# SUMÁRIO

[**SUMÁRIO 1**](#_11agubohmyoa)

[**Programação Orientada a Objetos 2**](#_t4yjix6t6a0)

[**Apresentação da disciplina e preparação do ambiente de desenvolvimento 2**](#_hzps4ktscmq4)

[Configuração do ambiente de desenvolvimento 3](#_aidqtei601f)

[Princípios norteadores e um guia de estilo para Python 4](#_2ljwcxbj03m2)

[**Revisão de Python básico 5**](#_3d06m9q1ytrk)

[Tipos de dados 6](#_n65hm0zdto4i)

[int 6](#_ciw0o2qkkzvl)

[float 6](#_vc06p2fuok5v)

[bool 7](#_qq70m07kql76)

[str 7](#_daxp3u2jxy8g)

[Variáveis 9](#_tarrhaoo14j0)

[Operadores 10](#_t7a605coxo94)

[Funções 11](#_b5owgbu5yj5j)

[**Listas, Tuplas e Dicionários em Python 12**](#_fbxi7wjze4mf)

[Sequências 12](#_ruxpuh76twub)

[Listas 12](#_pmshgo2uu63q)

[Acessar um item da lista 13](#_cwwt3rpkwiqu)

[Substituir um item da lista 13](#_wqdmnaax8e8e)

[Inserir um item em uma dada posição na lista 14](#_b42andpptkl)

[Remover um item da lista 14](#_17yid6y1tary)

[Acrescentar um item ao final da lista 15](#_lejvsaczpviu)

[Concatenar duas listas 15](#_timar2jr6kdc)

[Tuplas 15](#_wbqilo5dfe1t)

[Dicionários 16](#_5n35cmb5fov0)

[Acessar um valor do dicionário 17](#_y3vuqj3qrerc)

[Inserir um valor no dicionário 18](#_ngsm045mpw4h)

[Excluir um item do dicionário 19](#_toz6ta87w8iu)

[Operações de pertencimento em um dicionário 19](#_hkonsrp3cynh)

[Métodos especiais para iterar sobre um dicionário 20](#_cjjs2op2dz23)

[Operações de pertencimento em listas, tuplas e strings 21](#_e3132bqemugr)

[Desempacotamento de sequências 21](#_jpbr25ni2kbu)

[**Estruturas de controle de fluxo 22**](#_jogedoangld0)

[Estruturas de seleção 22](#_7hiyvwslul4q)

[Estruturas de repetição 24](#_kzyboiq4on5h)

[Estruturas de repetição indefinidas - While 24](#_to2zvavopbci)

[Estruturas de repetição definidas - For 24](#_1r0wli4scfq5)

[**VSCode - modo de depuração 25**](#_um31awbo0ee7)

[**Recomendações PEP 8 28**](#_lt0zb4ut955w)

[Configurações extras no ambiente de desenvolvimento 28](#_2thp03bp8zo7)

[Corrigindo o código 30](#_ps805aovi42q)

# Programação Orientada a Objetos

Esta disciplina irá aprofundar o estudo da linguagem Python e apresentar os conceitos de Programação Orientada a Objetos (POO), tanto a teoria, que pode ser aplicada também em diversas outras linguagens de programação, quanto a sua aplicação em Python, habilitando-nos dessa forma a resolver problemas cada vez mais complexos.

O conteúdo visto na disciplina de Linguagem de Programação será essencial para a continuação dos estudos. Todos os conceitos e estruturas aprendidos lá serão utilizados ao longo desta disciplina.

A disciplina está organizada da seguinte forma: começando com uma revisão de Python básico, a fim de retomar os conceitos vistos em Linguagem de Programação, antes de seguir para a Programação Orientada a Objetos. Serão apresentados os pilares de POO e como eles são aplicados em Python, o que é e como se construir uma classe, criar objetos e fazer com que eles interajam entre si, através de seus métodos.

Essa disciplina também inclui tópicos como a manipulação de arquivos com Python, a criação e utilização de ambientes virtuais, a modularização e criação de pacotes, o gerenciamento de dependências e a utilização de testes automatizados, além de uma introdução aos princípios do SOLID e aos Padrões de Projeto. Esses recursos são essenciais para o bom desenvolvimento de software e irão aprimorar suas habilidades como desenvolvedor e contribuir para sua formação.

Disciplina elaborada e apresentada pelo professor Rafael Maximo Carreira Ribeiro.

# Apresentação da disciplina e preparação do ambiente de desenvolvimento

A disciplina de Programação Orientada a Objetos é uma continuação do que foi visto em Linguagem de Programação (LP). Nesta disciplina vocês conhecerão mais estruturas de programação utilizadas em Python, e que podem também ser aplicadas em outras linguagens de programação, para poder resolver problemas ainda mais complexos.

Tudo que você aprendeu em LP será aproveitado aqui: variáveis, tipos de dados, estruturas de decisão, estruturas de repetição, listas, tuplas e funções. Isso é na realidade a base para todas as disciplinas que envolvem programação no curso e não será diferente para esta. Utilizaremos tais conceitos para aprender um novo paradigma de desenvolvimento de software, no qual abordamos a representação dos problemas baseada na construção de objetos e na interação entre eles, traçando um paralelo mais próximo entre a realidade e a programação.

Esse paradigma é conhecido como Programação Orientada a Objetos, que chamaremos de POO de agora em diante. Aprenderemos o que são objetos, classes, métodos e atributos, veremos qual a relação entre eles e conceitos como abstração, encapsulamento, herança e polimorfismo, que formam os pilares de POO.

Além de POO, aprenderemos também sobre como criar módulos em Python, estudaremos como podemos testar nossas aplicações de maneira automatizada e veremos uma introdução aos princípios do SOLID e aos padrões de projeto. Estes fatores contribuem para um código de manutenção mais fácil, com menos erros ou bugs e portanto com maior qualidade.

Esta disciplina é fundamental para sua formação na área de tecnologia da informação, mesmo que você não venha a trabalhar diretamente como desenvolvedor, os conceitos aprendidos aqui com certeza serão um diferencial na sua carreira.

