# SUMÁRIO

[**SUMÁRIO 1**](#_11agubohmyoa)

[**Programação Orientada a Objetos 4**](#_t4yjix6t6a0)

[**Conceitos Essenciais da Programação Orientada a Objetos (POO) 4**](#_ohuwnqu0od79)

[Classes e Objetos 4](#_g2stlqvja39)

[Exemplo 4](#_ysysfs7ogodz)

[Encapsulamento 4](#_ftjy0uxres3k)

[Herança 5](#_8xs9yrqx399p)

[Exemplo 5](#_msygdlpv6gtu)

[Polimorfismo 5](#_ulvbf6o0bq2m)

[Exemplo 5](#_k2u4l6eqrxp6)

[**Apresentação da disciplina e preparação do ambiente de desenvolvimento 5**](#_hzps4ktscmq4)

[Configuração do ambiente de desenvolvimento 6](#_aidqtei601f)

[Princípios norteadores e um guia de estilo para Python 7](#_2ljwcxbj03m2)

[**Revisão de Python básico 9**](#_3d06m9q1ytrk)

[Tipos de dados 9](#_n65hm0zdto4i)

[int 9](#_ciw0o2qkkzvl)

[float 10](#_vc06p2fuok5v)

[bool 10](#_qq70m07kql76)

[str 10](#_daxp3u2jxy8g)

[Variáveis 12](#_tarrhaoo14j0)

[Operadores 13](#_t7a605coxo94)

[Funções 14](#_b5owgbu5yj5j)

[**Listas, Tuplas e Dicionários em Python 15**](#_fbxi7wjze4mf)

[Sequências 15](#_ruxpuh76twub)

[Listas 15](#_pmshgo2uu63q)

[Acessar um item da lista 16](#_cwwt3rpkwiqu)

[Substituir um item da lista 17](#_wqdmnaax8e8e)

[Inserir um item em uma dada posição na lista 17](#_b42andpptkl)

[Remover um item da lista 17](#_17yid6y1tary)

[Acrescentar um item ao final da lista 18](#_lejvsaczpviu)

[Concatenar duas listas 18](#_timar2jr6kdc)

[Tuplas 19](#_wbqilo5dfe1t)

[Dicionários 19](#_5n35cmb5fov0)

[Acessar um valor do dicionário 20](#_y3vuqj3qrerc)

[Inserir um valor no dicionário 21](#_ngsm045mpw4h)

[Excluir um item do dicionário 22](#_toz6ta87w8iu)

[Operações de pertencimento em um dicionário 22](#_hkonsrp3cynh)

[Métodos especiais para iterar sobre um dicionário 23](#_cjjs2op2dz23)

[Operações de pertencimento em listas, tuplas e strings 24](#_e3132bqemugr)

[Desempacotamento de sequências 24](#_jpbr25ni2kbu)

[**Estruturas de controle de fluxo 25**](#_jogedoangld0)

[Estruturas de seleção 25](#_7hiyvwslul4q)

[Estruturas de repetição 27](#_kzyboiq4on5h)

[Estruturas de repetição indefinidas - While 27](#_to2zvavopbci)

[Estruturas de repetição definidas - For 27](#_1r0wli4scfq5)

[**VSCode - Modo de Depuração 28**](#_um31awbo0ee7)

[**Recomendações PEP 8 31**](#_lt0zb4ut955w)

[Configurações extras no ambiente de desenvolvimento 31](#_2thp03bp8zo7)

[Corrigindo o código 33](#_ps805aovi42q)

[**Paradigmas de Programação e os Pilares de POO 34**](#_qghejtfqpvoc)

[Paradigmas de programação 35](#_9x14hyroxysf)

[Paradigmas Imperativos 36](#_p0zuv6kfm1f7)

[Paradigmas Declarativos 36](#_93fbm99a0hoy)

[Pilares de POO 37](#_b92mjtpchs2x)

[Abstração 37](#_okredbunxgli)

[Encapsulamento 38](#_6o4ikqtuns8n)

[Herança 39](#_x5aki7lir7np)

[Polimorfismo 39](#_7jhuikj46xf0)

[**Criação de classes em Python e Encapsulamento 40**](#_1knv1qtrev36)

[PEP-8 aplicada às classes 41](#_n4167huox824)

[POO em Python 41](#_ue45jmv8brcr)

[Implementando classes em Python 41](#_dyrtvootzwa8)

[Instanciando objetos a partir de uma classe em Python 42](#_9np7pv31tt3a)

[Como inicializar um objeto em Python 43](#_ezbdse6thgm5)

[Entendendo o parâmetro self 44](#_44jkia5o7oz0)

[Personalizando a inicialização dos objetos em Python e incluindo novos métodos 45](#_xyty9ymqfydz)

[Encapsulamento 47](#_f7nmp8hy4b4p)

[Trabalhando com atributos não-públicos em Python 48](#_s5ne6o3clnps)

[Utilizando os decoradores @property e @property.setter 50](#_6axcut9ttz0e)

[**Herança e Polimorfismo em Python 52**](#_lk7b4mxr37ic)

[Herança 52](#_ke5vtwst9r9g)

[Usando a função integrada super em Python 56](#_7rce8864jfm8)

[Atributos e métodos “protegidos” em Python 59](#_4rdkjrg3npe3)

[Polimorfismo 59](#_ety46n49mkvk)

[Sobrecarga 60](#_km6dd95hc3i8)

[Sobrescrita 61](#_40rp4ldg83vo)

[Ordem de resolução dos métodos 63](#_i2zsn4utx01i)

[**SOLID 63**](#_40vj59tmpss9)

[SRP - Single Responsibility Principle (Princípio da Responsabilidade Única) 63](#_uyalee167km4)

[OCP - Open-Closed Principle (Princípio Aberto-Fechado) 63](#_qqzabvfc3zez)

[LSP - Liskov Substitution Principle (Princípio da Substituição de Liskov) 63](#_6nhrj1u48f0s)

[ISP - Interface Segregation Principle (Princípio da Segregação da Interface) 64](#_b2sz8drbs27o)

[DIP - Dependency Inversion Principle (Princípio da Inversão de Dependência) 64](#_x6liew7kbcas)

[Benefícios do SOLID 64](#_2oum6up996pb)

[**Módulos e Pacotes 64**](#_t8653b6mmhde)

[Módulos em Python 65](#_c4ewrr625ct3)

[VOCÊ SABIA? 66](#_a2hx65xxolct)

[Espaço de nomes de um módulo 66](#_57sh7tk0kpuv)

[A variável \_\_name\_\_ 67](#_st8fdvpq5mth)

[A variavel \_\_all\_\_ em Python 69](#_rjxtmu3bponk)

[Formas de importar um módulo 70](#_d09bsxe88i18)

[O caminho de busca de um módulo 70](#_utnfcz2z5v1i)

[Importação com a criação de um alias para o módulo 70](#_9mbhubsj5iej)

[Importação de elementos do módulo 71](#_xe5darifwe40)

[Importação de elemento com a criação de alias 72](#_7mmctuksnzlp)

[Organização de módulos em pacotes e sub-pacotes 72](#_ryjt8kdc7f5u)

[O arquivo \_\_init\_\_.py 73](#_h558hncvjs5m)

[Inicialização dos módulos 75](#_qfxyjedd9dja)

[Nota sobre a criação de pacotes em Python 76](#_tmgx2k5q5phw)

[PEP 8 76](#_s0dpc8gug29a)

[**Ambientes virtuais, utilização de bibliotecas de terceiros e gerenciamento de dependências 77**](#_5rrbelxhydd5)

[Ambientes virtuais 77](#_uk7bzga126st)

[Observações 81](#_m8ueiui59lvb)

[Instalando pacotes com PIP 83](#_ndcpxqh32kmy)

[Gerenciando dependências 84](#_iu0887romrif)

[**Introdução ao Desenvolvimento Guiado por Testes (TDD); testes unitários e tratamento de erros 85**](#_vntps0xnwzbr)

[Desenvolvimento Guiado por Testes (TDD) 86](#_t47iki6gdxk4)

[Ciclos do TDD 86](#_tt1nohdj4tj2)

[Principais vantagens do TDD 86](#_c9t1eikqq484)

[Limitações do TDD 87](#_dpwj5lrzh1ij)

[Testes Unitários e Testes Automatizados 87](#_2ufiicmjdani)

[Instalando o Pytest 90](#_fdc3rg8tsfj9)

[Escrevendo funções de teste com o Pytest 91](#_mjdkhypkbofe)

[Executando os testes com o Pytest 92](#_y4xhre27eqvd)

[Interpretando os resultados do Pytest 92](#_cf1b4p694txx)

[Resumo dos testes coletados 93](#_rj989k1h0r6)

[Saída detalhada dos testes 93](#_iasbrxc15wwq)

[Saída resumida dos testes 93](#_c87f9bmytip2)

[Erros e exceções 95](#_tm0ufnduihi9)

[Lançamento de exceções 96](#_wjtin0dqht7c)

[Criação de exceções 98](#_c6nmdv3zdbn0)

[Tratamento de exceções 98](#_ka1mp143qwji)

# 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.

# Conceitos Essenciais da Programação Orientada a Objetos (POO)

O POO é um paradigma de programação que organiza o código em torno de "objetos". Cada objeto possui características e comportamentos específicos. Essa organização torna o código mais modular, reutilizável e fácil de manter.

## Classes e Objetos

**Classe:** Define um modelo que representa um tipo de objeto. Ela define as características (atributos) e comportamentos (métodos) que todos os objetos desse tipo terão.

**Objeto:** É uma instância de uma classe. Ele possui os atributos e métodos definidos na classe, com valores específicos para cada objeto.

### Exemplo

* Classe: Carro
* Atributos: marca, modelo, ano, cor
* Métodos: dirigir, frear, acelerar

## Encapsulamento

* Esconde os detalhes internos de um objeto, expondo apenas sua interface pública.
* Permite modificar a implementação interna do objeto sem afetar o código que o utiliza.
* Promove a modularidade e a reutilização de código.

## Herança

* Permite que uma classe herde atributos e métodos de outra classe.
* Facilita o reuso de código e promove a organização do código.
* Permite criar classes mais complexas a partir de classes mais simples.

### Exemplo

* Classe: Animal
* Atributos: nome, idade
* Métodos: comer, dormir
* Classe: Cachorro (herda de Animal)
* Atributos: raça, porte
* Métodos: latir, buscar

## Polimorfismo

* Permite que diferentes objetos respondam à mesma mensagem de forma diferente.
* Torna o código mais flexível e adaptável.
* Permite escrever código mais genérico e reutilizável.

### Exemplo

* Método: falar()
* Classe: Animal
* Implementação: Animal faz um som genérico
* Classe: Cachorro (herda de Animal)
* Implementação: Cachorro late
* Classe: Gato (herda de Animal)
* Implementação: Gato mia

# 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")

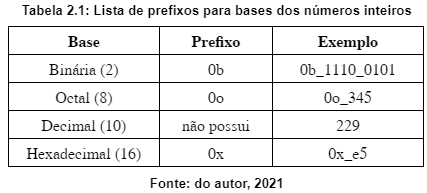
<class 'str'>

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

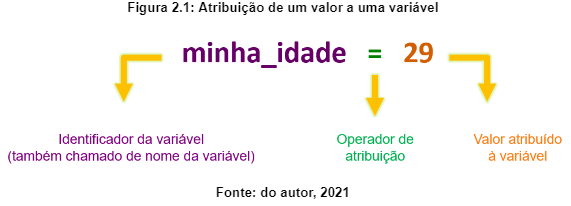
' 3.142'

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:

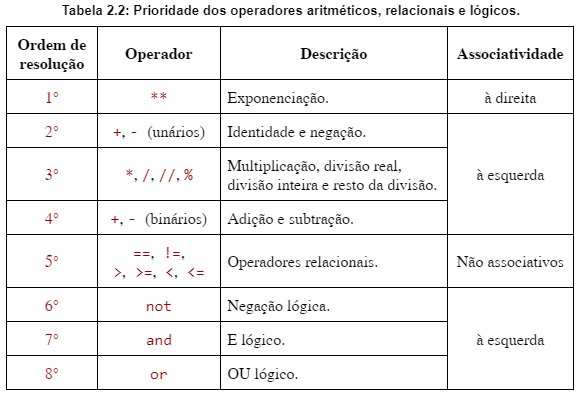
>>> help('keywords')

## 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:

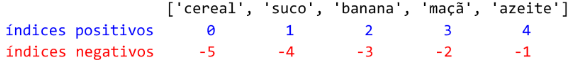
>>> inteiros = [1, 10, 3]

>>> 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

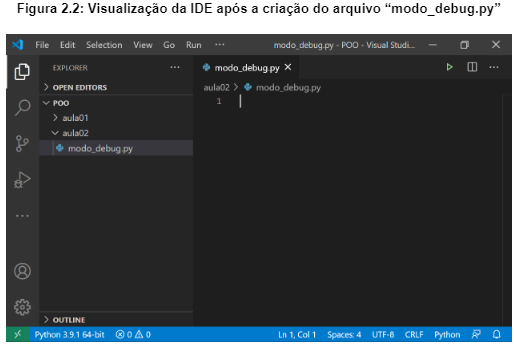
Ao executar um arquivo Python no terminal, ele é executado do início ao fim, de modo que todas as variáveis criadas são perdidas e temos acesso apenas àquilo que foi exibido na tela. Um método de depuração básico, que pode nos ajudar a resolver muitos problemas, consiste em colocar diversos “prints”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

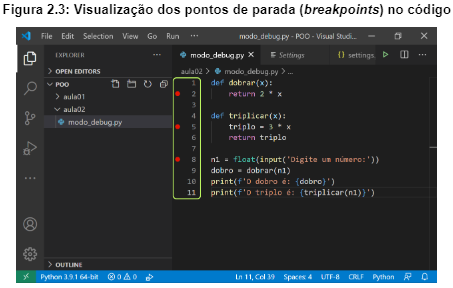
n1 = float(input('Digite um número:'))

dobro = dobrar(n1)

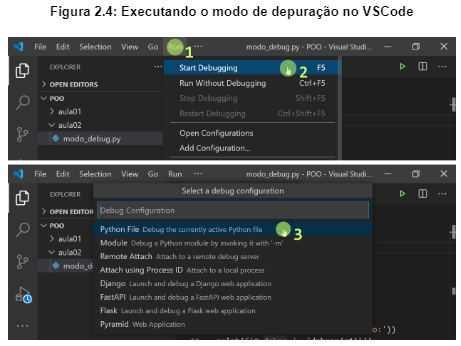
print(f'O dobro é: {dobro}')

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.

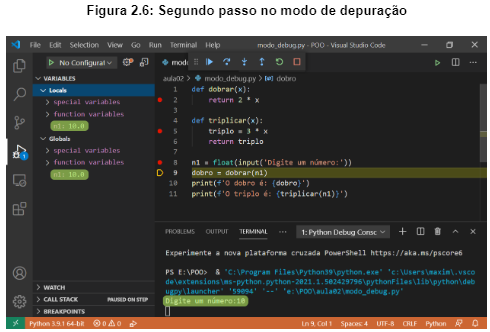


Observe que estamos agora na aba lateral esquerda referente ao modo de depuração, na qual temos algumas janelas, vamos focar agora na de variáveis, portanto podemos minimizar as outras clicando nos símbolos de seta ao lado dos nomes de cada uma. Além disso, surgiu no topo da janela um menu extra com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

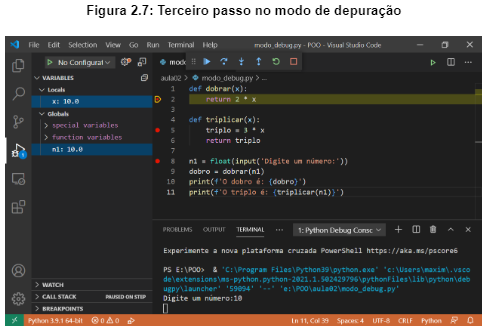
Nos botões que surgiram temos as seguintes ações:

* 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

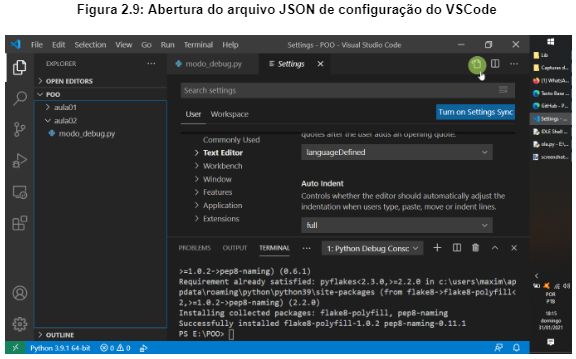
Após a instalação finalizar, digite:

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,

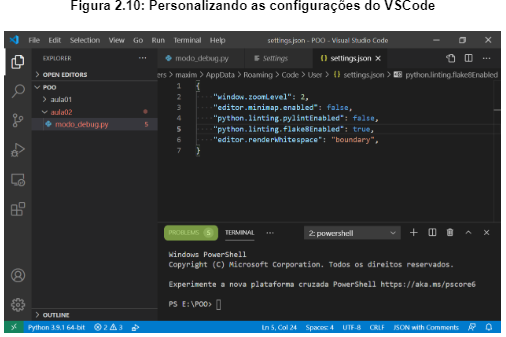
"python.linting.flake8Enabled": true,

"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.

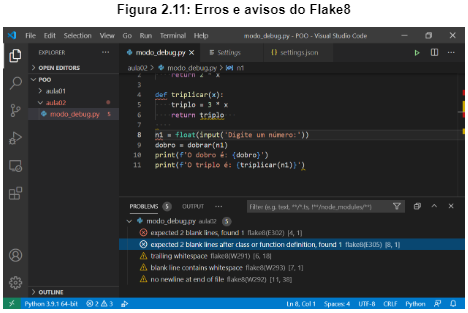
O resultado final pode ser visto na Figura 2.10. Observe que no arquivo do exemplo há uma configuração a mais para definir o zoom da IDE (que pode também ser alterado com os atalhos Ctrl + ‘+’ e Ctrl + ‘-’) e outra para esconder o mini-mapa que aparece por padrão no canto direito da tela. Estas duas 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.

# Paradigmas de Programação e os Pilares de POO

A programação orientada a objetos, que chamaremos de POO a partir de agora, é um paradigma de programação no qual o mundo real é modelado com base no conceito de objetos. Esse conceito aproxima a programação do mundo real, onde enxergamos as coisas à nossa volta de fato como objetos (televisão, mesa, carro, etc.) e isso facilita muito a resolução de diversos tipos de problemas.

Cada objeto possui características, uma TV por exemplo possui resolução e tamanho da tela, voltagem, cor, número de portas HDMI, e assim por diante. Em POO dizemos que estes são os atributos do objeto.

Os objetos possuem também ações, no exemplo da TV, podemos ligar, desligar, mudar de canal, ajustar o volume, alterar a entrada de vídeo, configurar o modo de exibição de cores, entre outros. Em POO dizemos que estes são os métodos do objeto.

Sendo assim, um programa em POO consiste na criação de diferentes objetos que irão interagir entre si para representar a situação que pretendemos modelar. Podemos dizer ainda que um objeto possui consciência de si mesmo e pode manipular seus próprios dados. A TV do exemplo é capaz de alterar o valor do seu próprio volume, usamos aqui um dos métodos (ações) da TV para alterar um de seus atributos (características).

Existem basicamente duas abordagens para POO, baseada em classes e em protótipos. Atualmente a maioria das linguagens de programação orientadas a objetos é baseada em classes, termo que iremos falar bastante de agora até o final do curso. Uma classe é a abstração de um objeto, na qual definimos quais serão os atributos e métodos que os objetos de um mesmo tipo devem possuir. Podemos pensar na classe como um molde ou forma, que podemos usar para criar objetos.

Os conceitos aplicados atualmente em POO surgiram há muito tempo, com os termos “orientado” e “objetos” sendo usados neste contexto pela primeira vez no MIT, no final da década de 1960 (McCARTHY, 1960), em referência a elementos da linguagem LISP. Ao longo da década de 60, diversos estudos contribuíram para o desenvolvimento inicial desse paradigma, resultando no lançamento das primeiras linguagens orientadas a objetos, SIMULA e SMALLTALK, ainda no final da década.

Desde então, diversas novas linguagens foram criadas e ficaram conhecidas mundialmente, em especial a partir da década de 1990, impulsionadas também pela popularização das interfaces gráficas, que em geral se baseiam nas técnicas de POO. Dentre tais linguagens podemos citar Objective-C, C++, Java, C#, Delphi e Python.

Podemos dizer que praticamente todas as linguagens de programação se baseiam no conhecimento e nas experiências com uma ou mais linguagens anteriores, afinal, é assim que toda a ciência evolui, e com a ciência da computação não seria diferente. Portanto é muito comum observarmos semelhanças entre muitas dessas linguagens, que podem ser vistas no tratamento e manipulação dos dados em memória, no funcionamento do compilador ou interpretador[[6]](#footnote-5) da linguagem, nos recursos disponíveis para o programador, ou ainda na escolha das regras de sintaxe.

## Paradigmas de programação

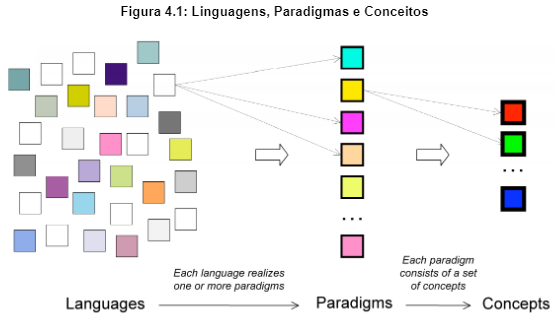
Um paradigma é uma forma de ver e interpretar a realidade, de acordo com o dicionário Michaelis (2021), um paradigma é “algo que serve de exemplo ou modelo; [um] padrão”. Podemos dizer então que um paradigma de programação é um modelo que usamos para representar a realidade em nossos programas; é um conceito abstrato que nos diz como enxergar o mundo e suas relações e como traduzir isso para a programação. Podemos ainda pensar em um paradigma de programação como um estilo de programação.

