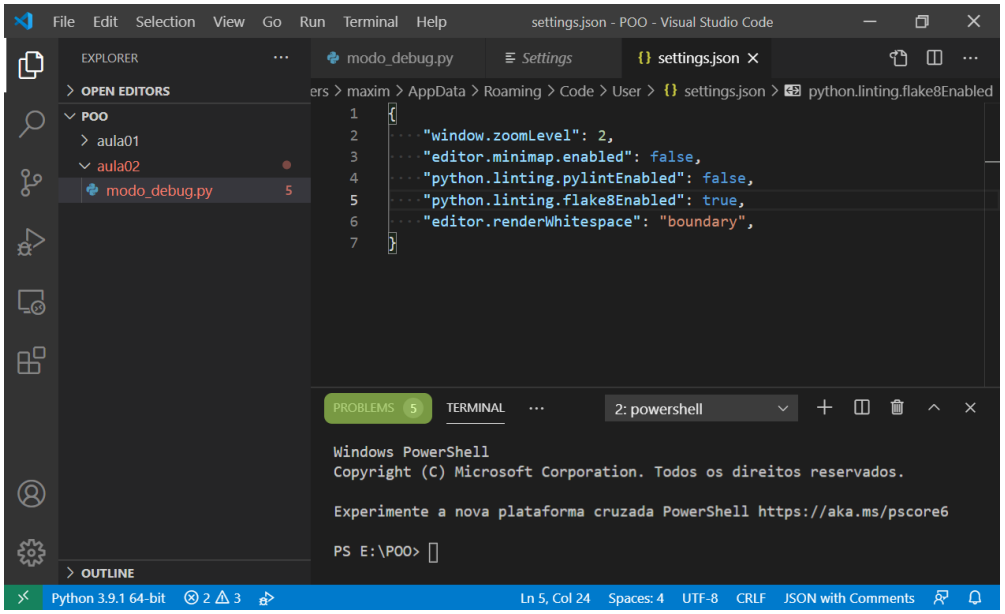
Após abrir o arquivo de configuração JSON, adicione as seguintes opções a este arquivo e salve-o (Ctrl + s). Caso este arquivo já tenha alguma configuração salva, inclua as configurações a seguir dentro do mesmo par de chaves já existente, mantendo-as com a mesma indentação das demais configurações e lembrando que é necessário uma vírgula ao final de cada linha (exceto da última, que é opcional).

```
{
    "python.linting.pylintEnabled": false,
    "python.linting.flake8Enabled": true,
    "editor.renderWhitespace": "boundary",
}
```

A primeira linha desativa o Pylint, pois ter dois mecanismos de verificação da PEP8 não é recomendado; a segunda linha ativa o Flake8; e a terceira linha diz para o VSCode renderizar os espaços em branco adjacentes, no começo e no final da linha.

O resultado final pode ser visto na Figura 2.10. Observe que no arquivo do exemplo há uma configuração a mais para definir o zoom da IDE (que pode também ser alterado com os atalhos Ctrl + '+' e Ctrl + '-') e outra para esconder o mini-mapa que aparece por padrão no canto direito da tela. Estas duas configurações são opcionais, com o tempo, você irá aos poucos personalizando sua IDE da maneira que melhor funciona para você.

Figura 2.10: Personalizando as configurações do VSCode



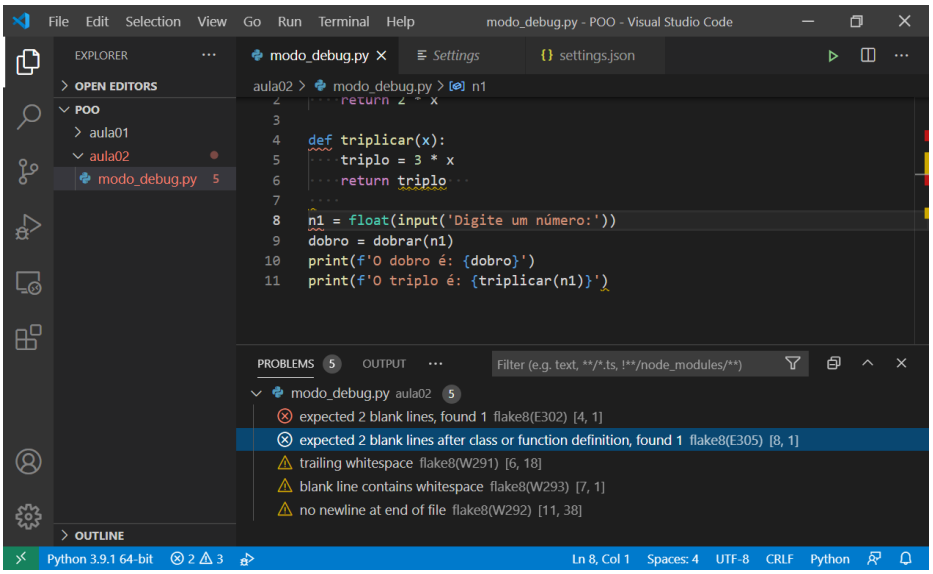
Fonte: do autor, 2021

2.3.2. Corrigindo o código

Observe que após salvar o arquivo de configurações, apareceu na janela do terminal um indicador com o número cinco na aba *PROBLEMS*. Volte para a aba do arquivo "modo_debug.py" e clique na aba de problemas na janela do terminal. A Figura 2.11 lista os problemas encontrados no arquivo usado neste exemplo. Ao clicar em um dos erros, o VSCode realça a linha em que ele acontece.

