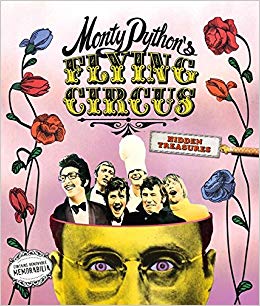
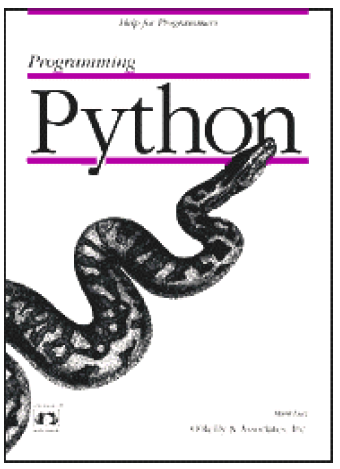
O que é o Python?

Em 1982, na cidade de Amsterdã, capital da Holanda, Guido Van Hossum, um dos desenvolvedores da linguagem de programação ABC, trabalhava no CWI (Instituto de Pesquisa Nacional para Matemática e Ciência da Computação) em um sistema operacional distribuído chamado amoeba. Devido as grandes falhas deste sistema com a linguagem C, Guido resolve então criar uma linguagem que possa resolver tais problemas. Van Hossum Queria desenvolver uma tecnologia fácil e intuitiva pois, segundo ele, determinados softwares programados em C eram bastante complexos pelo fato de, possuírem uma codificação extensa e, apenas programadores experientes conseguiam entender alguns programas escritos em C. Após Criação da nova linguagem veio a parte da nomeação. Para esse fim o CWI possuía um padrão de nomeações, que era baseado em algum nome referente a Televisão. Logo o Holandês batizou a linguagem de Python devido a seu programa favorito, o Monty Python’s Flying Circus. Até então seu nome não tinha nenhuma relação com a serpente píton, porém, o primeiro livro de programação em Python foi produzido pela editora O’Reilly, e cada livro dessa editora, possui um animal em sua capa, e o animal para este livro foi a própria serpente píton.



Na década de 90, fora finalizado o projeto Python e foi lançada em 1991 no Instituto de Pesquisa Nacional para Matemática e Ciência da Computação (CWI), nos Países Baixos. Com isso Guido Van Hossum se muda para os Estados Unidos e, inicia o projeto CP4E (Computer programming for everybody – programação de computadores para todos) financiado pela DARPA (Defense Advanced Research Projects Agency - Agência de Projetos de Pesquisa Avançada de Defesa) onde ele ensina programação para todos que tenham interesse. Em 2001 foi fundada a PSF (Python Software Foundation) que é uma organização sem fins lucrativos que, mantém e coordena um modelo de desenvolvimento comunitário, aberto do uso da linguagem. Atualmente a PSF é apoiada por grandes empresas como o Google, Microsoft e a Globo.com que também utiliza o Python nos seus sistemas.

Uma nova versão da linguagem é lançada a cada um ano e meio mais ou menos, então, na média, um sistema feito com o Python mais novo vai poder contar com correções de bugs por 3 anos, e correções de segurança por 6 anos.

|  |  |  |
| --- | --- | --- |
| Versão | Lançamento | Ferramentas |
| 0.9.0 | [1991](https://pt.wikipedia.org/wiki/1991) | [Classes](https://pt.wikipedia.org/wiki/Classe_(programa%C3%A7%C3%A3o)) com [herança](https://pt.wikipedia.org/wiki/Heran%C3%A7a_(programa%C3%A7%C3%A3o)), tratamento de exceções, [funções](https://pt.wikipedia.org/wiki/Sub-rotina) e os [tipos de dado](https://pt.wikipedia.org/wiki/Tipo_de_dado) nativos [list](https://pt.wikipedia.org/wiki/Lista), [dict](https://pt.wikipedia.org/wiki/Dicion%C3%A1rio), [str](https://pt.wikipedia.org/wiki/Cadeia_de_caracteres), e assim por diante.  O modelo de exceções e sistema de módulos emprestado do [Modula-3](https://pt.wikipedia.org/wiki/Modula-3). |
| 1.0 | janeiro de 1994 | [programação funcional](https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional) como [lambda](https://pt.wikipedia.org/wiki/C%C3%A1lculo_lambda), map, filter e reduce |
| 1.4 | 1995 | parâmetros nomeados (a capacidade de passar parâmetro pelo nome e não pela posição na lista de parâmetros) e suporte nativo a [números complexos](https://pt.wikipedia.org/wiki/N%C3%BAmero_complexo), assim como uma forma de [encapsulamento](https://pt.wikipedia.org/wiki/Encapsulamento) |
| 2.0 | 2000 | [List comprehension](https://pt.wikipedia.org/wiki/List_comprehension), uma relevante funcionalidade de linguagens funcionais como [SETL](https://pt.wikipedia.org/wiki/SETL) e [Haskell](https://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programa%C3%A7%C3%A3o)). Sistema [coletor de lixo](https://pt.wikipedia.org/wiki/Coletor_de_lixo_(inform%C3%A1tica)) capaz de identificar e tratar ciclos de referências. [Analisador sintático](https://pt.wikipedia.org/wiki/Analisador_sint%C3%A1tico) [SAX](https://pt.wikipedia.org/wiki/SAX) |
| 2.2 | 2002 | Unificação dos tipos Python (escritos em [C](https://pt.wikipedia.org/wiki/C_(linguagem_de_programa%C3%A7%C3%A3o))) e classes (escritas em Python) em somente uma hierarquia. Isto tornou o modelo de objetos do Python consistentemente orientado a objeto. Também foi adicionado [generator](https://pt.wikipedia.org/wiki/Generator_(ci%C3%AAncia_da_computa%C3%A7%C3%A3o)), inspirado em [Icon](https://pt.wikipedia.org/wiki/Icon_(linguagem_de_programa%C3%A7%C3%A3o)).  Mudança na especificação para suportar escopo aninhado, assim como outras linguagens com escopo estático |
| 2.6 | 01/10/[2008](https://pt.wikipedia.org/wiki/2008) | incluídas [bibliotecas](https://pt.wikipedia.org/wiki/Biblioteca_(computa%C3%A7%C3%A3o)) para [multiprocessamento](https://pt.wikipedia.org/wiki/Multiprocessamento), [JSON](https://pt.wikipedia.org/wiki/JSON) e [E/S](https://pt.wikipedia.org/wiki/E/S), além de uma nova forma de formatação de [cadeias de caracteres](https://pt.wikipedia.org/wiki/Cadeia_de_caracteres) |
| Python 3.0 ou Python 3000 | dezembro de 2008 | Quebra de compatibilidade com a família 2.x para corrigir falhas que foram descobertas neste padrão, e para limpar os excessos das versões anteriores.  Alteração da [palavra reservada](https://pt.wikipedia.org/wiki/Palavra_reservada) print, que passa a ser uma função, tornando mais fácil a utilização de uma versão alternativa da rotina.  Mudança para [Unicode](https://pt.wikipedia.org/wiki/Unicode) de todas as [cadeias de caracteres](https://pt.wikipedia.org/wiki/Cadeia_de_caracteres) |

A linguagem oferece suporte a desktops, desenvolvimento web, aplicações mobile, geoprocessamento, processamento de imagens, robótica, Data Science, programação para hardware (Harduíno e RaspbarryPi), desenvolvimento de games, biotecnologia e também no desenvolvimento científico, pois, trabalha com números grandes e complexos, e possui também, diversas bibliotecas para essa finalidade como Scipy e NumPy. Além de aplicações comercias, empresariais e científicas, com linguagem Python é possível produzir filmes com computação 3D, ou desenvolver games. Alguns jogos desenvolvidos com Python foi, Civilization 4, QuArK e um dos mais conhecidos, Battlefield 2.

Atualmente o Python se encontra no cotidiano de muitos usuários, pois, ele está presente em vários lugares, como por exemplo, nos buscadores do Google processando pesquisas, ou nas transmissões de vídeo do YouTube, e também nos algoritmos bem elaborados da NetFlix. A linguagem faz parte de diversas outras grandes outras empresas como Dropbox, Yahoo, Zope Corporation, Industrial Light & Magic, Walt Disney Feature Animation, Pixar, NASA, NSA, Red Hat, Nokia, IBM, Yelp, Intel, Cisco, HP, Qualcomm, e JPMorgan Chase.

|  |  |  |  |
| --- | --- | --- | --- |
| **Ano** | **Linguagem** | **Ano** | **Linguagem** |
| **1956** | Fortran I | **1984** | Common Lisp, C++. SML |
| **1958** | Lisp | **1986** | Eiffel, Perl, Caml |
| **1960** | Cobol, Algol 60 | **1988** | Tel |
| **1964** | PL/I | **1990** | Fortran 90. Python. Java |
| **1968** | Smalltalk | **1994** | Ruby, Perl 5 |
| **1970** | Pascal, Prolog | **1995** | JavaScript |
| **1974** | Scheme | **1996** | OCaml |
| **1976** | Fortran 77, ML | **1998** | Scheme R5RS, C++(ISO), Haskell 98 |
| **1978** | C (K&R) | **2000** | Python 2.0, C# |
| **1980** | Smalltalk 80 | **2004** | C# 2.0(beta), Java 2 (beta) |
| **1982** | Ada 83 | **2008** | Python 3.0 |

O Python trabalha de forma isolada, porém, com esta linguagem também é possível realizar integrações a outras linguagens, como por exemplo com Java, C, JavaScript, linguagens de marcação como Html entre outras.

Python é uma linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. A linguagem foi projetada com a filosofia de enfatizar a importância do esforço do programador sobre o esforço computacional. Prioriza a legibilidade do código sobre a velocidade ou expressividade. Combina uma sintaxe concisa e clara com os recursos poderosos de sua biblioteca padrão e por módulos e frameworks desenvolvidos por terceiros.

|  |  |  |  |
| --- | --- | --- | --- |
| **Linguagem** | **Tipagem** | **Conversões** | **Obs.** |
| Smalltalk | dinâmica | forte |  |
| Python | dinâmica | forte |  |
| Ruby | dinâmica | forte |  |
| C (K&R) | estática | fraca |  |
| C (ANSI) | estática | forte |  |
| Java | estática | forte |  |
| C# | estática | forte |  |
| JavaScript | dinâmica | fraca | Bugs sorrateiros |
| PHP | dinâmica | fraca | Bugs sorrateiros |

Devido às suas características, ela é principalmente utilizada para processamento de textos, dados científicos e criação de CGIs para páginas dinâmicas para a web. Foi considerada pelo público a 3ª linguagem "mais amada", de acordo com uma pesquisa conduzida pelo site Stack Overflow em 2018,[5] e está entre as 5 linguagens mais populares, de acordo com uma pesquisa conduzida pela RedMonk.[6]

* Linguagem de alto nível, ou seja, possui sintaxe se aproxima mais da nossa linguagem e se distanciam mais da linguagem de máquina.
* Sintaxe simples: fácil de aprender e intuitiva
* Permite que o programa execute em múltiplas plataformas, sem alterações.

Extremamente portável Unix/Linux, Windows, Mac, PalmOS, WindowsCE, RiscOS, VxWorks, QNX, OS/2, OS/390, AS/400, PlayStation, Sharp Zaurus, BeOS, VMS…Pode ser usada em Windows, Linux, FreeBSD, Mac OS, etc.. . .

* Implementação distribuída como código livre. Grande comunidade
* Muitas bibliotecas disponíveis
* Usada no “mundo real”: Google, Microsoft, Yahoo!, NASA,
* Os tipos pré-definidos em Python são poderosos e simples de usar;
* O interpretador Python permite aprender e testar rapidamente trechos de código
* Python é expressivo, com abstrações de alto nível => código curto e rápido desenvolvimento;
* É fácil escrever extensões para Python em C e C++, quando for necessário desempenho máximo, ou quando necessitar de interfacear alguma ferramenta nestas linguagens;
* Possui tratamento de exceções (moderno mecanismo de tratamento de erros);
* Gerenciamento automático memória
* “Segura”: sem “core dumps” causados por bugs do usuário
* Pouco punitiva: poucas regras arbitrárias; torna prazeroso seu uso

Por que aprender Python?

Aplicabilidade

Python, apesar de tipicamente usado em aplicações web e como linguagem de scripting para administração de sistemas, é uma linguagem de uso geral que pode ser empregada em vários tipos de problemas. A biblioteca padrão inclui módulos para processamento de texto e expressões regulares, protocolos de rede (HTTP, FTP, SMTP, POP, XMLRPC, IMAP), acesso aos serviços do sistema operacional, criptografia, interface gráfica etc. Além da biblioteca padrão, existe uma grande variedade de extensões adicionais para todo tipo de aplicação.

Facilidade de aprendizado

É uma linguagem de alto nível, com alta capacidade de abstração o que torna essa simplicidade poderosa e expressiva. Python possui um conjunto reduzido de estruturas de controle, de forma a reduzir a complexidade da linguagem. Além disso a forma de expressar um loop for para percorrer elementos de uma lista ou dicionário ou qualquer objeto que implemente alguns métodos especiais é simples e poderosa.

Legibilidade:

* Python possui uma sintaxe clara e concisa, que favorece a legibilidade do código fonte, tornando a linguagem mais produtiva.
* Uso da endentação para delimitar.

Redigibilidade:

* Tipagem dinâmica (não é preciso explicitar tipos);
* Controle de laços é feito por endentação, não necessita de chaves;

Eficiência:

* Por ser interpretada, é mais lenta que C e C++;
* Em geral é uma linguagem eficiente, mas dependente da aplicação;
* Verificação dinâmica de tipos;
* Passagem de parâmetros normal (eager);
* Apresenta uma ampla variedade de comandos e expressões.

Confiabilidade:

* É uma linguagem dinamicamente tipada e ao mesmo tempo fortemente tipada e também possui inferência a tipos.
* O sistema de verificação de tipos não realiza a conversão implícita de um tipo de dados para outro, levantando uma exceção quando tipos inconsistentes são utilizados.

Custo:

* O custo de treinamento e para escrever programas em Python é baixo, uma vez que a que capacidade de escrita e a legibilidade são altas.
* O tempo do entendimento de códigos na aprendizagem e para a manutenção é baixo.

Portabilidade:

A implementação padrão do Python é escrita em ANSI C portável, compila e executa em praticamente todas as principais plataformas em uso atualmente. Por exemplo, os programas em Python são executados em tudo, de PDAs até supercomputadores.

Reusabilidade:

* Oferece reuso de funções, tipos e variáveis distribuídas em bibliotecas.
* Oferece também o conceito de classes e possuem mecanismo de pacotes.
* O polimorfismo universal também auxilia na criação de código reusável e frameworks.

Integração

A facilidade de integração com C faz de Python uma linguagem embutida atrativa em aplicações de maior porte. Atualmente, o código Python pode chamar bibliotecas de C/C++, pode ser chamado a partir de programas em C/C++, pode ser integrado com componentes Java, pode se comunicar pela rede.

Gerenciamento de memória

A linguagem Python disponibiliza um mecanismo de gerência de memória automático que é responsável por alocar memória para seus objetos e desalocá-la quando esses objetos não possuem mais referência para eles.

Exceções

Python apresenta mecanismos embutidos para tratamento de exceções;

Concorrência

Python apresenta bibliotecas para uso de concorrência;

Multiparadigma

Python suporta múltiplos paradigmas de programação, incluindo orientação a objetos (class), imperativa (def), e programação funcional (lambda). Multiparadigma ou hibridas, pois misturam ou possibilitam a utilização de vários recursos de programação pois suporta os paradigmas:

1. Programação Imperativa

Exemplo: Algol, Pascal, C. Em Imperativa, vamos dar ao nosso programa uma sequência de passos para resolver determinado problema, ou seja, linha embaixo de linha, você sai dando ordens uma embaixo da outra, e o computador obedece.

1. Programação Funcional

Exemplo: LISP, Scheme, ML, OCaml, Haskell ou F#. É a programação “orientada a funções”. Trata os programas como avaliando funções matemáticas e evita dados de estado e mutáveis (imutabilidade).

1. Programação lógica

Exemplo: Prolog. O sentido da programação lógica é trazer o estilo da lógica matemática à programação de computadores.

Prolog foi desenvolvida em 1972 por Alain Colmerauer. Ela veio de uma colaboração entre Colmerauer em Marselha e Robert Kowalski em Edinburgo. Colmerauer estava trabalhando na compreensão da linguagem natural, usando lógica para representar semânticas e usando resolução para questionamento-resposta. Durante o verão de 1971, Colmerauer e Kowalski descobriram que a forma clausal da lógica poderia ser usada para representar gramáticas formais e que demonstrações do teorema da resolução poderia ser usado para análise gramatical.