## Configuração do ambiente de desenvolvimento

A disciplina de Programação Orientada a Objetos utiliza como base a linguagem de programação Python, dando sequência ao que foi visto em LP. Você precisará ter instalado em seu computador a versão 3 do Python, de preferência a mais recente, mas para esta disciplina qualquer versão igual ou superior a 3.7 será suficiente. Se precisar, baixe a versão mais recente em <https://www.python.org/downloads/>. Este é o mesmo instalador utilizado na disciplina de LP, e a configuração de instalação será exatamente a mesma. O que muda agora é o editor que utilizaremos para escrever os programas.

Um programa de computador ou um projeto pode ser desenvolvido utilizando apenas o bloco de notas. Um arquivo de código-fonte em Python nada mais é que um arquivo de texto codificado utilizando a tabela UTF-8 e com a extensão \*.py ao invés de \*.txt. O programa pode ter sido feito no bloco de notas, e podemos abrir um prompt de comando no windows e executá-lo diretamente, o único requisito é possuir o interpretador do Python instalado e acessível.

A linguagem Python possui o próprio Ambiente de Desenvolvimento Integrado ou IDE, da sigla em inglês Integrated Development Environment, chamado IDLE. A IDLE é uma IDE voltada para o aprendizado da linguagem Python, o L vem de Learning, e é instalado automaticamente junto com a instalação do interpretador do Python (nos sistemas Windows, para Linux pode ser necessária a instalação à parte da IDLE).

A IDLE é um programa que contém um console interativo para executar comandos do Python, chamado Shell, e um editor de texto próprio, com realce de sintaxe e outras ferramentas simples.

Uma IDE, de forma geral, é um software que provê facilidade quando estamos escrevendo um programa, com o intuito de aumentar a produtividade ao integrar diversas ferramentas em um mesmo ambiente. Como vimos, a IDLE provê a coloração do código (realce de sintaxe), que facilita a identificação de comandos e estruturas, e a integração com a Shell, que nos permite executar o programa escrito e inspecionar as variáveis criadas após sua execução.

A definição do que um programa precisa para ser considerado uma IDE pode variar, mas deve incluir pelo menos:

* Um editor de texto para a linguagem em questão;
* Possibilidade de gerenciamento de arquivos;
* Construção automática, isto é, um compilador ou interpretador capaz de gerar o código binário e realizar os processos necessários para que o código possa ser executado;
* Uma ferramenta de depuração (debugger).

Outras funcionalidades muito comuns em IDEs modernas incluem:

* Função de autocompletar os comandos digitados;
* Integração com ferramentas para, entre outros:
* Versionamento (GIT);
* Construção de interfaces gráficas (GUI); e
* Construção de diagramas.

Utilizaremos o Visual Studio Code, que abreviaremos para VSCode, uma IDE moderna desenvolvida pela Microsoft, de código aberto, com suporte a diversas linguagens e totalmente configurável. O VSCode (<https://code.visualstudio.com/>) está disponível para Windows, Linux e MacOS, vem com o básico para começarmos a programar e permite a instalação de extensões disponibilizadas pela comunidade para atender uma grande variedade de necessidades.

O VSCode é uma IDE de propósito geral, ou seja, pode ser utilizado para diversas linguagens simultaneamente. Ele provê grande parte das facilidades mencionadas, além de ser leve e ter uma ótima integração com a linguagem Python. No entanto, a instalação padrão vem somente com as funcionalidades básicas, e as ferramentas específicas para cada linguagem devem ser instaladas como extensões, usando a própria interface do VSCode. Para esta disciplina será utilizada a extensão do Python, desenvolvida pela própria Microsoft.

## Princípios norteadores e um guia de estilo para Python

Python é uma linguagem de código aberto, e como tal, alterações e melhorias na linguagem são propostas[[1]](#footnote-0), discutidas, avaliadas e implementadas pela própria comunidade de desenvolvedores. Para gerenciar todo esse processo, existe a Python Software Foundation, organização sem fins lucrativos responsável por manter e avançar o desenvolvimento da linguagem, veja mais em https://www.python.org/psf-landing/.

Uma das principais ideias que guiaram o criador da linguagem Python, Guido van Rossum, é que programas são lidos com muito mais frequência do que são escritos (Python Software Foundation, 2021).

Pense em uma aplicação qualquer, um trecho de código é escrito apenas uma vez, mas conforme a aplicação evolui, eventualmente precisaremos revisitá-lo, seja para corrigir um bug, para aprimorar o algoritmo utilizado ou para adicionar uma nova funcionalidade às já existentes, e para fazer isso corretamente, é imprescindível entender o que o código que já está lá faz, antes de podermos incluir as modificações necessárias.

Agora imagine que você precise editar um código escrito por outro desenvolvedor, que há anos não trabalha mais na empresa, e que já teve alterações feitas por diversas outras pessoas. Se cada um programar da forma que bem entender, muito rapidamente a situação fica insustentável e você gastaria muito mais tempo apenas tentando entender o que já está ali do que de fato implementando algo novo.

Por isso podemos dizer que a legibilidade e consistência do código são extremamente importantes, tão importantes quanto o algoritmo em si. Para isso, temos um guia de estilo desenvolvido pela comunidade Python e apresentado na proposta de melhoria número 8, a PEP 8 (Rossum, G. V., Warsaw, B., Coghlan, N., 2013).

Tim Peters (1999), resumiu os princípios que guiaram o desenvolvimento da linguagem Python, totalizando 19 aforismas, em um poema que leva o nome de “Zen do Python”. Estes princípios foram incluídos oficialmente na PEP 20, em 2004 (Peter, T., 2004) e podem ser acessados na Shell do Python, com o seguinte comando:

Code:

>>> import this

PEP 8 - Guia de Estilo

O guia de estilo para o Python é bastante detalhado e foi desenvolvido pensando justamente em aprimorar tanto a legibilidade quanto a consistência do código. Ele cobre os principais pontos, de como nomear variáveis e funções a como quebrar linhas que ficam muito compridas, passando por quando é recomendado pular linhas no código e como devemos usar variáveis booleanas em estruturas condicionais, entre muitas outras recomendações.