Já uma linguagem de programação é algo concreto, que define um conjunto de regras de sintaxe com as quais podemos nos comunicar com o computador. Tais regras podem permitir o uso de um ou mais paradigmas, portanto uma linguagem não está limitada a um dado paradigma, e podemos classificá-las de acordo com quais paradigmas aceitam.

Podemos agrupá-los em dois grupos principais (COENEN, 1999):

1. Imperativos: o programa é constituído de uma sequência de instruções (ordens ou comandos) que dizem ao computador exatamente como manipular os dados. Fazem parte deste grupo os paradigmas procedural e orientado a objetos.
2. Declarativos: ao contrário do imperativo, o programa é composto de um conjunto de definições ou equações que descrevem o “que” o programa deve fazer, mas não o “como” fazer, que é delegado a implementação da linguagem. Fazem parte deste grupo os paradigmas funcional e lógico, entre outros.

É importante ressaltar que, assim como diferentes linguagens podem trabalhar com o mesmo paradigma, diferentes paradigmas podem compartilhar conceitos. A Figura 4.1 ilustra a relação entre estes três elementos: linguagens, paradigmas e conceitos.



Agora você pode estar se perguntando qual o melhor paradigma de programação, e a resposta mais uma vez depende do problema que você estiver resolvendo. A melhor forma de se programar é multiparadigma, pois diferentes problemas de programação precisam de diferentes conceitos para serem resolvidos de maneira elegante, e uma linguagem que suporte diferentes paradigmas dá ao programador a liberdade de aplicar o que for mais adequado à situação (ROY, 2009).

Por exemplo, a orientação a objetos é boa para problemas com uma grande quantidade de dados relacionais agrupados em uma estrutura hierárquica; para um problema com estruturas simbólicas complexas, o paradigma lógico é o mais adequado; já problemas com forte teor matemático, como análise de risco financeiro por exemplo, podem se beneficiar do uso do paradigma funcional.

### Paradigmas Imperativos

Nos paradigmas imperativos, há um estado implícito que pode ser alterado através de comandos (HUDAK, 1989). Um programa escrito sob este modelo deve definir exatamente o que o computador deve fazer e em qual ordem, isto é, garantindo que o estado dos dados seja alterado de maneira determinística. Nós definimos os comandos que irão manipular os dados e o fluxo de execução destes comandos, podemos então, com base no estado atual, determinar o estado seguinte a partir dos comandos dados.

Em LP estudamos o paradigma procedural, no qual usamos estruturas de controle de fluxo para definir a ordem de execução dos comandos que queremos executar. Podemos ainda abstrair um conjunto de comandos ao agrupá-los em um procedimento (função), que pode então ser chamado como se fosse um comando único.

Na programação orientada a objetos, continuaremos a usar as mesmas estruturas de controle de fluxo para definir o que o computador deve fazer e em qual ordem. Então o que muda? Muda a forma como agrupamos estas instruções e os dados que elas podem manipular.

No paradigma procedural, aprendemos sobre o escopo global e local, nos quais podemos criar variáveis para salvar e manipular nossos dados. Mas agora, ao invés de agrupar as instruções em um procedimento, vamos agrupá-las em um objeto, que poderá conter dados internos e instruções sobre como manipular tais dados. Um programa então irá consistir na criação de diversos objetos que irão interagir entre si a partir da troca de mensagens.

### Paradigmas Declarativos

Nos paradigmas declarativos, não há um estado implícito dos dados, e a programação é feita com base na avaliação de expressões ou termos (HUDAK, 1989). Os principais paradigmas declarativos são o lógico e o funcional.

O paradigma lógico é baseado na lógica formal, na qual definimos um conjunto de sentenças lógicas que expressam os fatos e regras pertinentes ao problema que queremos resolver, e a solução é deduzida a partir da aplicação de tais regras e fatos.

Já no paradigma funcional, as funções atuam como funções matemáticas puras, isto é, não alteram o estado do programa. Em outras palavras, não produzem nenhum efeito colateral, retornando sempre o mesmo resultado se chamadas com os mesmos argumentos. Além disso, aspectos importantes do paradigma funcional são o tratamento de funções como objetos de primeira classe e a existência de funções de ordem superior. Ou seja, funções podem ser atribuídas a variáveis e passadas como argumento para outras funções, que podem também retornar uma nova função como valor de resposta.

Em um programa puramente funcional o foco está em declarar o que cada função deve fazer e através da composição destas funções, chegar a solução de problemas. Em muitas linguagens de programação podemos aplicar conceitos da programação funcional à programação imperativa (procedural ou orientada a objetos).

## Pilares de POO

Vamos trabalhar com a programação orientada a objetos baseada no conceito de classes, que são os blocos essenciais para a construção de um programa em POO. Há quatro ideias ou conceitos fundamentais da programação orientada a objetos:

* Abstração;
* Encapsulamento;
* Herança; e
* Polimorfismo.

### Abstração

O primeiro dos conceitos é provavelmente o mais importante de todos para a programação de maneira geral, pois ao representar algo do mundo real em um programa de computador, precisamos decidir o que iremos representar e o que iremos ignorar.

No contexto de POO, esse conceito ganha também o significado de generalização. Pense na TV que falamos na introdução, em qual TV você pensou? Uma TV de tubo? de LED? Grande? Pequena? FullHD? Smart? A TV que você pensou é um exemplar específico de uma TV, mas ela possui características em comum com todas as outras TVs, pois o conceito de TV é o mesmo e isso é o que chamamos de abstração.

Estamos interessados em uma generalização de TV, ou seja, a abstração em POO significa escolher as qualidades em comum de todas as TV que sejam relevantes para a situação que estamos querendo modelar. Pense que estamos escrevendo um programa para uma loja de eletrodomésticos e precisamos criar os anúncios das TVs que estarão disponíveis no site da loja. Para isso podemos criar uma classe TV que irá agrupar tudo que seja relevante nesta situação e a partir dessa classe, criar cada um dos objetos que irão representar os modelos específicos das TVs em estoque no momento.

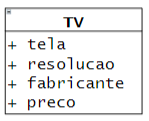


Figura 4.2: Diagrama de classe simplificado

Na Figura 4.2 vemos uma versão simplificada do diagrama de classe da UML[[7]](#footnote-6), com o nome da classe e alguns dos possíveis atributos (características) que podemos escolher para uma TV. O sinal de + na frente de um atributo indica que ele é público, voltaremos neste assunto mais pra frente no curso.

E a partir dessa classe podemos criar quantos objetos de TV forem necessários. Cada objeto será único e terá seus próprios valores para cada atributo, como mostrado na Figura 4.3.

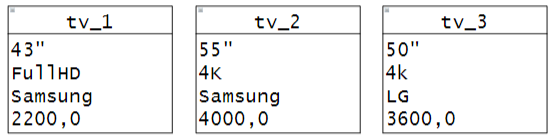


Figura 4.3: Diagrama de objetos simplificado

### Encapsulamento

O termo encapsulamento se refere a colocar algo no interior de um cápsula, em geral com os objetivos de manter junto e proteger. Em POO, isso pode fazer referência a própria classe em si, na qual agrupamos as propriedades e comportamentos que abstraímos do objeto real, e estamos modelando em uma unidade, um compartimento.

Mas mais do que isso, esse termo é usado para indicar a ocultação de informações. Podemos esconder partes da nossa classe do restante da aplicação, e isso serve a dois propósitos: proteger os dados do objeto de serem alterados por outra parte da aplicação que não seja o próprio objeto e esconder do restante da aplicação partes internas do funcionamento do objeto que podem sofrer alterações no futuro. Dessa forma, caso algum detalhe interno da implementação seja alterado, isso não irá afetar a forma como os objetos são usados, contribuindo para uma boa manutenção do código.

Vamos pensar no exemplo de uma conta bancária, pense em quais atributos e quais métodos podemos abstrair para uma classe que represente as contas bancárias de um determinado banco.

A Figura 4.4 mostra uma possível modelagem para esta classe. Observe que não seria interessante ter o saldo da conta como um atributo público, que pudesse ser alterado por qualquer parte da aplicação, então para evitar isso, podemos esconder esse atributo internamente e deixar que a interação com o restante da aplicação se dê por meio de métodos públicos.

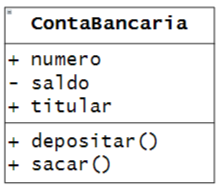
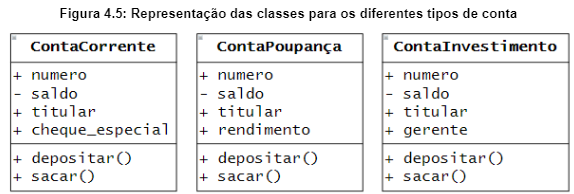


Figura 4.4: Exemplo de representação da classe para uma conta bancária

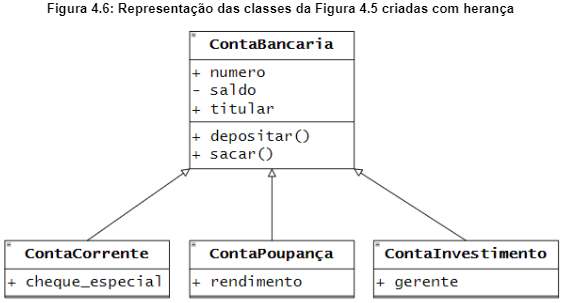
No diagrama de classes, o sinal de menos indica que o atributo ou método é privado, ou seja, só está acessível ao próprio objeto, e não pode ser alterado por outra parte da aplicação. Neste exemplo as únicas formas de alterar o saldo da conta são através dos métodos sacar e depositar, e isso aumenta a segurança do funcionamento do nosso programa, pois podemos implementar restrições e verificações no interior desses métodos antes de fazer a alteração do valor do atributo.

### Herança

A ideia de herança é uma das várias formas que podemos reutilizar código em POO. Digamos que no exemplo do banco precisamos agora criar outros tipos de contas, como uma conta corrente, poupança, investimento, etc. Poderíamos criar uma classe para cada tipo de conta, como mostra a Figura 4.5.



Observe no entanto que há várias características e comportamentos em comum entre estas classes, então a herança é uma forma de reaproveitarmos o código, criando uma classe nova a partir de outra classe já existente. Dizemos que a classe original é a superclasse ou classe mãe, e as classes derivadas são subclasses ou classes filhas. Veja a representação dessa relação na Figura 4.6.



Com a herança, precisamos apenas adicionar os atributos ou métodos específicos de cada subclasse, todos os métodos e atributos em comum são herdados da classe mãe.

### Polimorfismo

O polimorfismo é uma característica que aparece também em outros paradigmas de programação e pode ser dividido em dois tipos diferentes, sobrecarga e sobrescrita. Mas antes de falar sobre os dois tipos, vamos entender de onde vem esta palavra. Polimorfismo é uma palavra de origem grega:

* *Poly*: muito, numeroso, frequente;
* *Morph*: forma; e
* *Ismos*: processo, estado;

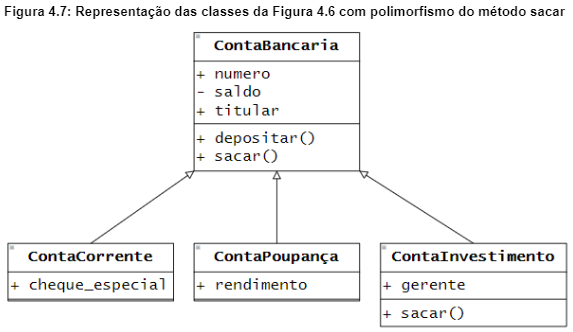
Ou seja, polimorfismo é uma propriedade daquilo que pode apresentar muitas formas ou aspectos, e em programação se refere a funções ou métodos com o mesmo nome, mas com comportamentos diferentes. Vejamos um exemplo que está presente em praticamente todas as linguagens de programação para entender melhor.

Em LP usamos o operador + para somar dois números e também para concatenar duas strings, duas listas ou duas tuplas. Em cada uma destas situações, a operação realizada é diferente, mas o operador é o mesmo. Dizemos então que este operador está sobrecarregado, isto é, carregado com mais de uma implementação, com mais de uma forma, daí o nome polimorfismo. A escolha de qual operação deve ser realizada é feita automaticamente em função dos operandos utilizados.

Agora vamos ver como isso se aplica às nossas classes do exemplo da conta bancária. No caso da conta de investimento, é comum haver um custo para sacar um valor investido, que em geral diminui em função do tempo de duração do investimento, além de eventuais cálculos de impostos ou taxas que podem ser cobrados. Com isso, não é interessante ter a conta investimento herdando o método sacar da classe mãe, pois ele estará incorreto.

No entanto, não adianta corrigir este método na classe mãe, pois isso também estaria incorreto, já que não é uma boa prática colocar a responsabilidade de um cálculo específico em uma classe genérica. Teríamos um método que só seria usado por uma das classes filha, mas todas as outras também o herdariam.

Uma possível solução seria criar esta classe do zero, sem a herança, mas com isso perderíamos a reutilização dos demais métodos e atributos que nos eram úteis, então a solução é mais simples do que pode parecer, basta ignorar a implementação da classe mãe e sobrescrever este método na classe filha, com a implementação das regras específicas para aquela classe. Veja na Figura 4.7 como ficaria tal representação.



Agora, imagine que tenhamos 200 contas (objetos) diferentes em nosso programa, podemos seguramente chamar o método sacar() em qualquer uma delas e sabemos que todos os cálculos e verificações pertinentes serão realizados corretamente, seja ela uma conta corrente, poupança ou de investimento.

# Criação de classes em Python e Encapsulamento

Python é uma linguagem multiparadigma, que permite a programação simultânea em diferentes paradigmas de programação, em especial procedural, funcional e orientado a objetos. Este é um dos motivos que contribuiu para a popularidade atual do Python nos mais diversos contextos.

O Python trata todas as entidades da linguagem como objetos, o que faz dele uma linguagem naturalmente apta ao paradigma de POO. No entanto, devido às características da linguagem, a aplicação de alguns conceitos difere do que vemos em linguagens exclusivamente orientadas a objetos, como Java, C# e PHP, por exemplo.

## PEP-8 aplicada às classes

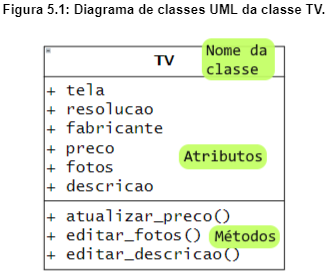
Antes de ver propriamente como implementar as classes em Python, vamos ver as regras para nomeá-las, pois isso irá nos ajudar a identificar e diferenciar classes de instâncias ao lermos um código que segue as recomendações da PEP8.

A convenção para nomes de classes é MinhaClasse, na qual todas as palavras possuem a primeira letra maiúscula e são unidas diretamente, sem espaço ou sublinhado. Para os atributos e métodos das classes, seguimos a mesma convenção das variáveis e funções, todas as letras minúsculas, unidas por sublinhado (PSF, 2021a).

Uma exceção a essa regra são as classes integradas do Python, como int, float, etc. que possuem nomes curtos e todos minúsculos. O raciocínio por trás disso é que tais classes são usadas pelo programador principalmente como funções para converter dados entre tipos diferentes, e seguem a convenção para nomear funções.

## POO em Python

Vamos começar relembrando a definição que vimos para a classe TV (Figura 5.1), incluindo alguns atributos novos e também alguns métodos. Lembre-se que estamos modelando uma TV que irá representar um produto a ser vendido, então precisamos pensar nos métodos e atributos que sejam relevantes ao contexto. Por exemplo, um método aumentar\_volume faz sentido na programação do sistema operacional da TV, mas não é necessário aqui. Isso faz parte do processo de abstração do objeto real para sua representação no código.



A Figura 5.1 mostra o diagrama de classe UML para a classe TV. É comum que este diagrama indique os tipos de cada atributo, os parâmetros de cada método e seus tipos e muitas vezes o tipo do retorno de cada método. Por hora vamos usar esta versão simplificada para focar na tradução desse diagrama para o Python.

### Implementando classes em Python

Em Python, a criação de uma classe é feita com o uso da palavra chave class, seguida do nome da classe. Todas as classes em Python herdam de uma classe especial chamada object, que garante que novas classes terão os métodos comumente esperados de uma classe em Python. Isso facilita muito a programação e a partir da versão 3 do Python, foi introduzida uma nova sintaxe para a criação de classes, na qual não é preciso mais indicar explicitamente tal herança.

code:

class NomeDaClasse:

<bloco de código da classe>

Vamos fazer alguns exemplos na Shell do Python para entender o funcionamento deste comando. Abra a IDLE ou digite o comando do Python referente ao seu sistema operacional (python, py ou python3) no terminal do VSCode para abrir uma Shell.

code:

>>> class TV:

... pass

O comando acima cria uma classe cujo nome é TV, mas por hora não definimos nenhum atributo ou método ainda. Podemos verificar isso acessando o nome da classe.

code:

>>> TV

<class '\_\_main\_\_.TV'>

### Instanciando objetos a partir de uma classe em Python

Já para criar um objeto a partir desta classe devemos chamá-la, de maneira análoga a forma como chamamos funções, com parênteses.

code:

>>> TV()

<\_\_main\_\_.TV object at 0x7f2c6f106310>

Ao chamarmos a classe, ela nos retorna um objeto de TV, dizemos que esse objeto é do tipo TV, pois uma classe define um novo tipo de dado. Em Python, podemos adicionar atributos diretamente ao objeto, então se quisermos criar uma nova TV podemos fazer:

code:

>>> tv\_1 = TV()

>>> tv\_1.tela = 43

>>> tv\_1.resolucao = 'FullHD'

>>> tv\_1.fabricante = 'Samsung'

>>> tv\_1.preco = 2400.0

>>> tv\_1.fotos = []

>>> tv\_1.descricao = 'TV FullHD 43" - Samsung'

Podemos agora usar a função integrada vars, que devolve um dicionário com os atributos de um objeto em Python, para conferir que nosso objeto de fato possui os atributos que criamos nele.

code:

>>> vars(tv\_1)

{'tela': 43, 'resolucao': 'FullHD', 'fabricante': 'Samsung', 'preco': 2400.0, 'fotos': [], 'descricao': 'TV LED FullHD de 43" - Samsung'}

Agora se quisermos criar um segundo objeto, podemos repetir o processo. Vamos criar outro objeto e verificar os atributos que ele possui inicialmente.

code:

>>> tv\_2 = TV()

>>> vars(tv\_2)

{}

Como não colocamos nada em nossa classe, o novo objeto começa vazio, então criar novos objetos dessa forma não é algo prático na maioria das situações.

### Como inicializar um objeto em Python

Para resolver esse problema, o Python possui um método especial que podemos definir para que o objeto seja inicializado. Esse método deve obrigatoriamente ser chamado \_\_init\_\_ e será executado uma vez no momento da criação de cada objeto. Os métodos especiais em Python são métodos que começam e terminam com dois sublinhados, também chamados de dunder[[8]](#footnote-7) methods. A lista completa desses métodos pode ser vista na documentação (PSF, 2021b).

Vamos agora pro editor do VSCode. Crie um arquivo chamado “classes.py” na pasta “aula05” e digite o seguinte código nele:

code:

class TV:

def \_\_init\_\_(self):

self.tela = 43

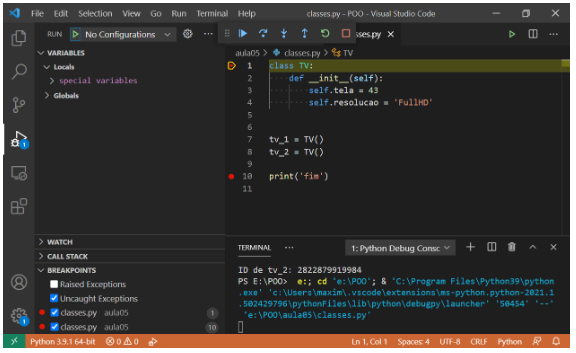
self.resolucao = 'FullHD'

tv\_1 = TV()

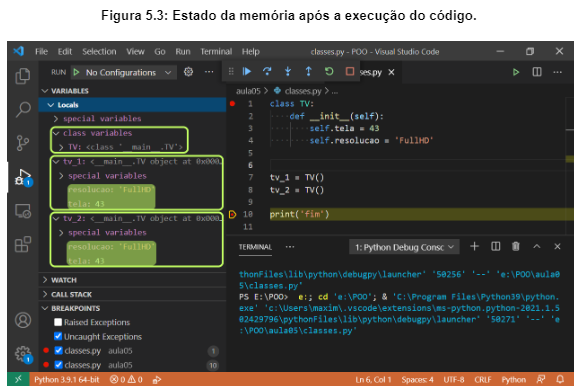
tv\_2 = TV()

print('fim')

Em seguida salve o arquivo, adicione pontos de parada na primeira e na última linha e aperte F5 para executá-lo no modo de depuração. O resultado é mostrado na Figura 5.2. Figura 5.2: Início da execução no modo de depuração.



Agora avance na execução passo a passo e observe o que ocorre na memória com a execução de cada instrução deste código. A Figura 5.3 mostra o estado da memória após a criação dos dois objetos de TV.



Podemos observar que ambos objetos foram criados com os mesmos valores para os atributos. Isso acontece pois deixamos os valores fixos no código, mas, antes de ver como podemos alterar esse comportamento, precisamos entender o que é esse parâmetro self que colocamos em nossos métodos.

### Entendendo o parâmetro self

Quando comparamos os paradigmas procedural com orientado a objetos, vimos que em POO, cada objeto tem consciência sobre si mesmo. Isso se dá através desse parâmetro self, que pode ser traduzido para “próprio” ou “si mesmo”. Por padrão, o Python injeta uma referência para o objeto em questão como primeiro argumento de todos os métodos em uma classe.