A linguagem de programação Prolog foi explicitamente apresentada como baseada na lógica matemática. A base dessa alegação era que um programa Prolog podia literalmente ser lido como um conjunto de fórmulas em um fragmento da lógica de primeira ordem, herdando o modelo de teoria e demonstração da lógica de primeira ordem.

1. Programação Orientada a Objetos

Exemplo: Smalltalk, C++, Java, C#. Conhecida também pela sigla POO ou OOP(em inglês, Object-Oriented Programming) . Organiza programas como objetos: estruturas de dados que consistem em campos de dados e métodos, juntamente com suas interações. POO, vai a modelar os códigos e algoritmos pensando em entidades que possuem características e comportamentos, vulgo objetos.

* Encapsulamento e proteção através dos mecanismos de classes e pacotes.
* Polimorfismo universal por inclusão que é caraterizado por herança simples e múltipla.

1. Programação Procedural

Exemplo: ALGOL, BASIC, Delphi, Fortran . Também conhecida como Programação Estruturada, ela especifica as etapas que um programa deve executar para atingir o estado desejado.

INTERFACES

* XML
  + DOM, expat
  + XMLRPC, SOAP, Web Services
* Base de dados relacional
  + MySQL, PostgreSQL, Oracle, ODBC,
  + Sybase, Informix
* Java (via Jython)
* Objective C
* COM, DCOM (.NET too)
* Many GUI libraries
  + cross-platform
    - Tk, wxWindows, GTK, Qt
  + platform-specific
    - MFC, Mac (classic, Cocoa), X11

Por que NÃO aprender Python?

DATA DE VALIDADE DO PYTHON

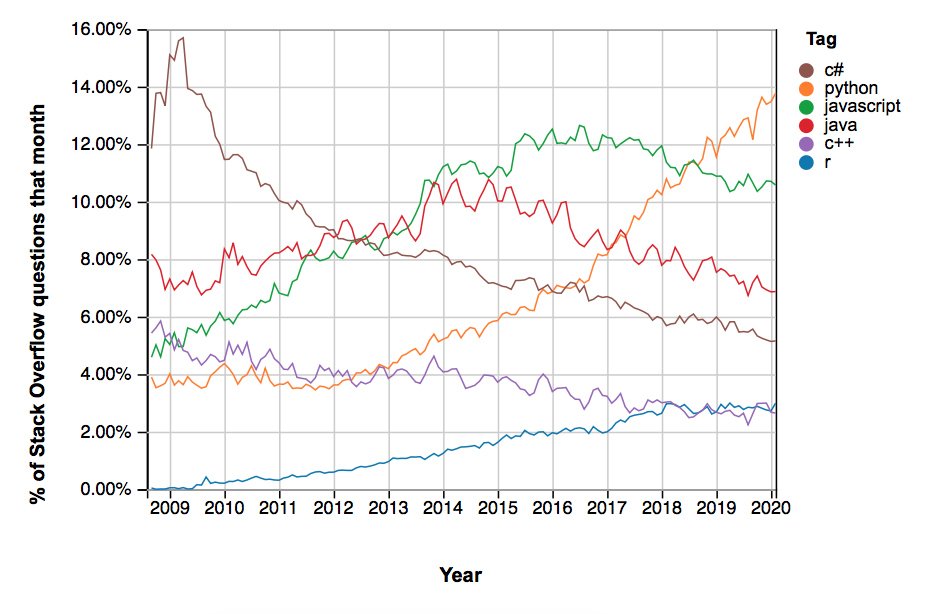
Até quando, Python, o "garotão" em scripts, será a linguagem de programação do futuro?

Na década passada, um idioma subiu em popularidade e superou todos os seus antecessores em popularidade, e esse idioma é obviamente Python. O Python é uma linguagem de programação, interpretada por C, orientada a objetos, fácil de usar, de ler e de mutar. A comunidade de programação levou algumas décadas para apreciar o Python, mas, desde o início de 2010, ele vem crescendo - e, eventualmente, superando a popularidade de C, C#, Java e JavaScript.

Há muitas razões pelas quais o Python foi recentemente coroado a linguagem de programação mais popular do mundo e, mesmo com alta demanda por mais alguns anos, também há muitas razões que poderia o levar a perder esse título. Colocar uma data de validade exata no Python seria tanta especulação que poderia passar como ficção científica. Em vez disso, vou avaliar as virtudes que estão aumentando a popularidade do Python, os pontos fracos do momento que poderá leva-lo a quebrar no futuro.

O que torna o Python popular agora

O sucesso do Python se reflete nas tendências do Stack Overflow, que medem a contagem de tags nas postagens na plataforma. Dado o tamanho do StackOverflow, este é um bom indicador para a popularidade do idioma.



Embora o R tenha atingido o platô nos últimos anos, e muitas outras línguas estejam em constante declínio, o crescimento do Python parece imparável. Quase 14% de todas as perguntas do StackOverflow são marcadas como "python" e a tendência está subindo. E há várias razões para isso.

É velho

O Python existe desde os anos noventa. Isso não significa apenas que ela teve muito tempo para crescer. Também adquiriu uma comunidade grande e solidária.

Portanto, se você tiver algum problema enquanto estiver codificando em Python, é grande a probabilidade de resolvê-lo com uma única pesquisa no Google. Simplesmente porque alguém já encontrou seu problema e escreveu algo útil sobre ele.

É fácil de aprender

Não é apenas o fato de existir há décadas, dando aos programadores tempo para fazer tutoriais brilhantes. Mais do que isso, a sintaxe do Python é muito legível por humanos.

Para começar, não há necessidade de especificar o tipo de dados. Você acabou de declarar uma variável; O Python entenderá a partir do contexto se é um número inteiro, um valor flutuante, um booleano ou qualquer outra coisa. Esta é uma vantagem enorme para iniciantes. Se você já teve que programar em C ++, sabe como é frustrante o seu programa não ser compilado porque você trocou um float por um número inteiro.

E se você já teve que ler códigos Python e C++ lado a lado, saberá o quão compreensível é o Python. Embora o C++ tenha sido projetado com o inglês em mente, é uma leitura bastante irregular, em comparação com o código Python.

Uma grande diferença entre uma linguagem como Python e linguagens de script semelhantes é o quão legível e fácil de entender é o Python. Muitas vezes, ler Python pode ser como ler um livro abstrato e estranho sobre variáveis ​​iguais a números. Isso não apenas facilita para iniciantes, como também mutar, modificar e decifrar, o que é muito importante e, especialmente, para grandes pools de códigos com milhares de desenvolvedores trabalhando neles. Sejamos honestos, todos preferimos escrever Python a C.

O Python para o iniciante, por ser uma linguagem de programação muito usada, traz benefícios como:

* Fácil de pesquisar no Google
* Fácil de tirar dúvidas
* Grandes pacotes
* Patches frequentes

É FOSS

Free and open source software (F/OSS, FOSS). A Python Foundation funciona principalmente com doações e certificados educacionais, o que significa que o Python é um software totalmente gratuito e aberto que alguém codificou para você usar.

Python é uma linguagem interpretada, o que significa que não há compilador ou montador capaz de colocar a linguagem no código da máquina. Em vez disso, outra linguagem, C no caso do Python, é usada para interpretar a linguagem com o cabeçalho Python.h. Normalmente, isso colocaria o Python na categoria de script, no entanto, acho importante não ignorar a posição do Python como um grampo de programação universal.

É versátil

Como o Python existe há tanto tempo, os desenvolvedores criaram um pacote para todos os fins. Hoje em dia, você pode encontrar um pacote para quase tudo.

* Deseja triturar números, vetores e matrizes? NumPy é seu cara.
* Deseja fazer cálculos para tecnologia e engenharia? Use SciPy.
* Deseja se destacar na manipulação e análise de dados? Dê uma chance aos pandas.
* Deseja começar com Inteligência Artificial? Por que não usar o Scikit-Learn.

Qualquer que seja a tarefa computacional que você esteja tentando gerenciar, é provável que exista um pacote Python por aí. Isso faz com que o Python fique atualizado sobre os desenvolvimentos recentes, como pode ser visto no aumento do Machine Learning nos últimos anos.

Aprendizado de máquina (Machine Learning)

O aprendizado de máquina é outro uso proeminente do Python que certamente ajudou a aumentar sua popularidade nos últimos anos. O aprendizado de máquina está na vanguarda da tecnologia, e o Python, juntamente com seu relacionamento próximo com C, é surpreendentemente eficaz e útil para o aprendizado de máquina. Embora o Python certamente não tenha o desempenho de algumas linguagens semelhantes como Nim, Julia e Go, em muitos aspectos, ele é compensado por ser rápido, fácil, conciso, mas talvez mais importante, universal.

Desenvolvimento Web

Uma coisa que o Python provou ser chocantemente talentoso é administrar o back-end de sites. Essa também é uma vantagem que o Python possui para muitas outras linguagens. O Python possui muitos pacotes incríveis para implantar APIs e até criar aplicativos da Web com todos os recursos.

Unix

É difícil falar sobre Python sem falar sobre sistemas modernos semelhantes ao Unix. O Python 2.7 está obsoleto há dois meses e meu ambiente de área de trabalho, assim como muitos outros recursos no meu sistema operacional, ainda usam o Python 2.7. A combinação de Bash e Python pode criar alguns scripts bastante úteis para executar servidores, preencher dados, concluir solicitações, editar arquivos e certamente muito mais.

DESVANTAGENS DO PYTHON

Com base nas elaborações anteriores, você pode imaginar que o Python permanecerá no topo pelas próximas gerações. Mas, como toda tecnologia, o Python tem suas fraquezas. Analisarei as falhas mais importantes, uma a uma, e avaliarei se são fatais ou não.

Rapidez

Embora o Python seja certamente uma ótima linguagem e tenha tido um enorme impacto no mundo inteiro, toda linguagem tem suas desvantagens em um aspecto ou outro, e o Python não é diferente. Em primeiro lugar, e mais notável:

Python exigirá o Python

Uma desvantagem enorme do Python é que o código Python não pode ser compilado em um executável. Qualquer aplicativo ou ferramenta escrita em Python exigirá que o Python, assim como suas dependências, seja instalado no sistema do usuário final. Junto com isso, estão todas as versões corretas de cada pacote que correspondem aos pacotes usados ​​para desenvolver o referido aplicativo.

Velocidade - Python é lento.

Embora eu certamente concorde que o uso de logs e o método para cada método e geralmente apenas a criação de um código melhor certamente acelere o Python, existem definitivamente algumas situações em que o código precisará ser lento. O aprendizado de máquina é um ótimo exemplo, pois o treinamento de redes neurais geralmente requer um uso mortal de recursão. Não sei dizer quantas vezes escrevi um script para extrair dados em Python e experimentei tempos limite e lentidão dentro da minha CLI (Command Line Interface).

Existem esforços para mitigar isso, com o sempre impressionante Cython, mas geralmente a transição não é tão simples quanto maçãs e laranjas ao tentar usar o Cython. Embora existam muitos cenários em que as limitações do Python não possam ser sentidas, eu mentiria se dissesse que nunca tive que mudar para Julia, Nim ou C para fazer alguma coisa.

Python é lento. Tipo, muito lento. Em média, você precisará de 2 a 10 vezes mais para concluir uma tarefa com Python do que com qualquer outro idioma.

Existem várias razões para isso. Uma delas é que ela é digitada dinamicamente - lembre-se de que você não precisa especificar tipos de dados como em outros idiomas. Isso significa que muita memória precisa ser usada, porque o programa precisa reservar espaço suficiente para cada variável para que ele funcione em qualquer caso. E muito uso de memória se traduz em muito tempo de computação.

Outro motivo é que o Python pode executar apenas uma tarefa por vez. Isso é uma consequência de tipos de dados flexíveis - o Python precisa garantir que cada variável tenha apenas um tipo de dados, e processos paralelos podem atrapalhar isso.

Em comparação, seu navegador da Web comum pode executar uma dúzia de threads diferentes ao mesmo tempo. E há outras teorias por aí também.

Mas no final do dia, nenhum dos problemas de velocidade é importante. Computadores e servidores ficaram tão baratos que estamos falando de frações de segundos. E o usuário final realmente não se importa se o aplicativo é carregado em 0,001 ou 0,01 segundos.

Dependências

Outra desvantagem significativa do Python são as dependências e os ambientes virtuais. No que diz respeito a uma linguagem de script no domínio do Python, acho que o Python se sai muito bem com dependências e ambientes virtuais. Gosto de comparar o Python com as duas linguagens que uso para fazer coisas semelhantes, Julia e Nim, e o mundo de dependências de Julia é bastante semelhante ao do Python. Uma vantagem que Julia tem é, na minha opinião subjetiva, que os ambientes virtuais são muito melhores e mais fáceis de usar.

Escopo

Originalmente, o Python tinha um escopo dinâmico. Isso basicamente significa que, para avaliar uma expressão, um compilador primeiro pesquisa o bloco atual e, em seguida, sucessivamente todas as funções de chamada.

O problema do escopo dinâmico é que toda expressão precisa ser testada em todos os contextos possíveis - o que é tedioso. É por isso que a maioria das linguagens de programação modernas usa escopo estático.

O Python tentou fazer a transição para o escopo estático, mas estragou tudo. Geralmente, os escopos internos - por exemplo, funções dentro de funções - seriam capazes de ver e alterar os escopos externos. No Python, os escopos internos podem ver apenas os escopos externos, mas não os alteram. Isso leva a muita confusão.

Lambdas

Apesar de toda a flexibilidade no Python, o uso do Lambdas é bastante restritivo. Lambdas só podem ser expressões em Python, e não declarações.

Por outro lado, declarações e declarações variáveis ​​são sempre declarações. Isso significa que o Lambdas não pode ser usado para eles.

Essa distinção entre expressões e declarações é bastante arbitrária e não ocorre em outros idiomas.

Espaços em branco

No Python, você usa espaços em branco e recuos para indicar diferentes níveis de código. Isso o torna opticamente atraente e intuitivo de entender.

Outras linguagens, por exemplo C ++, contam mais com chaves e ponto e vírgula. Embora isso possa não ser visualmente atraente e amigável para iniciantes, torna o código muito mais sustentável. Para projetos maiores, isso é muito mais útil.

Idiomas mais recentes, como Haskell, resolvem esse problema: eles contam com espaços em branco, mas oferecem uma sintaxe alternativa para aqueles que desejam ficar sem.

Desenvolvimento Móvel

Enquanto assistimos à mudança do computador para o smartphone, fica claro que precisamos de linguagens robustas para criar software móvel.

Mas não há muitos aplicativos móveis sendo desenvolvidos com o Python. Isso não significa que não possa ser feito - existe um pacote Python chamado Kivy para esse fim.

Mas o Python não foi feito com o celular em mente. Portanto, mesmo que possa produzir resultados aceitáveis ​​para tarefas básicas, sua melhor aposta é usar um idioma criado para o desenvolvimento de aplicativos para dispositivos móveis. Algumas estruturas de programação amplamente usadas para dispositivos móveis incluem React Native, Flutter, Iconic e Cordova.

Para deixar claro, laptops e computadores de mesa devem estar presentes por muitos anos. Mas como os dispositivos móveis ultrapassam o tráfego de computadores há muito tempo, é seguro dizer que aprender Python não é suficiente para se tornar um desenvolvedor experiente e versátil.

Erros de tempo de execução

Um script Python não é compilado primeiro e depois executado. Em vez disso, ele é compilado toda vez que você o executa, portanto, qualquer erro de codificação se manifesta no tempo de execução. Isso leva a um desempenho ruim, consumo de tempo e necessidade de muitos testes. Como, muitos testes.

Isso é ótimo para iniciantes, pois os testes os ensinam muito. Mas para desenvolvedores experientes, ter que depurar um programa complexo em Python os faz dar errado. Essa falta de desempenho é o maior fator que define um carimbo de data / hora no Python.

Aprendizado de máquina

Uma pergunta que me fazem muito é: "Quanto tempo você acha que o Python vai ficar por aí?" Muitas vezes, essa pergunta surge no espaço de aprendizado de máquina, porque o Python tem uma série de problemas associados ao aprendizado de máquina. É engraçado pensar que qualquer linguagem poderia ser estática e permanecer como a linguagem de programação mais usada por muito tempo.

Fortran foi o grande problema antes de C, C foi o grande problema antes de C++, C++ foi o grande problema antes de Java, e essa lista continua e continua, e sempre será. Os computadores são emocionantes porque estão em constante evolução e, com eles, a tecnologia em que trabalham. Apenas 30 anos atrás, a idéia de 16 GB de RAM era um conceito totalmente estranho, então não há como dizer o que o futuro reserva para as linguagens de programação.