O guia pode ser lido na íntegra em <https://www.python.org/dev/peps/pep-0008/>, mas não é algo que deve ser lido uma vez e decorado. Pelo contrário, deve ser incorporado organicamente como uma filosofia de desenvolvimento, sendo consultado sempre que necessário.

Como todo guia ou recomendação, isso não diz respeito às regras sintáticas do Python, portanto o fato de um trecho de código não segui-lo não surtirá nenhum efeito do ponto de vista de execução desse código, mas pode reduzir bastante sua legibilidade por outras pessoas que precisarem editá-lo (incluindo você mesmo no futuro).

Conforme avançamos na disciplina, falaremos um pouco mais sobre uma ou outra recomendação, de modo que vocês possam aprendê-las aos poucos, aplicando-as de maneira orgânica em seus códigos de acordo com a necessidade. Outra coisa muito importante quando se trata de um guia de estilo geral como este, é saber quando não se deve usá-lo. Não existe uma resposta correta para essa questão, portanto sempre que estiver em dúvida, não hesite em perguntar e discutir com colegas a melhor abordagem (use o “Zen do Python” para guiar a discussão).

Como dito anteriormente, o guia de estilo foi pensado para melhorar a consistência, mas não adianta ser consistente com o guia se isso faz o código inconsistente com outra parte do código. É comum projetos ou empresas terem o seu próprio guia de estilo, que pode ser ou não baseado no guia geral, portanto é importante saber quando seguir e quando ignorar uma recomendação.

*Um guia de estilo fala sobre consistência. Consistência com este [PEP 8] guia é importante. Consistência interna em um projeto é mais importante. Consistência interna de um módulo ou função é ainda mais importante. (Rossum, G. V., Warsaw, B., Coghlan, N., 2013)*

A PEP8 lista também algumas razões e situações em que o mais sensato é ignorar a recomendação, como por exemplo:

1. Se por algum motivo aplicar a recomendação torna o código menos legível, incluindo para alguém já familiarizado com o guia da PEP8;
2. Para manter a consistência com o código ao redor, que também quebra determinada recomendação (talvez por motivos históricos, embora esta possa ser uma boa oportunidade de “limpar” a bagunça de outra pessoa);
3. Porque o código em questão foi escrito antes do lançamento da recomendação e não há nenhuma outra razão para modificá-lo;
4. Quando o código em questão precisa ser compatível com uma versão mais antiga do Python que não suporta o recurso utilizado pela recomendação.

Para saber mais a respeito do guia de estilo do Python, acesse How to Write Beautiful Python Code With PEP 8 – Real Python - <https://realpython.com/python-pep8/>.

# 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.

## 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:

code:

>>> type(3)

<class 'int'>

>>> type("3")

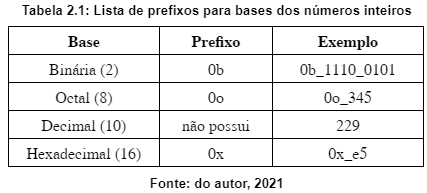
<class 'str'>

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.

### 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).



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.

### 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.

### 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).

### 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:

code:

>>> '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):

code:

>>> 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).

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:

code:

>>> 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:

code:

>>> 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.

code:

>>> 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

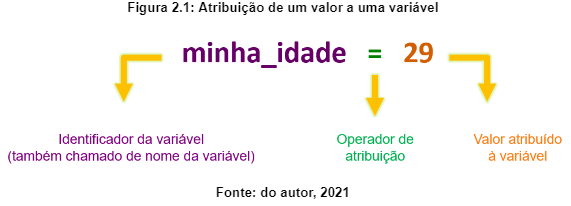
' 3.142'

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).

## 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.



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.

code:

>>> 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:

code:

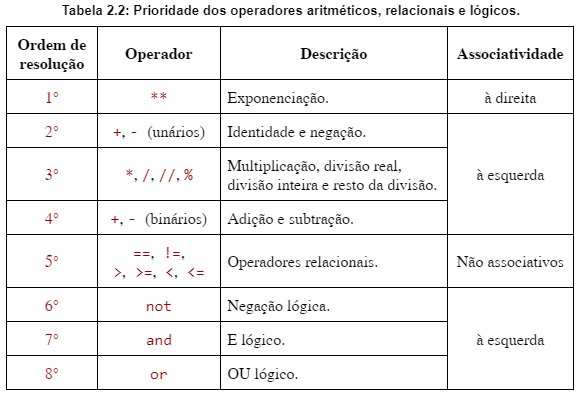
>>> help('keywords')

## Operadores

Vamos relembrar três tipos de operadores aqui:

* Aritméticos: usados para realizar operações matemáticas entre os operandos;
* Relacionais: usados para comparar dois objetos em Python;
* 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.



## 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[[2]](#footnote-1) da 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.

code:

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:

code:

>>> 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 (DOWNEY, 2016).

# Listas, Tuplas e Dicionários em Python

Neste capítulo vamos fazer uma breve revisão de sequências (listas e tuplas) e das estruturas básicas de controle de fluxo: decisão e repetição. Além disso, vamos aprender sobre dicionários em Python, estruturas que facilitam a manipulação de dados dentro de um programa e são extremamente úteis em diversas situações. Praticamente todas as linguagens modernas possuem estruturas semelhantes ou análogas.

## Sequências

O Python possui três tipos básicos de sequências: listas, tuplas e intervalos (range), e mais dois tipos feitos especificamente para lidar com sequências de caracteres (strings) e de dados binários. Nesta aula vamos revisar as sequências de lista e tupla.

A principal diferença entre elas é que a lista é uma sequência mutável, cujos itens podem ser alterados, e a tupla é imutável, ou seja, após criada não pode mais ser alterada. Lembre-se que aqui estamos falando dos objetos na memória, e não do conteúdo de uma variável, pois sempre podemos atribuir um novo valor a uma variável, sobrescrevendo o valor anterior.

A lista completa de operações que podemos fazer em todas as sequências é chamada de Operações Comuns de Sequências (PSF, 2021a). Para as listas, temos ainda as operações listadas em Tipos de Sequências Mutáveis (PSF, 2021b). Não deixe de conferir na bibliografia o link para a documentação do Python em cada caso.