Para confirmar isso, podemos usar a função id, que nos retorna o número de identificação do objeto, que é único para cada objeto criado na memória. No exemplo inicial na Shell, podemos ver que os dois objetos criados possuem identidades diferentes, pois são objetos diferentes na memória.

code:

>>> id(tv\_1)

2280534403472

>>> id(tv\_2)

2280534403440

Vamos agora adicionar um atributo para guardar a identidade do objeto referenciado pelo self, e depois compará-la com a identidade de cada objeto de TV instanciado. Altere o código do arquivo “classes.py” para:

class TV:

code

def \_\_init\_\_(self):

self.tela = 43

self.resolucao = 'FullHD'

self.id = id(self)

tv\_1 = TV()

tv\_2 = TV()

print(f'ID de tv\_1: {id(tv\_1)}')

print(f'ID de tv\_2: {id(tv\_2)}')

Veja na Figura abaixo o resultado da execução desse código no modo de depuração.



Como podemos ver, efetivamente o parâmetro self do método \_\_init\_\_ recebe uma cópia da referência para o objeto a partir do qual é chamado. Isto é válido tanto para os métodos com nomes especiais quanto para os demais métodos que criamos, com exceção dos métodos de classe e métodos estáticos, que veremos mais adiante no curso.

### Personalizando a inicialização dos objetos em Python e incluindo novos métodos

Vamos então fazer a implementação completa da classe TV que vimos no diagrama da Figura 5.1. Como você talvez já tenha deduzido, para adicionar um novo método à classe basta definir uma função dentro do bloco de código da classe, que deverá obrigatoriamente ter o self como primeiro parâmetro.

Podemos ainda receber quantos argumentos forem necessários para o método adicionando mais parâmetros após o self, então para inicializar o objeto basta simplesmente adicionar mais parâmetros ao método especial \_\_init\_\_.

Agora, no momento de instanciar um objeto, podemos passar os valores que queremos atribuir para aquele objeto, e o Python ao criar o objeto, automaticamente executa o método \_\_init\_\_, repassando-lhe os valores dos argumentos que passamos à classe. Veja o código a seguir:

code:

class TV:

def \_\_init\_\_(self, tela, resolucao, fabricante, preco):

self.tela = tela

self.resolucao = resolucao

self.fabricante = fabricante

self.preco = preco

self.fotos = []

self.descricao = f'TV {resolucao} {tela}" - {fabricante}'

def atualizar\_preco(self, novo\_preco):

self.preco = novo\_preco

def editar\_fotos(self):

pass

def editar\_descricao(self):

pass

tv\_1 = TV(43, 'FullHD', 'Samsung', 2400)

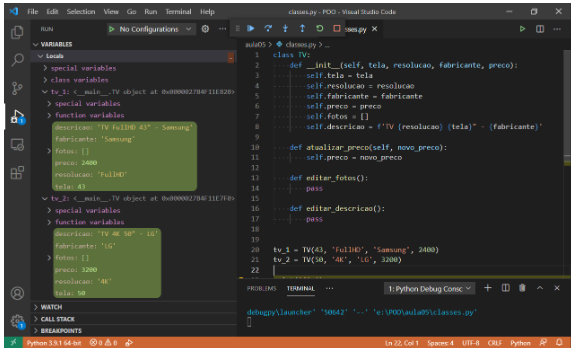
tv\_2 = TV(50, '4K', 'LG', 3200)

print('fim')

Aqui omitimos o código dos métodos de editar fotos e descrição, pois o objetivo é ver como funciona a criação dos métodos em uma classe. A palavra chave pass indica ao Python que nada deve ser feito ali. Podemos usá-la quando queremos criar primeiro a estrutura de uma classe, método ou função, e testar parcialmente seu funcionamento, antes de implementá-la completamente, evitando que haja um erro de sintaxe ao deixar um bloco de código vazio.

Caso seja passado um número diferente de argumentos para a classe TV no momento de criação de um novo objeto, isso irá gerar um erro de execução, pois em Python não podemos chamar uma função ou método com o número incorreto de parâmetros obrigatórios. Portanto, podemos agora instanciar quantas TVs forem necessárias de maneira prática e cada uma já irá possuir os valores adequados aos seus atributos.

A Figura 5.5 abaixo mostra o resultado da execução deste código no modo de depuração.



Vamos ver agora como podemos usar um dos métodos de um objeto. Nesse nosso exemplo, o único método funcional é o de atualizar o preço, os outros dois foram colocados apenas como exemplo e não fazem nada. Vamos então alterar o preço da primeira TV aplicando um desconto de 10%.

Com a implementação atual, precisamos informar diretamente o novo preço final, que com o desconto será de R$ 2160,00. Para isso, adicione as seguintes linhas ao final do código:

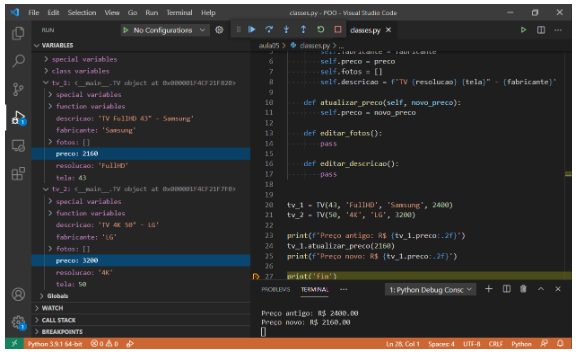
code:

print(f'Preço antigo: R$ {tv\_1.preco:.2f}')

tv\_1.atualizar\_preco(2160)

print(f'Preço novo: R$ {tv\_1.preco:.2f}')

Na Figura 5.6 abaixo podemos ver que o preço do primeiro objeto foi alterado, mas não o do segundo, pois cada objeto só é capaz de alterar seus próprios atributos.



### Encapsulamento

A ideia principal do encapsulamento é ocultar atributos e métodos que não devem ser acessíveis a partes externas da aplicação. Dentre as razões para fazermos isso, podemos citar:

* Possibilita a inclusão de regras de negócio para validar ou preparar os dados antes de fazermos a alteração do valor de um atributo;
* Aumenta a segurança do código contra bugs ou erros inesperados, pois limita quem pode alterar os atributos de um objeto;
* Facilita a manutenção do código, pois caso seja preciso alterar um método interno, só precisamos alterar o código da própria classe e o restante da aplicação ou programa não é afetado por esta alteração.

Quando falamos de encapsulamento, precisamos entender três conceitos importantes de POO, e como esses conceitos são aplicados no Python. Em POO podemos ter atributos e métodos de um objeto classificados de acordo com sua visibilidade e acessibilidade em:

* Públicos: são aqueles que podem ser acessados diretamente por qualquer parte da aplicação com acesso ao objeto em si. No diagrama de classes da UML, são marcados precedidos do sinal +.
* Protegidos: são aqueles que podem ser acessados apenas pelo próprio objeto e seus descendentes, isto é, objetos de classes que tenham estendido a classe original, através de herança. São indicados na UML pelo sinal de # e falaremos mais a respeito deles no capítulo sobre herança.
* Privados: são aqueles que só podem ser acessados pelo próprio objeto e por mais nenhuma outra parte da aplicação ou programa. Indicados na UML por -.

Cada linguagem implementa estes conceitos de uma maneira diferente, havendo pontos positivos e negativos em todas elas. No caso do Python, tais conceitos não são impostos pela linguagem, isto é, todos os atributos e métodos de um objeto estão sempre visíveis e acessíveis. Mas isso não significa que não seja possível utilizar tais conceitos a nosso favor em Python.

A PEP8 não utiliza a nomenclatura padrão vista acima, classificando os atributos e métodos apenas em públicos e não-públicos, mas isso não significa que os conceitos não possam ser aplicados ao design das nossas classes. Portanto a recomendação para nomear os atributos e métodos é:

* Públicos: seguem a mesma regra para nomenclatura de variáveis e funções letras\_minusculas\_separadas\_por\_sublinhado.
* Não-públicos, quando tratados como:
  + Protegidos: devem ser precedidos por um sublinhado \_atributo\_protegido.
  + Privados: devem ser precedidos por dois sublinhados \_\_atributo\_privado.

Lembrando que os nomes especiais, como o \_\_init\_\_, são precedidos e sucedidos por dois sublinhados, e não podem ser inventados, isto é, só podemos usar os nomes especiais definidos na documentação do Python.

Quando criamos um atributo que queremos tratar como privado, precedendo-o com dois sublinhados, o Python realiza o que chamamos de “desfiguração de nomes”, alterando o identificador (nome do atributo ou método) para incluir o nome da classe: um atributo \_\_atributo\_privado que seja definido em qualquer parte da classe MinhaClasse terá seu identificador transformado pelo Python em \_MinhaClasse\_\_atributo\_privado (PSF, 2021c).

Sendo assim, quando estamos escrevendo uma classe em Python, sabemos que os clientes ou consumidores desta classe, isto é, qualquer parte da aplicação ou programa que faça uso da nossa classe, deverão utilizar apenas os métodos e atributos que sejam públicos. Seguindo esta recomendação, sabemos que no futuro, podemos alterar os nomes e a implementação de qualquer método ou atributo não-público sem nos preocupar em quebrar o código que faz uso atualmente da nossa classe.

### Trabalhando com atributos não-públicos em Python

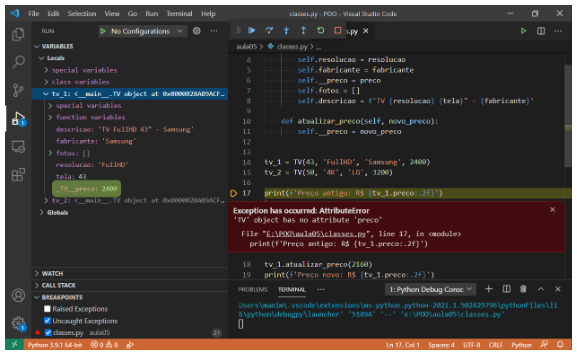
Vamos ver como isso funciona na prática. No exemplo da TV, o preço é um atributo público, assim como todo o resto, então qualquer parte do nosso programa pode acessá-lo e alterá-lo com facilidade, basta atribuir um novo valor ao atributo:

code:

tv\_1.preco = 300

Que o atributo preco terá seu valor alterado no objeto. Portanto, podemos transformá-lo em um atributo não-público, de modo que o único responsável por alterá-lo seja o próprio objeto. O mais comum é usar apenas um sublinhado para atributos não-públicos, deixando o uso de dois sublinhados para casos específicos. Falaremos mais a respeito ao estudar herança, por hora, vamos fazer um exemplo com o uso de dois sublinhados para visualizarmos o funcionamento da “desfiguração de nomes” do Python.

Para isso altere o nome do atributo de self.preco para self.\_\_preco, em todas as suas ocorrências, e vamos executar novamente nosso código. Veja a Figura 5.7 abaixo.



Como podemos ver, quebramos a aplicação pois agora não podemos mais acessar o atributo tv\_1.preco, ele não existe! O nome deste atributo agora é \_TV\_\_preco, e poderíamos fazer tv\_1.\_TV\_\_preco que isso funcionaria. Mas ao fazer isso estamos violando a privacidade deste atributo, pois ele não é mais público e só deve ser acessado diretamente no interior do objeto.

Então, ao criar um atributo não-público, podemos escolher se queremos que ele seja 100% não-público, aberto para leitura ou aberto para leitura e escrita, e fazemos isso através dos métodos conhecidos por getters e setters. Para abrir o atributo para leitura, definimos um método getter, que ficará responsável por recuperar e retornar o valor do atributo; e para abrir o atributo para escrita, definimos um método setter, que ficará responsável por atribuir um novo valor ao atributo. No exemplo que fizemos, o método para atualizar o preço já está fazendo o papel de um setter.

Tradicionalmente em POO, estes métodos devem ser nomeados com o nome do atributo precedido de da palavra get ou set. Portanto, faça uma cópia do arquivo “classes.py” e renomeie-o “classes\_getters\_e\_setters.py”, e vamos alterar neste novo arquivo o nome do método atualizar\_preco para set\_preco e criar outro método chamado get\_preco.

O código dos métodos deverá ficar assim:

code:

class TV:

def \_\_init\_\_(self, tela, resolucao, fabricante, preco):

self.\_\_preco = preco

... # demais atributos não são alterados

def get\_preco(self):

return self.\_\_preco

def set\_preco(self, novo\_preco):

self.\_\_preco = novo\_preco

Agora, não precisamos mais acessar o atributo não-público diretamente, e podemos então alterar as linhas que estão utilizando nossos objetos de TV para usar o getter e o setter:

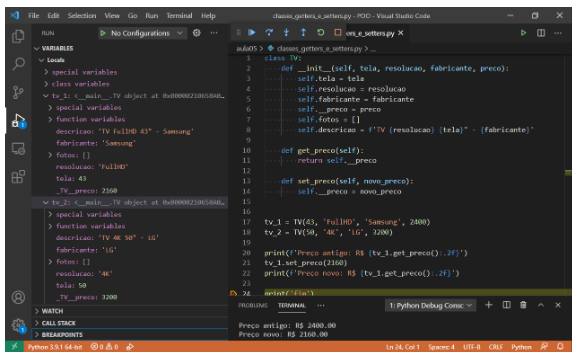
code:

print(f'Preço antigo: R$ {tv\_1.get\_preco():.2f}')

tv\_1.set\_preco(2160)

print(f'Preço novo: R$ {tv\_1.get\_preco():.2f}')

Veja na Figura 5.8 abaixo a execução do arquivo “classes\_getters\_e\_setters.py”.



A principal vantagem de se fazer isso é que agora o acesso, tanto para escrita quanto para leitura, é feito por um método, no qual podemos introduzir qualquer lógica que seja necessária para validar as ações sendo feitas. Por exemplo, em set\_preco, poderíamos antes de alterar o atributo de preço, verificar se o funcionário que está logado no sistema atualmente tem a permissão para fazer isso, ou então pedir que seja entrada uma senha para continuar com a operação. Um exemplo de código ilustrativo de tal verificação é:

from modulo\_validacao import valida\_autorizacao

code:

class TV:

# restante do código sem alterações

def set\_preco(self, novo\_preco):

senha = input('Digite a senha de autorização: ')

if not valida\_autorizacao(senha):

return

self.\_\_preco = novo\_preco

Dessa forma, estamos colocando uma camada extra de proteção na alteração do preço. Agora apenas os funcionários que possuírem uma senha de autorização para editar os preços das TVs poderão fazê-lo. Esse é o conceito de encapsulamento, o acesso a um atributo está encapsulado dentro de métodos que o protegem.

### Utilizando os decoradores @property e @property.setter

Na criação dos getters e setters tradicionais, como vimos acima, precisamos alterar um código que já fazia uso do nosso atributo público preco para uma chamada de método. No entanto, o Python possui uma forma mais natural de criarmos getters e setters, que facilitam o acesso dos atributos pelos clientes da nossa classe ao mesmo tempo que permitem as verificações que fizemos ao criar os métodos tradicionais.

Isto é feito através de um decorador chamado property. Veremos em mais detalhes o que são decoradores mais pra frente no curso, por hora é suficiente entender que são funções especiais que podemos usar para decorar nossos métodos, fornecendo-lhes uma funcionalidade extra. E aplicamos um decorador colocando-o, precedido do símbolo @, na linha imediatamente anterior à definição do método.

Ao decorarmos um método com o decorador property, estamos criando um getter. O Python irá criar um identificador público com o mesmo nome do método, que funcionará como um atributo padrão para o mundo exterior ao objeto, e toda vez que o atributo público for acessado, por baixo dos panos o Python irá chamar o método associado a ele pelo decorador property.

Para criar um setter, devemos decorar o método do setter com o decorador <nome>.setter, onde <nome> deve ser o nome público criado pelo decorador property. Com isso, toda vez que um valor for atribuído ao atributo público criado pelo Python, ele irá repassar esse valor para o método associado ao setter.

Importante: para criar um setter, é obrigatório antes criar uma property. Mas é possível criar apenas a property, sem criar o setter associado a ela.

Vamos ver agora como ficaria o exemplo do preço utilizando estes decoradores. Para isso vamos voltar para o arquivo “classes.py” e alterar os métodos para:

code:

class TV:

def \_\_init\_\_(self, tela, resolucao, fabricante, preco):

self.\_\_preco = preco

... # demais atributos não são alterados

@property

def preco(self):

return self.\_\_preco

@preco.setter

def preco(self, novo\_preco):

self.\_\_preco = novo\_preco

Veja na Figura 5.9 abaixo o resultado da execução deste código no modo de depuração.



Pontos importantes a se observar:

* Na criação da property:
  + Podemos escolher o nome que quisermos para o método, e ele será adotado como o nome do atributo que será exposto publicamente;
  + No interior do método decorado com @property, não podemos jamais acessar o atributo público que ela cria, pois isso irá criar uma recursão infinita, e o programa não irá funcionar. No exemplo acima, no interior do método preco() definido na linha 11, é proibido acessar o atributo self.preco;
  + Esse método pode conter verificações, se necessário, mas o mais comum é apenas retornar o valor do respectivo atributo não-público.
* Na criação do setter, que é opcional, devemos:
  + Sempre usar o mesmo nome criado pela property;
  + Sempre receber um único parâmetro, além do self que o Python automaticamente injeta em todos os métodos. O nome desse parâmetro não importa, basta usar o mesmo nome dentro do método, portanto escolha um nome representativo;
  + Em geral, esse método não possui retorno de valor, e é comum realizarmos verificações antes de alterarmos de fato o valor do respectivo atributo não-público, sendo também comum haver um retorno antecipado (vazio) quando alguma validação falha.
* Este atributo criado pela property será usado pelos clientes da classe (demais módulos da nossa aplicação) como se fosse um atributo padrão, sem que eles tenham conhecimento da implementação por trás. Portanto, não devemos implementar métodos que tenham um alto custo computacional ou que levem muito tempo para serem processados. Se for este o caso, evite usar uma property/setter e crie um método tradicional para alterar o atributo não-público, pois ao utilizar o setter, os clientes da classe estarão esperando interagir com um atributo de dados, cujo acesso é extremamente rápido e de baixo custo computacional.

# Herança e Polimorfismo em Python

Continuando no estudo dos conceitos básicos de POO, vamos aprender agora como herança e o polimorfismo podem ser implementados em Python. Como vimos nos demais capítulos, a herança é uma das formas que nos permite reutilizar código, mas precisamos tomar bastante cuidado com o seu uso.

Se tentarmos resolver todos os problemas de reutilização de código apenas com o uso da herança, podemos criar uma teia hierárquica de relacionamentos entre as classes, que pode levar a dificuldades de manutenção e comportamentos inesperados no código quando precisarmos editar algo em uma das superclasses nesta rede.

Outra forma de reutilização de código é a composição, que nos permite compor diversas classes para podermos separar as responsabilidades de cada uma e melhorar a manutenibilidade do código.

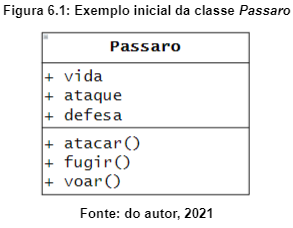
## Herança

Em POO, um dos objetivos é aproximar a modelagem de um programa do mundo real, para facilitar a escrita e leitura de código. Portanto, o conceito de herança em POO é muito parecido com a taxonomia dos seres vivos, isto é, a forma que os classificamos em grupos de acordo com suas características.

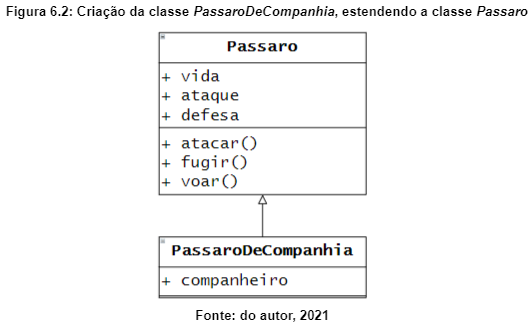
Colocamos seres com características semelhantes em uma espécie, em seguida agrupamos espécies com características semelhantes em um gênero, gêneros semelhantes em uma família, famílias semelhantes em uma ordem, e assim por diante passando por classe e filo, até chegar no reino (animal, vegetal, etc.).

Vamos pensar no reino animal, que inclui todos os animais que conhecemos no mundo, é um conceito muito genérico, portanto ele define apenas os traços comuns a todos os animais, que coloca no mesmo grupo esponjas do mar, jacarés, grilos, pássaros e nós humanos. Em seguida temos diversas subdivisões de acordo com outras características destes seres, até chegar na espécie, cujos indivíduos possuem o maior grau de semelhança entre si.

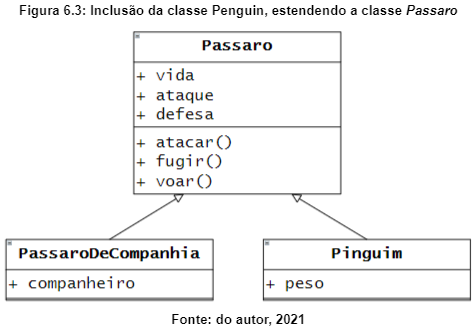
Com esse exemplo em mente, vamos modelar um exemplo comumente utilizado para ilustrar a herança e um dos problemas que podem surgir se a implementamos sem a devida consideração. Digamos que estejamos programando um jogo de mundo aberto, que irá possuir diversos animais, e nossa tarefa é programar as classes para os pássaros do jogo. Criamos então a classe da Figura 6.1 para agrupar todos os pássaros do jogo.



Agora imagine que no design do jogo, alguns dos personagens principais poderão em determinado ponto ganhar um pássaro de companhia, que irá ajudá-los nas missões. Podemos então estender a classe pássaro e criar uma nova classe com um atributo para guardar o personagem associado ao pássaro. Veja a Figura 6.2.



Inicialmente essa classe funciona perfeitamente, mas imagine agora que em uma vila do jogo, há um mercador de animais exóticos que vende pinguins, precisamos de uma classe para eles também, então podemos criá-la herdando de Passaro e teremos a situação da Figura 6.3.



Antes de seguir na leitura, veja se consegue encontrar um problema que introduzimos na modelagem das classes quando fizemos a classe Pinguim herdar de Passaro.

Exatamente, pinguins não voam, mas a classe Passaro implementa um método voar() que será herdado por Pinguim, pois um pinguim é um pássaro. O que deve acontecer quando esse método for chamado em um objeto do tipo Pinguim? Podemos sobrescrever o método e fazer com que um erro seja levantado, indicando que aquela ação não pode ser realizada, mas essa não é uma boa abordagem.