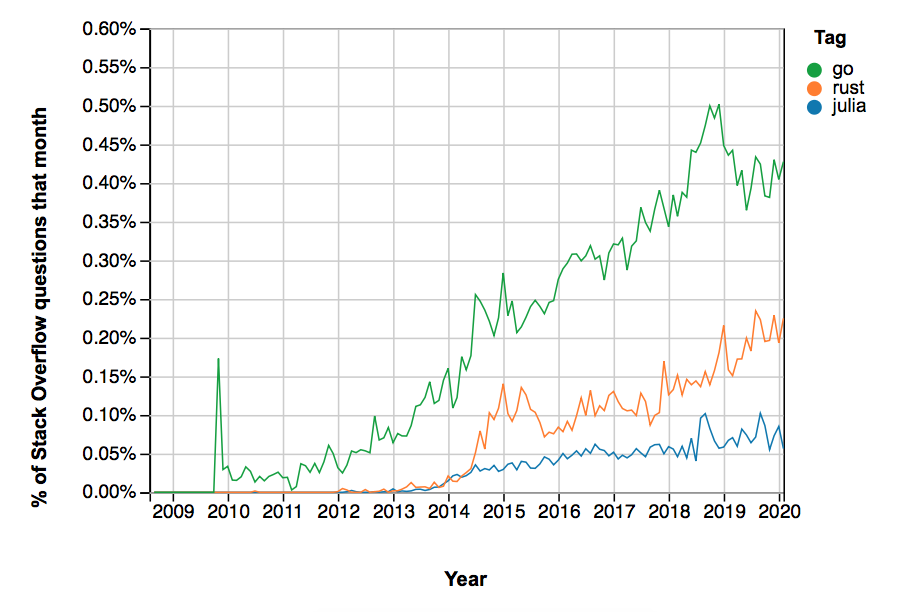
Embora seja certamente verdade que, eventualmente, o Python provavelmente será ultrapassado por outra linguagem de programação, acho importante lembrar que as pessoas ainda escrevem Fortran, C, Java e C++; portanto, o próprio Python provavelmente estará conosco e será usado com bastante frequência por um longo período de tempo, por mais impopular que possa se tornar.

Outras línguas - O que poderia substituir o Python no futuro - e quando

Esses idiomas podem não ter ganhado tanto vapor ainda, mas estão mostrando uma coisa que considero importante: Existem alguns novos concorrentes no mercado de linguagens de programação:

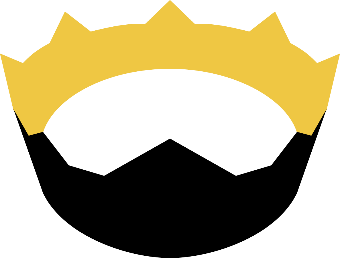
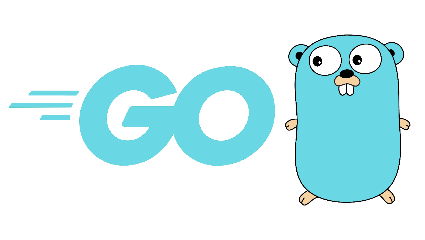
1. O Rust oferece o mesmo tipo de segurança que o Python possui - nenhuma variável pode ser substituída acidentalmente. Mas resolve o problema de desempenho com o conceito de propriedade e empréstimo. É também a linguagem de programação mais amada dos últimos anos, de acordo com o StackOverflow Insights.
2. Go é ótimo para iniciantes como Python. E é tão simples que é ainda mais fácil manter o código. Ponto interessante: os desenvolvedores Go estão entre os programadores mais bem pagos do mercado.
3. Julia é uma linguagem muito nova que compete diretamente com o Python. Ele preenche a lacuna dos cálculos técnicos em larga escala: normalmente, alguém usaria Python ou Matlab e corrigia tudo com bibliotecas C ++, necessárias em larga escala. Agora, pode-se usar Julia em vez de fazer malabarismos com dois idiomas. Julia certamente poderia mudar a maneira como fazemos o aprendizado de máquina
4. Nim – é uma linguagem imperativa, de propósito geral, multi-paradigma, tipicamente estatal, para sistemas, projetada e desenvolvida por Andreas Rumpf. Ele foi projetado para ser "eficiente, expressivo e elegante", suportando estilos de programação metaprogramação, funcional, passagem de mensagens, processuais e orientados a objetos, fornecendo vários recursos, como geração de código de tempo de compilação, tipos de dados algébricos , uma interface de função externa (FFI) com C e C ++ e compilando em C, C ++, Objective-C e JavaScript. O Nim supera facilmente o Python e a Julia sobre como as dependências e os pacotes são tratados para os usuários finais e a implantação. Com o Nim, você pode criar um executável compilado contendo todas as dependências necessárias. E Nim é definitivamente um grande divisor de águas para scripts de alto nível.

Embora existam outras línguas no mercado, Rust, Go e Julia são as que corrigem patches fracos do Python. Todas essas linguagens se destacam nas tecnologias ainda por vir, principalmente na Inteligência Artificial. Embora sua participação no mercado ainda seja pequena, como refletido no número de tags StackOverflow, a tendência para todas elas é clara: para cima.



Dada a popularidade onipresente do Python no momento, certamente levará meia década, talvez até um todo, para que qualquer uma dessas novas linguagens o substitua.

Qual dos idiomas será - Rust, Go, Julia ou um novo idioma do futuro - é difícil dizer neste momento. Mas, considerando os problemas de desempenho que são fundamentais na arquitetura do Python, um inevitavelmente será substituído.

A maior ameaça à popularidade do Python é provavelmente outras novas linguagens de programação. As linguagens que discuti, Julia e Nim, são simplesmente o que eu acredito serem os dois grandes concorrentes do Python neste momento.

É possível avançar.

Isto é, o Python não é uma parede de tijolos sólida e o avanço das linguagens de programação, do aprendizado de máquina e do script certamente não vai parar com isso. Parte da razão pela qual acho que essas linguagens não foram bem-sucedidas é porque ambas são de tipo estatístico e, na maioria das vezes,

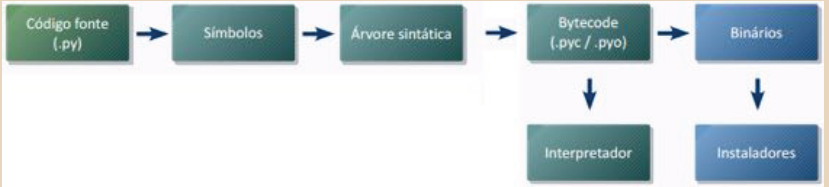
Linguagens Funcionais

E eu entendo que esse pode ser um conceito muito dissonante para quem se apega e ama o Python.

INTERPRETADOR DE PYTHON

Ao contrário das linguagens compiladas, que transformam o código escrito dos programas para uma plataforma especifica, por exemplo, Windows ou Linux, Python é implementado com um interpretador híbrido:

* Programa Python é traduzido para um código intermédio (bytecode);
* O bytecode é executado por um interpretador especial, o que possibilita o aplicativo ser executado em várias plataformas com poucas ou mesmo nenhuma alteração.
* O bytecode é armazenado em arquivos com extensão “.pyc” (bytecode normal) ou “.pyo” (bytecode otimizado).
* O bytecode também pode ser empacotado junto com o interpretador em um executável, para facilitar a distribuição da aplicação, eliminando a necessidade de instalar Python em cada computador.



Vantagens do Interpretador:

* Fácil de usar interativamente
* Fácil testar e modificar componentes
* Mais eficiente do que um interpretador clássico

Desvantagem:

* Não é tão eficiente como uma linguagem compilada tradicional (ex: C, C++, Fortran)

A maioria das distribuições Linux já vem com o interpretador do Python instalado. Para plataforma Windows há um instalador, que inclui o interpretador, a documentação e um ambiente de desenvolvimento integrado (IDE) – o IDLE.

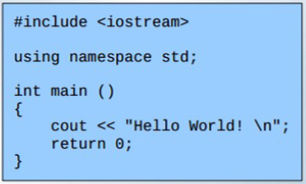
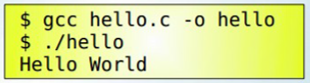
O instalador do Python no Windows inclui o IDLE, cujo nome é tanto uma homenagem a Eric Idle, quanto abreviação de Integrated DeveLopment Editor, e possui recursos como coloração de sintaxe (syntax highlighting), autocompletar (autocomplete) e um debugger.

* **Interpretadores**: tradução é efetuada sempre que o programa executa. Ex.: Python, Pearl, Basic tradicional, Shell Script...



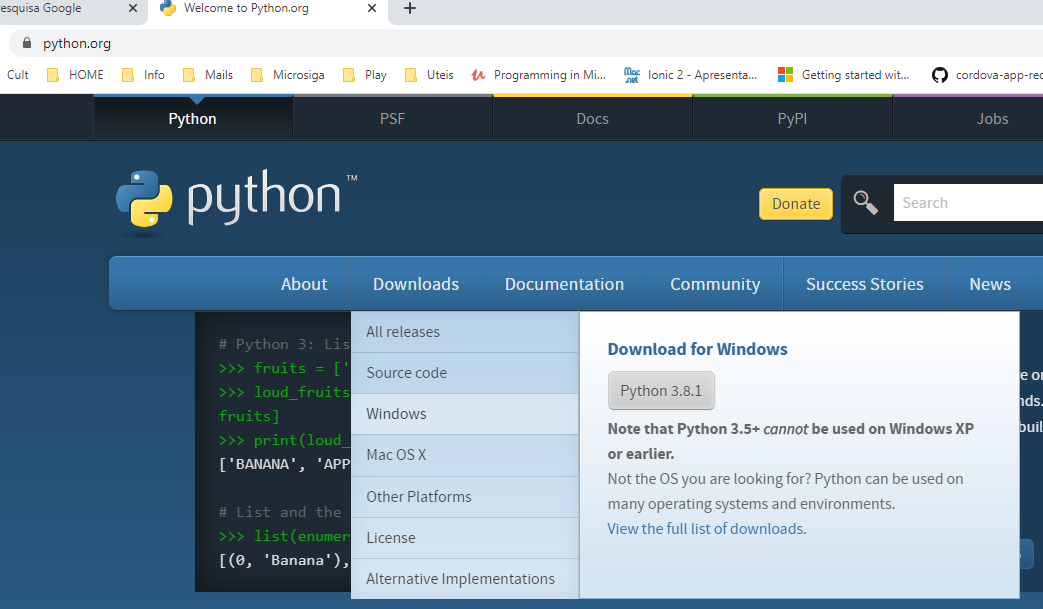
* **Compiladores**: tradução é efetuada uma só vez; produz um programa código-máquina independente. Ex.: Fortran, C, C++, Visual Basic...

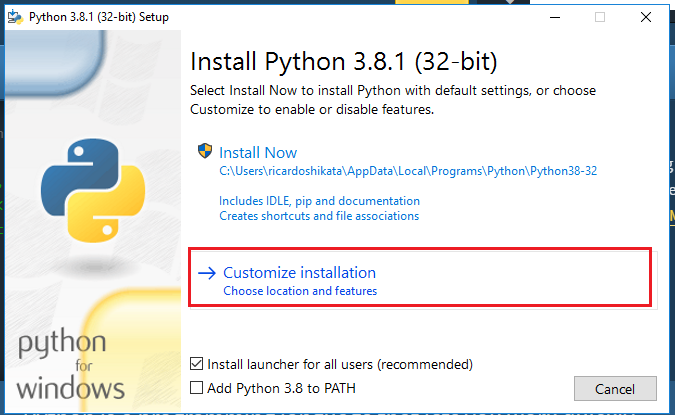


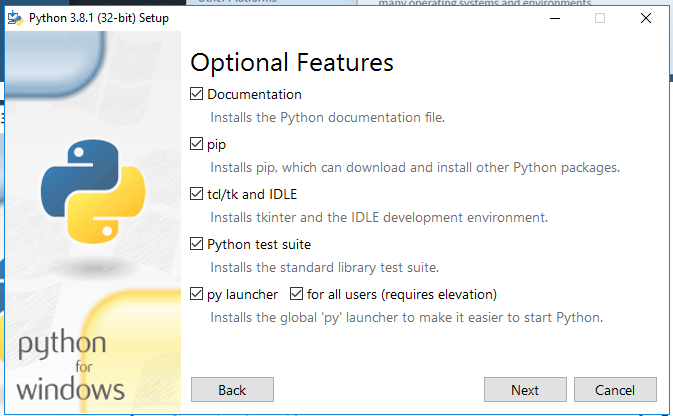
 

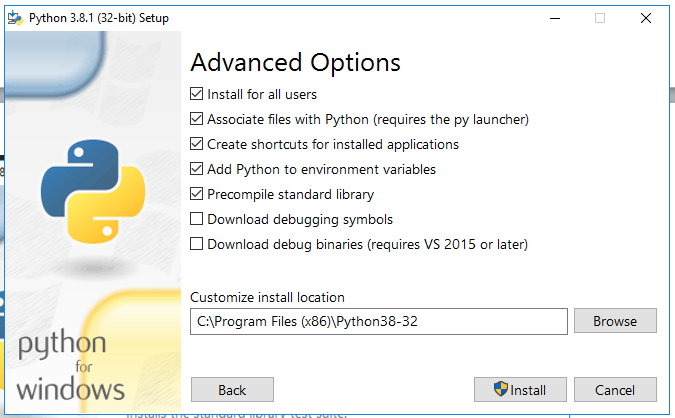
INSTALAÇÂO

<https://www.python.org/>



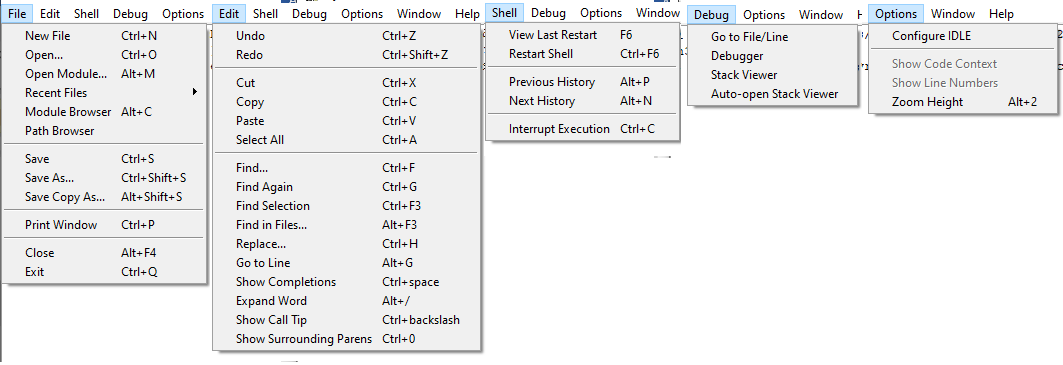


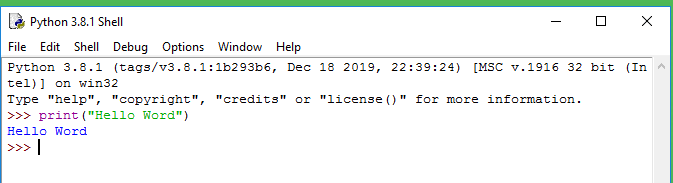




Command Line Interface (CLI)

A interface de linha de comando (CLI) é um modo eficiente de interaçãoo ser humano-máquina que é feita através da entrada de comandos na forma de textos digitados em um no terminal.