Faremos aqui uma breve revisão de listas e tuplas em Python, recuperando os conceitos vistos em Linguagem de Programação.

## Listas

Uma lista em Python é uma estrutura de dados linear, ordenada, mutável e heterogênea, isto é, os itens de uma lista tem uma posição fixa, podem ser modificados e podem ser de tipos diferentes entre si. Ela é representada em Python por colchetes, sendo os itens separados por vírgula. Vejamos alguns exemplos:

code:

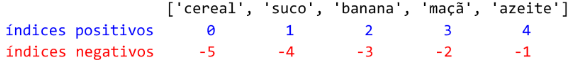
>>> inteiros = [1, 10, 3]

>>> compras = ['cereal', 'suco', 'banana', 'maçã', 'azeite']

>>> lista\_mista = [5, 'bla', True, 10.3]

>>> lista\_vazia = []

Cada elemento adicionado em uma lista ganha automaticamente um índice, referente à sua posição na lista (primeiro, segundo, etc.). Em Python, os índices começam no zero e aumentam constantemente para a direita conforme necessário. Há também um índice negativo, que começa em -1, a partir do final da lista, e diminui conforme caminhamos de volta para o começo da lista, como mostra a Figura 3.1.



Veremos a seguir algumas das ações que podemos realizar em uma lista, confira a lista completa na documentação do Python (PSF, 2021a; PSF, 2021b).

### Acessar um item da lista

Para acessar um elemento da lista, usamos também um par de colchetes com o índice do elemento em questão.

code:

>>> inteiros[1]

10

>>> compras[0]

'cereal'

Lembrando que os índices começam em zero, então se tentarmos acessar o quinto elemento da lista de compras definida acima, devemos usar o índice 4.

code:

>>> compras[4]

'azeite'

Se tentarmos usar o valor 5 para o índice, iremos receber um erro dizendo que o valor está fora do intervalo de índices da lista.

code:

>>> compras[5]

(...)

IndexError: list index out of range

Já o índice negativo nos permite acessar de maneira fácil o último elemento de uma lista, sem precisarmos nos preocupar com o tamanho da lista.

code:

>>> compras[-1]

'azeite'

### Substituir um item da lista

As posições na lista funcionam de maneira análoga a uma variável, então podemos simplesmente atribuir um valor a um elemento existente, sobrescrevendo assim o valor anterior:

code:

>>> compras[1] = 'limão'

>>> compras

['cereal', 'limão', 'banana', 'maçã', 'azeite']

### Inserir um item em uma dada posição na lista

Para inserir um item na lista usamos o método insert, passando a ele como argumentos o índice e o valor a ser inserido na lista. Em breve ficará mais claro como funcionam os métodos de um objeto, por hora, precisamos saber que devemos chamá-los a partir do objeto que queremos alterar, usando a notação de ponto:

code:

>>> compras.insert(1, 'pão')

>>> compras

['cereal', 'pão', 'limão', 'banana', 'maçã', 'azeite']

Observe que foi inserido um novo valor na segunda posição (índice 1), e que todos os valores da lista a partir dessa posição foram deslocados para a direita.

### Remover um item da lista

Existem três formas de se remover um item de uma lista:

1. Com o comando del: esta palavra chave deleta um objeto da memória, podemos usá-la também para deletar uma variável por exemplo, então basta acessar o item que queremos excluir e passá-lo ao comando del:

code:

>>> del compras[-1]

>>> compras

['cereal', 'pão', 'limão', 'banana', 'maçã']

1. Com o método pop: este método recebe um índice como argumento, exclui o respectivo item e retorna o seu valor, caso nenhum índice seja passado, por padrão remove e retorna o último item:

code:

>>> compras.pop(3)

'banana'

>>> compras

['cereal', 'pão', 'limão', 'maçã']

1. Com o método remove: este método recebe um valor e faz uma busca na lista, removendo o primeiro item que corresponder ao valor passado. Se houverem itens repetidos, apenas o primeiro é removido:

code:

>>> compras.remove('pão')

>>> compras

['cereal', 'limão', 'maçã']

### Acrescentar um item ao final da lista

Para acrescentar um item ao final de uma lista, usamos o método append, que recebe um objeto e o adiciona ao final da lista:

code:

>>> compras.append('sorvete')

>>> compras

['cereal', 'limão', 'maçã', 'sorvete']

VAMOS PRATICAR!

1) Tente usar o append para adicionar uma lista ao final de outra lista e veja o que acontece.

code:

>>> alex\_primeiro = [1,2,3]

>>> alex\_primeiro

[1, 2, 3]

>>> alex\_primeiro.append([3,2,1])

>>> alex\_primeiro

[1, 2, 3, [3, 2, 1]]

2) Agora refaça o teste usando o método extend e compare os resultados.

code:

>>> alex\_primeiro = [1,2,3]

>>> alex\_primeiro

[1, 2, 3]

>>> alex\_primeiro.extend([3,2,1])

>>> alex\_primeiro

[1, 2, 3, 3, 2, 1]

### Concatenar duas listas

Podemos também concatenar duas listas usando o operador +, de maneira análoga a concatenação de strings:

code:

>>> compras + inteiros

['cereal', 'limão', 'maçã', 'sorvete', 1, 10, 3]

Na concatenação, uma nova lista é gerada e retornada pela operação, sem que as listas originais sofram qualquer alteração[[3]](#footnote-2).