Os dois primeiros erros se referem a quantidade de linhas em branco após a definição de uma função; o terceiro e o quarto são referentes a espaços em branco em uma linha vazia e no final da linha, respectivamente; e o quinto nos diz que o arquivo não tem uma linha vazia no final.

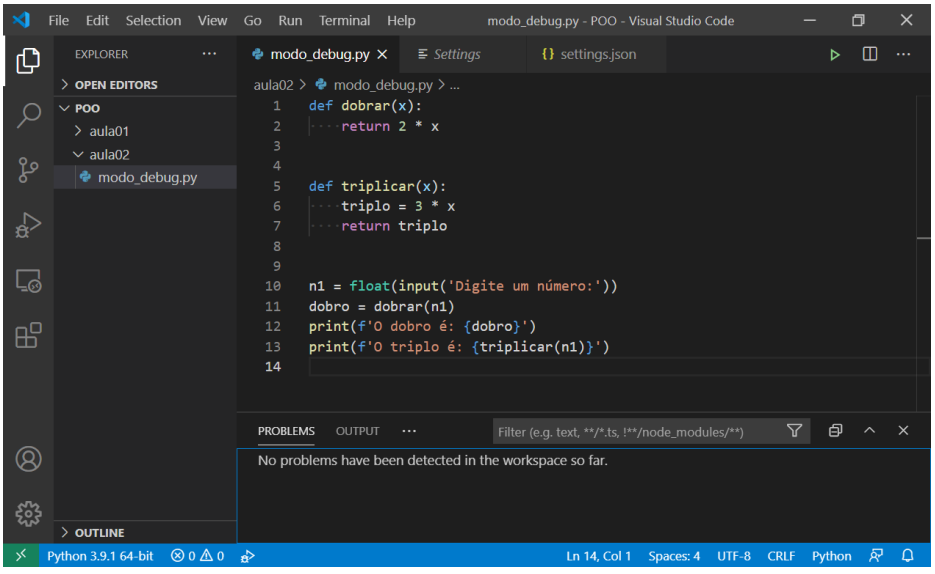
Figura 2.11: Erros e avisos do Flake8



Fonte: do autor, 2021

Toda vez que salvamos o arquivo, essa lista é automaticamente atualizada. Portanto, corrija os erros que estiverem aparecendo em seu código e o resultado será semelhante ao mostrado na Figura 2.12.

Figura 2.12: Código formatado conforme as recomendações da PEP8



Fonte: do autor, 2021

Com o uso dessas ferramentas, podemos aprender sobre as recomendações da PEP 8 naturalmente conforme cada erro aparece em nosso código, usando o guia de estilo como um documento para consultar em caso de dúvidas, por exemplo.

Referências

BADER, D. **Python string formatting best practices**. Real Python, 2018. Disponível em: <https://realpython.com/python-string-formatting/>. Acesso em: 26 jan. 2021.

DOWNEY, A. B. **Pense em python**. São Paulo: Novatec Editora Ltda., 2016.

PETERS, T. **The python way**. 1999. Disponível em: <https://mail.python.org/pipermail/python-list/1999-June/001951.html>. Acesso em: 29 jan. 2021.

PETERS, T. **PEP 20 - The zen of python**. 2004. Disponível em: <https://www.python.org/dev/peps/pep-0020/>. Acesso em: 27 jan. 2021.

PSF. **Lexical analysis:** integer literals. 2021a. Disponível em: <https://docs.python.org/3/reference/lexical_analysis.html#integer-literals>. Acesso em: 29 jan. 2021.

PSF. **Floating point arithmetic:** issues and limitations. 2021b. Disponível em: <https://docs.python.org/3/tutorial/floatingpoint.html>. Acesso em: 29 jan. 2021.

PSF. **Built-in types**. 2021c. Disponível em: <https://docs.python.org/3/library/stdtypes.html>. Acesso em: 29 jan. 2021.

PSF. **Built-in types:** boolean values. 2021d. Disponível em: <https://docs.python.org/3/library/stdtypes.html#boolean-values>. Acesso em: 29 jan. 2021.

PSF. **Lexical analysis:** formatted string literals. 2021e. Disponível em: <https://docs.python.org/3/reference/lexical_analysis.html#f-strings>. Acesso em: 29 jan. 2021.

PSF. **Strings:** format specification mini-language. 2021f. Disponível em: <https://docs.python.org/3/library/string.html#format-specification-mini-language>. Acesso em: 29 jan. 2021.

PSF. **An informal introduction to python:** strings. 2021g. Disponível em: <https://docs.python.org/3/tutorial/introduction.html#strings>. Acesso em: 29 jan. 2021.

PSF. **The python tutorial:** functions. 2021h. Disponível em: <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>. Acesso em: 29 jan. 2021.

ROSSUM, G. V., WARSAW, B., COGHLAN, N. **PEP 8** - style guide for python code. 2013. Disponível em: <https://www.python.org/dev/peps/pep-0008/>. Acesso em: 27 jan. 2021.