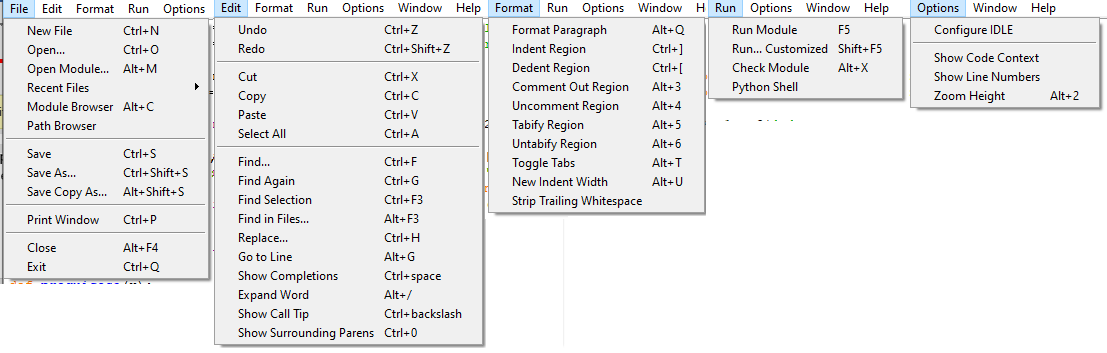


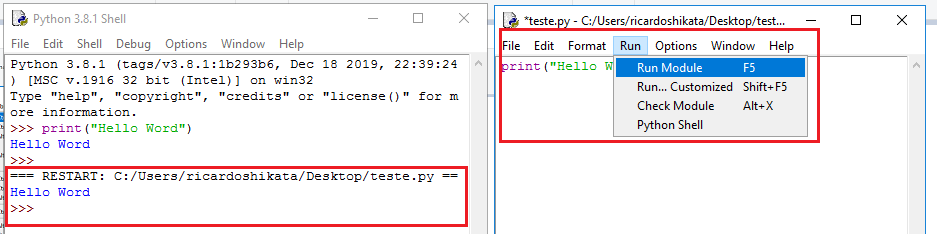


Uso Interativo do Python

Cálculos numéricos complexos podem ser executados com simplicidade no Command Line Interface do Python. Os ficheiros de programas Python têm extensão .py

A forma interativa é usada para testar pequenas partes de código. Devemos escrever programas com mais do que algumas linhas num script. Ambientes de desenvolvimento como o IDLE e o IEP combinam: uma janela para testes interativos; uma ou mais janelas para scripts





Antes de iniciar os testes, é necessário conhecer alguns comandos básicos:

|  |  |
| --- | --- |
| **Comandos** | **Decrição** |
| python | Inicializa uma sessao do software no modo interativo. |
| Alt + P | Repete o ultimo comando digitado |
| Backspace | Apaga caracteres a esquerda do cursor. |
| Ctrl + C | Interromper execucao. |
| Ctrl + F6 | Reiniciar a sessao. |
| Delete | Apaga caracteres a direita do cursor. |
| F6 | Ver ultima inicializacao |
| quit () ou exit () ou Ctrl-D (“end-of-file”) | Termina uma sessao do programa. |
| Tab ou Ctrl + Space | Auto completar a chamada de funções, comandos da aplicação e nomes das variáveis. |

PIP

0 pip trata-se de um gerenciador de pacotes do Python, no qual é possivel instalar novos pacotes ou atualizar os existentes, além do que, é possivel executar a atualização do próprio pip o que é extremamente importante, sobretudo quando se quer configurar o interpretador da distribuição Python Anaconda no PyCharm e/ou instalar pacotes a partir deste.

|  |  |
| --- | --- |
| **Comandos** | **Decrição** |
| pip --version | Para verificar a versão do pip instalada |
| pip install nome\_do\_pacote | Para instalar um pacote a partir clo pip |
| pip uninstall nome\_do\_pacote | Para desinstalar um pacote a partir clo pip |
| py -3.6 -m pip install --upgrade | Para atualizar o pip, em que, o 3 . 6 é a versao instalada |
| Digitar exit | Para sair do console |
| De preferência execute os comandos no Anaconda Prompt como administrador | |

VARIÁVEIS E TIPOS DE DADOS

Identificadores são nomes utilizados para identificar objetos - variáveis, funções e classes, por exemplo. Os identificadores devem começar com uma letra sem acentuação ou com um sublinhado ( \_ ), e podem conter números, letras sem acentuação e sublinhados. Python é case-sensitive, ou seja, o identificador python é diferente dos identificadores PYTHON e Python.

Variáveis são espaços reservados na memória utilizados para armazenar valores, como por exemplo, textos, resultados de cálculos, entrada de usuário, resultados de consultas a uma base de dados, etc. Variáveis devem seguir as regras de identificadores, vistos anteriormente. Em Python, os tipos de dados básicos são:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Grupo** | | **Variável** | **Sintaxe** | **Exemplo** |
| **Numéricos** | **Primitivos** | Integer | int | 1, 2, 3, -1, -2, 0, 7, 234, ... |
| Float | float | 1.23, 0.3566, 3.53e+63 |
| Complex | complex | 2+3j, 2j, 5+27j |
| Boolean | bool | True, False |
| **Estruturados** | **Ordenados**  **(Sequenciais)** | Strings | str, unicode | 'ATGCCCAATTG' |
| Listas | list | [1, 5, 0.2, ‘A’, ‘xxx’] |
| Tuplas | tuple | (1,2,3) |
| **Não Ordenados** | Conjuntos | set | set([‘A’, ‘T’, ‘G’, ‘C’]) |
| frozenset | frozenset([4.0, 'string', True]) |
| Dicionarios | dict | {‘A’ : ‘Ala’, ‘V’ : ‘Val’, ‘1’ : ‘lie’, ‘L’ : ‘Leu’) |

Além destes, a linguagem Python possui uma representação dos números complexos através do tipo complex. Os complexos são números com parte imaginária definida por j, tal que j2 = -1. Isto é, são números do tipo a + bj, onde a e b são números reais, sendo a o coeficiente da parte real e b, o coeficiente da parte imaginária.

TIPAGEM DINÂMICA

Python possui o que se chama de tipagem dinâmica, ou seja, a tipagem pode mudar a cada nova entrada de dados em uma variável. Isso significa que o tipo de uma variável é inferido pelo interpretador em tempo de execução (isto é conhecido como Duck Typing).

A tipagem dinâmica reduz a quantidade de tempo de planejamento prévio e é um mecanismo importante para garantir flexibilidade e simplicidade das funções Python.

No interpretador de Python podemos usar type(...) para obter o tipo de um resultado:

a = 1   
print(type(a)) *# <class 'int'>*a = **'abacaxi'**print(type(a)) *# <class 'str'>*a = 1.0   
print(type(a)) *# <class 'float'>*a = 3.1 + 2j   
print(type(a)) *# <class 'complex'>*

Função INPUT

Inserir uma entrada a partir do console.

LI = int(input(**"Digite o Limite Inferior (LI) ="**))

Função PRINT

Imprimir valores (textos e/ou numeros) na tela do console.

delta = b \*\* 2 - 4 \* a \* c  
print(**"delta = "**, delta)

COPY e DEEPCOPY

Para obter um novo objeto, ao invés de um ponteiro para um objeto existente deve-se usar o módulo copy. O módulo de cópia implementa operações de cópia superficial e profunda.

c = **'copia de texto'***#c2 NÃO é cópia, é referência de c*c2 = c  
  
*# ´Cópias do texto*c2 = list(c)  
c2 = c[:]  
**import** copy  
c2 = copy.copy(c)

No entanto, em listas as técnicas anteriores criam cópias rasas. Isso significa que objetos aninhados não serão copiados. Para contornar esse problema, você deve executar uma cópia profunda:

**import** copy  
lista\_a = [1,2,[3,4]]  
lista\_b = lista\_a[:]  
copylista\_b = copy.copy(lista\_a)  
deepcopylista\_b = copy.deepcopy(lista\_a)  
  
*# Subtituiu o 3 pelo 10*lista\_a[2][0] = 10  
  
print(lista\_a) *# [1, 2, [10, 4]]*print(lista\_b) *# [1, 2, [10, 4]]*print(copylista\_b) *# [1, 2, [10, 4]]*print(deepcopylista\_b) *# [1, 2, [3, 4]]*

A atribuição de valor para uma variável pode ser feita utilizando o comando **input()**, que solicita ao usuário o valor a ser atribuído à variável.

nome = input(**"Entre com o seu name: "**)  
print(nome) *# Fulano da Silva*

Com a atribuição múltipla, atribuímos valores a mais de uma variável ocupando apenas uma linha de código, mas isto se aplica somente quando o interesse for atribuir valores idênticos. Para atribuir valores distintos a variáveis distintas ocupando uma mesma linha do programa, a linguagem Python exibe a particularidade da atribuição em lista. Por exemplo, podemos preencher as variáveis Nome, ok, a e b usando um único comando:

**Nome, ok, a, b = 'Maria', True, 0, 0**

Observação: A atribuição acima citada aplica, na verdade, uma propriedade do tipo predefinido tupla. Por enquanto, é útil saber que (Nome, ok, a, b) e ('Maria', True, 0, 0) são tuplas e a atribuição em lista obedece à correspondência entre a lista das variáveis e a lista dos seus respectivos valores.

O comando **input()**, sempre vai retornar uma string. Nesse caso, para retornar dados do tipo inteiro ou float, é preciso converter o tipo do valor lido. Para isso, utiliza-se o **int**(string**)** para converter para o tipo inteiro, ou **float(**string **)**para converter para o tipo float.

num = (input(**"Entre com um numero: "**))  
print(num) *# 100*altura = float(input(**"Entre con a sua altura: "**))  
print(altura) *# 1.77*

IDENTIFICADORES

Embora a linguagem adotada nesse curso (a linguagem Python, em sua mais recente versão) admita caracteres acentuados, tentaremos ao máximo evitá-los apenas para reduzir a multiplicidade de formas e consequente risco de confusão para o iniciante. Seguiremos as regras já citadas acima nesta subunidade.

Acrescenta-se a restrição de que um identificador não pode ser palavra reservada (palavras da linguagem de propósitos específicos). Toda linguagem de programação possui um conjunto de palavras reservadas. No caso de Python, são seguintes:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| and | del | for | lambda | raise |
| as | elif | from | none | return |
| assert | else | global | nonlocal | true |
| break | except | if | not | try |
| class | exec | import | or | while |
| continue | false | in | pass | with |
| def | finally | is | print | yield |

Em Python, os nomes das variáveis devem ser iniciados com uma letra, mas podem possuir outros tipos de caracteres, como números e símbolos. O símbolo sublinha ( \_ ) também é aceito no início de nomes de variáveis.

|  |  |
| --- | --- |
| **NOMES VALIDOS** | **NOMES INVALIDOS** |
| nome\_candidato | nome candidato |
| endereco | endereço |
| RG | R.G. |
| mes\_ferias | mês\_férias |
| dataNasc | data-Nasc |
| fone1 | 1fone |

STRINGS

Uma string é uma sequência de caracteres simples. Na linguagem Python, as strings são utilizadas com aspas simples ('... ') ou aspas duplas ("...").

Para exibir uma string, utiliza-se o comando print().

print(**"Olá Mundo!!"**) *# Olá Mundo!!*

Os literais de string podem ser colocados entre aspas simples (') ou aspas duplas ("). Eles também podem ser colocados em grupos correspondentes de três aspas simples ou duplas (geralmente são chamadas de strings com aspas triplas). O caractere de barra invertida (\) é usado para escapar de caracteres que, de outra forma, possuem um significado especial, como nova linha, a própria barra invertida ou o caractere de citação.

|  |  |
| --- | --- |
| Escape Sequence | Meaning |
| \*newline* | Ignored |
| \\ | Backslash (\) |
| \' | Single quote (') |
| \" | Double quote (") |
| \a | ASCII Bell (BEL) |
| \b | ASCII Backspace (BS) |
| \f | ASCII Formfeed (FF) |
| \n | ASCII Linefeed (LF) |
| \N{*name*} | Character named *name* in the Unicode database (Unicode only) |
| \r | ASCII Carriage Return (CR) |
| \t | ASCII Horizontal Tab (TAB) |
| \u*xxxx* | Character with 16-bit hex value *xxxx* (Unicode only) |
| \U*xxxxxxxx* | Character with 32-bit hex value *xxxxxxxx* (Unicode only) |
| \v | ASCII Vertical Tab (VT) |
| \*ooo* | ASCII character with octal value *ooo* |
| \x*hh* | ASCII character with hex value *hh* |

Os literais de string podem opcionalmente ser prefixados com uma letra "r" ou "R"; chamadas raw strings e usam regras diferentes para interpretar seqüências de escape de barra invertida. Um prefixo de "u" ou "U" torna a cadeia uma cadeia Unicode. As cadeias Unicode usam o conjunto de caracteres Unicode, conforme definido pelo Unicode Consortium e pela ISO 10646. as seqüências descritas abaixo estão disponíveis nas seqüências de caracteres Unicode. Os dois caracteres do prefixo podem ser combinados; nesse caso, "u" deve aparecer antes de "r". Observe que o prefixo “u” só era necessário no Python 2 (“Legacy Python”), pois as strings nativas são o padrão agora no Python 3.

stringprefix = "r" | "u" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"

Uma string r é uma string bruta, ela ignora caracteres de escape. Por exemplo, "\ n" é uma sequência que contém um caractere de nova linha e r"\ n" é uma sequência que contém uma barra invertida e a letra n. Normalmente você usaria uma string-r se estivesse passando a string para outra coisa que usa um monte de caracteres estranhos.

Semelhante à forma como você cria uma string bruta (prefixando-a com “r”), você pode prefixar uma string com a letra “f” para obter uma "f-strings". A letra "f" indica que uma seqüência de caracteres é usada para formatação, e são realmente a maneira mais simples e prática de formatar strings.

name = **'Fred'**age = 42  
print(**f'He said his name is {**name**} and he is {**age**} years old.'**)

Para concatenar strings, utiliza-se o operador +.

print(**"Apostila"** + **"Python"**) *# ApostilaPython*a = **'Programação'**b = **'Python'**c = a + b  
print(c) *# ProgramaçãoPython*

|  |  |
| --- | --- |
| **Operação** | **Descrição** |
| x in s | True se x pertence a s |
| x not in s | True se x não pertencer a s |
| s + t | Concatenacao |
| s\*n | Produz n repeticoes de s |
| s[i] | i-esimo elemento de s |
| s[i:j] | Slice: elementos de i a j |
| s[i:j:k] | Slice: elementos de i a j com intervalo k |
| len(s) | Comprimento de s |
| min(s) | Menor elemento de s\* |
| max(s) | Maior elemento de s\* |

Em Python, existem várias funções (métodos) para manipular strings. Na tabela a seguir são apresentados os principais métodos para a manipulação das strings.

|  |  |  |
| --- | --- | --- |
| **Método** | **Descrição** | **Exemplo** |
| len() | Retoma o tamanho da string. | teste = "Apostila de Python" len(teste) 18 |
| capitalize() | Retorna a string com a primeira letra maiuscula | a = "python" a.capitalize() ‘Python' |
| count() | Informa quantas vezes um caractere (ou uma sequência de caracteres) aparece na string. | b = "Linguagem Python" b.count(“n") 2 |
| startswith() | Verifica se uma string inicia com uma determinada sequência. | c = "Python" c. startswith("Py") True |
| endswith() | Verifica se uma string termina com uma determinada sequência. | d = "Python" d.endswith("Py") False |
| isalnum() | Verifica se a string possui algum conteúdo alfanumérico (letra ou número). | e«="!g=$%" e.isalmun() False |
| isalpha() | Verifica se a string possui apenas conteúdo alfabetico. | f= "Python" f.isalpha() True |
| islower() | Verifica se todas as letras de uma string são minúsculas | g = "pytHon" g.islower() False |
| isupper() | Verifica se todas as letras de uma string são maiúsculas. | h = "# PYTHON 12" h.isupper() True |
| lower0 | Retoma uma cópia da string trocando todas as letras para minúsculas. | i = "# PYTHON 3" i.lower() '# python 3' |
| upper() | Retoma uma cópia da string trocando todas as letras para maiúsculas. | j = "Python" j.upper() 'PYTHON' |
| swapcase() | Inverte o conteúdo da string (Minusculo Maiuscu-lo). | k = "Python" k. swapcase() 'pYTHON' |
| title() | Convertee para maiusculo todas as primeiras letras de cada palavra da string. | 1 = "apostila de python" l.title() 'Apostila De Python' |
| split() | Transforma a string em uma lista. utilizando os espaços como referência. | m = "cana de açúcar" m.split() ['cana', 'de', ‘açúcar] |
| replace(Sl. S2) | Substitui na string o trecho SI pelo trecho S2. | u = "Apostila teste" n.replace("teste". "Python") 'Apostila Python' |
| find() | Retoma o indice da primeira ocorrência de um determinado caractere na string. Se o caractere nao estiver na string retoma -1 | o = "Python" o.find("h") 3 |
| ljust() | Ajusta a string para urn tamanho mínimo, acrescentando espaços a direita se necessário. | p = " Python" p.ljust(15) ‘Pytlion ‘ |
| rjust() | Ajusta a string para urn tamanho mínimo, acrescentando espaços a esquerda se necessário. | q = "Python" q.rjust(15) ‘ Python' |
| center() | Ajusta a string para um tamanlio mínimo, acrescentando espaços a esquerda e a direita, se necessário. | r = "Python" r.center(10) ' Python ' |
| Istrip() | Remove todos os espaços em branco do lado esquerdo da string. | s = " Python " s. Istrip() 'Python ' |
| rstrip() | Remove todos os espaços em branco do lado direito da string. | t = " Python " t.rstrip() ' Python' |
| strip() | Remove todos os espaços em branco da string. | u = " Python " u.strip() 'Python' |

O fatiamento é uma ferramenta usada para extrair apenas uma parte dos elementos de uma string. Retorna uma string com os elementos das posições do limite inferior até o limite superior -1.