## Tuplas

Assim como listas, tuplas são estruturas de dados lineares, ordenadas e heterogêneas, no entanto, ao contrário de listas, tuplas são imutáveis. Em Python, as tuplas são definidas por um par de parênteses, com os itens separados por vírgula.

code:

>>> tupla = (1, 10, 3)

>>> tupla\_vazia = ()

Para definir uma tupla com um único elemento, devemos colocar uma vírgula extra para que os parênteses não sejam interpretados com uma expressão aritmética.

code:

>>> tupla\_de\_um\_item = (8,)

>>> nao\_eh\_uma\_tupla = (5)

Ao separar elementos por vírgula, mesmo sem os parênteses, o Python também irá criar uma tupla.

code:

>>> nova\_tupla = 14, 25, 91

>>> nova\_tupla

(14, 25, 91)

Como tuplas não podem ser modificadas, apenas os métodos comuns a todas as sequências são aplicáveis. Dos exemplos visto em lista, podemos apenas acessar um valor da tupla e concatenar duas tuplas, pois em nenhum dos casos os objetos originais são modificados. Da mesma forma que em listas, acessar um item inexistente irá gerar um erro de execução.

code:

>>> tupla[1]

10

>>> tupla + tupla\_de\_um\_item

(1, 10, 3, 8)

Dica: essas mesmas duas operações também funcionam com strings, que são um tipo especial de sequência imutável. Para aprender mais sobre as estruturas de dados em Python veja o tutorial oficial (PSF, 2021c).

## Dicionários

Dicionários são uma estrutura de mapeamento, atualmente a única estrutura padrão desse tipo em Python, mas o que é uma estrutura de mapeamento? Uma estrutura de mapeamento cria uma associação entre dois objetos, uma chave e um valor. Em outras linguagens, estruturas semelhantes são chamadas de hashmaps, hash tables ou arrays associativos.

Para criar um dicionário em Python, colocamos, entre chaves, uma lista de pares chave: valor separados por vírgula. Vejamos um exemplo:

code:

>>> dicionario\_vazio = {}

>>> notas = {'Jack': 8.3, 'Anna': 9.0, 'Cris': 7.5}

Essa é a forma literal de se criar um dicionário em Python, mas há diversas outras formas que podem ser mais vantajosas em algumas situações, em especial quando precisamos converter dados entre diferentes tipos de estruturas. Estas formas podem ser vistas na documentação (PSF, 2021f), como por exemplo criar um dicionário a partir de uma lista de tuplas com dois valores.

O exemplo a seguir cria o mesmo dicionário atribuído a variável notas no exemplo anterior, só que a partir de uma lista já existente, usando a função dict().

code:

>>> lista\_notas = [('Jack', 8.3), ('Anna', 9.0), ('Cris', 7.5)]

>>> notas\_2 = dict(lista\_notas)

>>> notas == notas\_2

True

As chaves em um dicionário precisam ser únicas, então ao criarmos dois itens com a mesma chave, o valor do segundo item sobrescreve o do primeiro.

code:

>>> notas = {'Jack': 8.3, 'Anna': 9.0, 'Jack': 7.5}

{'Jack': 7.5, 'Anna': 9.0}

Podemos pensar na chave como se fosse o “índice” do valor a que estamos nos referindo, mas agora temos a liberdade de criar esses “índices”. Os dicionários só aceitam como chave objetos que sejam imutáveis, como por exemplo strings, inteiros, floats e tuplas cujos itens sejam também imutáveis, mas vale ressaltar que não é uma boa ideia usar um float como chave pois, devido a sua natureza, só é possível guardar um valor aproximado dele na memória, o que pode levar a erros ou inconsistências. As chaves mais comumente usadas são strings e inteiros.

Quanto aos valores de um dicionário, não há nenhuma restrição de tipo, então podemos guardar quaisquer objetos do Python, inclusive outros dicionários. Ainda, os valores podem ser repetidos sem nenhum problema.

Veremos agora alguns métodos de dicionários, a lista completa pode ser encontrada na documentação (PSF, 2021f).

### Acessar um valor do dicionário

Para acessar os valores de um dicionário, de maneira análoga a listas ou tuplas, devemos indicar o valor da chave entre colchetes:

code:

>>> notas['Jack']

7.5

>>> notas['Anna']

9.0

Se tentamos acessar uma chave que não existe no dicionário, obtemos um erro:

code:

>>> notas['Megan']

(...)

KeyError: 'Megan'

Outra forma de acessar o valor de um dicionário é usar o método get, cuja sintaxe é dicionario.get(key, default=None), e ele retorna o valor associado à chave, ou o valor default caso a chave não exista.

code:

>>> notas.get('Megan')

>>> notas.get('Megan', 'Nome não encontrado')

'Nome não encontrado'

Observe que a primeira chamada retorna None, portanto a Shell não exibe nada.

### Inserir um valor no dicionário

Ao contrário de listas, para acrescentar um novo valor ao dicionário podemos fazer uma atribuição diretamente a uma chave, sendo ela existente ou não.

code:

>>> notas['Megan'] = 8.0

>>> notas

{'Jack': 7.5, 'Anna': 9.0, 'Megan': 8.0}

Novos itens são sempre inseridos no final, pois desde a versão 3.7 do Python, os dicionários guardam a ordem de inserção dos pares chave-valor. Caso a chave já exista, o seu valor será sobrescrito e a ordem não é alterada.

code:

>>> notas['Jack'] = 6.0

>>> notas

{'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}

Para atualizar diversos valores de uma única vez, podemos usar o método update, que pode receber como argumento uma lista de tuplas, como vimos na criação de um dicionário com a função dict, ou então um outro dicionário.

Caso haja chaves repetidas, os valores mais à direita terão prioridade, pois serão inseridos por último e irão sobrescrever os anteriores.

code:

>>> inteiros = {}

>>> inteiros.update({1: 'um', 2: 'dois'})

>>> inteiros

{1: 'um', 2: 'dois'}

>>> inteiros.update([(3, 'três'), (4, 'quatro')])

>>> inteiros

{1: 'um', 2: 'dois', 3: 'três', 4: 'quatro'}

Na versão 3.9 do Python, foi introduzido o operador de união para dicionários também (este operador já existia para conjuntos em Python - set), representado pelo caractere de barra vertical (pipe): |.

coe:

>>> d1 = {'a': 1, 'b': 2}

>>> d2 = {'c': 3: 'd': 4}

>>> d1 | d2

{'a': 1, 'b': 2, 'c': 3: 'd': 4}

### Excluir um item do dicionário

Para excluir um par chave-valor do dicionário podemos acessar o objeto e passá-lo ao comando del, como vimos em listas, e caso a chave não exista, objetos um KeyError.

code:

>>> del inteiros[2]

>>> inteiros

{1: 'um', 3: 'três', 4: 'quatro'}

Ou podemos usar o método pop, que nos retorna o valor associado à chave e exclui o par chave-valor do dicionário.

code:

>>> inteiros.pop(4)

'quatro'

>>> inteiros

{1: 'um', 3: 'três'}

Caso a chave não esteja no dicionário, também obtemos um KeyError. Para evitar o erro, podemos passar mais um argumento para o método, que será usado como valor padrão a ser retornado quando a chave não existir.

code:

>>> inteiros.pop(5)

(...)