É esperado que um objeto de uma classe mãe possa ser substituído por um objeto de qualquer uma das classes filhas, sem que isso altere a expressão do código, ou seja, um método que funciona em uma classe não pode deixar de existir ou levantar um erro em uma classe derivada (esse é um dos princípios do SOLID, que veremos em outro capítulo).

Você pode estar se perguntando agora “mas e o polimorfismo, não é justamente a alteração de um método nas classes filhas, isso não contradiz o parágrafo anterior?”, e essa é uma pergunta válida. Como veremos ainda neste capítulo, a forma como o método funciona pode ser diferente no polimorfismo, mas o resultado final não. No caso do método voar, poderíamos ter um pássaro mecânico que implementa tal método com o uso de motores e hélices, ao invés de bater as asas, então a forma (implementação) é diferente, mas o resultado é o mesmo: ao chamar o método voar(), o pássaro voa.

Agora, como podemos resolver o problema mencionado acima em nossas classes? Nessa situação, a melhor coisa é repensar a hierarquia que definimos de maneira a acomodar tais mudanças, e por isso este é um processo extremamente importante de ser feito no começo do projeto para evitar mudanças drásticas no futuro, que impactam toda a aplicação e muitas vezes são inviáveis. A Figura 6.4 traz um exemplo da nova hierarquia que evita o problema do método voar sendo passado para a classe Pinguim.



Note que conforme subimos na hierarquia de classes, vamos para classes mais genéricas, e conforme descemos, chegamos a classes mais específicas.

Com isso, vemos que antes de programar qualquer linha de código, é importante definir qual será a responsabilidade de cada classe, o que ela está modelando, quais objetos estamos abstraindo e agrupando em uma classe, como estes objetos irão se relacionar, etc. Pois assim evitamos ou reduzimos a necessidade de alterar trechos de código por toda a aplicação para acomodar uma reorganização das classes.

Em uma situação real, estaríamos trabalhando com potencialmente muito mais classes, e é quase certo que novos recursos sejam adicionados com o passar do tempo, então é preciso desenvolver nosso programa ou aplicação de modo que os módulos e classes possam ser estendidos e reutilizados de maneira simples e fácil.

Para nos ajudar nessa tarefa, existe um conjunto de princípios de POO, desenvolvidos por Robert C. Martin entre o final da década de 1990 e começo dos anos 2000, com foco em como projetar um programa ou aplicação para que o código seja reutilizável, robusto e flexível (MARTIN, R. C., 2000). Os primeiros 5 princípios são popularmente conhecidos pelo acrônimo [SOLID](#_40vj59tmpss9), e fazem referência específica ao projeto de classes em POO.

PARA ASSISTIR! - Sandi Metz fez uma palestra a respeito dos princípios do SOLID, disponível no Youtube em <https://www.youtube.com/watch?v=v-2yFMzxqwU>, com possibilidade de legendas em português geradas automaticamente. No vídeo ela explica que Robert C. Martin não inventou sozinho todos os princípios, mas foi o responsável por juntar diferentes ideias que estavam circulando à época e nomeá-las em seu artigo, que serviu desde então de base para a discussão do desenvolvimento de software segundo a POO.

Agora vamos implementar a herança das classes acima em Python. Para fazer com que uma classe herde de outra, indicamos a classe mãe entre parênteses no momento de criação da classe filha:

Code:

class ClasseFilha(ClasseMae):

pass

Começamos então definindo a classe base inicial, para isso crie um arquivo “passaros.py” na pasta “aula06”, com o seguinte código:

Code:

class Passaro:

def \_\_init\_\_(self, vida, ataque, defesa):

self.vida = vida

self.ataque = ataque

self.defesa = defesa

def atacar(self, alvo):

pass

def fugir(self, destino):

pass

Não nos importamos com a implementação dos métodos, pois o objetivo aqui é demonstrar as características da herança em POO, então em seguida, vamos implementar as duas subclasses de Passaro.

Adicione o seguinte código ao arquivo passaros.py:

Code:

class PassaroAereo(Passaro):

def voar(self):

pass

class PassaroAquatico(Passaro):

def nadar(self):

pass

Nestas classes, não precisamos adicionar nenhum atributo, então não há necessidade de implementar o método inicializador, basta implementar os métodos específicos de cada classe, que o Python irá usar o inicializar herdado da classe mãe para inicializar os objetos. Vejamos um exemplo no modo de depuração do VSCode.

Adicione o seguinte código ao arquivo, adicione também um ponto de parada na última linha do código e execute-o modo debug pressionando F5.

Code:

p\_ar = PassaroAereo(100, 300, 250)

p\_agua = PassaroAquatico(140, 80, 400)

print('fim')

### Usando a função integrada super em Python

Para implementar as próximas duas classes, precisamos adicionar mais um atributo ao objeto, então a primeira ideia que poderíamos fazer seria:

Code:

class PassaroDeCompanhia(PassaroAereo):

def \_\_init\_\_(self, companheiro):

self.companheiro = companheiro

class Pinguim(PassaroAquatico):

def \_\_init\_\_(self, peso):

self.peso = peso

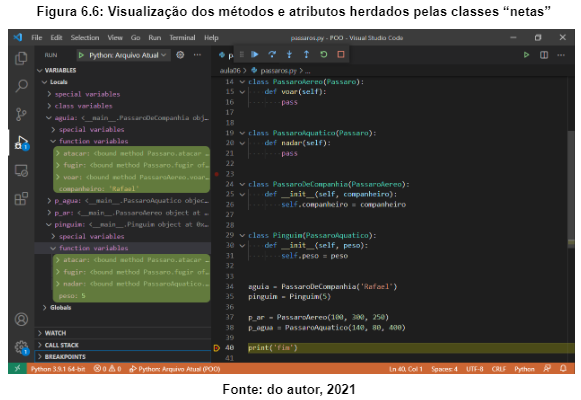
Em seguida podemos instanciar um objeto de cada classe, da seguinte forma:

Code:

aguia = PassaroDeCompanhia('Rafael')

pinguim = Pinguim(5)

Adicione os trechos de código acima ao arquivo “passaros.py”, após a definição das classes já existentes e antes das linhas em que instanciamos tais classes para testar. O resultado pode ser visto na Figura 6.6.



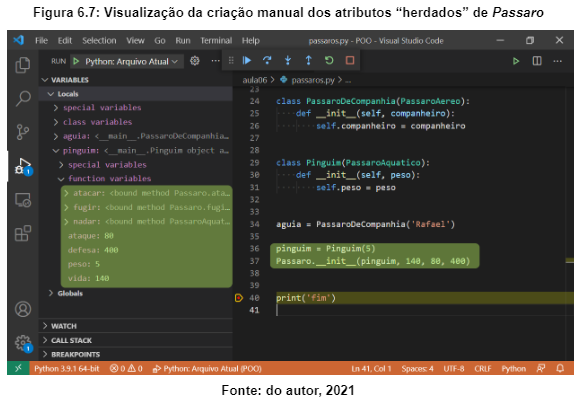
Podemos ver na Figura 6.6 que tanto os métodos gerais quanto os métodos das classes intermediárias foram herdados corretamente, mas o que aconteceu com os demais atributos? Os nossos objetos possuem apenas os atributos específicos, tendo perdido os atributos gerais de Passaro. Isso ocorreu porque sobrescrevemos o método inicializador, então foi executado o método \_\_init\_\_ de Pinguim e não mais o de Passaro, por exemplo.

Esse mecanismo é chamado de Ordem de Resolução dos Métodos, e a explicação detalhada de como ele funciona é dada no item 6.3.3 deste capítulo.

Para resolver o problema precisamos chamar manualmente o método inicializador da classe Passaro, e isso poderia ser feito usando diretamente o nome da classe e passando para ela os argumentos necessários, incluindo a referência para o objeto que deve ser alterado. Por exemplo, após criar o pinguim, podemos fazer:

Passaro.\_\_init\_\_(pinguim, 140, 80, 400)

E isso irá executar o método inicializador de Passaro com a referência para o objeto pinguim. Observe que não estamos invocando o método de um objeto, mas sim diretamente da classe, portanto o Python não fará a injeção automática do self e por isso devemos passar o objeto que queremos alterar. Observe o resultado na Figura 6.7.



Com a experiência que você já tem até aqui, o código acima deveria tocar um alarme de “acho que estou fazendo isso errado”, pois imagine a confusão que seria se para cada objeto criado precisássemos adicionar atributos manualmente, em um sistema com dezenas ou até centenas de classes. A chance de introdução de bugs no código dessa forma é altíssima e algo que queremos minimizar.

Portanto, a solução é usar a função integrada super para colocar essa chamada que fizemos manualmente no interior da classe, de modo que o Python ficará responsável por buscar o método inicializador das classes mãe e executá-los conforme nossas instruções. De acordo com a documentação da função super (PSF, 2021a), podemos usá-la em qualquer parte do nosso código, mas aqui estamos interessados no seu funcionamento quando usada no interior da definição de uma classe.

Quando isso acontece, podemos chamar a função super sem passar nenhum argumento e ela irá nos retornar um objeto que automaticamente saberá a ordem em que precisa buscar um determinado método ou atributo, seguindo o conceito de MRO que veremos no item 6.3.3 deste capítulo. Edite as classes de Pinguim e PassaroDeCompanhia para corresponder a:

Code:

class PassaroDeCompanhia(PassaroAereo):

def \_\_init\_\_(self, vida, ataque, defesa, companheiro):

self.companheiro = companheiro

super().\_\_init\_\_(vida, ataque, defesa)

class Pinguim(PassaroAquatico):

def \_\_init\_\_(self, vida, ataque, defesa, peso):

self.peso = peso

super().\_\_init\_\_(vida, ataque, defesa)

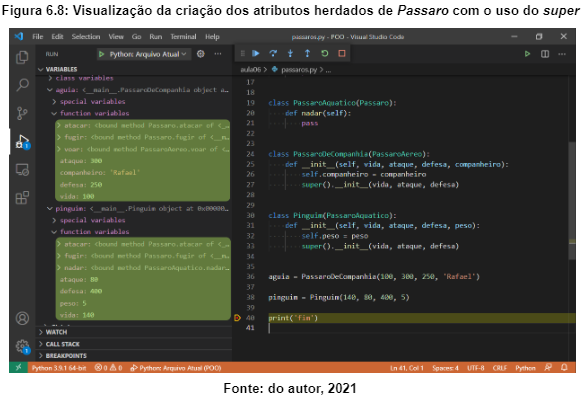
E em seguida, edite a criação dos objetos para:

Code:

aguia = PassaroDeCompanhia(100, 300, 250, 'Rafael')

pinguim = Pinguim(140, 80, 400, 5)

A Figura 6.8 mostra o resultado após a execução deste código.



Nesta situação estamos atribuindo na classe mais específica os atributos que são pertinentes à ela, e em seguida, delegamos ao Python o trabalho de procurar nas classes que estão acima na hierarquia de heranças o inicializador que deverá ser executado para atribuição dos demais atributos.

Podemos efetivamente criar uma cadeia, em que cada vez que precisarmos adicionar um atributo específico no inicializador, sobrescrevemos o método \_\_init\_\_ com os todos os parâmetros da classe mãe, seguidos pelos parâmetros específicos. Atribuímos os parâmetros específicos localmente e chamamos a função super para lidar com os demais parâmetros. Um exemplo desta cadeia é dado no item 6.3.3.

### Atributos e métodos “protegidos” em Python

Quando estamos trabalhando com herança em POO, podemos criar atributos que não sejam públicos nem privados, isto é, que estejam “escondidos” do mundo exterior mas acessíveis não só à própria classe, mas também a seus descendentes. São os atributos ditos protegidos.

Mas como já vimos, o Python não força as regras de acessibilidade. Na realidade, a documentação da PEP8 diz explicitamente que em Python não se usam os termos “privado” e “protegido”, apenas havendo a diferenciação entre os atributos [[9]](#footnote-8)públicos dos não-públicos (PSF, 2021b). Sendo assim, ela recomenda a adoção dos seguintes critérios para criação dos atributos de uma classe (incluindo os métodos):

* Sempre decida com antecedência quais atributos de uma classe serão públicos e não-públicos, na dúvida, escolha não-públicos. É mais fácil tornar público um atributo interno (não-público), do que internalizar um atributo público. Na primeira situação só precisamos mexer no código da própria classe, já na segunda, precisamos editar também todo código que seja cliente daquela classe e que usavam tal atributo público, e isso pode ser uma tarefa que, além de complicada, introduza comportamentos inesperados na aplicação
* Atributos públicos não devem ser precedidos por nenhum sublinhado;
* Se o nome de um atributo público colide com o nome de uma palavra chave e usar outro nome não é desejável (por exemplo, por piorar a legibilidade do código), deve-se adicionar um sublinhado ao final. Por exemplo: for\_. A única exceção é que caso esse nome seja usado para referenciar uma classe, a convenção é usar o nome cls.
* Para atributos de dados (o que chamamos até agora apenas de atributos) públicos simples, recomenda-se expor diretamente o atributo, e se for necessária a utilização de alguma lógica de validação em seu acesso, deve-se usar o decorador que vimos ao estudar encapsulamento: a property.
* Para atributos não-públicos, recomenda-se preceder-los de um único sublinhado.
* Se a classe foi projetada para ser extendida, e há atributos que você não quer que sejam editados pelas classes filhas, nomeie-os precedidos com dois sublinhados, já que isso irá invocar a “desfiguração de nomes” do Python.

Por fim, o mais importante é lembrar que, de acordo com com a PEP8, um guia de estilo deve prezar pela consistência, em especial a consistência interna de um módulo e do projeto no qual está inserido. Então, ao entender o funcionamento da linguagem, podemos tomar uma decisão consciente por uma ou outra abordagem.

## Polimorfismo

Como vimos na introdução dos pilares de POO, há dois tipos de polimorfismo, o de sobrecarga e o de sobrescrita.

Na sobrecarga, temos a mesma função ou método executando uma ação diferente em função da assinatura da função, isto é, se passamos um número diferente de argumentos e/ou argumentos de tipos diferentes, o comportamento muda.

Já a sobrescrita ocorre quando objetos diferentes possuem implementações diferentes de um mesmo método, como citado no exemplo de um pássaro mecânico vs. um pássaro normal em relação ao método voar.

### Sobrecarga

Em Python, devido a sua natureza de tipagem dinâmica, não é possível fazer a sobrecarga de um método ou função da maneira tradicional, como é feita em linguagens de tipagem estática como Java, C# ou C++. Isso ocorre porque em Python, o interpretador só saberá o tipo de uma variável ou parâmetro em tempo de execução, então a única forma de diferenciarmos a assinatura de um método é pela quantidade de argumentos que ele recebe.

Com isso, podemos implementar uma variação do polimorfismo de sobrecarga ao utilizarmos valores padrões para alguns dos parâmetros do método ou função, tornando tais parâmetros opcionais e permitindo assim que o método seja chamado com diferentes assinaturas.

Para exemplificar, podemos usar a função que foi provavelmente a primeira a aprendermos em Python, o print. Como talvez você já saiba, o print aceita alguns parâmetros nomeados[[10]](#footnote-9) como end e sep, que alteram, respectivamente, o caractere adicionado ao final da string e o caractere usado na união dos parâmetros posicionais passados previamente para formar a string, antes de exibi-la na tela.

Temos dessa forma, efetivamente comportamentos diferentes para uma mesma função quando chamada com assinaturas diferentes. O comportamento padrão é incluir um espaço entre os valores posicionais passados e finalizar a string com uma quebra de linha, denotada pelo caractere '\n'.

Faça o teste executando o código a seguir e observe o resultado.

Code:

print('chamada original')

print(1, 2, 3)

print()

print('alterando o sep')

print(1, 2, 3, sep='...')

print()

print('alterando o end')

print('prints separados', end=' ')

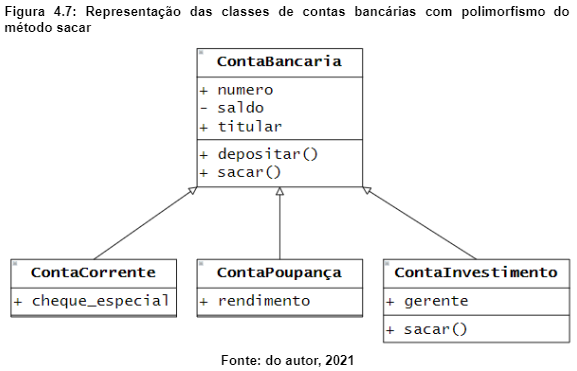
print('mas impressos', end=' ')

print('em uma mesma linha')

### 

### Sobrescrita

O polimorfismo por sobrescrita ocorre quando sobrescrevemos na classe filha um método herdado da classe mãe. Vamos retomar o exemplo da aula de introdução a POO, na qual fizemos classes para representar contas bancárias, mostrado na Figura 6.9.



Vamos implementar um exemplo ilustrativo destas classes para visualizar melhor como funciona o polimorfismo de sobrescrita em Python. Crie um arquivo “contas\_bancarias.py” na pasta “aula06” e adicione os trechos de código a seguir.

Na classe mãe, implementamos a inicialização do objeto e os métodos para realizar depósitos e saques, que inclui a lógica para verificar o saldo e realizar o saque:

Code:

class ContaBancaria:

def \_\_init\_\_(self, numero, titular):

self.numero = numero

self.titular = titular

self.\_\_saldo = 0

def depositar(self, valor):

self.\_\_saldo += valor

print(f'Deposito realizado. Saldo: R$ {self.\_\_saldo}')

def sacar(self, valor):

if valor > self.\_\_saldo:

print(f'Saque falhou. Saldo: R$ {self.\_\_saldo}')

return 'Saldo insuficiente.'

self.\_\_saldo -= valor

print(f'Saque realizado. Saldo: R$ {self.\_\_saldo}')

return valor

Ao criar a classe da conta poupança, precisamos apenas sobrescrever o inicializador do objeto, para incluir o atributo específico referente ao rendimento da poupança, que neste exemplo será fixo, e chamamos o super para continuar com a inicialização do objeto:

Code:

class ContaPoupanca(ContaBancaria):

def \_\_init\_\_(self, numero, titular):

self.rendimento = 0.5

super().\_\_init\_\_(numero, titular)

Já na classe da conta investimento, reescrevemos o inicializador e também o método de sacar, para incluir a lógica específica relacionada a este tipo de conta:

Code:

class ContaInvestimento(ContaBancaria):

def \_\_init\_\_(self, numero, titular, gerente):

self.gerente = gerente

super().\_\_init\_\_(numero, titular)

def sacar(self, valor):

print('verificando prazo do investimento...')

print('calculando impostos e taxas...')

print('realizando saque...')

return super().sacar(valor)

Observe que usamos o super também no método de sacar, para delegar à classe mãe a realização de fato do saque. Como criamos o atributo saldo com dois sublinhados, indicando que não devemos alterá-lo fora da classe que o criou, isso é necessário para não violarmos a não-publicidade do atributo. Vale lembrar aqui que em uma situação real, as instruções de print do exemplo seriam substituídas pela lógica que faria as verificações de fato.

Agora crie um outro arquivo “test\_contas.py” na mesma pasta, com o seguinte código para testar as classes:

Code:

from contas\_bancarias import ContaPoupanca, ContaInvestimento

# Criação das contas

conta\_poupanca = ContaPoupanca('001', 'Rafael')

conta\_investimento = ContaInvestimento('001', 'Rafael', 'Ana')

print('\n----Operações na conta poupança----')

conta\_poupanca.depositar(1000)

saque\_1 = conta\_poupanca.sacar(100)

saque\_2 = conta\_poupanca.sacar(3000)

print(f'Primeiro saque da poupança: R$ {saque\_1}')

print(f'Segundo saque da poupança: R$ {saque\_2}')

print('\n----Operações na conta investimento----')

conta\_investimento.depositar(500)

saque\_3 = conta\_investimento.sacar(300)

saque\_4 = conta\_investimento.sacar(300)

print(f'Primeiro saque da conta investimento: R$ {saque\_3}')

print(f'Segundo saque da conta investimento: R$ {saque\_4}')

### Ordem de resolução dos métodos

Quando acessamos um método ou atributo de um objeto usando a notação de ponto: objeto.atributo ou objeto.método(), o Python irá buscá-lo na classe atual e, caso não encontre, ele sobe um nível na hierarquia e busca novamente. Esse processo é repetido até chegar ao fim da linha, que é a classe object, da qual todas as outras classes herdam automaticamente em Python. Se então o atributo ou método não for encontrado, o Python levanta um erro de atributo (AttributeError).

Para visualizar a ordem das classes em que o Python irá buscar pelos métodos e atributos usando o método mro[[11]](#footnote-10) a partir da classe que queremos investigar.

No exemplo dos pássaros, temos:

Code:

>>> Pinguim.mro()

[<class '\_\_main\_\_.Pinguim'>, <class '\_\_main\_\_.PassaroAquatico'>, <class '\_\_main\_\_.Passaro'>, <class 'object'>]

Podemos ver que ao buscar um método ou atributo, o Python irá percorrer toda a hierarquia que definimos no começo do capítulo, até chegar em object, parando a busca na primeira ocorrência encontrada.

# SOLID

O SOLID é um acrônimo que representa cinco princípios fundamentais do design de software orientado a objetos (POO), introduzidos por Robert C. Martin, também conhecido como Uncle Bob. Esses princípios visam criar código mais limpo, modular, flexível e fácil de manter. Eles são frequentemente usados como diretrizes para desenvolvedores de software escreverem código de alta qualidade e sistemas bem projetados. Abaixo os cinco princípios do SOLID.

## SRP - Single Responsibility Principle (Princípio da Responsabilidade Única)

Uma classe deve ter apenas uma única responsabilidade, ou seja, um único motivo para mudar. Isso torna a classe mais coesa e facilita sua compreensão e manutenção.

## OCP - Open-Closed Principle (Princípio Aberto-Fechado)

Uma classe deve estar aberta para extensões, mas fechada para modificações. Isso significa que você pode adicionar novas funcionalidades à classe sem precisar modificar o código existente.