**Nome\_String [Limite\_Inferior : Limite\_Superior]**

s = **"Python"***# Seleciona os elementos das posições 1,2,3*print(s[1:4]) *# yth  
# Seleciona os elementos a partir da posição 2*print(s[2:]) *# thon  
# Seleciona os elementos até a posição 3*print(s[:4]) *# Pyth*

As sequências de escape, também denominadas como constantes de barra invertida, tem o propósito de definir caracteres (bytes) especiais dentro de uma string. A mais comum é \n que é utilizada para encerrar a linha atual e ir para a próxima linha (pula linha). Sempre que necessário definir uma string que deva conter os caracteres aspas simples ou duplas, pode-se utilizar \’ e \", já que estes caracteres são utilizados para definir fim de strings. A Tabela 5.1 lista as possíveis sequências de escape e seus respectivos significados.

|  |  |
| --- | --- |
| **Sequência** | **Significado** |
| \a | som de bip no auto falante |
| \b | backspace |
| \f | formatação cm cascata |
| \n | pula linha (ENTER) |
| \r | carriage return |
| \t | tabulação horizontal |
| \v | tabulação vertical |
| \ooo | caracter com valor octal ooo |
| \xhh | caracter com valor hexadecimal hh |
| \’ | aspas simples |
| \" | aspas duplas |
| \\ | barra invertida |

texto = **"Meu texto \n\n\t pulei duas linhas e tabulei!"**print(texto)  
**"""  
Meu texto   
  
 pulei duas linhas e tabulei!  
"""**

letra r (maiúscula ou minúscula) antes de uma string indica que a mesma é uma string bruta (raw string), o que significa que possíveis sequências de escape presentes na mesma não serão interpretadas, conforme o Código 5.4:

texto = **r"\n não pulou linha"**print(texto) *#\n nao pulou linha*

ORD e CHR

As funções ord e chr se destinam a relacionar caracteres e sua respectiva codificação ASCII. Seja x uma string com um único caractere. ord(x) retornará o código ASCII do caractere em x. Por exemplo, para obter o código ASCII do "B", podemos fazer:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Comando** | **Saída** | **Comando** | **Saída** | **Comando** | **Saída** |
| ord('B') | 66 | ord('C') | 67 | ord('Z') | 90 |

A função chr, por sua vez, realiza a operação reversa à de ord. Enquanto ord recebe um caracter e retorna seu respectivo código ASCII, chr recebe um código ASCII e retorna seu respectivo caracter associado:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Comando** | **Saída** | **Comando** | **Saída** | **Comando** | **Saída** |
| chr(65) | A | chr(98) | b | chr(43) | + |

**for** i **in** range(1, 255):  
 ch = chr(i)  
 ordem = ord(ch)  
 print(i, **"="**, ch)  
 print(ch, **"="**, ordem)

NÚMEROS

Os quatro tipos numéricos simples, utilizados em Python, são números inteiros (int), números longos (long), números decimais (float) e números complexos (complex).

A linguagem Python também possui operadores aritméticos, lógicos, de comparação e de bit.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Operador** | **Descricao** | **Exemplo** |  | **Operador** | **Descricao** | **Exemplo** |
| **+** | Soma | 5 + 5 = 10 | **+=** | x = x + 1 | x += 1 |
| **-** | Subtração | 7 - 2 = 5 | **-=** | x = x - 1 | x -= 1 |
| **\*** | Multiplicação | 2 \* 2 = 4 | **\*=** | x = x \* 1 | x \*= 1 |
| **/** | Divisão | 4 / 2 = 2 | **/=** | x = x / 1 | x /= 1 |
| **%** | Resto da divisão | 10 % 3 = 1 | **//=** | x = x // 1 | x //= 1 |
| **\*\*** | Potência | 4 \*\* 2 = 16 | **\*\*** | x = x \*\* 1 | x \*\*= 1 |

O interpretador interativo do Python também pode ser utilizado como uma calculadora, digitando os cálculos e recebendo os valores. Exemplos:

2 + 2 # Imprime 4

5 / 2 # Imprime 2

10 % 5 # Imprime 0

12.356 / 2 # Imprime 6.1779999999999999

5 \* 5 # Imprime 25

2 \*\* 3 # Imprime 8

5 - 2 # Imprime 3

12.356 // 2 # Imprime 6.0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Operadores de Comparação** | | |  | **Operadores lógicos** | | |
| **Operador** | **Descricao** | **Exemplo** | **Operador** | **Descricao** | **Exemplo** |
| **<** | Menor que | a < 10 | **Not** | NÃO | Not a |
| **<=** | Menor ou igual | b <= 5 | **And** | E | (a<10) and (a>1) |
| **>** | Maior que | c > 4 | **Or** | OU | (a<10) or (a>1) |
| **>=** | Maior ou igual | d >= 2 |  |  |  |
| **==** | Igual | e == 1 |
| **!=** | Diferente | f != 16 |

DATA e HORA

Um programa Python pode lidar com data e hora de várias maneiras. Converter entre formatos de data é uma tarefa comum para computadores. Os módulos de hora e calendário do Python ajudam a rastrear datas e horas.

## TICK

Existe o módulo time disponível no Python, que fornece funções para trabalhar com horários e para converter entre representações. A função time.time() retorna a hora atual do sistema em ticks ( ponto flutuante em unidades de segundos ) desde as 00:00 de 1º de janeiro de 1970, denominada época.

**import** time  
  
ticks = time.time()  
*# Ticks desde 12:00am, January 1, 1970: 1583115679.9241986*print(**"Ticks desde 12:00am, January 1, 1970:"**, ticks)

A aritmética de datas é fácil de fazer com os ticks. No entanto, as datas anteriores à época não podem ser representadas neste formulário. Datas no futuro distante também não podem ser representadas dessa maneira, o ponto de corte é em algum momento de 2038 para UNIX e Windows.

## TIMETUPLE

Muitas das funções de tempo do Python lidam com o tempo como uma tupla de 9 números, como mostrado abaixo.

|  |  |  |
| --- | --- | --- |
| **Índice** | **Campo** | **Valores** |
| 0 | Ano de 4 dígitos | 2008 |
| 1 | Mês | 1 a 12 |
| 2 | Dia | 1 a 31 |
| 3 | Hora | 0 a 23 |
| 4 | Minuto | 0 a 59 |
| 5 | Segundo | 0 a 61 (60 ou 61 são segundos bissextos) |
| 6 | Dia da semana | 0 a 6 (0 é segunda-feira) |
| 7 | Dia do ano | 1 a 366 (dia juliano) |
| 8 | Horário de verão | -1, 0, 1, -1 significa que a biblioteca determina o horário de verão |

**import** datetime  
  
dateToday = datetime.datetime.today()  
attributesInTuple = dateToday.timetuple()  
  
*# 2020 3 3 23 9 10 1 63 -1***for** attribute **in** attributesInTuple:  
 print(attribute, end=**' '**)

A tupla acima é equivalente à estrutura struct\_time. Essa estrutura possui os seguintes atributos:

|  |  |  |
| --- | --- | --- |
| **Índice** | **Atributos** | **Valores** |
| 0 | tm\_year | 2008 |
| 1 | tm\_mon | 1 a 12 |
| 2 | tm\_mday | 1 a 31 |
| 3 | tm\_hour | 0 a 23 |
| 4 | tm\_min | 0 a 59 |
| 5 | tm\_sec | 0 a 61 (60 ou 61 são segundos bissextos) |
| 6 | tm\_wday | 0 a 6 (0 é segunda-feira) |
| 7 | tm\_yday | 1 a 366 (dia juliano) |
| 8 | tm\_isdst | -1, 0, 1, -1 significa que a biblioteca determina o horário de verão |

print(**f"Ano: {**agora.tm\_year**}"**) *# Ano: 2020*print(**f"Mes: {**agora.tm\_mon**}"**) *# Mes: 3*print(**f"Dia: {**agora.tm\_mday**}"**) *# Dia: 4*print(**f"Hora: {**agora.tm\_hour**}"**) *# Hora: 10*print(**f"Minuto: {**agora.tm\_min**}"**) *# Minuto: 41*print(**f"Segundo: {**agora.tm\_sec**}"**) *# Segundo: 19*print(**f"Oia da senana: {**agora.tm\_wday**}"**) *# Oia da senana: 2*print(**f"Dia no ano: {**agora.tm\_yday**}"**) *# Dia no ano: 64*print(**f"Horario de verao: {**agora.tm\_isdst**}"**) *# Horario de verao: 0*

## Módulo time

Existe um módulo time popular disponível no Python, que fornece funções para trabalhar com horários e para converter entre representações.

Para converter um instante de tempo de segundos após o valor do ponto flutuante da época em uma tupla de tempo, passe o valor do ponto flutuante para uma função (por exemplo, hora local) que retorna uma tupla de tempo com todos os nove itens válidos.

**import** time;  
  
localtime = time.localtime(time.time())  
print(**"Local current time :"**, localtime)  
**"""  
Local current time : time.struct\_time(tm\_year=2020,  
tm\_mon=2, tm\_mday=29, tm\_hour=20, tm\_min=52,  
tm\_sec=12, tm\_wday=5, tm\_yday=60, tm\_isdst=0)  
"""**

## Hora Formatada

Você pode formatar a qualquer momento, conforme sua exigência, mas o método simples para obter horas no formato legível é asctime () -

localtime = time.asctime(time.localtime(time.time()))  
print(**"Local current time :"**, localtime)  
*# Local current time : Sat Feb 29 20:54:00 2020*

Aqui está a lista de todos os métodos disponíveis.

|  |  |  |
| --- | --- | --- |
| **Função** | **Descrição** | **Exemplo** |
| altzone | O deslocamento do fuso horário local do horário de verão, em segundos a oeste do UTC, se um estiver definido. Isso é negativo se o fuso horário local do horário de verão estiver a leste do UTC (como na Europa Ocidental, incluindo o Reino Unido). Use-o somente se a luz do dia for diferente de zero. | time.altzone >> 7200 |
| [asctime ([tupletime])](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_asctime.htm&usg=ALkJrhi0xsm0naprxcQg2Jzhj-WXvY3BsQ) | Aceita uma tupla de tempo e retorna uma sequência legível de 24 caracteres, como 'Ter 11 de dezembro, 18:07:14 2008'. | time.asctime(timetuple) >> Tue Mar 3 23:31:53 2020 |
| process\_time() | Retorna o tempo atual da CPU como um número de ponto flutuante de segundos. Para medir custos computacionais de diferentes abordagens, o valor de time.process\_time() é mais útil que o de time.time(). | time.process\_time() >> 0.046875 |
| [ctime ([segundos])](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_ctime.htm&usg=ALkJrhiDO-8dfB4VK7nAb_MTrPSnCEq3Vg) | Como asctime (localtime(segundos)) e sem argumentos é como asctime() | time.ctime(time.time()) >>  time.ctime() >> Wed Mar 4 10:59:30 2020 |
| [gmtime ([segundos])](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_gmtime.htm&usg=ALkJrhhY8jyB_gsSeJR0sDUs9M0u7b8W-w) | Aceita um instante expresso em segundos e retorna uma tupla de tempo t com a hora UTC. Nota: t.tm\_isdst é sempre 0 | time.gmtime(ticks) >>  time.gmtime() >>  time.struct\_time.... |
| [localtime ([segs])](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_localtime.htm&usg=ALkJrhhA0CxK-4tYtIq3C5CaIzfwpFyrsQ) | Aceita um instante expresso em segundos e retorna uma tupla de tempo t com o horário local (t.tm\_isdst é 0 ou 1, dependendo se o horário de verão se aplica a segundos instantâneos pelas regras locais). | time.localtime() >> time.struct\_time.... |
| [mktime (tupletime)](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_mktime.htm&usg=ALkJrhj_t23u7iPVGEO7Nh0-0FczLUNcDg) | Aceita um instante expresso como uma tupla de tempo no horário local e retorna um valor de ponto flutuante com o instante expresso em segundos. | time.mktime(hora\_tupla)  >> 1583290216.0 |
| [sleep (segundos)](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_sleep.htm&usg=ALkJrhiNsP6ar1PHgdt_-HcrkF5hLXNzHg) | Suspende o segmento de chamada por segundos. | time.sleep(1) |
| [strftime (fmt [, tupletime])](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_strftime.htm&usg=ALkJrhjekI99iAZeJeOmqzQGs2sAw-C5oQ) | Aceita um instante expresso como uma tupla de tempo no horário local e retorna uma sequência que representa o instante conforme especificado pela sequência fmt. | Exemplo abaixo |
| [strptime (str, fmt = '% a% b% d% H:% M:% S% Y')](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_strptime.htm&usg=ALkJrhg7tUReOYkgr02m0SFNaBdcuWPB5A) | Analisa str de acordo com o formato string fmt e retorna o instante no formato de tupla de tempo. | Exemplo abaixo |
| [time()](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_time.htm&usg=ALkJrhiYy8e8MCErusjCO7BQDXg_Kb45UQ) | Retorna o instante de hora atual, um número de ponto flutuante de segundos. | time.time() >> 1583330687.0985613 |
| [tzset()](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&pto=nl&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialspoint.com/python/time_tzset.htm&usg=ALkJrhhMwLkx6rkw486gNqR6rr-foxiwRg) | Redefine as regras de conversão de horário usadas pelas rotinas da biblioteca. A variável de ambiente TZ especifica como isso é feito. | time.tzset()  >> None |

Existem dois atributos importantes disponíveis no módulo de tempo -

|  |  |  |
| --- | --- | --- |
| **Atributo.** | **Descrição** | **Exemplo** |
| timezone | É o deslocamento em segundos do fuso horário local (sem DST) do UTC (> 0 nas Américas; <= 0 na maior parte da Europa, Ásia, África). | time.timezone  >> 10800 |
| tzname | É um par de sequências dependentes do código do idioma, que são os nomes do fuso horário local sem e com o horário de verão, respectivamente. | time.tzname >> ('Hora oficial do Brasil', 'Horário brasileiro de verão') |

A função time.strftime permite a formatação do tempo em string. Voce pode passar o formato desejado para a string, seguindo os códigos de formatação da tabela abaixo.

|  |  |
| --- | --- |
| **Código** | **Descrição** |
| %a | dia da semana abreviado |
| %A | nome do dia da semana |
| %b | nome do mes abreviado |
| %B | nome do mês completo |
| %c | data e hora conforme configuração regional |
| %d | dia do mês (01-31) |
| %H | hora no formato 24 h (00-23) |
| %I | hora no formato 12 h |
| %j | dia do ano 001-366 |
| %m | mês (01-12) |
| %M | minutos (00-59) |
| %p | AM ou PM |
| %S | segundos (00-61) |
| %U | número da semana (00-53), em que a semana 1 começa após o primeiro domingo. |
| %w | dia da semana (0-6), em que 0 e o domingo |
| %W | número da semana (00-53), em que a semana 1 comeca apos a primeira segunda-feira |
| %x | representação regional da data |
| %X | representação regional da hora |
| %y | ano (00-99) |
| %Y | ano com 4 digitos |
| %Z | nome do fuso horário |
| % | símbolo de % |

*# time.struct\_time(tm\_year=2015, tm\_mon=7, tm\_mday=29, tm\_hour=0,   
# tm\_min=0, tm\_sec=0, tm\_wday=2, tm\_yday=210, tm\_isdst=-1)*print(time.strptime(**"29 Jul 2015"**, **"%d %b %Y"**))  
estrutura = time.strptime(**"Monday July 29 2015"**, **"%A %B %d %Y"**)  
timetup = time.gmtime()  
print(estrutura)  
  
*# time.struct\_time(tm\_year=2020, tm\_mon=3, tm\_mday=4, tm\_hour=13,  
# tm\_min=37, tm\_sec=39, tm\_wday=2, tm\_yday=64, tm\_isdst=0)*print(timetup)  
print(time.strftime(**'%Y-%m-%dT%H:%M:%SZ'**, timetup)) *# 2020-03-04T13:37:39Z*print(time.strftime(**'%Y-%m-%dT%H:%M:%SZ'**, estrutura)) *# 2015-07-29T00:00:00Z*

Módulo datetime

É possível no Python também fazer manipulação de data e hora no módulo datetime. Para evitar repetir o nome datetime, voce tambem pode importar apenas o tipo que vai usar:

**from** datetime **import** datetime **as** dt  
print(dt.now()) *# 2020-03-04 11:48:51.058566*

**import** datetime  
  
hoje = datetime.datetime.today()  
  
print(hoje) *# 2020-03-04 11:32:31.113566*print(datetime.date.today()) *# 2020-03-04*print(datetime.date(2018, 11, 1)) *# 2018-11-01*print(datetime.datetime.now()) *# 2020-03-04 11:36:23.811566  
  
# 2018-11-01 13:45:11.877875*print(datetime.datetime(2018, 11, 1, 13, 45, 11, 877875))   
  
print(hoje.year) *# 2020*print(hoje.month) *# 3*print(hoje.day) *# 4*

O tipo datetime, alem das propriedades da data (date), permite acessar a hora, os minutos e os segundos independentemente:

hoje = datetime.datetime.today()  
  
print(datetime.datetime.now().time()) *# 11:36:23.812561*print(datetime.time(13, 45, 26, 437575)) *# 13:45:26.437575*print(hoje.hour) *# 11*print(hoje.minute) *# 42*print(hoje.second) *# 46*print(hoje.microsecond) *# 355561*print(hoje.weekday()) *# 2*

print(hoje.isoweekday()) *# 3*

No caso, 2 representa a quarta-feira, pois começamos a contar de 0 e a partir de segunda-feira. Para obter um valor mais compativel com o que usamos no Brasil, utilize isoweekday.