KeyError: 5

>>> inteiros.pop(5, 'chave não encontrada')

'chave não encontrada'

### Operações de pertencimento em um dicionário

Os operadores de pertencimento, quando usados em um dicionário, fazem a verificação na lista de chaves do dicionário. Podemos então usá-los para verificar se uma chave existe ou não em um dado dicionário.

code:

>>> 5 not in inteiros

True

>>> 'Jonas' in notas

False

Como os valores associados às chaves podem ser qualquer objeto do Python, não há um método padrão para fazer uma verificação da existência de valores no dicionário. Quando isso for necessário, devemos escrever nossos próprios métodos ou funções para tal. Ao aprendermos a criar classes poderemos usar o conceito de herança para estender um tipo do Python, criando nosso próprio tipo personalizado com recursos e funcionalidades extras.

### Métodos especiais para iterar sobre um dicionário

É comum precisarmos iterar sobre todos os itens de um dicionário, e usando um laço for, podemos fazer isso iterando chave a chave. Crie um arquivo “dicionarios.py”, na pasta “aula03”, com o seguinte código:

code:

notas = {'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}

for chave in notas:

print(chave, notas[chave])

Agora, com o VSCode aberto na pasta da disciplina, execute no terminal:

code:

> python aula03/dicionarios.py

Jack 6.0

Anna 9.0

Megan 8.0

Neste exemplo, estamos iterando sobre as chaves do dicionário, e precisamos acessar o elemento usando a notação dicionario[chave]. Isso é uma operação bastante eficiente, devido a natureza dos dicionários em Python, mas há uma forma ainda melhor de fazer esta iteração, com os métodos dict.keys(), dict.values() e dict.items(). Estes métodos retornam um objeto especial do Python chamado Objeto de Visualização de Dicionários (PSF, 2021g), que nos fornece uma visão dinâmica sobre as entradas do dicionário.

Podemos pensá-los como “listas” que podem ser iteradas e nas quais podemos fazer operações de pertencimento. Altere o laço for do arquivo “dicionarios.py” para:

code:

for nome, nota in notas.items():

print(nome, nota)

Ao executar o arquivo, o resultado obtido será o mesmo, mas o código está muito mais legível, uma vantagem ainda maior quando trabalhamos com situações mais complexas que a do exemplo.

O método dict.keys() retorna uma visualização das chaves do dicionário, o dict.values() dos valores, e o dict.items() dos pares chave-valor. Para ver o efeito destes métodos na Shell podemos converter o retorno para uma lista estática:

code:

>>> notas = {'Jack': 6.0, 'Anna': 9.0, 'Megan': 8.0}

>>> list(notas.keys())

['Jack', 'Anna', 'Megan']

>>> list(notas.values())

[6.0, 9.0, 8.0]

>>> list(notas.items())

[('Jack', 6.0), ('Anna', 9.0), ('Megan', 8.0)]

Observe que essa conversão para lista não é necessária quando estivermos usando estes métodos em um laço for, e não deve ser feita, pois reduz a legibilidade do código e retira a dinamicidade do objeto original, podendo afetar seu desempenho.

## Operações de pertencimento em listas, tuplas e strings

O operador de pertencimento tem a mesma precedência dos operadores relacionais e pode ser usado para todos os tipos de sequências, verificando se um objeto está ou não contido na sequência.

code:

>>> tupla = (1, 10, 3)

>>> compras = ['cereal', 'limão', 'maçã', 'sorvete']

>>> texto = 'Olá mundo!'

>>> 3 in tupla

True

>>> 'uva' not in compras

True

>>> 'M' in texto

False

Observe que a negação do operador retorna True quando o objeto não é encontrado na lista, e que a letra M maiúscula de fato não existe na string da variável texto, pois o Python diferencia letras maiúsculas de minúsculas tanto em identificadores (nomes de variáveis, funções, etc.) quanto em strings.

## Desempacotamento de sequências

O desempacotamento de sequências (listas, tuplas e conjuntos), é uma forma de atribuir os itens de uma sequência a diferentes variáveis, que pode ser usado com o operador de atribuição, em laços do tipo for e na passagem de argumentos para funções. Vejamos o uso com o operador de atribuição:

code:

>>> a, b = [1, 2]

>>> a

1

>>> b

2

No exemplo anterior, o número de itens precisa ser igual ao número de variáveis e vice-versa, caso contrário o Python irá levantar um erro dizendo que não foi possível realizar o desempacotamento.

Se tivermos uma sequência cujos itens sejam em si uma sequência, podemos usar esse recurso também em laços do tipo for. Imagine a seguinte lista, onde cada item é uma tupla com um par (<letra>, <número>):

code:

>>> lista = [('a', 1), ('b', 2), ('c', 3)]

Iterando sobre esta lista com um for tradicional, podemos fazer:

code:

>>> for item in lista:

... print(f'{item} --> {item[0]}: {item[1]}')

('a', 1) --> a: 1

('b', 2) --> b: 2

('c', 3) --> c: 3

Ou seja, se quisermos acessar a letra e o número em cada item, precisamos usar os índices como no exemplo anterior, mas com o desempacotamento, podemos fazer:

code:

>>> for letra, numero in lista:

... print(f'{letra}: {numero}')

a: 1

b: 2

c: 3

Agora podemos acessar diretamente a letra e o número de cada item dentro do for, que estão disponíveis em variáveis que deixam o código mais legível, quando comparamos a item[0] e item[1]. Observe que esse desempacotamento ocorre no momento de atribuição de valor que é realizado pelo laço for.