## LSP - Liskov Substitution Principle (Princípio da Substituição de Liskov)

Subclasses devem ser substituíveis por suas superclasses em qualquer contexto sem afetar o comportamento correto do programa. Isso garante a flexibilidade e a reutilização do código.

## ISP - Interface Segregation Principle (Princípio da Segregação da Interface)

As interfaces devem ser pequenas e coesas, definindo apenas um conjunto específico de funcionalidades. Isso torna as interfaces mais fáceis de entender e implementar.

## DIP - Dependency Inversion Principle (Princípio da Inversão de Dependência)

As classes devem depender de abstrações, e não de implementações concretas. Isso torna as classes mais desacopladas e facilita sua modificação e teste.

## Benefícios do SOLID

* **Código mais modular:** O SOLID ajuda a criar código mais modular, composto por classes menores e coesas com responsabilidades bem definidas.
* **Código mais flexível:** O SOLID torna o código mais flexível e adaptável a mudanças, facilitando a adição de novas funcionalidades e a correção de bugs.
* **Código mais reutilizável:** O SOLID promove a reutilização de código, pois as classes e interfaces são mais genéricas e desacopladas.
* **Código mais fácil de manter:** O SOLID facilita a manutenção do código, pois as classes são mais coesas e fáceis de entender.

Os princípios do SOLID podem ser aplicados a qualquer linguagem de programação orientada a objetos. Ao seguir esses princípios, você poderá escrever software mais robusto, eficiente e sustentável.

# Módulos e Pacotes

Já vimos que em Python, todo arquivo “\*.py” é automaticamente um módulo, que pode ser importado em outros arquivos “\*.py” ou diretamente em uma Shell do Python, mas pouco foi falado sobre como podemos usar isso para melhor organizar nosso código.

Quando iniciamos o aprendizado de uma linguagem de programação, é comum utilizarmos um interpretador interativo para testar os recursos da linguagem em tempo real e aprender sobre os diferentes tipos de dados, mas isso não nos permite salvar nosso progresso, pois a cada vez que fechamos o programa, tudo é perdido e precisamos começar do zero na próxima vez.

Para resolver isso, usamos um editor de texto para salvar as instruções em pequenos trechos de código, e então executamos o código a partir deste arquivo em um interpretador. A esses arquivos damos o nome de scripts.

Conforme avançamos nos estudos e no desenvolvimento de um projeto, esse arquivo cresce e começa a ficar responsável por muitas tarefas, o que dificulta a manutenção do código. Então é comum dividirmos esse script em diversos arquivos, para facilitar a organização e manutenção do nosso projeto. A esses arquivos damos o nome de módulos.

Dependendo do tamanho do projeto, pode ser necessário um número muito grande de arquivos, então é comum dividirmos estes arquivos em pastas para melhor organizar o que cada conjunto de arquivos é responsável por fazer. A estas pastas damos os nomes de pacotes[[12]](#footnote-11). Um pacote pode conter quantos módulos e sub-pacotes forem necessários.

Observe que do ponto de vista prático, não existe uma diferença real entre um script e um módulo, a não ser pela forma como se pretende utilizá-los. Quando dividimos nosso código em módulos, podemos importar o conteúdo de um módulo para outro, o que facilita a reutilização de código, sem necessidade de copiá-lo.

Se construirmos os módulos de maneira que eles sejam o mais independentes possível, isto é, dependentes apenas de interfaces bem definidas para as funções e classes, mas agnósticos à sua implementação, nosso projeto será mais fácil de testar e manter, já que alterações em um módulo causam pouco (idealmente nenhum) impacto em outros módulos.

## Módulos em Python

Podemos separar os módulos do Python em 3 tipos, de acordo com a sua origem, mas vale ressaltar que do ponto de vista do Python só existe um tipo de módulo.

* módulos integrados;
* módulos de terceiros; e
* módulos próprios.

Os primeiros módulos que utilizamos são aqueles que já vêm integrados à linguagem, como por exemplo os módulos time e math. Em Python, os módulos integrados à implementação padrão[[13]](#footnote-12) da linguagem podem ser escritos tanto em C quanto em Python.

O módulo de matemática é um exemplo de módulo escrito em C, cujo código fonte pode ser visto no github oficial do Python (PSF, 2021a). Já o módulo turtle, cuja demo pode ser acessada a partir do menu de ajuda do IDLE, é escrito inteiramente em Python e seu código fonte pode ser visto no diretório de instalação do Python. Na Shell do Python, digite o código da Codificação 7.1 em uma Shell do Python e navegue até a pasta que for exibida para visualizar o código fonte do módulo turtle.

Codificação 7.1. Visualização do diretório do módulo turtle no Python

code:

>>> import turtle

>>> print(turtle.\_\_file\_\_)

C:\Program Files\Python39\lib\turtle.py

Além dos módulos integrados, podemos também instalar módulos ou pacotes desenvolvidos por terceiros para resolver um determinado problema, como por exemplo, criar interfaces gráficas, ler e escrever dados em planilhas, criar aplicações web e API’s, fazer análises estatísticas, manipular matrizes n-dimensionais, criar jogos, entre muitas outras possibilidades.

A principal forma de se obter tais módulos é através do PyPI, repositório oficial da linguagem Python, e a ferramenta mais popular e recomendada de se instalar tais pacotes é o pip. O guia completo de utilização do pip pode ser visto na página do grupo responsável por manter o PyPI (PyPA, 2021).

E por fim, podemos ainda utilizar nossos próprios módulos em nossos projetos, ou seja, podemos subdividir nosso projeto em diversos arquivos e importar as classes e funções de um lugar para outro conforme necessário. então veremos agora como organizar nossos arquivos em módulos e como o Python trabalha com diferentes arquivos em um mesmo projeto.

### VOCÊ SABIA?

Em Python, uma das principais ferramentas para trabalhar com vetores e matrizes é a biblioteca Numpy, desenvolvida para realizar com maior eficiência cálculos complexos com diferentes tipos de sequências. É uma biblioteca utilizada em áreas como ciência de dados, engenharia, matemática aplicada, estatística, economia, entre outras.

Essa biblioteca, como outras do CPython, foi desenvolvida em C, mas sua interface é Python, possibilitando a eficiência dos códigos em C, com a simplicidade da sintaxe Python. Portanto, escreve-se um código mais fácil de ler e entender, sem abrir mão do desempenho necessário para aplicações que processam enorme quantidade de dados.

O Numpy, juntamente com outras bibliotecas do Python que dependem dele, como Scikit-image, SciPy, MatPlotLib e Pandas, foi usado em duas situações com repercussão mundial: a geração da primeira imagem de um buraco negro (Numpy, 2020a) e a detecção de ondas gravitacionais (Numpy, 2020b) pelos cientistas do Observatório de Ondas Gravitacionais por Interferômetro Laser (LIGO).

Em ambos os casos o Python foi usado para coletar, tratar, analisar e gerar visualizações processando terabytes de dados diariamente. São aplicações do Python em problemas complexos, com muitos dados e com exigência de excelente desempenho computacional.

### Espaço de nomes de um módulo

O primeiro ponto a ser observado é o que podemos chamar de escopo global do módulo. Quando aprendemos a definir funções, estudamos a diferença entre o escopo global e o escopo local da função. Cada módulo em Python define um escopo global, no qual há uma tabela que relaciona os identificadores existentes naquele módulo com os objetos na memória.

Então, ao importarmos um módulo, o Python adiciona uma entrada neste escopo com uma referência para o módulo que foi importado. Isso pode ser verificado com o uso da função dir, que retorna uma lista dos nomes existentes em um dado escopo. Se chamada sem argumento, retorna os nomes do escopo atual.

Code:

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

>>> import math

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'math']

Podemos observar que o nome math foi adicionado ao escopo atual, e podemos então utilizar as funções que este módulo traz. Para ver a lista completa: dir(math).

### A variável \_\_name\_\_

Todo arquivo Python contendo definições e declarações é um módulo, e o nome do módulo é o próprio nome do arquivo sem a extensão. No interior do módulo, esse nome fica disponível, como string, na variável global \_\_name\_\_. Vamos criar um módulo de exemplo para praticar.

1. Crie um arquivo Python vazio com o nome meu\_modulo.py;
2. Abra um terminal no próprio VSCode ou externamente (cmd ou powershell, se estiver no Windows);
3. Se necessário, navegue até a pasta do arquivo que você criou, com o comando: cd caminho\para\pasta;
4. Abra o interpretador do Python, com o comando: py no Windows, e python ou python3 no Linux/Mac.

Na Codificação 7.3 importamos nosso módulo e inspecionamos o conteúdo da variável \_\_name\_\_.

Codificação 7.3. Inspeção do nome de um módulo importado na Shell

Code:

>>> import meu\_modulo

>>> meu\_modulo.\_\_name\_\_

'meu\_modulo'

Quando importamos um módulo, o Python cria um nome no escopo global para o qual o módulo foi importado e associa este nome ao objeto do módulo na memória (veja o que retorna a função dir() após a importação de meu\_modulo). Dessa forma, evitamos qualquer conflito de nomes entre as variáveis globais do módulo atual, que está sendo executado, e dos módulos que são importados. Isso garante que podemos desenvolver nossos módulos sem nos preocupar com a unicidade dos nomes das variáveis, classes e funções, em relação a outros módulos.

Quando executamos um módulo diretamente, seja pressionando F5 na IDLE, usando o modo de depuração do VSCode, executando o interpretador do Python na linha de comando ou usando qualquer outra IDE, dizemos que ele é o módulo (ou script) principal. Nesse caso, a variável especial \_\_name\_\_ recebe o como nome a string '\_\_main\_\_' .

Ao importarmos um módulo, o interpretador do Python executa todo o seu conteúdo uma única vez, criando um objeto na memória que contém internamente todos os objetos criados pelo módulo. Vamos então incluir a seguinte linha no nosso arquivo meu\_modulo.py: print(f'O nome deste módulo é: {\_\_name\_\_!r}'[[14]](#footnote-13)).

Agora podemos executar o arquivo na linha de comando:

Codificação 7.4. Execução do arquivo meu\_modulo.py na linha de comando

Code:

E:\POO\aula07> py meu\_modulo.py

O nome deste módulo é: '\_\_main\_\_'

E podemos também importar esse arquivo para uma. Shell do Python:

Code:

E:\POO\aula07> py

Python 3.9.1

>>> import meu\_modulo

O nome deste módulo é: 'meu\_modulo'

Observe que quando o módulo é executado diretamente, o nome da variável especial \_\_name\_\_ é '\_\_main\_\_', já quando o módulo é importado, ou seja, não é o módulo principal sendo executado, essa variável guarda o nome do próprio módulo meu\_modulo. No segundo exemplo, o “módulo” principal é a execução atual da Shell do Python, que define um espaço de nomes e também possui a variável especial \_\_name\_\_. Podemos confirmar isso executando os comando da Codificação 7.6 na mesma Shell:

Codificação 7.6. Inspeção dos nomes do módulo e do escopo atual

Code:

>>> \_\_name\_\_

'\_\_main\_\_'

>>> meu\_modulo.\_\_name\_\_

'meu\_modulo'

Você deve ter observado que a instrução para exibir o nome do módulo foi executada automaticamente no momento da importação. Em geral, esse é um comportamento que queremos evitar ao importar um módulo, pois na grande maioria das vezes, queremos apenas carregar suas definições (classes, funções, constantes, etc.) para podermos utilizá-las conforme necessário. Vamos trocar a exibição direta para a definição de uma função que quando chamada exibe o nome do módulo.

Edite o arquivo meu\_modulo.py para corresponder à Codificação 7.7.

Codificação 7.7. Conteúdo do arquivo meu\_modulo.py

Code:

def exibe\_nome():

print(f'O nome deste módulo é: {\_\_name\_\_!r}')

if \_\_name\_\_ == '\_\_main\_\_':

exibe\_nome()

Antes de seguir para o próximo teste, feito na Codificação 7.8, feche a Shell que utilizamos para o teste anterior, com o comando exit(), ou abra um novo terminal. Isso é necessário pois o Python importa os módulos apenas uma vez, ao tentarmos importar um módulo que já está importado, o Python identifica que aquele módulo já existe e ignora o comando de importação[[15]](#footnote-14).

Code:

E:\POO\aula07> py meu\_modulo.py

O nome deste módulo é: '\_\_main\_\_'

E:\POO\aula07> py

Python 3.9.1

>>> import meu\_modulo

>>> meu\_modulo.exibe\_nome()

O nome deste módulo é: 'meu\_modulo'

Para que nosso módulo continue funcionando também como um script, o que pode ser útil durante o desenvolvimento, para realização de testes por exemplo, podemos adicionar uma verificação do conteúdo da variável especial \_\_name\_\_, se o módulo estiver sendo executado como módulo principal, fazemos a chamada à função que exibe o nome, mas quando importamos o módulo, esse código não é executado.

Observe que para chamar a função que exibe o nome do módulo, usamos a notação de ponto, indicando que queremos executar a função exibe\_nome que pertence ao módulo meu\_modulo.

### A variavel \_\_all\_\_ em Python

A função \_\_all\_\_ em Python é uma variável especial que serve para controlar quais nomes (módulos, funções, classes, etc.) dentro de um pacote ou módulo são expostos quando o pacote ou módulo é importado. Isso significa que, quando você usa from <nome\_do\_pacote> import \*, apenas os nomes presentes na lista \_\_all\_\_ serão importados para o seu namespace.

Considere um módulo meu\_modulo.py com as seguintes funções:

Code:

def funcao\_publica():

...

def \_funcao\_privada():

...

\_\_all\_\_ = ['funcao\_publica']

Ao importar este módulo em outro script:

Code:

from meu\_modulo import \*

funcao\_publica() # Função pública pode ser importada e utilizada

#\_funcao\_privada() # Erro: função privada não está na lista \_\_all\_\_

Observações:

* A função \_\_all\_\_ é uma variável, não uma função. Você não precisa chamá-la, apenas precisa defini-la como uma lista contendo os nomes que deseja exportar.
* O valor de \_\_all\_\_ é ignorado quando você importa um módulo usando import <nome\_do\_modulo>. Nesse caso, todos os nomes do módulo serão importados.
* É uma boa prática usar \_\_all\_\_ para documentar quais nomes são públicos em seu pacote ou módulo.

## Formas de importar um módulo

Além da forma padrão que vimos até agora: import modulo, existem algumas outras formas de se importar um módulo, ou parte dele, para o escopo atual, como veremos agora.

### O caminho de busca de um módulo

Ao executarmos um comando de importação de um módulo, o Python irá seguir as seguintes regras sobre onde procurar pelo arquivo do módulo:

* Antes de realizar qualquer busca, o Python verifica se aquele módulo já foi importado naquela sessão, se sim, ele reutiliza a mesma referência para o módulo já existente em memória e interrompe o processo.
* Se o módulo ainda não foi importado, o Python irá fazer uma busca a partir de uma lista de diretórios que pode ser vista na variável sys.path. Esta lista é composta por 3 partes:

1. O primeiro lugar buscado é a partir do diretório em que se encontra o arquivo de entrada, que está executando a importação, ou do diretório atual se não for especificado nenhum arquivo (como em uma Shell, por exemplo);
2. Em seguida, ele busca a lista de diretórios especificada em uma variável de ambiente com caminhos padrão do Python; e
3. Por último, há uma lista de caminhos padrão que depende da instalação e do sistema operacional.

É possível visualizar e alterar em tempo de execução essa lista para incluir ou excluir diretórios de busca conforme a necessidade, através da variável sys.path, utilizada como no exemplo da Codificação 7.9.

Codificação 7.9. Visualização da variável sys.path em uma instalação do Python 3.8 no Ubuntu

Code:

>>> import sys

>>> sys.path

['',

'/usr/lib/python38.zip',

'/usr/lib/python3.8',

'/usr/lib/python3.8/lib-dynload',

'/usr/local/lib/python3.8/dist-packages',

'/usr/lib/python3/dist-packages']

Essa variável é uma lista comum do Python, e portanto podemos usar todos os métodos de lista para alterá-la. As strings com cada caminho devem seguir o padrão do sistema operacional em questão, e tais alterações devem ser feitas com cautela, pois uma alteração incorreta poderá fazer com que a importação de módulos pare de funcionar ou tenha efeitos inesperados.

### Importação com a criação de um alias para o módulo

Para criarmos um alias, ou apelido, para um módulo, usamos a sintaxe a seguir:

Codificação 7.10. Sintaxe para importação de um módulo atribuído a um novo nome (apelido)

Code:

import <nome do módulo> as <apelido>

Isso irá associar o módulo importado ao nome definido por <apelido>. Traduzindo para o português, podemos ler a instrução acima como: “importe o módulo <nome do módulo> como <apelido>”. Veja o exemplo com módulo math na Codificação 7.11.

Codificação 7.11. Exemplo de utilização da importação com apelido

Code:

>>> import math as mat

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'mat']

>>> mat

<module 'math' (built-in)>

>>> mat.pi

3.141592653589793

Observe que no escopo atual, foi criado o nome mat, e devemos utilizá-lo para acessar o objeto do módulo de matemática que contém todas as funções que este módulo nos traz. O nome do módulo (que pode ser acessado em mat.\_\_name\_\_) continua o mesmo, mas sua referência está associada a um novo nome no escopo atual.

### Importação de elementos do módulo

É possível importar apenas um sub-pacote, classe ou função específica a partir de um módulo, com a seguinte sintaxe:

Codificação 7.12. Sintaxe da importação parcial de um elemento do módulo

Code:

from <nome do módulo> import <nome do elemento>

Isso irá importar apenas o elemento para o escopo atual, e não o módulo inteiro. Em geral utilizamos essa abordagem quando não queremos carregar o módulo todo pois só utilizaremos poucas de suas funções ou quando queremos simplificar o código do escopo atual, pois agora não precisamos mais da notação de ponto para chegar até um determinado elemento do módulo. Traduzindo para o português, podemos ler a instrução acima como: “a partir do módulo <nome do módulo> importe o <elemento>”.

Code:

>>> from math import pi

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'pi']

>>> pi

3.141592653589793

Note que podemos utilizar o nome pi diretamente, sem o nome do módulo, mas ao mesmo tempo, não temos acesso a nenhuma outro nome definido pelo módulo de matemática, como por exemplo a função log, sqrt, etc. Devemos também estar atentos a um eventual conflito que pode ocorrer caso nosso código atual também defina uma variável chamada pi.

É possível importar mais de um elemento de um mesmo módulo separando-os por vírgula, como no exemplo da Codificação 7.14, no qual importamos as funções de seno, cosseno e tangente, além da constante 𝛑.

Codificação 7.14. Exemplo de importação parcial de múltiplos elementos de um módulo.

Code:

>>> from math import cos, pi, sin, tan

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'cos', 'pi', 'sin', 'tan']

### Importação de elemento com a criação de alias

É possível também unir as duas situações que acabamos de ver, e criarmos um apelido no escopo atual para o elemento que foi importado, com a seguinte sintaxe:

Codificação 7.15. Sintaxe da importação parcial de um elemento do módulo atribuído a um novo nome (apelido)

Code:

from <nome do módulo> import <nome do elemento> as <apelido>

Podemos ler esse comando como “a partir do módulo <nome do módulo> importe o elemento <nome do elemento> como <apelido>”. Voltando ao exemplo do módulo de matemática, veja a Codificação 7.16.

Codificação 7.16. Exemplo de utilização conjunta da importação parcial com apelido

Code:

>>> from math import sin as seno

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'seno']

E podemos ainda fazer a importação de mais de um elemento usando a atribuição de apelidos, como mostra a Codificação 7.17.

Codificação 7.17. Exemplo de utilização conjunta da importação parcial de múltiplos elementos do módulo com apelidos

Code:

>>> from math import cos as cosseno, pi, sin as seno, tan as tangente

>>> dir()

['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'cosseno', 'pi', 'seno', 'tangente']

## Organização de módulos em pacotes e sub-pacotes

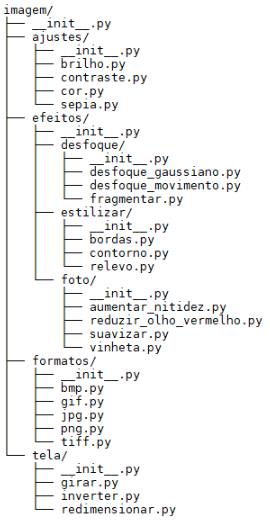
Para entender como podemos criar e organizar módulos em pacotes e sub-pacotes, veremos um exemplo de uma ferramenta para edição de imagens, cuja estrutura de diretórios é mostrada na Figura 7.1. Vale lembrar que o objetivo desta disciplina não é discutir a arquitetura do projeto, apenas mostrar um exemplo simplificado dessa organização, para entendermos como funciona a importação e utilização de tais módulos.

Neste exemplo, não estamos preocupados com o conteúdo de cada arquivo Python, mas imagine que em uma situação real eles teriam as classes e funções necessárias para realizar o processamento dos dados das imagens e a leitura e escrita em disco dos arquivos.

Com essa estrutura, podemos desenvolver cada módulo isoladamente, sem nos preocupar com os demais módulos nem com possíveis conflitos de nomes entre os módulos, pois como vimos, cada módulo possui seu próprio escopo global, no qual são definidos os nomes de variáveis, classes e funções.

Para acessar as funções e classes de um módulo, podemos utilizar a notação de ponto para chegar até o módulo que for necessário e importá-lo.

Figura 7.1: Exemplo de estrutura de um módulo para tratamento de imagens em Python.



### O arquivo \_\_init\_\_.py

Você deve ter observado a presença de um arquivo com o nome \_\_init\_\_.py em cada uma das pastas. Desde a versão 3.3 do Python, esse arquivo é opcional e, se estiver presente, será executado quando o módulo for importado. Podemos deixar o arquivo vazio, apenas para indicar ao Python que aquela pasta deve ser interpretada como um pacote, ou colocar algum código de inicialização do nosso módulo, como por exemplo fazer a importação de outros nomes para o escopo atual.

Outra vantagem de colocar esse arquivo \_\_init\_\_.py é evitar que pastas com nomes comuns ao Python, como string por exemplo, acabem escondendo outros pacotes mais adiante no caminho de busca de pacotes.