Uma conversão muito usada na web e principalmente para converter data e hora em string e o formato ISO 8601, que representa uma data no formato ano, mes, dia, hora, minutos, segundos e microssegundos.

print(hoje.isoformat()) *# 2020-03-04T11:50:36.512566*

Novos objetos podem ser criados passando valores como em:

data = datetime.date(year=2019, month=9, day=7)  
print(data) *# 2019-09-07*data = datetime.date(2020, 11, 3)  
print(data) *# 2020-11-03*

Podemos tambem calcular valores futures ou passados usando timedelta:

daqui\_a\_90\_dias = hoje + datetime.timedelta(days=90)  
print(daqui\_a\_90\_dias) *# 2020-06-02 11:55:51.779566*calc\_data = hoje - datetime.timedelta(minutes=30)  
print(calc\_data) *# 2020-03-04 11:25:51.779566*calc\_data = hoje - datetime.datetime(2018, 11, 1, 13, 16, 17, 94101)  
print(calc\_data) *# 488 days, 22:39:34.685465*

now = parse(**"Sat Oct 11 17:13:46 UTC 2003"**)  
print(now) *# 2003-10-11 17:13:46+00:00*today = now.date()  
print(**"Today is: %s"** % today) *# Today is: 2003-10-11*year = rrule(YEARLY,dtstart=now,bymonth=8,bymonthday=13,byweekday=FR)[0].year  
*# Year with next Aug 13th on a Friday is: 2004*print(**"Year with next Aug 13th on a Friday is: %s"** % year)  
  
rdelta = relativedelta(easter(year), today)  
*# How far is the Easter of that year: relativedelta(months=+6)*print(**"How far is the Easter of that year: %s"** % rdelta)  
*# And the Easter of that year is: 2004-04-11*print(**"And the Easter of that year is: %s"** % (today+rdelta))

Se precisar converter uma tupla em segundos, utilize a função timegm do modulo calendar.

## Módulo calendário

O módulo de calendário oferece uma ampla variedade de métodos para jogar com calendários anuais e mensais. Aqui, imprimimos um calendário para um determinado mês.

**import** calendar  
  
cal = calendar.month(2020, 3)  
print(cal)  
**"""  
 March 2020  
Mo Tu We Th Fr Sa Su  
 1  
 2 3 4 5 6 7 8  
 9 10 11 12 13 14 15  
16 17 18 19 20 21 22  
23 24 25 26 27 28 29  
30 31  
"""**

Por padrão, o calendário assume segunda-feira como o primeiro dia da semana e domingo como o último. Para alterar isso, chame a função calendar.setfirstweekday().

Aqui está uma lista de funções disponíveis com o módulo de calendário.

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| calendar (ano, w = 2, l = 1, c = 6) | Retorna uma sequência multilinha com um calendário para o ano do ano formatado em três colunas separadas por espaços em c. w é a largura em caracteres de cada data; cada linha tem comprimento 21 \* w + 18 + 2 \* c. l é o número de linhas para cada semana. |
| firstweekday () | Retorna a configuração atual para o dia da semana que começa a cada semana. Por padrão, quando o calendário é importado pela primeira vez, é 0, o que significa segunda-feira. |
| isleap (ano) | Retorna True se ano é um ano bissexto; caso contrário, False. |
| leapdays (y1, y2) | Retorna o número total de dias bissextos nos anos dentro do intervalo (y1, y2). |
| month (ano, mês, w = 2, l = 1) | Retorna uma sequência multilinha com um calendário para mês mês do ano ano, uma linha por semana mais duas linhas de cabeçalho. w é a largura em caracteres de cada data; cada linha tem comprimento 7 \* w + 6. l é o número de linhas para cada semana. |
| monthcalendar (ano, mês) | Retorna uma lista de listas de entradas. Cada sub-lista indica uma semana. Dias fora do mês, mês do ano ano são definidos como 0; os dias do mês são definidos para o dia do mês 1 ou superior. |
| monthrange (ano, mês) | Retorna dois números inteiros. O primeiro é o código do dia da semana para o primeiro dia do mês mês no ano ano; o segundo é o número de dias no mês. Os códigos dos dias da semana são de 0 (segunda-feira) a 6 (domingo); os números dos meses são de 1 a 12. |
| prcal (ano, w = 2, l = 1, c = 6) | Como imprimir calendar.calendar (ano, w, l, c). |
| prmonth (ano, mês, w = 2, l = 1) | Como imprimir calendar.month (ano, mês, w, l). |
| setfirstweekday (weekday) | Define o primeiro dia de cada semana como código de dia da semana dia da semana. Os códigos dos dias da semana são de 0 (segunda-feira) a 6 (domingo). |
| timegm (tupletime) | O inverso de time.gmtime: aceita um instante de tempo na forma de tupla de tempo e retorna o mesmo instante que um número de segundos de ponto flutuante desde a época. |
| weekday (ano, mês, dia) | Retorna o código do dia da semana para a data especificada. Os códigos dos dias da semana são de 0 (segunda-feira) a 6 (domingo); os números dos meses são de 1 (janeiro) a 12 (dezembro). |

Módulo pytz

Ou seja, dependendo da configuração do ambiente no qual o código está rodando, você pode obter uma data e hora diferentes. Para não depender desta configuração e usar um timezone específico, uma alternativa é usar o módulo pytz (disponível no PyPI):

**from** datetime **import** datetime  
**from** pytz **import** timezone  
  
*# obter a data e hora em um timezone específico*d = datetime.fromtimestamp(1556322834.483199999, tz=timezone(**'America/Sao\_Paulo'**))  
print(d) *# 2019-04-26 20:53:54.483200-03:00  
  
# obter o timestamp a partir de uma data/hora e timezone*d = timezone(**'America/Sao\_Paulo'**).localize(datetime(2019, 4, 26, 10, 30, 0, 0))  
print(d.timestamp()) *# 1556285400.0*

Repare que agora, ao imprimir o datetime, também é mostrado o offset -03:00 (que é a diferença em relação a UTC, que o timezone America/Sao\_Paulo usa naquele instante específico). No código acima também há um exemplo para converter uma data e hora específicas em um timestamp (lembrando que o timezone utilizado faz com que o valor do timestamp seja diferente, já que a mesma data e hora acontece em instantes diferentes em cada parte do mundo). E se você usar apenas o construtor de datetime (sem usar nenhum timezone), o retorno de timestamp() usará o timezone do ambiente no qual o código está rodando (e portanto pode variar).

Usar um timezone é importante para que o código não fique dependente da configuração de timezone do ambiente no qual o código roda - a menos que este seja o comportamento desejado, claro. Mas se quiser que o timestamp corresponda a uma data e hora em um timezone específico, é melhor usar o pytz. As informações de timezones mudam o tempo todo (há épocas em que há horário de verão e o offset muda, entre outros detalhes explicados neste link), e tentar manter isso manualmente é inviável. O pytz é atualizado de acordo com as versões do TZDB da IANA (o banco de informações de fusos-horários que várias linguagens e sistemas usam) e basta você usar o timezone correto (como America/Sao\_Paulo, Europe/London, etc), que o pytz se encarrega de verificar qual a data e hora correspondentes.

Módulo dateutil

Em Python você pode usar o módulo datetime. Se quiser trabalhar com somente a data (apenas o dia, mês e ano), pode usar um date. Infelizmente, não é possível usar timedelta, como foi feito para somar dias, já que timedelta só possui dias, mas não meses ou anos.

Nesse caso, uma alternativa é usar o módulo dateutil, disponível no PyPI, que possui a classe relativedelta:

**from** datetime **import** date  
**from** dateutil.relativedelta **import** relativedelta  
  
d = date(2016, 1, 31)  
print(d) *# 2016-01-31*d = d + relativedelta(months=1)  
print(d) *# somar 1 mês = 2016-02-29*d = d + relativedelta(years=1)  
print(d) *# somar 1 ano = 2017-02-28*

Como podemos ver, também são feitos os devidos ajustes semânticos nos resultados.

Se quiser, também pode usar um datetime, a diferença é que este também possui o horário. E se você criá-lo com d = datetime(2016, 1, 31), o horário é automaticamente setado para meia-noite.

LISTAS - ARRAY

Lista é um conjunto sequencial de valores, onde cada valor é identificado através de um índice. O primeiro valor tem índice 0. Uma lista em Python é declarada da seguinte forma:

**Nome\_Lista = [ valor1, valor2, ...,valorN]**

Definidas por colchetes ([int/float/str/object]), uma array é uma estrutura de dados mutável finita, ou seja, que pode ser alterada e que pode conter diversos tipos de dados aninhados na mesma array. Isto é, uma array NÂO precisa ser apenas homogenea em termos de tipos de dados. Um item adicionado a uma array, é sempre inserido no final da array. Entretanto, como a intenção tambem é usar uma array para representar uma matrix/vector, para que isso seja feita da maneira correta, ou seja, a qual possibilite a execução de cálculos, a array deve conter apenas números.

Observações:

1. O primeiro elemento de uma array tem índice 0 (zero);

2. O último indice de uma array é a quantidade de elementos menos 1 (um) (n - 1):

3. A quantidade de elementos de uma array é o índice do último elemento mais 1 (um) (n+ 1);

4. Um vector é uma matrix coluna:

5. Toda matrix/vector é uma array, mas nem toda array e uma matrix/vector;

Uma lista pode ter valores de qualquer tipo, incluindo outras listas .

L = [3, **'abacate'**, 9.7, [5, 6, 3], **"Python"**, (3, **'j'**)]  
print(L[2]) *# 9.7*print(L[3]) *# [5, 6, 3]*print(L[3][1]) *# 6*

Para alterar um elemento da lista, basta fazer uma atribuição de valor através do índice. O valor existente será substituído pelo novo valor.

L[3] = **'morango'**print(L) *# [3, 'abacate', 9.7, 'morango', 'Python', (3, 'j')]*

A tentativa de acesso a um índice inexistente resultará em erro.

L[7] = **'banana'**

File "C:/Users/x\_kat/Documents/Projetos/ProjectTestEnv/Teste\_02.py", line 72, in <module>

L[7] = 'banana'

IndexError: list assignment index out of range

Operações com listas

Na tabela a seguir estão algumas funções utilizadas para manipular listas

|  |  |  |
| --- | --- | --- |
| **Função** | **Descrição** | **Exemplo para L = [1,2,3,4]** |
| append(x) | adiciona urn novo valor na no final da lista. | L.append(100) >> [1, 2, 3, 4, 100] |
| count(x) | Conta o número de ocorrencias de x | L.count(3) >> 1 |
| del list[index] | remove um elemento da lista pelo indice. | del L[3] >> [1, 2, 3] |
| extend([ x, y, z]) | insere uma lista no final de outra lista. | L.extend([5.6,7]) >> [1, 2, 3, 4, 5, 6, 7] |
|  | a = [1, 7, 9] e b = [6, -1, 4] | c = a + b >> [1, 7, 9, 6, -1, 4] |
| in | verifica se um valor pertence a lista. | 3 in L >> True |
| index(x) | Retorna a posição do primeiro elemento x | L.index(3) >> 2 |
| insert(index, x) | insere uma elemento x no index | L.insert(2, 3) >> [1, 2, 3, 3, 4] |
| len | retoma o tamaimo da lista. | len(L) >> 4 |
| max | retoma o maior valor da lista. | max(L) >> 4 |
| min | retoma o menor valor da lists. | min(L) >> 1 |
| pop(index) | remove um elemento da lista pelo indice. | L.pop(3) >> [1, 2, 3] |
| remove(index) | Remove o elemento da lista no índice | L.remove(3) >> [1, 2, 4] |
| reverse() | inverte os elementos de uma lista. | L.reverse() >> [4, 3, 2, 1] |
| sort() | ordena em ordem crescente | L.sort() >> [1, 2, 3, 4] |
| sum | retoma soma dos elementos da lista. | sum(L) >> 10 |

Concatenação ( + )

a = [0, 1, 2]  
b = [3, 4, 5]  
c = a + b  
print(c) *# [0, 1, 2, 3, 4, 5]*

Repetição ( \* )

L = [1, 2]  
R = L \* 4  
print(R) *# [1, 2, 1, 2, 1, 2, 1, 2]*

Fatiamento

O fatiamento de listas é semelhante ao fatiamento de strings.

L = [3, **'abacate'**, 9.7, [5, 6, 3], **"Python"**, (3, **'j'**)]  
*# Seleciona os elementos das posições 1,2,3*print(L[1:4]) *# ['abacate', 9.7, [5, 6, 3]]  
# Seleciona os elementos a partir da posição 2*print(L[2:]) *# [9.7, [5, 6, 3], 'Python', (3, 'j')]  
# Seleciona os elementos até a posição 3*print(L[:4]) *# [3, 'abacate', 9.7, [5, 6, 3]]*

TUPLAS

Tupla, assim como a Lista é um conjunto sequencial de valores, onde cada valor é identificado através de um índice. A sintaxe básica é:

**Nome\_tupla = (valor1, valor2, ..., valorN) ou “valor1”, “valor2”, ..., “valorN”**

Definidas por parentesis ((int/float/str/object)), uma tuple pode conter diversos tipos de dados aninhados numa mesma tuple, como numero (int ou float), str e/ou outros tipos, por exemplo, object ou atá mesmo outra tuple. Em geral, usa-se este tipo de estrutura de dados para armazenar informações de entrada (input do banco de dados), para a aplicação que é desenvolvida.

Observação:

1. O primeiro item inserido na tuple, será sempre o primeiro elemento da tuple, o segundo será sempre o segundo e, assim por diante;

2. A principal diferença compara com uma list é de que uma vez inserido numa tuple o elemento NÂO pode ser alterado.

T = (1, 2, 3, 4, 5)  
print(T)  
print(T[3])  
T[3] = 8

File "C:/Users/x\_kat/Documents/Projetos/ProjectTestEnv/Teste\_02.py", line 258, in <module>

T[3] = 8

TypeError: 'tuple' object does not support item assignment

Dentre as utilidades das tuplas, destacam se as operações de empacotamento e desempacotamento de valores

, que permite atribuir os elementos armazenados em uma tupla a diversas variáveis.

T = (10, 20, 30, 40, 50)  
a, b, c, d, e = T  
print(**"a="**, a, **"b="**, b) *# a= 10 b= 20*print(**"d+e="**, d + e) *# d+e= 90*

Podemos utilizar \* para indicar vários valores a desempacotar

\*a, b = [1, 2, 3, 4, 5]  
print(a) *# [1, 2, 3, 4]*print(b) *# 5*a, \*b = [1, 2, 3, 4, 5]  
print(a) *# 1*print(b) *# [2, 3, 4, 5]  
  
# SyntaxError: two starred expressions in assignment*\*a, \*b = [1, 2, 3, 4, 5]

DICIONÁRIOS

Definida por chaves ({}), um dicionário é uma estrutura de dados bidimensional mutável, ou seja, que pode ser alterada e conter diversos tipos de dados aninhados na mesma dict, porém, um dado inserido está relacionado com uma chave de acesso (key) e apenas este dado possui esta atribuição. Isto é, uma dict é um tipo de set que pode ter valores repetidos, entretanto, possui uma palavra-chave única atribuida a cada dado.