Para realizar o desempacotamento para os argumentos de uma função, devemos usar um \* antes da sequência que será desempacotada na chamada da função:

code:

>>> def teste\_desempacotamento(a, b, c):

... print(f'{a=}, {b=}, {c=}')

>>> sequencia = 35, 21, 9

>>> teste\_desempacotamento(\*sequencia)

a=35, b=21, c=9

# Estruturas de controle de fluxo

São as estruturas que nos permitem alterar o fluxo sequencial de execução do código, seja escolhendo um determinado caminho em função da avaliação de uma condição (estruturas de seleção), seja repetindo um trecho de código (estruturas de repetição).

## Estruturas de seleção

Em Python, a estrutura de seleção é feita com o comando if, seguido de uma condição. Caso a condição seja verdadeira, o bloco de código definido por esse comando será executado, e caso seja falsa, será ignorado.

code:

if <condição>:

<bloco de código>

O guia de estilo recomenda não utilizar parênteses em torno da condição e caso seja utilizada uma flag booleana, não é recomendado a comparação com os tipos True e False, deve-se usar diretamente a flag com o operador not se necessário: if flag: ou if not flag:, respectivamente.

É possível também definir um bloco de código para ser executado quando a condição do comando if é avaliada para False, com o comando else.

code:

if <condição>:

<bloco de código se verdadeira>

else:

<bloco de código se falsa>

Observe que não há nenhuma condição após o comando else, pois este bloco irá se e somente se a condição do comando if for falsa.

Há ainda o comando elif, que é uma contração dos comandos else + if, quando temos laços encadeados, com o objetivo de reduzir uma indentação excessiva.

code:

if <condição C1>:

<bloco de código se verdadeira>

else:

if <condição C2>:

<bloco de código se C1 verdadeira e C2 falsa>

else:

<bloco de código se C1 e C2 falsas>

Podemos reescrever o código do exemplo anterior usando o comando elif, da seguinte forma:

if <condição C1>:

<bloco de código se verdadeira>

elif <condição C2>:

<bloco de código se C1 verdadeira e C2 falsa>

else:

<bloco de código se C1 e C2 falsas>

Observe que os blocos de execução são exatamente os mesmos, mas agora temos apenas um nível de indentação, independentemente de quantos comandos elif houver. Veja o seguinte exemplo, retirado do tutorial oficial do Python (PSF, 2021d).

code:

>>> x = int(input("Por favor entre um inteiro: "))

Por favor entre um inteiro: 42

>>> if x < 0:

... x = 0

... print('Negativo alterado para zero')

... elif x == 0:

... print('Zero')

... elif x == 1:

... print('Unitário')

... else:

... print('Mais que um')

## Estruturas de repetição

As estruturas de repetição podem ser divididas em dois grupos, indefinidas e definidas. No primeiro, não sabemos a priori quantas vezes a instrução será executada pois dependemos da avaliação de uma condição. Já no segundo, estamos iterando sobre uma sequência, portanto o número de repetições será definido pelo seu tamanho, e as instruções executadas uma vez para cada valor da sequência.

### Estruturas de repetição indefinidas - While

As estruturas de repetição indefinidas são feitas com o comando while:

code:

while <condição>:

<bloco de código>

Esta estrutura é muito parecida com a estrutura de seleção simples (o comando if), mas a principal diferença é que, ao contrário do if, após a execução do bloco de código, o while irá reavaliar a condição e o processo se repete enquanto esta avaliação resultar em True.

code:

soma = 0

while soma < 21:

carta = int(input('Digite o valor da carta: '))

soma += carta

print(f'Seu resultado é {soma}')

No exemplo acima, é pedido o valor de uma carta e este valor é acumulado em uma variável soma, caso esta soma seja maior ou igual a 21, a condição resultará em falso e o laço é encerrado. Em outras palavras, enquanto a soma for menor que 21, será pedido o valor de mais uma carta e o processo se repete.

### Estruturas de repetição definidas - For

As estruturas de repetição definidas são feitas com o comando for:

code:

for <variável> in <sequência>:

<bloco de código>

O bloco de código de um laço for será executado uma vez para cada valor da sequência, com o valor da vez sendo atribuído à variável no início do laço. Isto é, ao começar um laço for, o Python irá atribuir o primeiro valor da sequência à variável definida no laço, executar o bloco de código e repetir o processo enquanto houver valores na sequência. Após o último valor, o laço é encerrado automaticamente e segue-se o fluxo de execução.

Execute o código do exemplo a seguir para ver o funcionamento deste laço.

code:

lista = ['a', 1, True, 3.5]

for valor in lista:

print(f'valor: {valor}, do tipo: {type(valor)}')

print('-----\nfim').

Uma função muito utilizada em conjunto com os laços definidos é a função range, que cria um intervalo de números inteiros em Python. Esta função pode receber 1, 2 ou 3 parâmetros, que devem sempre ser números inteiros. Ao receber um único parâmetro, é gerada uma sequência começando em zero e indo até o número anterior ao número dado, ou seja, o número dado será o tamanho da sequência.

# VSCode - modo de depuração

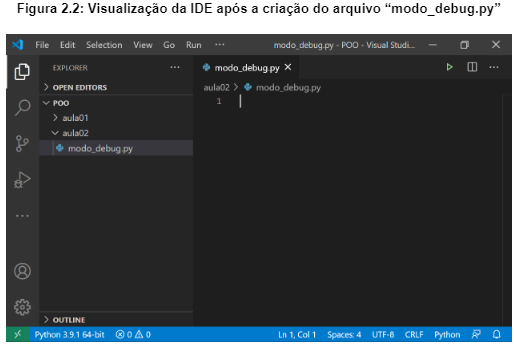
Ao executar um arquivo Python no terminal, ele é executado do início ao fim, de modo que todas as variáveis criadas são perdidas e temos acesso apenas àquilo que foi exibido na tela. Um método de depuração básico, que pode nos ajudar a resolver muitos problemas, consiste em colocar diversos “prints”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

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.