Aqui estamos usando a palavra módulo para nos referenciar também ao pacote. Acontece que em python, todo pacote é também um módulo, que pode ser importado e define um espaço de nomes, mas nem todo módulo é um pacote (PSF, 2021b). Por exemplo, a maioria dos módulos são arquivos que contém as funções e classes que desejamos reutilizar.

Se deixarmos os arquivos de inicialização vazios, podemos utilizar esse módulo externamente sem nenhum problema, importando cada um dos submódulos que precisarmos.

No da Codificação 7.18, podemos importar cada módulo necessário e acessá-los utilizando a notação de ponto. Alternativamente, podemos fazer uma importação parcial apenas das funções que iremos precisar, como mostra na codificação 7.19.

Codificação 7.18. Exemplo 1 de utilização do módulo imagem

Code:

import imagem.ajustes.brilho

import imagem.efeitos.foto.vinheta

import imagem.formatos.jpg

import imagem.formatos.png

import imagem.tela.redimensionar

arquivo = input('Digite o nome do arquivo jpg: ')

dados = imagem.formatos.jpg.carregar(arquivo)

dados = imagem.tela.redimensionar.pixels(dados, 600, 400)

dados = imagem.ajustes.brilho.ajustar\_brilho(dados, 0.7)

dados = imagem.efeitos.foto.vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')

imagem.formatos.png.salvar(dados, f'{nome}.png')

Codificação 7.19. Exemplo 2 de utilização do módulo imagem

Code:

from imagem.ajustes.brilho import ajustar\_brilho

from imagem.efeitos.foto.vinheta import aplicar

from imagem.formatos.jpg import carregar

from imagem.formatos.png import salvar

from imagem.tela.redimensionar import pixels

arquivo = input('Digite o nome do arquivo jpg: ')

dados = carregar(arquivo)

dados = pixels(dados, 600, 400)

dados = ajustar\_brilho(dados, 0.7)

dados = aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')

salvar(dados, f'{nome}.png')

A forma da codificação 7.19 deixa o código mais simples, mas ao mesmo tempo perdemos parte da semântica trazida ao lermos o caminho completo de importação de cada função utilizada. Portanto, a decisão entre qual delas usar irá depender do contexto e neste exemplo, é razoável pensar que haverá mais de uma função carregar ou salvar, então poderia fazer mais sentido a importação apresentada na Codificação 7.20.

Codificação 7.20. Exemplo 3 de utilização do módulo imagem

Code:

from imagem.ajustes.brilho import ajustar\_brilho

from imagem.efeitos.foto import vinheta

from imagem.formatos import jpg, png

from imagem.tela.redimensionar import pixels

arquivo = input('Digite o nome do arquivo jpg: ')

dados = jpg.carregar(arquivo)

dados = pixels(dados, 600, 400)

dados = ajustar\_brilho(dados, 0.7)

dados = vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')

png.salvar(dados, f'{nome}.png')

Neste terceiro exemplo, continuamos com um código mais simples e evitamos um possível conflito de nomes entre a função salvar do formato jpg e png, por exemplo, além de deixar o código mais expressivo e mais fácil de ler.

### Inicialização dos módulos

Ao deixarmos os arquivos de inicialização vazios, os submódulos não são automaticamente importados ao importarmos simplesmentes o módulo principal, isto é, ao executar o comando import imagem, mas ao realizar este comando, o arquivo imagem/\_\_init\_\_.py será executado, então podemos incluir nele o código da Codificação 7.21.

Codificação 7.21. Código do arquivo de inicialização do módulo imagem (imagem/\_\_init\_\_.py)

Code:

import imagem.ajustes

import imagem.efeitos

import imagem.formatos

import imagem.tela

Tais importações irão executar os arquivos de inicialização de cada um destes submódulos, então se repetirmos esse procedimento para todos os arquivos de inicialização, importando em cada um deles os submódulos ali presentes, teremos todo o módulo de imagem carregado para o escopo atual ao fazermos simplesmente o comando: import imagem. Dessa forma, podemos simplificar a importação do módulo como mostra a Codificação 7.22.

Codificação 7.22. Exemplo 4 de utilização do módulo imagem

Code:

import imagem

arquivo = input('Digite o nome do arquivo jpg: ')

dados = imagem.formatos.jpg.carregar(arquivo)

dados = imagem.tela.redimensionar.pixels(dados, 600, 400)

dados = imagem.ajustes.brilho.ajustar\_brilho(dados, 0.7)

dados = imagem.efeitos.foto.vinheta.aplicar(dados, 0.8, '0032af')

nome = input('Salvar como png: ')

imagem.formatos.png.salvar(dados, f'{nome}.png')

### Nota sobre a criação de pacotes em Python

A criação de pacotes em Python é um assunto muito mais complexo do que o que foi apresentado neste capítulo, e o seu entendimento por completo se faz necessário apenas ao precisarmos criar um pacote Python para distribuição.

Para os exemplos que veremos ao longo do curso, é suficiente deixar os arquivos na mesma pasta, sem a necessidade de criação do arquivo de inicialização. E podemos utilizar tanto a importação completa (import <módulo>) quanto a parcial (from <módulo> import <elemento>[[16]](#footnote-15)), conforme desejarmos, que isso irá funcionar.

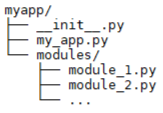
Para continuar seus estudos a respeito dos módulos em Python, veja o tutorial da documentação oficial (PSF, 2021c).

## PEP 8

A recomendação da PEP 8 em relação à nomenclatura de pacotes e módulos, cujo exemplo pode ser visto na Figura 7.2, é a seguinte:

* Módulos (arquivos) devem ter nomes curtos e com todas as letras minúsculas, e as palavras podem ser separadas por sublinhado se isso melhorar a legibilidade.
* Pacotes (pastas) devem seguir a mesma regra, nomes curtos com letras minúsculas, mas sem a separação com sublinhados.

Figura 7.2: Exemplo de nomenclatura de módulos e pacotes segundo a PEP 8.



# Ambientes virtuais, utilização de bibliotecas de terceiros e gerenciamento de dependências

Podemos dizer que boa parte dos problemas que enfrentamos no dia a dia da programação são releituras de problemas comuns. Por exemplo, cada aplicação web existente é única e feita para atender a uma regra de negócio específica, mas ao mesmo tempo, boa parte do que ela precisa fazer é comum a todas as aplicações, como criar rotas, renderizar uma template html, gerar um arquivo JSON, fazer uma requisição http, etc. Portanto, existem ferramentas como Flask e Django, que são exemplos famosos no caso de aplicações web, que facilitam nosso trabalho de desenvolvimento, automatizando e abstraindo boa parte das funções comuns, de modo que podemos focar em desenvolver a parte da aplicação que é única para o nosso problema.

Considerando a maturidade da linguagem Python, que existe há mais de 30 anos e conta com uma comunidade global extremamente ativa e participativa, é seguro dizer que, para todos os problemas que precisamos resolver, provavelmente alguém em algum lugar já precisou resolver um problema parecido e fez um módulo para isso. E no raro caso de isso não ser verdade, esta pode ser uma ótima oportunidade para criar seu primeiro módulo e disponibilizá-lo para a comunidade. Qualquer um pode fazer isso e não é preciso décadas de experiência para tanto, pois uma vez lançado, seu projeto poderá ganhar outros contribuidores que o ajudarão a avançar no seu desenvolvimento[[17]](#footnote-16).

Além de aprender sobre a instalação e utilização de pacotes, outro conceito muito importante é o de ambientes virtuais, virtualenv’s ou venv’s [[18]](#footnote-17)como são muitas vezes chamados. Eles são essenciais quando precisamos trabalhar em mais de um projeto ou aplicação, pois é praticamente certo que em algum momento elas apresentarão uma incompatibilidade de dependências, então o uso de um ambiente virtual nos permite isolar não só os módulos mas também o interpretador do Python que será usado para cada aplicação ou projeto, que podem ter inclusive versões diferentes.

## Ambientes virtuais

Imagine a seguinte situação, um projeto A precisa trabalhar com a versão 1.3 de um determinado módulo, para ser compatível com a interface que foi desenvolvida no seu lançamento por exemplo, mas outro projeto B, que está começando agora poderá utilizar a versão 3.0, mais recente e que traz novas ferramentas, mas que não é retrocompatível com as versões anteriores do módulo.

Nesta situação, não é possível ter os requisitos de ambos os projetos satisfeitos simultaneamente, pois ou instalamos a versão 1.3 para que o projeto A possa rodar ou instalamos a versão 3.0 para desenvolver o projeto B, e não conseguimos mais rodar o projeto A.

Para isso podemos criar um ambiente virtual, que nada mais é do que uma pasta com uma cópia da instalação do Python, na qual serão instalados os módulos e que será utilizada para rodar o projeto. Cada projeto ou aplicação passa então a ter o seu próprio ambiente virtual, de modo que evitamos qualquer tipo de conflito entre suas dependências.

Outra vantagem que decorre do uso de ambientes virtuais é que podemos manter a nossa instalação principal do Python limpa, apenas com os módulos principais e mais genéricos, como por exemplo o IPython, que é uma Shell alternativa ao IDLE.

Para criar um ambiente virtual do Python, podemos usar o módulo integrado venv, como mostra a Codificação 8.1.

Codificação 8.1: Sintaxe para criação de um ambiente virtual do Python

Code:

> <executável do python> -m venv <nome do ambiente a ser criado>

A Codificação 8.2 mostra alguns exemplos desse comando para Windows e sistemas Unix (Linux e Mac).

Codificação 8.2: Exemplos de criação de um ambiente virtual

Code:

> py -m venv meu-venv # Windows (1)

> C:\Program Files\Python39\python -m venv venv39 # Windows (2)

> C:\Program Files\Python37\python -m venv venv37 # Windows (3)

> phython3 -m venv meu-venv # Linux e Mac (1)

> /usr/bin/phython3.5 -m venv venv35 # Linux e Mac (2)

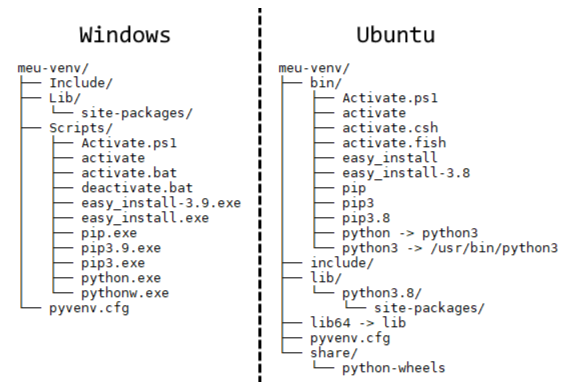
> /usr/bin/phython3.8 -m venv venv38 # Linux e Mac (3)

Nos exemplos da Codificação 8.2 marcados com (1), será criado um ambiente virtual com a versão que estiver configurada na variável de ambiente PATH do sistema, em geral é a versão mais recente do Python. Nos exemplos marcados com (2) e (3), estamos passando o caminho completo para o binário (executável) do Python, portanto o ambiente virtual será criado com uma cópia do binário usado para sua criação. Dessa forma podemos controlar de maneira muito fácil qual versão do Python será usada em cada aplicação, bastando que tenhamos tal versão já instalada em nosso computador.

A Figura 8.1 mostra uma comparação entre a hierarquia inicial das pastas do ambiente virtual criado em um sistema Windows e Ubuntu, com a execução dos comandos > py -m venv meu-venv e > python3 -m venv meu-venv, respectivamente.

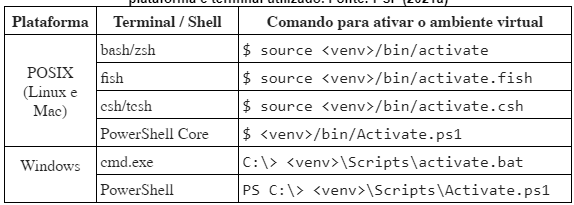
Nela podemos notar que há uma pasta com o executável do Python (Scripts no Windows e bin no Ubuntu), junto com alguns outros scripts que falaremos em breve, e há também uma pasta para a instalação dos pacotes que forem adicionados a este ambiente (Lib/site-packages no Windows e lib/python3.8/site-packages no Ubuntu).

A estrutura de diretórios criada para o ambiente virtual pode sofrer variações de acordo com o sistema operacional ou a versão do Python, mas o único momento que iremos acessar diretamente uma destas pastas é para ativar o ambiente virtual, portanto essas diferenças não são importantes. A interface que utilizaremos para instalar os pacotes será a mesma independente do sistema operacional ou versão do Python, e essa é uma das vantagens da portabilidade do Python, que irá internamente realizar as operações corretas em cada situação.



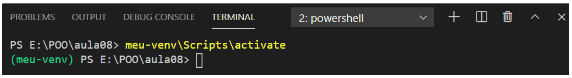
Após criado, o ambiente virtual precisa ser ativado para que possamos começar a utilizá-lo. O processo de criação apenas faz a cópia do arquivo binário do Python e demais arquivos necessários para seu funcionamento. O modo de ativar o ambiente virtual vai depender novamente do sistema operacional e de qual terminal, também chamado de Shell, estamos usando, como mostrado na Tabela 8.1 (PSF, 2021a).

Tabela 8.1: Comandos para ativar o ambiente virtual por plataforma e terminal utilizado. Fonte: PSF (2021a)



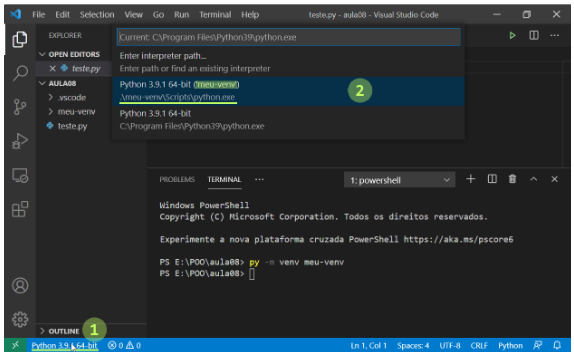
Após a ativação do ambiente virtual, a Shell irá mostrar o nome do ambiente em uso entre parênteses no começo da linha, como mostrado na Figura 8.2.

Figura 8.2: Ativação do ambiente virtual no terminal do VSCode (PowerShell)



Caso você queira executar o modo de depuração do VSCode em um ambiente virtual, deverá escolher o interpretador que será utilizado. Caso o ambiente virtual tenha sido criado na raiz da pasta que está aberta no VSCode, ele irá automaticamente identificar a existência de um ambiente virtual e podemos então selecionar o executável abrindo um arquivo Python qualquer e clicando no canto inferior esquerdo da tela, como mostra a Figura 8.3.

Figura 8.3: Seleção do interpretador do Python no VSCode.



Se você estiver com outra pasta aberta ou pretende usar um ambiente virtual que não esteja na raiz, é possível indicar manualmente o caminho para o executável, como mostrado nas Figuras 8.4 e 8.5.

Figura 8.4: Seleção manual do interpretador do Python no VSCode - parte 1

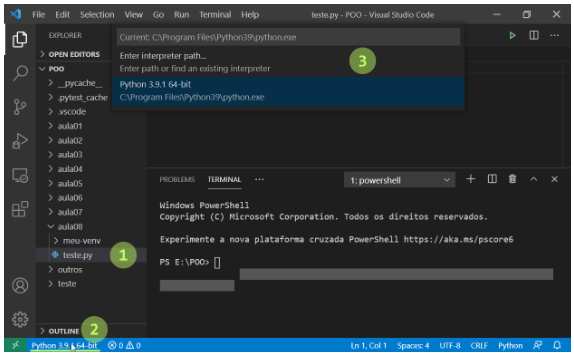
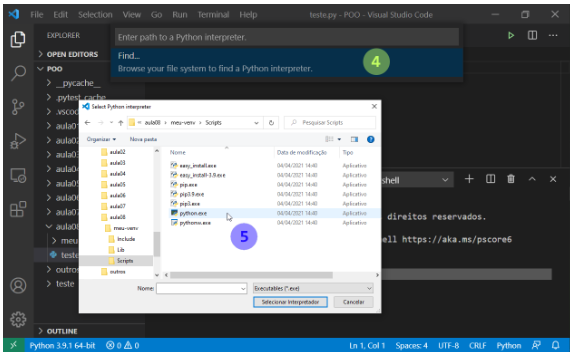


Figura 8.5: Seleção manual do interpretador do Python no VSCode - parte 2



Primeiramente crie ou abra um arquivo Python (passo 1), em seguida clique na informação sobre o interpretador, no canto inferior esquerdo (passo 2), e então clique em “entrar caminho para o interpretador”. Aqui você pode digitar o caminho se souber, ou então fazer como no passo 4 da Figura 8.4 e clicar em “encontrar” para navegar até o interpretador do Python (arquivo python.exe no Windows ou apenas python no Linux e Mac) que se encontra no seu ambiente virtual, pasta Scripts no Windows ou bin no Linux e Mac.

Após a alteração, a barra de status inferior do VSCode será atualizada para exibir o interpretador do Python selecionado.

Um nome comumente utilizado para essa pasta é .venv, em especial em sistemas Unix, nos quais o terminal trata nomes que começam com um ponto como arquivos ou pastas ocultos.

Por fim, para encerrar o ambiente virtual, apenas execute o comando deactivate, como mostrado na Figura 8.7.

Figura 8.7: Desativando o ambiente virtual.



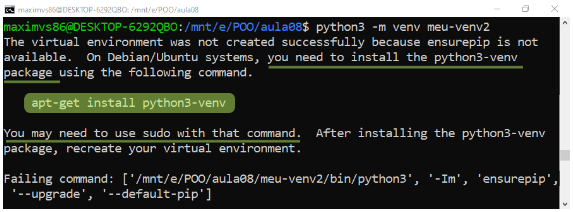
### Observações

Caso você esteja usando Linux, é possível que o módulo venv não esteja completamente instalado, então ao tentar criar o ambiente você verá um erro. Em geral o próprio sistema avisa o que deve ser feito para corrigir o problema, com uma mensagem semelhante a da Figura 8.8. Se for esse o caso, execute o comando dado como administrador (sudo) para instalar o módulo. No exemplo da figura, o comando [[19]](#footnote-18)é:

Code:

> sudo apt install python3-venv

Figura 8.8: Erro ao criar um ambiente virtual pela primeira vez em sistemas Debian/Ubuntu



Caso você esteja usando a PowerShell no Windows, é possível que precise alterar as permissões de execução de scripts para poder ativar o ambiente virtual. Para isso, abra como administrador uma janela da PowerShell independente e execute o comando da Codificação 8.2. Para executar como administrador, vá ao menu iniciar, pesquise por “PowerShell” e clique em "Executar como administrador”, como mostra a Figura 8.9.

Codificação 8.3: Alteração das permissões da PowerShell no Windows para permitir a execução do script de ativação do ambiente virtual

Code:

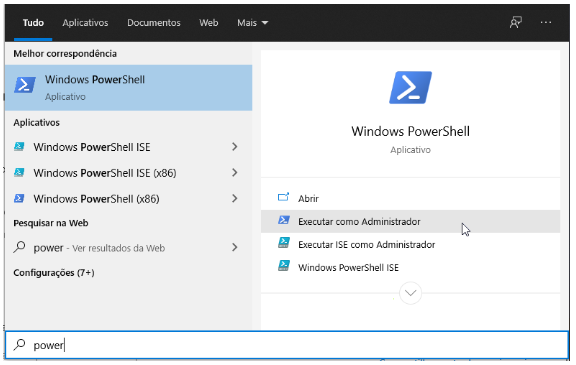
PS C:> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser

Para voltar às configurações padrão das permissões de execução de scripts na PowerShell, execute no terminal com privilégios de administrador o comando da Codificação 8.3. Lembrando que isso efetivamente impede que um ambiente virtual seja ativado via um terminal da PowerShell.

Codificação 8.4: Restabelece a configuração alterada pelo comando da Codificação 8.2 para o valor padrão

Code:

PS C:> Set-ExecutionPolicy -ExecutionPolicy Undefined -Scope CurrentUser



## Instalando pacotes com PIP

Uma vez com o ambiente virtual ativado, o gerenciador de pacotes do Python, pip, irá automaticamente instalar os pacotes e módulos no lugar certo, portanto independente de onde estamos instalando um módulo Python, seja na instalação do sistema ou em um ambiente virtual, os comandos são exatamente os mesmos.

O pip consegue instalar pacotes do índice PyPI, de urls de projetos em sistemas de versionamento (github, gitlab, etc.), projetos locais e arquivos fonte locais ou remotos. Além disso, é possível instalar com um único comando todos os pacotes listados em um arquivo texto de requisitos, comumente nomeado requirements.txt, ou alguma variação, como dev-requirements.txt, por exemplo, para indicar a finalidade do arquivo em questão.

A lista completa de comandos e opções disponíveis para gerenciar pacotes com o pip pode ser vista na documentação oficial (PyPA, 2021), mas os comandos mais utilizados são:

* pip install: para instalar um módulo ou uma lista de módulos em um arquivo de requisitos, o comando a seguir instala o módulo requests:

Code:

> pip install requests

Já o comando a seguir instala todos os módulos listados no arquivo meus\_requisitos.txt, que está na mesma pasta a partir da qual o comando está sendo executado.

Code:

> pip install -r meus\_requisitos.txt

* pip show: mostra informações sobre um módulo específico, como por exemplo:

Code:

> pip show requests

Name: requests

Version: 2.25.1

Summary: Python HTTP for Humans.

Home-page: https://requests.readthedocs.io

Author: Kenneth Reitz

Author-email: me@kennethreitz.org

License: Apache 2.0

Location: e:\poo\aula08\meu-venv\lib\site-packages

Requires: certifi, urllib3, chardet, idna

Required-by:

* pip list: lista todos os módulos atualmente instalados no ambiente em questão, como por exemplo:

Code:

> pip list

Package Version

---------- ---------

certifi 2020.12.5

chardet 4.0.0

idna 2.10

pip 20.2.3

requests 2.25.1

setuptools 49.2.1

urllib3 1.26.4

Observe que nesse ambiente foi instalado apenas o módulo requests, mas como este módulo depende de outros para funcionar, o pip os instalou automaticamente.