A sintaxe básica é:

**Nome\_dicionario = { chave1 : chave2 : chave3 : chaveN :v alorN]:**

Observagao (oes):

1. NÂO se pode ter um ou mais elementos com a mesma key (palavra-chave);

2. O primeiro item inserido na dict, será sempre o primeiro item da diet;

3. Um item adicionado a uma dict, e sempre inserido no final da diet.

Os valores do dicionário não possuem ordem, por isso a ordem de impressão dos valores não é sempre a mesma

D = {**"arroz"**: 17.30, **"feijao"**: 12.50, **"carne"**: 23.90, **"alface"**: 3.40}  
  
*# {'arroz': 17.3, 'feijao': 12.5, 'carne': 23.9, 'alface': 3.4}*print(D)   
print(D[**"carne"**]) *# 23.9*

É possível acrescentar ou modificar valores no dicionário:

D[**"arroz"**] = 25.0  
D[**"feijao"**] = 8.80  
*# {'arroz': 25.0, 'feijao': 8.8, 'carne': 23.9, 'alface': 3.4}*print(D)

E emitirá mensagem de erro ao tentar acessar um item que não tem chave de acesso:

D = {**"arroz"**: 17.30, **"feijao"**: 12.50, **"carne"**: 23.90, **"alface"**: 3.40}  
print(D(**"tomate"**))

File "C:/Users/x\_kat/Documents/Projetos/ProjectTestEnv/Teste\_02.py", line 281, in <module>

print(D("tomate"))

TypeError: 'dict' object is not callable

|  |  |  |
| --- | --- | --- |
| **Comando** | **Descricao** | **Exemplo** |
| clear() | Limpa o dicionano a | D.clear() |
| del | Exclui um item infomando a chave k. | del D["alface"] >>  {'arroz': 25.0, 'feijao': 8.8, 'carne': 23.9} |
| get(k, m) | Retorna D[k], ou m se k nao estiver em D | D.get("arroz") >> 17.3  D.get("17.3") >> None  D.get("pepino", "não existe" ) >> "não existe" |
| in | Verifica se uma chave existe no dicionáno. | "alface" in D >> True |
| items() | Gera uma lista de pares (key, value) | D.items() >>  dict\_items([('arroz', 25.0), ('feijao', 8.8), ('carne', 23.9), ('alface', 3.4)]) |
| keys() | Obtém as chaves de um dicionáno | D.keys() >>  dict\_keys(['arroz', 'feijao', 'carne', 'alface']) |
| len(a) | Numero de elementos em a | len(D) >> 4 |
| not in | True se k nao for uma das chaves de a | "alface" not in D >> False |
| pop(k[,xJ) | Retorna D[k], ou m se k nao estiver em D | D.pop("arroz ") >> 17.3  D.pop("pepino ", "não existe")) >> "não existe" |
| update(b) | Adiciona | a = {"aaa":10, "bbb":20}  b = {"ddd":40, "eee":50}  a.update(b) >>  {'aaa': 10, 'bbb': 20, 'ddd': 40, 'eee': 50} |
| values() | Obtém os valores de um dicionáno. | D.values() >>  dict\_values([25.0, 8.8, 23.9, 3.4]) |

Os dicionários podem ter valores de diferentes tipos.

Dx = {2: **"carro"**, 3: [4, 5, 6], 7: (**'a'**, **'b'**), 4: 173.8}  
print(Dx[7]) *# ('a', 'b')*

Conversão list para dicionário:

d1 = [**'aaa'**, **'bbb'**, **'ccc'**]  
d2 = [**'1'**, **'2'**, **'3'**]  
  
lista = list(zip(d1, d2))  
*# [('aaa', '1'), ('bbb', '2'), ('ccc', '3')]*print(lista)  
  
dicio = dict(zip(d1, d2))  
*# {'aaa': '1', 'bbb': '2', 'ccc': '3'}*print(dicio)

Concatenar dicionários em Python não é tão fácil quanto strings, tuplas ou listas. Havia um método mais fácil no Python 2.7, mas ele não funciona mais no Python 3 e posteriores.

ages = {**"John"**: 30}  
all\_ages = dict(ages.items() + {**"Mary"**: 28}.items())

print(all\_ages) *# {‘John’: 30, ‘Mary’: 28}*

No Python 3 devia se utilizar de um for e utilizar a função update para atualizar um dicionário vazio:

all\_ages = {} *#declaring a empty dictionary first*ages = {**"John"**: 30}  
**for** x **in** (ages,{**"Mary"**: 28}): all\_ages.update(x)  
print(all\_ages) *# {‘John’: 30, ‘Mary’: 28}*

Como esta implantação contrariava princípios da programação funcional, a partir do 3.5 (PEP 448) foi proposto o operador \*\*:

ages = {**"John"**: 30}  
all\_ages = dict(ages, \*\*{**"Mary"**: 28})  
print(all\_ages) *# {‘John’: 30, ‘Mary’: 28}*

CONJUNTOS

Sets (ou, como iremos chamar daqui para a frente, conjuntos) são estruturas disponíveis como builtins do Python, utilizadas para representar coleções desordenadas de elementos únicos. Definidas por chaves ({}), uma set é uma estrutura de dados mutavel, ou seja, que pode ser alterada e conter diversos tipos de dados aninhados na mesma set, assim como e feito com uma list É importante sempre lembrar dos conjuntos por suas duas principais características:

* Os elementos NÃO são armazenados em uma ordem específica e confiável;
* Conjuntos NÃO contém elementos repetidos.
* O primeiro item inserido na set, será sempre o primeiro elemento da set;
* Um item adicionado a uma set, e sempre inserido no final da set.

A característica número 1 é importante, porque o desenvolvedor jamais deve confiar na ordenação de um conjunto, visto que a ordem em que os elementos são mantidos nos conjuntos varia de implementação para implementação do interpretador Python.

A diferença entre set e frozenset, é que o primeiro é mutável, já o segundo não. Os objetos existentes na linguagem podem ser classificados como:

* **Objeto imutável** => tem conteúdo fixo, ou seja, não pode ser alterado sem que o objeto seja reconstruído. Os tipos numéricos (int. float, complex), strings, tuplas e frozenset são imutáveis, de modo que, quando um novo conteúdo for atribuído ao objeto, sua instância anterior é removida, e uma nova instancia criada. Frozensets são como conjuntos, exceto que não podem ser alterados, ou seja, são imutáveis:

cities = frozenset([**"Frankfurt"**, **"3asel"**, **"Freiburg"**])  
cities.add(**"Strasbourg"**)

File "C:/Users/x\_kat/Documents/Projetos/ProjectTestEnv/Teste\_02.py", line 337, in <module>

cities.add("Strasbourg")

AttributeError: 'frozenset' object has no attribute 'add'

* **Objeto mutável** => Os tipos lista, dicionário e set são mutáveis, de modo que podem ter seu conteúdo alterado, sem que sua instancia seja recriada. Set são implementados de uma maneira que não permite objetos mutáveis. O exemplo a seguir demonstra que não podemos incluir, por exemplo, listas como elementos:

cities = set(((**"Python"**, **"Perl"**), (**"Paris"**, **"3erlin"**, **"London"**)))  
cities = set(([**"Python"**, **"Perl"**], [**"Paris"**, **"Berlin"**, **"London"**]))

File "C:/Users/x\_kat/Documents/Projetos/ProjectTestEnv/Teste\_02.py", line 336, in <module>

cities = set((["Python","Perl"], ["Paris", "Berlin", "London"]))

TypeError: unhashable type: 'list'

Embora conjuntos não possam conter objetos mutáveis, eles são mutáveis:

cities = set([**"Frankfurt"**, **"3asel"**, **"Freiburg"**])  
cities.add(**"Strasbourg"**)  
*# {'Freiburg', 'Frankfurt', '3asel', 'Strasbourg'}*print(cities)

Operações entre sets e frozensets são permitidas, veja que a ordem em que eles aparecem não importa

conjunto = set(**'234'**) | frozenset(**"45"**)  
print(conjunto) *# {'5', '3', '4', '2'}*print(set(**'45'**) | frozenset(**"234"**)) *# {'5', '3', '4', '2'}  
# frozenset({'5', '3', '4', '2'})*print(frozenset(**'45'**) | set(**'234'**))  
*# frozenset({'5', '3', '4', '2'})*print(frozenset(**'234'**) | set(**'45'**))

Outra coisa importante, é que podemos passar iteráveis tanto na criação do set como nas operações usando os métodos. Mas as operações usando os operadores devem ser obrigatoriamente entre sets:

*# {'4', '2', '5', '6', '3'}*print(set(**'234'**).union(**"56"**))

print(set(**'234'**) | [5, 6])

File "16\_conjuntos.py", line 28, in <module>

print(set('234') | [5, 6])

TypeError: unsupported operand type(s) for |: 'set' and 'list'

Os sets, por serem mutáveis, também possuem métodos que adicionar e remover novos elementos. Como os frozensets são imutáveis, não possuem esses métodos.

conjuntol = {**'a'**, **'b'**, 4, 7, 8, 9, **'c'**, **'d'**}  
*# {4, 7, 8, 9, 'c', 'd', 'b', 'a'}*print(conjuntol)  
  
conjunto2 = {**'nada, tudo'**}  
*# {'nada, tudo'}*print(conjunto2)  
  
conjunto3 = {**'nada'**, **'tudo'**}  
*# {'tudo', 'nada'}*print(conjunto3)  
  
conjunto4 = set(**'nada, tudo'**)  
*# {'o', 't', ' ', 'u', 'a', 'n', ',', 'd'}*print(conjunto4)  
  
conjuntol.update(conjunto2)  
*# {'nada, tudo', 4, 7, 8, 9, 'c', 'a', 'b', 'd'}*print(conjuntol)  
  
conjuntol.update(conjunto3)  
*# {'tudo', 'nada, tudo', 'nada', 4, 7, 8, 9, 'c', 'a', 'b', 'd'}*print(conjuntol)  
  
conjuntol.update(conjunto4)  
*# {'b', 'tudo', 4, ',', 7, 8, 9, 'nada, tudo', 'c',   
# 'nada', ' ', 'a', 'u', 'o', 'd', 't', 'n'}*print(conjuntol)

Contido e Contém

conjuntol = {**'a'**, **'b'**, 4, 7, 8, 9, **'c'**, **'d'**}  
print(conjuntol) *# {4, 'c', 7, 8, 9, 'd', 'b', 'a'}*conjunto2 = set([**'a'**, **'b'**, **'c'**])  
print(conjunto2) *# {'c', 'b', 'a'}*print(conjuntol.issubset(conjunto2)) *# False*print(conjunto2.issubset(set(**'abcdefg'**))) *# True*print(conjunto2 <= set(**'abcdefg'**)) *# True*print(set(**'abcdefg'**) >= conjunto2) *# True*

União

conjuntol = {**'a'**, **'b'**, 4, 7, 8, 9}  
*# {4, 7, 8, 'a', 9, 'b'}*print(conjuntol)  
  
conjunto2 = {**'a'**, **'b'**, **'c'**}  
*# {'c', 'b', 'a'}*print(conjunto2)  
  
*# OU conj\_union = conjuntol.union(conjunto2)*conj\_union = conjuntol | conjunto2  
*# {4, 7, 8, 9, 'a', 'c', 'b'}*print(conj\_union)

Interseção

*# OU conj\_intesect = conjuntol.intersection(conjunto2)*conj\_intesect = conjuntol & conjunto2  
*# {'b', 'a'}*print(conj\_intesect)

Essa operação é muito útil quando precisamos descobrir elementos que duas listas possuem em comum:

lista1 = [1, 2, 3]  
lista2 = [2, 4, 3]  
*# {2, 3}*print(set(lista1).intersection(lista2))

Perceba que convertemos lista1 para conjunto para podermos usar o método intersection; já lista2 não precisou ser convertida, pois esses métodos exigem que apenas o primeiro argumento seja um conjunto. Poderíamos obter o resultado da interseção como uma lista também:

lista3 = list(set(lista1).intersection(lista2))  
print(lista3) *# [2, 3]*

O método intersection não modifica os conjuntos recebidos como parâmetro. Se quisermos que o resultado da interseção seja gravado como novo valor do primeiro conjunto, ao invés de retornar o novo conjunto como resultado, podemos usar o método intersection\_update:

conjuntol = {**'a'**, **'b'**, 4, 7, 8, 9, **'c'**, **'d'**}  
*# {4, 7, 8, 9, 'c', 'd', 'b', 'a'}*print(conjuntol)  
  
conjunto2 = {**'nada, tudo'**}  
*# {'nada, tudo'}*print(conjunto2)  
  
conjunto3 = set(**'nada, tudo'**)  
*# {'o', 't', ' ', 'u', 'a', 'n', ',', 'd'}*print(conjunto3)  
  
conjuntol.intersection\_update(conjunto3)  
*# {'a', 'd'}*print(conjuntol)  
  
conjuntol.intersection\_update(conjunto2)  
*# set()*print(conjuntol)  
  
conjuntol.intersection\_update(conjunto3)  
*# set()*print(conjuntol)

Diferença

conjuntol = set(**'1234'**)  
*# {'2', '4', '3', '1'}*print(conjuntol)  
  
conjunto2 = {**'123, 1'**, 1, **'2'**, 2, 3}  
*# {'2', 1, 2, 3, '123, 1'}*print(conjunto2)  
  
*# OU conjuntol - conjunto2*conj\_diferenca = conjuntol.difference(conjunto2)  
*# {'3', '1', '4'}*print(conj\_diferenca)  
  
conj\_diferenca = conjunto2.difference(conjuntol)  
*# {1, 2, 3, '123, 1'}*print(conj\_diferenca)

Diferença simétrica

Diferença simétrica é uma operação sobre os dois conjuntos, que retorna todos os elementos (de ambos os conjuntos a e b) que pertencem a somente um dos conjuntos.

conj\_diferenca = conjuntol.symmetric\_difference(conjunto2)  
*# {1, 2, '123, 1', 3, '3', '4', '1'}*print(conj\_diferenca)

Pertinência

Além das operações tradicionais de união, interseção e diferença, também temos operações de verificação de pertinência. A seguir veremos algumas.

Para verificar se um determinado elemento pertence a um conjunto, podemos usar o já conhecido operador de pertinência in:

conjuntol = {**'a'**, **'b'**, 4, 7, 8, 9, **'c'**, **'d'**}  
print(**'a' in** conjuntol) *# True*print(**'7' in** conjuntol) *# False*print(5 **in** conjuntol) *# False*

Também podemos verificar se um conjunto é um subconjunto de outro ou é superconjunto de outro:

conjunto2 = set([**'a'**, **'b'**, **'c'**])  
print(conjunto2.issubset(conjuntol)) *# True*print(conjuntol.issubset(conjunto2)) *# False*print(conjuntol.issuperset(conjunto2)) *# True*

Outra relação importante que podemos checar é a disjunção entre dois conjuntos. Dois conjuntos são disjuntos se tiverem interseção nula. Exemplo:

conjuntol = {**'a'**, **'b'**, **'c'**, **'d'**}  
conjunto2 = set([**'a'**, **'b'**, **'c'**])  
conjunto3 = set([**'x'**, **'z'**])  
  
print(conjuntol.isdisjoint(conjunto2)) *# False*print(conjunto2.isdisjoint(conjunto3)) *# True*

Removendo elementos duplicados de uma sequência

Por definição, um conjunto é uma coleção de valores únicos (e desordenados). Assim sendo, se passarmos ao construtor de conjuntos uma lista com valores repetidos, esses valores serão eliminados de forma a permanecer apenas um deles. Exemplo:

lista1 = [1, 1, 1, 1, 2, 2, 3]  
conjuntol = set(lista1)  
*# {1, 2, 3}*print(conjuntol)  
*# OU voltar como lista [1, 2, 3]*print(list(conjuntol))

ATENÇÃO: a operação acima pode (e, com grandes chances, irá) bagunçar a lista. Ou seja, a ordem original irá se perder.

Métodos para conjuntos

|  |  |
| --- | --- |
| **Sintaxe** | **Descrição** |
| s.difference\_update(t) | Remove os itens que estão no conjunto t do conjunto s. |
| s.intersection\_update(t) | Faz com que o conjunto s contenha a interseção dele com t. |
| s.symetric\_difference\_update(t) | Faz com que o conjunto s contenha a diferença simétrica dele com t. |
| s.isdisjoin(t) | Retorna True se s e t não tem nenhum item em comum. |
| s.issubset(t) (s <= t) | Retorna True se s é igual a ou um subconjunto de t. |
| s.issuperset(t) (s >= t) | Retorna True se s é igual a ou t é um subconjunto de s. |
| s.discard(x) | Remove o item x do conjunto s. |
| s.remove(x) | Remove o item x se ele estiver em s se não retorna um KeyError. |
| s.update(t) | Adiciona cada item do conjunto s que não está no conjunto t para o conjunto t. |

GERADORES.

Iteradores Python

Simplificando, um iterador em Python é qualquer tipo Python que pode ser usado com um loop for. Listas, tuplas, dicionários e sets Python são todos exemplos de iteradores embutidos. Alguém pode perguntar: "O que faz desses tipos um iterador, e essa é uma propriedade apenas dos tipos embutidos do Python?"

Esses tipos são iteradores porque eles implementam o protocolo iterador. Então, O que é um protocolo iterador? Para responder esta pergunta, vamos precisar fazer outro pequeno desvio. Em Python, existem alguns métodos especiais, comumente chamados como métodos mágicos. Pode parecer estranho, mas apenas fique comigo e acredite pela fé no que digo, pelo menos, até chegarmos à orientação a objetos em Python.