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

code:

def dobrar(x):

return 2 \* x

def triplicar(x):

triplo = 3 \* x

return triplo

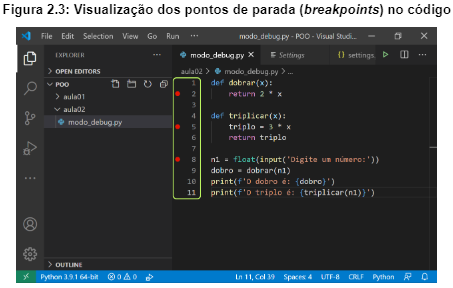
n1 = float(input('Digite um número:'))

dobro = dobrar(n1)

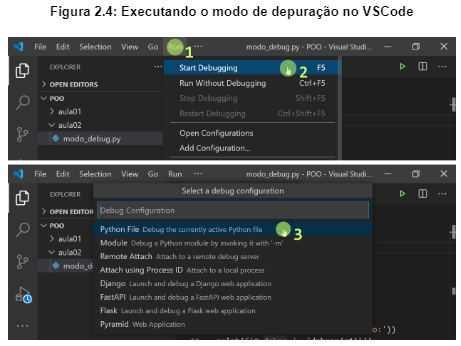
print(f'O dobro é: {dobro}')

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.



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.



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.

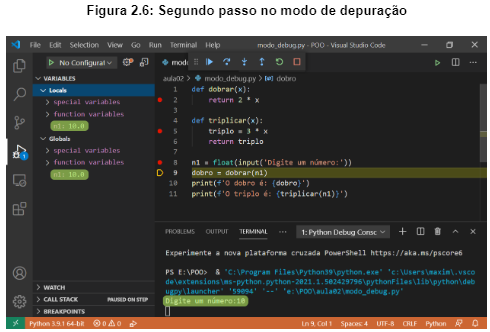


Observe que estamos agora na aba lateral esquerda referente ao modo de depuração, na qual temos algumas janelas, vamos focar agora na de variáveis, portanto podemos minimizar as outras clicando nos símbolos de seta ao lado dos nomes de cada uma. Além disso, surgiu no topo da janela um menu extra com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

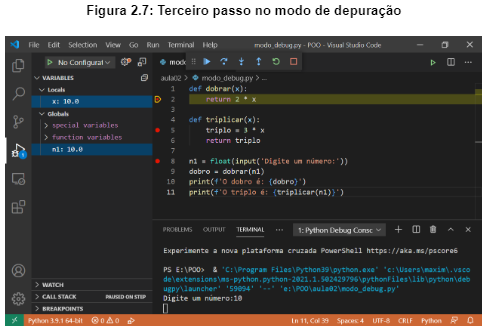
Nos botões que surgiram temos as seguintes ações:

* 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.



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 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.



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.

# 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.

## 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[[4]](#footnote-3) e pep8-naming[[5]](#footnote-4). Para instalar tais pacotes, vá no terminal do VSCode e digite:

code

> py -m pip install flake8 # Windows

$ python3 -m pip install flake8 # Linux e MacOS

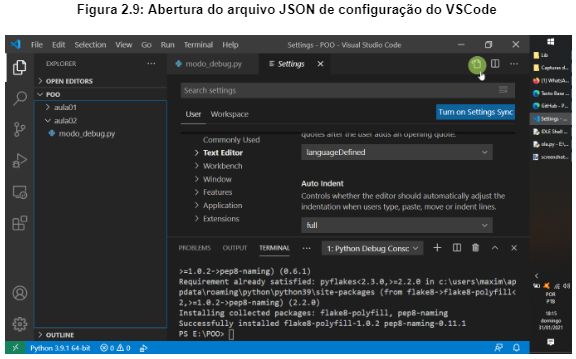
Após a instalação finalizar, digite:

code:

> 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.



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).

code:

{

"python.linting.pylintEnabled": false,

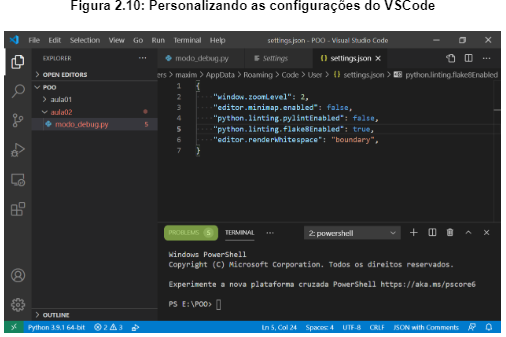
"python.linting.flake8Enabled": true,

"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.

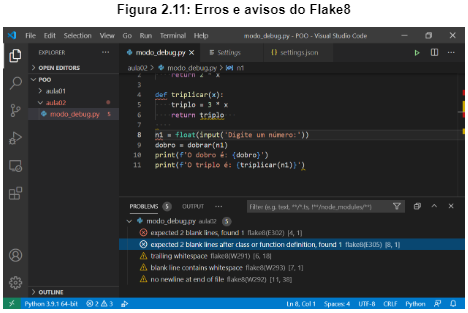
O resultado final pode ser visto na Figura 2.10. Observe que no arquivo do exemplo há uma configuração a mais para definir o zoom da IDE (que pode também ser alterado com os atalhos Ctrl + ‘+’ e Ctrl + ‘-’) e outra para esconder o mini-mapa que aparece por padrão no canto direito da tela. Estas duas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



## 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.



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.



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.

1. Esse processo é realizado através das PEP’s, ou Propostas de Melhoria do Python (do inglês Python Enhancement Proposal). Veja mais em https://www.python.org/dev/peps/. [↑](#footnote-ref-0)
2. 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. [↑](#footnote-ref-1)
3. Há uma exceção à essa regra quando usamos o operador de atribuição composta +=, com ele, o Python irá alterar o mesmo objeto na memória, de maneira análoga à utilização do método extend. [↑](#footnote-ref-2)
4. <https://flake8.pycqa.org/en/latest/index.html#installation-guide> [↑](#footnote-ref-3)
5. [GitHub - PyCQA/pep8-naming: Naming Convention checker for Python](https://github.com/PyCQA/pep8-naming) [↑](#footnote-ref-4)