* pip freeze: cria uma imagem da situação atual do ambiente, listando todos os módulos que estão instalados e qual versão de cada módulo. O retorno é mostrado no próprio terminal, como a seguir:

Code:

> pip freeze

certifi==2020.12.5

idna==2.10

requests==2.25.1

urllib3==1.26.4

Observe que nesta lista não aparecem o pip e o setuptools, mas eles apareciam com o comando list, isso ocorre pois ambos os módulos são instalados pelo Python durante a criação do ambiente virtual, então não há necessidade de serem incluídos na saída deste comando.

Uma forma fácil de criar um arquivo com essa lista é usar o mecanismo da própria Shell ou terminal para redirecionar e salvar o retorno de um comando para um arquivo, que é feito com o sinal de maior, como a seguir:

Code:

> pip freeze > requirements.txt

* pip uninstall: remove um módulo diretamente ou todos os módulos listados em um arquivo. A utilização é análoga ao comando pip install, mas com o efeito inverso.

Code:

> pip uninstall requests

Found existing installation: requests 2.25.1

Uninstalling requests-2.25.1:

Would remove:

e:\poo\aula08\meu-venv\lib\site-packages

\requests-2.25.1.dist-info\\*

e:\poo\aula08\meu-venv\lib\site-packages\requests\\*

Proceed (y/n)? y

Successfully uninstalled requests-2.25.1

Se você testar o comando pip list novamente, verá que apenas o módulo requests foi removido, mas não as suas dependências. Isso ocorre porque o pip é um gerenciador de pacotes mais simples e não faz uma verificação completa de todas as dependências que não estão sendo utilizadas. Uma forma simples de remover os demais módulos é gerar um arquivo de requisitos com o pip freeze (para um arquivo remover.txt, por exemplo), editá-lo para conter apenas os módulos que deseja remover, e por fim executar o seguinte comando:

Code:

> pip uninstall -r remover.txt

## Gerenciando dependências

Para um projeto simples, com poucas dependências, é possível fazer esse gerenciamento manualmente, usando o comando pip freeze que vimos. Ele cria uma imagem exata da condição atual do ambiente de desenvolvimento, fixando as versões exatas de cada módulo instalado para que o ambiente possa ser replicado com exatidão em outra máquina ou servidor.

No entanto, conforme o projeto cresce em complexidade e número de bibliotecas externas utilizadas, fica mais difícil gerenciar manualmente a compatibilidade de todas as subdependências do projeto. Nesta situação, é recomendado o uso de outras ferramentas que fazem uma verificação muito mais detalhada das lista de pacotes requeridos, avisando por exemplo se for encontrada alguma incompatibilidade entre as dependências que não seja possível resolver de maneira automática.

A ferramenta recomendada pela documentação do Python (PSF, 2021c) para uso geral é o Pipenv, que possibilita o gerenciamento de ambientes virtuais, dependências e importação de pacotes por uma interface de linha de comando mais avançada que o pip, pois inclui também o Pipfile[[20]](#footnote-19) e o virtualenv. Ou seja, com uma única ferramenta é possível gerenciar a criação de ambientes virtuais, instalação e verificação de dependências e a lista de pacotes do ambiente. Para aprender mais sobre o uso do Pipenv, leia a documentação citada neste parágrafo.

Caso o Pipenv não atenda as necessidades do projeto, o que pode ocorrer por diferentes motivos, a documentação recomenda também outras ferramentas similares:

* poetry: é uma ferramenta muito semelhante ao Pipenv, mas com foco em projetos que serão distribuídos como pacotes do Python.
* hatch: ferramenta com mais opções, como incremento de versão, aplicação de tags de lançamento e criação de templates.
* pip-tools: ferramenta para construção de processos de gerenciamento de dependências personalizados.
* micopipenv: Ferramenta que adiciona suporte para trabalhar com arquivos das demais ferramentas em um mesmo projeto, possibilitando a conversão entre os arquivos Pipenv, Poetry lock, requirements.txt e arquivos compatíveis com pip-tools.

# Introdução ao Desenvolvimento Guiado por Testes (TDD); testes unitários e tratamento de erros

Vimos em LP que linguagens de programação fazem parte do que chamamos de linguagens formais, pois possuem um conjunto definido de regras e não permitem nenhuma margem para ambiguidades. Portanto um computador irá executar exatamente aquilo que lhe é instruído, e pode nos dizer facilmente quando há uma instrução que não segue as regras da linguagem ou que tenta realizar uma operação não permitida, erros de sintaxe e execução, respectivamente. Já com relação a erros de lógica, somos nós os responsáveis por garantir que nosso código esteja fazendo a coisa certa.

Por exemplo, imagine que ao fazer um depósito na sua conta, o programa do banco tente adicionar um valor que não seja um número. Isso configuraria um erro de execução, pois não é possível somar um número a um não-número em Python, e o computador nos informaria desse erro.

Imagine agora outro erro, em que o programa subtraia o valor depositado do seu saldo ao invés de somá-lo. Isso caracteriza um erro de lógica, também chamado de erro de semântica, pois há uma divergência entre o que o código faz e o que esperávamos que fosse feito. Neste caso, computador não tem como saber que isso está errado, já que ele executa fielmente suas instruções, então se há um erro de lógica, somos nós os responsáveis por encontrá-lo e corrigi-lo.

## Desenvolvimento Guiado por Testes (TDD)

O desenvolvimento guiado por testes é uma técnica para o desenvolvimento de software em que os requisitos são convertidos em testes antes de começarmos a programar o software propriamente dito. Ou seja, primeiro escrevemos os testes e só depois escrevemos o código que deverá cumprir as tarefas para as quais os testes foram escritos.

Essa metodologia foi apresentada por Kent Beck (2002) no começo dos anos 2000. Segundo o próprio autor (BECK, K. 2012), ele apenas redescobriu essa técnica, que já era usada por programadores desde a época dos cartões perfurados.

### Ciclos do TDD

O TDD é constituído por ciclos de desenvolvimento e cada ciclo possui 3 fases: Red, Green, Refactor. A última fase é também conhecida como a fase azul (Blue). Na fase vermelha, escrevemos os testes para a nova funcionalidade (que não existe ainda), na fase verde escrevemos o código que implementa a funcionalidade sendo testada e na fase azul refatoramos o código, aprimorando a solução inicial da fase verde. Podemos descrever mais detalhadamente cada fase como:

* RED: Adicionamos um novo teste para a nova funcionalidade e rodamos todo o conjunto de testes. Nessa fase o teste deverá obrigatoriamente falhar, pois a funcionalidade ainda não foi implementada. Se o teste passar, há algo errado com ele e precisamos refatorá-lo até que ele falhe.
* GREEN: Implementamos a nova funcionalidade e executamos os testes até que o novo teste passe, sem quebrar nenhum outro teste. O objetivo aqui é escrever a solução mais simples possível que faça o teste passar, sem precisar se preocupar com padrões de projeto, com os princípios do SOLID, com deixar o código elegante ou com a eficiência do nosso algoritmo, tudo isso ficará para a etapa de refatoração do código.
* BLUE: Com o teste passando, sabemos duas coisas: 1) a nova funcionalidade está implementada; 2) não introduzimos nenhuma modificação que quebre outras funcionalidades. O objetivo nesta fase é melhorar desempenho, legibilidade e manutenabilidade do código, e com os testes, agora temos segurança para fazer melhorias na implementação, que podem envolver, entre outros:
  + renomear as variáveis e melhorar a legibilidade do código;
  + aplicar os princípios do [SOLID](#_40vj59tmpss9) que sejam relevantes;
  + reorganizar o código de acordo com a responsabilidade de cada classe;
  + aplicar um ou mais padrões de projeto que sejam pertinentes;
  + remover código duplicado e valores “hard-coded[[21]](#footnote-20)”;
  + subdividir os métodos e funções se necessário;

### Principais vantagens do TDD

O ciclo do TDD é então repetido para cada nova funcionalidade que precisamos implementar. No começo pode parecer contraintuitivo programar primeiro o teste, mas com o tempo irá se tornar natural e no médio e longo prazo, traz muitos benefícios, como por exemplo:

* Código mais simples e fácil de ler e dar manutenção;
* Código mais flexível e adaptável, com menos acoplamentos e interfaces mais claras e classes mais direcionadas. De certa forma, podemos dizer que a prática correta do TDD automaticamente leva à aplicação do princípio da responsabilidade única (o [S](#_uyalee167km4) do SOLID).
* Maior atendimento dos requisitos, uma vez que precisamos pensar nos testes antes de escrever o código, acabamos fazendo um exercício maior de interpretação dos requisitos e melhoramos o nosso entendimento sobre o comportamento da nova funcionalidade.
* Segurança ao desenvolver, pois sabemos que caso nossa implementação quebre alguma outra funcionalidade, seremos avisados pelos testes ainda durante o desenvolvimento e poderemos corrigir o problema antes de introduzir bugs silenciosamente na aplicação.

### Limitações do TDD

Assim como qualquer outra metodologia de desenvolvimento, o TDD também apresenta algumas limitações, como por exemplo:

* O TDD, por usar principalmente testes unitários, não é capaz de testar completamente as partes da aplicação que exigem outros tipos de testes, como por exemplo conexões com um banco de dados, configurações de rede no ambiente, interfaces de usuário, etc. Nestes casos, devemos testar a parte lógica interna da nossa aplicação, usando um *mock [[22]](#footnote-21)*para representar as partes externas às quais não temos controle.
* Para que as vantagens do TDD possam ser aproveitadas, toda a equipe envolvida no projeto precisa ter conhecimento da metodologia, caso contrário é possível que parte da equipe encare os testes como perda de tempo.
* Projetos com um grande número de testes podem passar uma falsa sensação de segurança, e isso pode levar a equipe a negligenciar outros tipos de testes.
* O conjunto de testes precisa de manutenção, assim como o restante do código, então testes mal escritos podem acabar aumentando o custo de manutenção da aplicação, pois será necessário mais tempo de desenvolvimento para corrigi-los. Um exemplo é o uso de testes com mensagens de erro escritas como strings literais no código, que poderão eventualmente falhar após uma atualização de algum módulo na qual as mensagens de erro sejam alteradas.
* O desenvolvedor irá escrever tanto os testes quanto o código, então caso haja um erro de interpretação dos requisitos do projeto, é provável que tanto o código quanto os testes compartilhem esse mesmo erro e isso passe sem ser percebido. Portanto é necessário estar atento a esse tipo de problema, sendo que ter o código/testes revisados por outros desenvolvedores ajuda a mitigá-lo.

## Testes Unitários e Testes Automatizados

O exemplo dado na introdução é simples, e pode parecer besta, mas em situações reais, as aplicações e programas são mais complexos, com diversos módulos diferentes que interagem entre si de maneira muitas vezes não linear e interdependente. Sendo assim, nem sempre conseguimos garantir que uma alteração em um trecho de código, não irá afetar outras áreas da aplicação de maneira inesperada.

Para resolver isso, após alterarmos alguma coisa no código, podemos testar nossa aplicação para garantir que tudo esteja funcionando como antes, mas conforme a aplicação cresce, fazer tais testes manualmente demandará cada vez mais tempo e eventualmente se tornará inviável.

A resposta óbvia é, portanto, a utilização de testes automatizados. Existem diversos tipos de testes, mas agora vamos estudar os testes unitários, ou testes de unidade. Testes unitários servem para testar uma unidade isolada de código, para garantir que ela faz aquilo a que se propõe, isso pode ser o teste de uma função, de um método, da criação de um objeto com as propriedades corretas, etc.

Vamos criar uma função que recebe um valor inteiro e nos retorna esse valor mais 2. Para isso crie o arquivo “meu\_modulo.py” na pasta aula09 e digite o código da Codificação 9.1:

Codificação 9.1: Exemplo de função que retorna número recebido mais dois

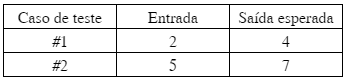
Code:

def soma\_2(numero):

return numero \* 2

Talvez você tenha percebido que há um erro na função. Esse erro é proposital e vamos supor que tenha passado despercebido por hora. Como sabemos o que a função deve fazer, podemos pensar em alguns casos de teste para verificar seu funcionamento, como mostra a Tabela 9.1.

Tabela 9.1: Casos de teste para a função soma\_2



Agora adicione o trecho de código da Codificação 9.2 ao arquivo anterior, para testar nossa função.

Codificação 9.2: Código de teste inicial

Code:

test1 = soma\_2(2) == 4

test2 = soma\_2(5) == 7

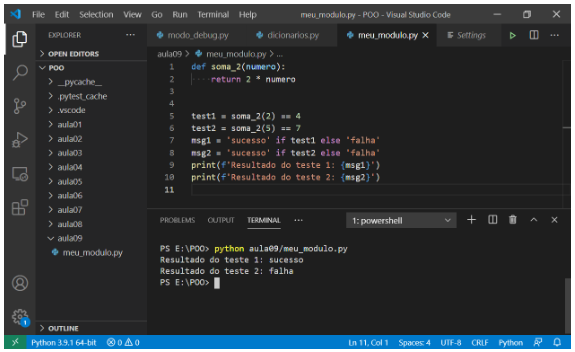
msg1 = 'sucesso' if test1 else 'falha'

msg2 = 'sucesso' if test2 else 'falha'

print(f'Resultado do teste 1: {msg1}')

print(f'Resultado do teste 2: {msg2}')

Neste código estamos guardando o resultado das verificações em uma flag booleana, uma variável que guarda um valor booleano (True ou False) referente ao status de algo em nosso código. Em seguida usamos o operador ternário para decidir qual mensagem deve ser exibida na tela, e por fim exibimos as mensagens do resultado de cada teste. A Figura 9.1 mostra o resultado da execução deste código no terminal.



Codificação 9.3: Comando utilizado na Figura 9.1 e respectiva saída no terminal

Code:

> python aula04/meu\_modulo.py

Resultado do teste 1: sucesso

Resultado do teste 2: falha

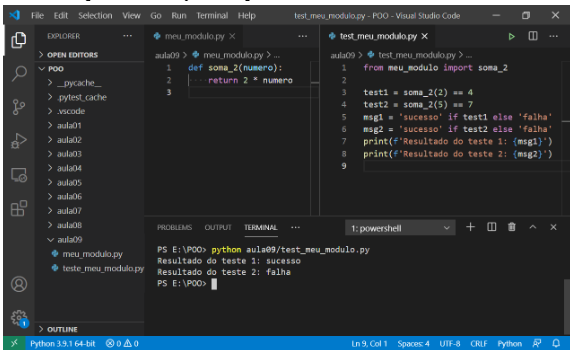
A primeira coisa que podemos fazer para melhorar nossos testes é criá-los em um arquivo separado, pois não é uma boa prática deixar nossas funções e nossos testes em um mesmo arquivo. Portanto crie na mesma pasta um arquivo chamado “test\_meu\_modulo.py” e copie para lá o código dos testes. Para executá-los, precisamos acessar a nossa função que continua no arquivo original, então para isso vamos importá-la colocando na primeira linha do arquivo a instrução da Codificação 9.4.

Codificação 9.4: Instrução para importação da função soma\_2 no arquivo de teste

Code:

from meu\_modulo import soma\_2

A criação e importação de módulos em Python é um assunto já visto em outra aula, mas da forma acima, não precisamos alterar o código do teste, pois estamos importando a partir de “meu\_modulo.py” a função soma\_2. O resultado pode ser visto na Figura 9.2.



Conseguimos separar as responsabilidades de cada arquivo, então agora poderíamos tentar melhorar as mensagens exibidas, incluindo informações dos valores esperados e obtidos, entre outras. No entanto, fazer isso para todos os testes nos daria muito trabalho extra com configuração e formatação, tirando o foco do desenvolvimento dos testes que verificam a lógica da aplicação em si.

Para nos ajudar nisso, vamos usar o pytest, uma ferramenta de testes automatizados que simplifica a construção e execução dos testes.

### Instalando o Pytest

A instalação do pytest é feita através do gerenciador de pacotes do Python, que já usamos em aulas passadas, o PIP. Execute no terminal o comando da Codificação 9.5 referente ao sistema operacional que estiver utilizando.

Codificação 9.5: Comando para instalação do pytest

> py -m pip install -U pytest # Windows

> python3 -m pip install -U pytest # Linux e Mac

Caso ao final da instalação o pip mostre um aviso (warning) informando que o caminho para a pasta de pacotes do Python não está na PATH do sistema[[23]](#footnote-22), adicione tal pasta a variável de ambiente PATH do seu usuário. Em seguida, verifique se a instalação foi bem sucedida conferindo a versão instalada, como na Codificação 9.6.

Codificação 9.6: Verificação da instalação correta do módulo pytest

> pytest --version

pytest 6.2.2

Caso o comando da Codificação 9.6 não funcione corretamente, tente executar o comando da codificação 9.7. Se isso tampouco funcionar, é possível que sua instalação do Python esteja com algum erro ou que não tenha sido adicionado à PATH do sistema (para Windows). Às vezes, reinstalar o Python pode resolver o problema.

Codificação 9.7: Verificação da instalação correta do módulo pytest

> py -m pytest --version # Windows

> python3 -m pytest --version # Linux e Mac

pytest 6.2.2

Ao executar o pytest, ele irá automaticamente buscar na pasta atual e em todas as subpastas por arquivos da forma “test\_\*.py” ou “\*\_test.py”, ou seja que comecem ou terminem com a palavra test seguida ou precedida, respectivamente, de um sublinhado[[24]](#footnote-23). Após encontrar os arquivos que correspondem ao padrão de nome, o pytest busca os testes em si, que podem ser funções ou classes. Vamos começar vendo a criação de testes usando funções.

O pytest irá executar todas as funções que começarem com “test\_”, caso a função execute até o final sem levantar nenhum erro, o teste passa, se ocorrer qualquer erro durante a execução da função, o teste falha, e a informação do erro é exibida junto com os resultados do teste.

### Escrevendo funções de teste com o Pytest

Para escrever nossos testes, usamos o comando assert para verificar se a condição que queremos é atendida ou não, ele nos permite avaliar qualquer expressão lógica e levanta um erro (do tipo AssertionError) caso ela seja avaliada para falso. É possível gerar erros personalizados, como veremos mais adiante na seção 9.4.

Abra uma Shell do Python e pratique o uso do assert. Siga os exemplos da Codificação 9.8.

Codificação 9.8: Exemplos de uso do comando assert do Python

Code:

>>> assert True

>>> assert 3 == 2 + 1

>>> assert False

As duas primeiras expressões avaliam para True, portanto não há nenhum retorno do comando assert, já a última avalia para False e levanta o erro mostrado na Codificação 9.9.

Codificação 9.9: Exemplo do erro levantado pelo comando assert

Code:

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AssertionError

O pytest captura o erro e faz uma introspecção para avaliar o motivo que o gerou, mostrando o resultado na tela para nós. Vamos então alterar nosso código criando duas funções de teste. Altere o código do arquivo “test\_meu\_modulo.py” para corresponder à Codificação 9.10.

Codificação 9.10: Funções de teste do arquivo test\_meu\_modulo.py

Code:

from meu\_modulo import soma\_2

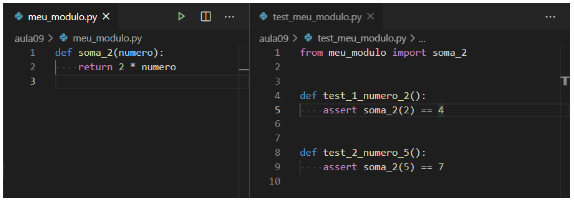
def test\_1\_numero\_2():

assert soma\_2(2) == 4

def test\_2\_numero\_5():

assert soma\_2(5) == 7

A Figura 9.3 mostra ambos os arquivos no VSCode, lembrando que para o teste funcionar da forma que fizemos, eles precisam estar na mesma pasta.



### Executando os testes com o Pytest

O VSCode nos traz alguns atalhos e consegue gerenciar a execução dos testes, de modo que podemos escolher executar um por um ou todos de uma vez, mas é importante aprender o funcionamento usando a linha de comando, assim não ficamos dependentes de uma ferramenta específica. Para executar o pytest no terminal, há pelo menos três maneiras diferentes:

1. > pytest

Digitando apenas o comando no terminal, o pytest irá buscar e executar todos os testes da pasta atual e suas subpastas.

1. > pytest caminho/para/arquivo\_test.py

Digitando o comando no terminal seguido do nome do arquivo que queremos executar, o pytest irá executar apenas os testes contidos neste arquivo, sem fazer a busca por arquivos de teste e, portanto, nesse caso o nome do arquivo não importa.

1. > py -m pytest # Windows

> python3 -m pytest # Linux e Mac

Precedendo qualquer uma das opções anteriores por python -m fará com que o pytest seja executado a partir do interpretador do python, como um módulo. Dependendo da forma como o pytest tenha sido instalado, pode ser que o seu comando, como mostrado nos itens 1 e 2, não seja reconhecido no terminal, então a forma do item 3 pode ser uma solução mais rápida que atualizar a PATH ou reinstalar o Python.

Um guia completo da utilização do pytest no terminal de comandos pode ser visto na documentação oficial (KREGEL, 2020).

### Interpretando os resultados do Pytest

Após a execução do pytest, será exibido no terminal um resumo sobre os testes encontrados e executados, uma saída com detalhes do erro para cada teste que falhou e por fim uma versão resumida dos erros nos testes que falharam. Vejamos a saída para o exemplo dado acima.

#### Resumo dos testes coletados

Codificação 9.11: Resumo dos testes do arquivo test\_meu\_modulo.py

Code:

PS E:\POO> pytest

========================= test session starts ========================

platform win32 -- Python 3.9.1, pytest-6.2.2, py-1.10.0, pluggy-0.13.1

rootdir: E:\POO

collected 2 items

aula09\test\_meu\_modulo.py .F [100%]

Podemos ver as informações da plataforma (sistema operacional) e versões do interpretador do Python e do pytest, diretório raiz a partir do qual o comando de testes foi executado, a quantidade de testes encontrados e coletados pela busca automática, e por fim, na última linha temos:

* o(s) arquivo(s) de teste executado(s);
* um “ponto” para testes bem sucedidos e uma letra F para testes com falha;
* a porcentagem de testes executados, independente de terem passado ou não.

Caso sejam encontrados mais de um arquivo de testes, haverá uma linha para cada um.

#### Saída detalhada dos testes

Codificação 9.12: Saída detalhada dos testes do arquivo test\_meu\_modulo.py

Code:

============================== FAILURES ==============================

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_2\_numero\_5 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

def test\_2\_numero\_5():

> assert soma\_2(5) == 7

E assert 10 == 7

E + where 10 = soma\_2(5)

aula09\test\_meu\_modulo.py:9: AssertionError

Podemos ver aqui uma seção com o nome do teste que falhou, em seguida a linha de definição deste teste e a introspecção do assert que levantou o erro, indicando tanto o código da linha que levantou o erro quanto os valores que foram avaliados durante a execução do teste. Por fim, temos novamente o nome do arquivo, mas dessa vez seguido de um número, que representa a linha do código em que houve o erro durante o teste, e o tipo do erro.

Nesse caso, podemos concluir que a chamada à função soma\_2, com o valor 5 retornou o valor 10, que quando comparado com o valor esperado 7, resultou em False e por isso o erro no assert.

#### Saída resumida dos testes

Codificação 9.13: Saída resumida dos testes do arquivo test\_meu\_modulo.py

Code:

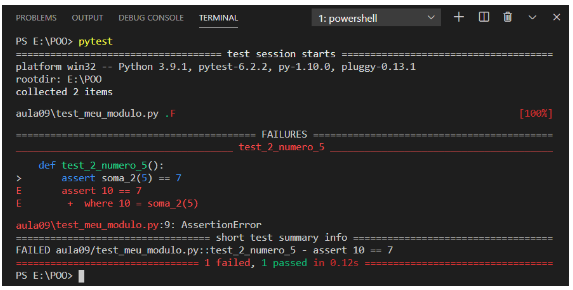
====================== short test summary info =======================

FAILED aula09/test\_meu\_modulo.py::test\_2\_numero\_5 - assert 10 == 7

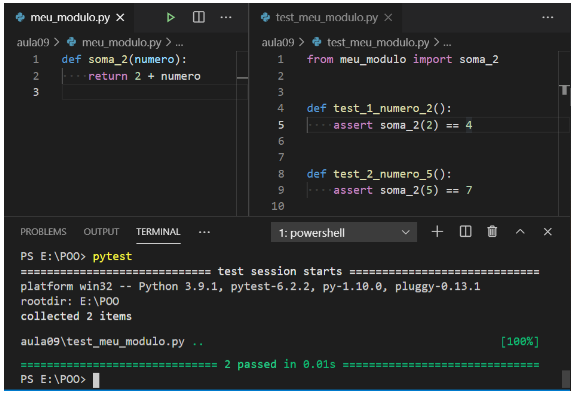
==================== 1 failed, 1 passed in 0.29s =====================

Na saída resumida, vemos uma lista de testes que falharam, que inclui o nome do arquivo, o nome do teste e a comparação que falhou, e em seguida um resumo geral da execução, indicando quantos testes falharam e passaram, e o tempo de execução. A Figura 9.4 mostra a saída vista no terminal do VSCode.

Figura 9.4: Visualização da saída dos testes no terminal do VSCode



Após analisar o resultado dos nossos testes, vemos que a função soma\_2 está incorreta, pois ao receber o valor 5, não retornou o valor esperado que era 5 + 2 → 7. Portanto, vamos corrigir o código e executar novamente os testes. Veja a Figura 9.5.



No momento de criar um teste, não nos importa a implementação da função, apenas o que ela faz, isto é, o que ela precisa receber e o que irá retornar. Com isso podemos ver um pouco da importância de se construir casos de teste adequados e de estudar o motivo que fez os testes não passarem.

Um outro ponto muito importante é que os testes são uma ferramenta para nos auxiliar, mas não são absolutos e não estão acima do nosso discernimento como programadores ao avaliar o código, pois mesmo estando incorreta, a primeira versão da função passou em um teste.

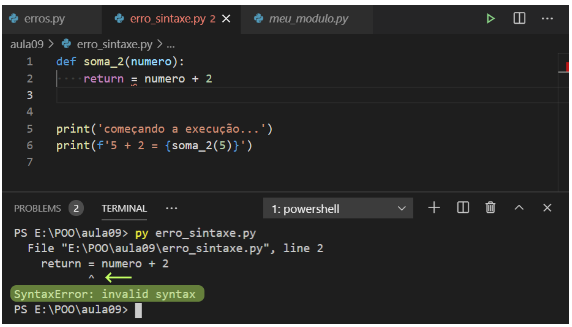
Devemos pensar nos casos de testes de maneira a cobrir o maior número de situações diferentes que seja viável. Tente pensar em outros testes para a função acima. Há outros valores que podem gerar situações semelhantes à do primeiro teste?

## Erros e exceções

Vimos em LP que podemos agrupar os erros em programação em três categorias, de acordo com sua origem quando eles ocorrem:

* Erros de sintaxe: são erros em que o interpretador não é capaz de entender as instruções pois elas não seguem as regras da linguagem;
* Erros de execução: ocorrem quando as regras estão corretas, mas por algum motivo não é possível executar a ação pedida;
* Erros de semântica: são erros na lógica implementada. Do ponto de vista do interpretador, nada está errado e a execução é bem sucedida, mas a ação realizada não condiz com o comportamento esperado.

No primeiro tipo de erro, os de sintaxe, o interpretador indica o arquivo e a linha na qual ele encontrou o erro, e inclui também um marcador na região em que o erro foi detectado. Os erros de sintaxe devem ser corrigidos para que o código possa ser executado, e como é um erro que está escrito diretamente no código fonte, é possível garantir que todos eles sejam eliminados. Veja no exemplo da Figura 9.6 que nenhum código é executado, pois o interpretador não é capaz de começar a execução já que estamos quebrando uma das regras de sintaxe do Python, no caso, estamos tentando atribuir um valor a uma palavra reservada da linguagem, o comando return.



Já com erros de execução, o segundo tipo, não é possível eliminá-los, pois muitas vezes eles ocorrem em virtude de elementos que fogem ao controle do programador, como por exemplo, uma falha na conexão com o banco de dados, uma entrada incorreta do usuário, uma falha de comunicação com uma API, etc. A estes erros, normalmente damos o nome de exceções, e para evitar que eles façam a nossa aplicação parar de funcionar, existe o tratamento de exceções, que veremos em seguida, na seção 9.4.3.

E por fim, em relação ao terceiro tipo de erro, os de semântica, a melhor forma de lidar com eles é a criação de testes automatizados, como começamos a ver neste capítulo, e que irão testar o funcionamento correto de uma aplicação ou software, com base no comportamento esperado definido no teste. Neste caso, é preciso que o programador escreva os testes manualmente, pois o interpretador não tem como saber o que está errado, dado que a execução é finalizada com sucesso, independentemente de produzir ou não o resultado esperado.

Em relação aos erros de execução (o segundo tipo), podemos distinguir duas situações: o lançamento ou levantamento de uma exceção e o seu respectivo tratamento. Ambas as partes não precisam acontecer sempre, por exemplo, em um trecho da aplicação, podemos apenas levantar exceções que serão tratadas (ou não) pelo código cliente[[25]](#footnote-24), e em outra podemos apenas tratar exceções que são levantadas por algum módulo que estamos usando (aqui nós seríamos o cliente). Ou podemos ter as duas coisas acontecendo na mesma aplicação: uma parte do código levanta uma exceção que será tratada em outra parte da mesma aplicação.

### Lançamento de exceções

Para lançar uma exceção em Python usamos a palavra reservada raise[[26]](#footnote-25), e podemos lançar qualquer exceção da lista de exceções integradas à linguagem, que pode ser vista na documentação (PSF, 2021b). Veja o exemplo da Codificação 9.14.

Codificação 9.14: Lançamento de uma exceção do tipo TypeError

Code:

def incrementa\_int(n):

if not isinstance(n, int):

raise TypeError('n deve ser um inteiro')

return n + 1

Neste exemplo, caso a função seja chamada com um valor que não seja um número inteiro, o fluxo de execução será interrompido pelo comando raise, que irá encerrar a função e “retornar” uma mensagem de erro. Esse retorno não é o mesmo retorno que ocorre quando usamos a palavra chave return, ou seja, não é possível atribuí-lo a uma variável, e ele continuará “subindo” na pilha de execução até que seja tratado ou chegue no módulo principal, momento em que o processo em execução é encerrado prematuramente, mostrando na tela a mensagem de erro.

Para entender um pouco melhor o que significa o erro “subir” ou ser “elevado”, podemos pensar que toda vez que chamamos uma função dentro de outra, a função atual fica “pausada”, esperando a função chamada encerrar a sua execução e retornar. Caso a função retorne seu valor normalmente, o fluxo de execução continua. Caso ela retorne um erro e este erro não seja tratado, a função que recebeu como “retorno” o erro repassa-o para quem a chamou, e assim sucessivamente. Veja o exemplo da codificação 9.15, na qual a função calcula\_idade é chamada com um número do tipo float como argumento.

Codificação 9.15: Código com lançamento de um erro do tipo TypeError

Code:

def incrementa\_int(n):

if not isinstance(n, int):

raise TypeError('n deve ser um inteiro')

return n + 1

def calcula\_idade(idade):

nova\_idade = incrementa\_int(idade)

print('esse código não é executado se der erro na linha acima')

return nova\_idade

def main():

print('executando a função principal...')

resposta = calcula\_idade(20.5)

print('esse código não será executado se der erro na linha acima')

print('a nova idade é:', resposta)

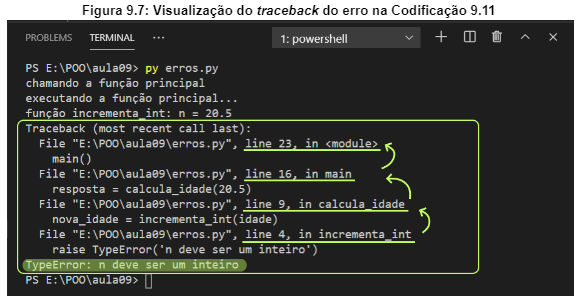
if \_\_name\_\_ == '\_\_main\_\_':

print('chamando a função principal')

main()

print('esse código não será executado se der erro na linha acima')

O erro que ocorreu na função incrementa\_int foi elevado para a função calcula\_idade, que por sua vez foi elevado para a função main, e por fim foi elevado para o programa principal, de onde a função main foi chamada. Como em nenhum momento fizemos o tratamento deste erro, a execução do programa foi interrompida e tal informação é mostrada na tela, onde conseguimos ver todo o caminho percorrido pelo erro, no que chamamos, em inglês, de Traceback, como mostra a figura 9.7.



No exemplo que acabamos de ver, faz sentido lançar um TypeError, pois estamos falando exatamente de um erro de tipo (o dado fornecido não é do tipo correto). No entanto, nem sempre o Python terá uma classe que seja adequada para representar as exceções que estamos prevendo em nossa aplicação, portanto é possível criarmos nossas próprias exceções, usando o conceito de herança que vimos na introdução a POO.

### Criação de exceções

Em Python toda exceção deve herdar da classe Exception, ou de outra classe que por sua vez herde de Exception. Isto é, toda nova exceção deve ser uma descendente de Exception (filha, neta, etc.), o que garante que o nosso erro ou exceção personalizado irá herdar todos os comportamentos mínimos necessários para que o Python possa identificá-la como uma exceção. E é especialmente importante para que um bloco except Exception seja capaz de pegar todas as exceções que não sejam específicas, como veremos a seguir na seção 9.4.3.

Imagine que estamos desenvolvendo um sistema de cadastro para uma clínica veterinária, uma funcionalidade que precisaremos implementar será a de criar a ficha dos animais que serão ali tratados, o que podemos modelar em uma classe Paciente. Caso aconteça alguma falha durante a criação, como por exemplo se o campo “nome” não for uma string ou estiver vazio, podemos levantar um erro neste método. Veja na Codificação 9.16 um exemplo de implementação simplificada da classe Paciente, na implementação real desta classe, o método inicializador teria mais parâmetros.

Codificação 9.16: Criação e utilização de exceção personalizada

Code:

class NameIsEmptyError(Exception):

pass

class Paciente:

def \_\_init\_\_(self, nome):

self.\_\_paciente = nome

... # restante do código que inicializa os dados do paciente

@property

def paciente(self):

return self\_.\_paciente

@paciente.setter

def paciente(self, nome):

if not isinstance(nome, str):

raise TypeError("'nome' inválido")

if nome == '':

raise NameIsEmptyError("'nome' é obrigatório")

self.\_\_paciente = nome

Na classe Paciente, da codificação 9.16, estamos usando uma property/setter para atribuir o valor do atributo não público \_paciente. E no setter, fazemos uma validação do valor recebido e só permitimos a atribuição caso este valor seja uma string e não seja vazio. É comum fazermos a verificação logo no começo e levantarmos os erros pertinentes, interrompendo o fluxo de execução. Dessa forma garantimos que o código que irá consumir esta classe irá “falhar” o mais cedo possível caso use-a de maneira incorreta, permitindo ao desenvolvedor perceber e corrigir o erro.

### Tratamento de exceções

O tratamento de exceções é extremamente importante para garantir que a aplicação ou programa continue funcionando ao encontrar algo inesperado em relação ao seu funcionamento normal. Ele captura o erro, impedindo que ele continue subindo na pilha de execução, e permite desvios no fluxo para que sejam tomadas as medidas necessárias em cada caso, o que pode incluir enviar mensagens aos usuários, escrever mensagens de log, reagendar a tarefa que falhou, fechar um arquivo aberto ou encerrar a conexão com o banco de dados, entre outros.

Observe que essa situação pode ser inesperada do ponto de vista do funcionamento padrão, mas do ponto de vista de quem está programando a aplicação, as falhas são sempre esperadas, pois em uma situação real, não é possível controlar todo o ambiente e garantir que o usuário não irá digitar um valor inválido, que o servidor do banco de dados não irá passar por uma instabilidade ou que a conexão não será interrompida porque uma API estava sobrecarregada e demorou demais para responder. Portanto, faz parte do nosso trabalho identificar os pontos suscetíveis a falha e implementar o tratamento adequado.

Em Python, esse tratamento é feito com o bloco try-except, como mostra a Codificação 9.17.

Codificação 9.17: Sintaxe do bloco try-except em Python

Code:

try:

# código suscetível a falha

except:

# código executado após ocorrer um erro

else:

# código executado apenas se nenhum erro ocorrer

finally:

# código executado sempre

Durante o tratamento de uma exceção, o único trecho de código seguro é aquele contido no bloco do try, ou seja, qualquer erro que ocorrer ali será capturado e o fluxo redirecionado para os blocos except. No entanto, se ocorrer um erro nos demais blocos, esse erro irá seguir o fluxo padrão de erros e será elevado na pilha de execução. É possível aninhar blocos try-except, mas em geral isso é considerado uma má prática, pois piora a legibilidade do código.

Veja na Codificação 9.18 um exemplo de código que utiliza a classe Paciente, da Codificação 9.16.

Codificação 9.18: Exemplo de uso do bloco try-except em Python

Code:

from paciente import Paciente, NameIsEmptyError

try:

nome = input('Digite o nome do paciente: ')

p = Paciente(nome)

except TypeError:

print('O nome deve ser uma string')

except NameIsEmptyError:

print('O nome não pode ser uma string vazia')

except Exception as e:

print('Ocorreu um erro inesperado ao criar o objeto')

print('informações do erro:', e)

Aqui podemos observar que há um encadeamento dos tipos de exceções que podem ser capturados, seguindo uma lógica parecida com a dos blocos elif no Python, isto é, será executado o bloco except da primeira exceção que corresponder, sendo os blocos subsequentes ignorados, pulando direto pro bloco do finally, se houver.

Devido a natureza da função input, que sempre retorna uma string, não é possível termos um erro de tipo, e devido a simplicidade deste exemplo, dificilmente teremos um erro genérico, portanto neste exemplo simplificado, poderíamos capturar apenas a exceção NameIsEmptyError.

Caso queira testar a captura dos demais erros, podemos introduzir erros propositais no código. Faça o teste substituindo, no bloco do try, a instrução do input por nome = 23, ou forçando, também no bloco do try, o levantamento de um erro genérico com raise Exception('erro genérico[[27]](#footnote-26)').

Há ainda três pontos importantes sobre o tratamento de exceções no Python:

1. Como no exemplo dado, é possível capturar o objeto de exceção levantado pelo Python em uma variável, usando o comando as para fazer a atribuição, como no exemplo da codificação 9.18.
2. É possível agrupar mais de um tipo de exceção no mesmo bloco except, colocando-as em uma tupla:

Code:

try:

...

except (TypeError, ValueError, ZeroDivisionError):

…

1. Podemos também apenas interceptar a exceção, fazer algum tipo de tratamento, como por exemplo salvar em um registro de log a ocorrência, e deixá-la seguir o seu fluxo natural para ser tratada em outra parte da aplicação. Para isso, basta usar o comando raise sozinho, dentro de um bloco except:

Code:

except:

logger.exception('salvando log da exceção')

raise

Com isso, a mesma exceção que ocorreu originalmente será relançada com o comando raise, após a execução do tratamento parcial.

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)
6. Um compilador converte o código-fonte para a linguagem de máquina de uma só vez, enquanto um interpretador faz isso linha por linha traduzindo código por código. [↑](#footnote-ref-5)
7. UML é a Linguagem Unificada de Modelagem, da sigla em inglês Unified Modelling Language. [↑](#footnote-ref-6)
8. Do inglês double underscore. [↑](#footnote-ref-7)
9. O termo atributo aqui faz referência tanto às características quanto aos comportamentos que uma classe define, pois para o Python a única diferença entre eles é que um método é um atributo “chamável”, ou seja, refere-se a um objeto que pode ser chamado, como fazemos com funções. [↑](#footnote-ref-8)
10. Para o polimorfismo, não é necessário que os parâmetros sejam nomeados, apenas que sejam opcionais. No caso do print, eles precisam ser nomeados pois o print pode receber um número variável de argumentos, então essa é a única forma de passar um argumento cujo objetivo não seja ser exibido na tela. [↑](#footnote-ref-9)
11. MRO é o acrônimo para Ordem de Resolução dos Métodos, na sigla em inglês: Method Resolution Order. [↑](#footnote-ref-10)
12. O termo pacote normalmente se refere à forma como o código é organizado para ser distribuído, mas em um contexto mais genérico, é comum usarmos os termos biblioteca, pacote ou módulo, de maneira intercambiável, para indicar um conjunto de ferramentas que lida com um tipo de problema ou domínio. [↑](#footnote-ref-11)
13. CPython é a implementação padrão do Python, escrita em C. [↑](#footnote-ref-12)
14. O modificador !r força a utilização da função repr, que gera uma representação do objeto em string, como \_\_name\_\_ já é uma string, a sua representação na tela inclui as aspas, evidenciando que é uma string, já que a função print, por padrão, exibe sempre o conteúdo da string, sem as aspas. [↑](#footnote-ref-13)
15. Para forçar a reimportação de um módulo, é necessário importar a biblioteca importlib e usar o método reload(), passando para ele o módulo a ser recarregado.

    >>> import meu\_modulo # importação inicial

    >>> import importlib

    >>> importlib.reload(meu\_modulo) # módulo recarregado [↑](#footnote-ref-14)
16. <elemento> é qualquer nome definido no escopo do módulo, pode ser um submódulo, classe, função ou variável global. [↑](#footnote-ref-15)
17. Essa é uma das principais vantagens de se trabalhar com uma linguagem de código aberto, pois o desenvolvimento de novas ferramentas não fica limitado aos recursos e interesses de uma única empresa ou companhia, e todos podem contribuir. [↑](#footnote-ref-16)
18. venv é a abreviação de Virtual Environment, em inglês. [↑](#footnote-ref-17)
19. Pode ser usado tanto apt quanto apt-get, com o mesmo resultado. [↑](#footnote-ref-18)
20. É a lista de pacotes do projeto, que o Pipenv utiliza para avaliar as dependências e gerar uma versão com as versões travadas, calculando as hashes dos arquivos e salvando todas essas informações em um arquivo Pipfile.lock, que pode ser usado em seguida para replicar a configuração do ambiente com muito mais confiabilidade que um arquivo padrão de requisitos (requirementes.txt). [↑](#footnote-ref-19)
21. Hard-coded, ou codificação rígida, é quando colocamos um valor fixo no próprio código fonte, por exemplo, quando colocamos a url para uma api como uma string no próprio código, ao invés de ler este valor de um arquivo de configurações ou variável de ambiente. [↑](#footnote-ref-20)
22. *mock* é um objeto que simula o objeto real, por exemplo, uma conexão com um banco de dados. No objeto de mock, podemos definir o comportamento esperado daquela conexão, se vai dar erro ou não, qual o erro, qual o retorno, etc. E assim podemos testar a nossa lógica interna em cada uma dessas situações. [↑](#footnote-ref-21)
23. Caso não saiba adicionar um caminho à variável de ambiente PATH, uma rápida pesquisa no google irá te trazer diversos tutoriais sobre como editar esta variável de ambiente do seu sistema operacional. [↑](#footnote-ref-22)
24. O asterisco indica que o resto do nome do arquivo não importa, podem ser quaisquer caracteres válidos para nomes de arquivos. [↑](#footnote-ref-23)
25. O termo cliente aqui se refere a qualquer programa ou código que consuma (faça uso) do nosso código. [↑](#footnote-ref-24)
26. raise pode ser traduzido para levantar ou elevar. [↑](#footnote-ref-25)
27. Observe que levantar um erro dentro de um bloco try fará com que esse erro seja capturado e tratado no próprio bloco, portanto isso não é uma prática utilizada normalmente, servindo aqui apenas para ilustrar o exemplo dado, já que no código do exemplo, não há margem para um erro genérico. Alternativamente, poderíamos ter editado a nossa classe para, ao receber um nome específico, levantar um erro genérico, este exemplo seria mais perto de um exemplo de utilização real do tratamento de exceções. [↑](#footnote-ref-26)