Esses métodos normalmente não são chamados explicitamente no código, mas são chamados implicitamente durante sua execução. Um exemplo muito familiar desses métodos mágicos, é o método \_\_init\_\_, que é mais ou menos como se fosse um construtor que é chamado durante a inicialização de um objeto Python. Semelhante a maneira como o método \_\_init\_\_ tem de ser implementado na inicialização de um objeto personalizado, o protocolo iterador tem uma série de métodos mágicos que precisam ser implementados em qualquer objeto que queira ser usado como um iterador.

Esses são os seguintes métodos:

O método \_\_iter\_\_ que é chamado na inicialização de um iterador. Ele deve retornar um objeto que tem o método next() (no Python 3 este método foi mudado para \_\_next\_\_).

O método next que é chamado sempre que a função global next() é invocada com o iterador como argumento. O método iterador nextdeve retornar o próximo valor do iterável. Quando um iterador é usado com um loop for, o for chama implicitamente o método next(). Este método levanta uma exceção StopIteration quando não existe mais nenhum novo valor, para sinalizar o fim da iteração.

Qualquer classe Python pode ser definida para agir como um iterador, desde que o protocolo iterador seja implementado. Isto é ilustrado através da implementação de um simples iterador que retorna os números da sequência Fibonacci até um determinado valor máximo.

Um gerador é, de fato, um iterador, mas com uma manipulação muito mais facilitada! Ele consegue fazer todo o trabalho que queremos que faça, mas sem precisar de todo o nosso esforço em construir uma classe que implemente \_\_iter\_\_() para ser iterável e enfim \_\_next\_\_() para ser iterador.

O código abaixo implementa o iterador na class Fib. Repare que def \_\_next \_\_ (self) é para Python 3, para Python 2, você precisa adicionar o método next (). O código funcionará no Python 3 e no Python 2:

**class** Fib:  
 **def** \_\_init\_\_(self, max):  
 super(Fib, self).\_\_init\_\_()  
 self.max = max  
  
 **def** \_\_iter\_\_(self):  
 self.a = 0  
 self.b = 1  
 **return** self  
  
 **def** \_\_next\_\_(self):  
 fib = self.a  
 **if** fib > self.max:  
 **raise** StopIteration  
 self.a, self.b = self.b, self.a + self.b  
 **return** fib  
  
 **def** next(self):  
 **return** self.\_\_next\_\_()  
  
*# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987***for** n **in** Fib(1000):  
 print(n, end=**' '**)

Nós também podemos ir mais em frente e implementar nossa própria função range personalizada para loop em números. Esta simples implementação apenas entra no loop partindo do 0 para cima.

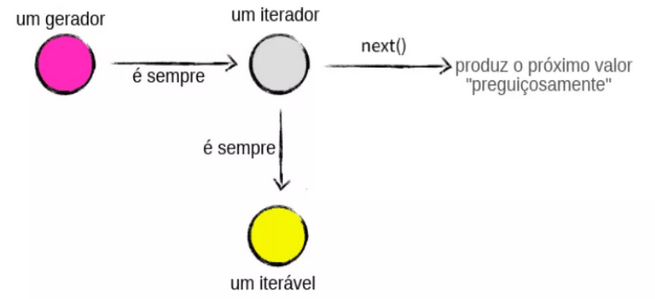
**class** CustomRange:  
 **def** \_\_init\_\_(self, max):  
 self.max = max  
  
 **def** \_\_iter\_\_(self):  
 self.curr = 0  
 **return** self  
  
 **def** \_\_next\_\_(self):  
 numb = self.curr  
 **if** self.curr >= self.max:  
 **raise** StopIteration  
 self.curr += 1  
 **return** numb  
  
 **def** next(self):  
 **return** self.\_\_next\_\_()  
  
*# 0 1 2 3 4 5 6 7 8 9***for** n **in** CustomRange(10):  
 print(n, end=**' '**)

Agora, nós temos um entendimento básico sobre iteradores, mas não como eles se relacionam com geradores. Em resumo, geradores são iteradores. A PEP 255, que descreve simples geradores, refere-se a geradores pelo seu nome completo: generator-iterator (gerador de iteradores). Geradores são utilizados quer chamando o método next no objeto gerador, ou usando o objeto gerador em um loop for.

Em Python, funções geradoras ou apenas geradores retornam objetos geradores. Esses geradores são funções que contêm a palavra reservada yield. Ao invés de ter que escrever cada gerador com o método \_\_iter\_\_ e next, que é bastante complicado, Python fornece a palavra reservada yield que provê uma maneira fácil para definir geradores. Por exemplo, o iterador de Fibonacci pode ser remodelado como um gerador usando a palavra reservada yield, como mostrado abaixo:

**def** fib(max):  
 a, b = 0, 1  
 **while** a < max:  
 **yield** a  
 a, b = b, a + b  
  
*# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987***for** n **in** fib(1000):  
 print(n, end=**' '**)

O uso da palavra reservada yield simplifica muito a criação do gerador.



Os geradores em Python são uma maneira simples de criar um objeto iterável sem a necessidade de construí-lo dentro de uma classe. De forma bem resumida, um objeto iterável consiste em um conjunto finito de dados¹ cujo conteúdo é tratado um elemento por vez, iniciando do primeiro e seguindo até o último, quando a iteração é terminada.

Também é possível produzir geradores que contém séries infinitas de dados. Alguns tipos de dados em Python têm também a característica de serem iteráveis. Por exemplo, é possível iterar uma variável do tipo string…

texto = **'um elemento por vez'***# u/m/ /e/l/e/m/e/n/t/o/ /p/o/r/ /v/e/z/***for** i **in** texto:  
 print(i + **'/'**, end=**''**)

Ou seja, o Python processa individualmente cada elemento (caractere) da string até que não reste mais nenhum e encerra o processo.

E além das strings outros tipos de dados nativos que também suportam iteração são os dicionários, listas e tuplas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Listas / Array** | **Tuplas** | **Dicionários** | **Conjuntos** |
| Simbolo | Conchetes [] | Parenteses () | Chaves {key:value} | Chaves {} |
| Ordem dos elementos | Fixa | Fixa | Mantida a partir do Python 3.7 | Indeterminada |
| Tamanho | Variável | Fixo | Variável | Variável |
| Elementos repetidos | Sim | Sim | Pode repetir valores, mas as chaves devem ser únicas | Não |
| Pesquisa | Sequencial, índice numérico | Sequencial, índice numérico | Direta por chave | Direta por chave |
| Alterações | Sim | Não | Sim | Sim |
| Uso Primário | Sequências | Sequências constantes | Dados indexados por chave | Verificação de unicidade, operações com conjuntos |

Em todos os aspectos uma função geradora é uma função como qualquer outra. Sua principal diferença está no fato de não utilizar a instrução return, que encerra a execução e envia o resultado, e sim a instrução yeld, que passa um elemento/resultado parcial, mas não interrompe a execução desta. Imagine uma função, que retorne todos os números inteiros entre 0 e max\_number:

**def** numbers\_up\_to(max\_number):  
 output = []  
 **for** number **in** range(max\_number + 1):  
 output.append(number)  
 **return** output  
  
*# 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15***for** i **in** numbers\_up\_to(15):  
 print(i, end=**' '**)

Escrita assim essa função cria uma lista [0, 1, 2, 3, 4, …] até chegar no max\_number. Ela já cria essa lista, aloca na memória, guardando toda a lista e seu conteúdo. Ocupa espaço, usa recursos de hardware para isso. E não tem problema algum se o max\_number for pequeno, mas tente usar essa função aí com um mol, ou seja, com numbers\_up\_to (623 \* 10 \*\* 21).

Para isso temos uma alternativa mais eficiente: geradores! Vamos transformar essa função em um gerador. É simples: não criamos lista nenhuma e usamos o yield ao invés do return:

**def** numbers\_up\_to(max\_number):  
 **for** number **in** range(max\_number + 1):  
 **yield** number  
  
*# 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 .... n***for** i **in** numbers\_up\_to(623 \* 10 \*\* 21):  
 print(i, end=**' '**)

Agora tente usar essa função com o número gigante: numbers\_up\_to\_as(623 \* 10 \*\* 21). Ele só vai calcular o primeiro elemento da sequência quando precisar dele. E vai te entregar 1 e parar. Não processa mais nada, não aloca nada em memória. Nadinha. Até você pedir o próximo número. Aí ele esquece do primeiro e te entrega o segundo. E assim vai. Você vai pedindo e ele vai entregando um por vez, o terceiro, depois o quarto, depois o quinto e assim por diante. Um por vez.

Ao invés de criar a lista toda, ele cria um gerador (de listas, por exemplo, mas um iterável) e vai calculando um a um os elementos, de acordo com a necessidade de acessá-los… e de fato ele só vai calcular alguma coisa a cada next() – que é a função chamada internamente se você passar um gerador para um for, por exemplo.

Mas o next() também pode ser usado manualmente pois utiliza avaliação preguiçosa, o que é ótimo para explorar:

my\_first\_generator = numbers\_up\_to(42)  
print(next(my\_first\_generator)) *# 0*print(next(my\_first\_generator)) *# 1*print(next(my\_first\_generator)) *# 2*print(next(my\_first\_generator)) *# 3*print(next(my\_first\_generator)) *# 4*

A função range define um intervalo de valores inteiros. Associada a list(), cria uma lista com os valores do intervalo.

A função range() pode ter de 1 a 3 parâmetros:

* range(n) => gera um intervalo de 0 a n-1
* range(i , n) => gera um intervalo de i a n-1
* range(i , n, p) => gera um intervalo de i a n-1 com intervalo p entre os números

ran\_1 = list(range(5))  
print(ran\_1) *# [0, 1, 2, 3, 4]*ran\_1 = list(range(3, 8))  
print(ran\_1) *# [3, 4, 5, 6, 7]*ran\_3 = list(range(2, 11, 3))  
print(ran\_3) *# [2, 5, 8]*

LIST COMPREHENSIONS

*List Comprehension* foi concebida na [PEP 202](https://www.python.org/dev/peps/pep-0202/) e é uma forma concisa de criar e manipular listas.

Sua sintaxe básica é:

**[expr for item in lista]**

Em outras palavras: aplique a expressão expr em cada item da lista. Dado o seguinte código:

lista = []  
**for** item **in** range(10):  
 lista.append(item \*\* 2)  
*# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]*print(lista)

Podemos reescrevê-lo, utilizando list comprehensions, da seguinte forma, aplique a potência de 2 em todos os itens da lista.

lista = [item \*\* 2 **for** item **in** range(10)]  
*# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]*print(lista)

Outro Exemplo: dado o seguinte código, que transforma os itens da lista em maiúsculos:

texto = **"teste"**maiusculas = []  
**for** item **in** texto:  
 maiusculas.append(str(item).upper())  
*# ['T', 'E', 'S', 'T', 'E']*print(maiusculas)

Podemos reescrevê-lo da seguinte forma:

texto =**"teste"**maiusculas = [str(item).upper() **for** item **in** texto]  
*# ['T', 'E', 'S', 'T', 'E']*print(maiusculas)

List Comprehensions com if

List comprehensions podem utilizar expressões condicionais para criar listas ou modificar listas existentes.

Sua sintaxe básica é:

**[expr for item in lista if cond]**

Exemplo:

pares = [numero **for** numero **in** range(20) **if** numero % 2 == 0]  
*# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]*print(pares)

List Comprehensions com vários if’s

Podemos verificar condições em duas listas diferentes dentro da mesma list comprehension. Utilizando múltiplos if's e list comprehensions, podemos criar o seguinte código para saber os Múltiplos Comuns de 5 e 6:

multiplos = [numero **for** numero **in** range(100) **if** numero % 5 == 0 **if** numero % 6 == 0]  
*# [0, 30, 60, 90]*print(multiplos)

List Comprehensions com if + else

Outra forma de se utilizar expressões condicionais e list comprehension é usar o conjunto if + else.

A sintaxe básica é:

**[resultado\_if if expr else resultado\_else for item in lista]**

Em outras palavras: para cada item da lista, aplique o resultado resultado\_if se a expressão expr for verdadeira, caso contrário, aplique resultado\_else. Por exemplo, queremos criar uma lista que contenha “1” quando determinado número for múltiplo de 5 e “0” caso contrário.

resultado = [**'1' if** numero % 5 == 0 **else '0' for** numero **in** range(10)]  
*# ['1', '0', '0', '0', '0', '1', '0', '0', '0', '0']*print(resultado)

Múltiplas List Comprehensions (aninhadas)

Vamos supor que queiramos transpor uma matriz, o que significa que, queremos transformar as linhas em colunas e vice-versa. Em Python, podemos fazer isso da seguinte forma:

matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]  
transposta = []  
**for** i **in** range(len(matrix[0])):  
 linha\_transposta = []  
 **for** linha **in** matrix:  
 linha\_transposta.append(linha[i])  
 transposta.append(linha\_transposta)  
  
*# [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]*print(transposta)

Podemos reescrever o código acima, de transposição de matrizes, da seguinte forma, utilizando list comprehension:

transposta = [[linha[i] **for** linha **in** matrix] **for** i **in** range(4)]  
*# [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]*print(transposta)

EXPRESSÕES GERADORAS

Uma desvantagem com list comprehensions é que os valores são todos calculados de uma vez, independente se esses valores são necessários naquele momento ou não. Isto pode algumas vezes consumir uma quantidade excessiva de memória. A PEP 289 propôs a expressão geradora para resolver isto, e esta proposta foi aceita e adicionada a linguagem.

Expressões geradoras são como list comprehensions; a única diferença é que os colchetes na list comprehensions são substituídos por parênteses. Nós contrastamos uma list comprehension com uma expressão geradora logo abaixo.

Para gerar uma lista com o quadrado dos números de 0 e 10 usando list comprehensions é feita da seguinte forma:

squares = [i \*\* 2 **for** i **in** range(10)]  
*# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]*print(squares)

Nós poderíamos usar uma expressão geradora, tal como mostrada abaixo, no lugar de uma list comprehension, e poderíamos então acessar os valores do gerador usando um loop for ou o método next, como mostrado abaixo. Cada valor é computado sob demanda, ou seja, só quando requisitado.

*# squares = <generator object <genexpr> at 0x000001E5BA502A48>*squares = (i \*\* 2 **for** i **in** range(10))  
**for** item **in** squares:  
 print(item, end=**' '**) *# 0 1 4 9 16 25 36 49 64 81*

Também é possível criar geradores através de expressões. Sua sintaxe é bastante parecida com a das list comprehensions, exceto pelo uso dos parênteses ao invés dos colchetes. E que apesar de definidas de um jeito diferente, funcionam da mesma maneira que as funções geradoras.

*# pares = <generator object <genexpr> at 0x000001C7B1E74B48>*pares = (i **for** i **in** range(20) **if not** i % 2)  
**for** item **in** pares:  
 print(item, end=**' '**) *# 0 2 4 6 8 10 12 14 16 18*

LAMBDA

Normalmente, usamos a palavra-chave def para definir uma função com um nome. A palavra-chave lambda é usada para criar funções anônimas. Geralmente, essa função é destinada ao uso único.

A função lambda pode ter apenas uma expressão (não instrução). Obviamente, ele não pode substituir uma função cujo corpo possa ter condicionais, loops etc.

As funções Lambda são úteis quando queremos atribuir a função como um dos argumentos a outra função. No Python, a função é um objeto de primeira ordem, o que significa que, assim como número, string, lista, etc., a função também pode ser usada como argumento.

Python possui funções internas que aceitam outras funções como argumentos. As funções map (), filter () e reduce () são importantes ferramentas de programação funcional. Todos eles assumem uma função como argumento.

A função de argumento pode ser uma função normal ou uma função lambda.

Sintaxe:

**lambda arg1, arg2 ...: expressão**

A função lambda pode ter qualquer número de argumentos, mas sempre há uma única expressão após o símbolo: Quando essa função é chamada, a expressão é seu valor de retorno.

>>> quadrado = lambda x: x \* x

Aqui, a expressão lambda x : x \* x é atribuída a um quadrado variável. Essa função lambda recebe x como parâmetro e retorna o quadrado de x (x \* x). Agora podemos chamá-lo como uma chamada de função regular.

square = **lambda** x: x \* x  
print(square(5)) *# 25*

A definição da função lambda acima é semelhante à definição de uma função normal, como mostrado abaixo:

**def** square\_func(x): **return** (x \* x)

Considere a seguinte função lambda com vários parâmetros:

soma = **lambda** x, y, z: x + y + z  
print(soma(5, 10, 15)) *# 30*

A expressão nem sempre precisa retornar um valor. Pode ser nulo.

disp = **lambda** str: print(**'Saída:'** + str.upper())  
disp(**"Olá, mundo!"**) *# Saída:OLÁ, MUNDO!*

MAP()

A função interna map() chama a função especificada para cada item de um iterável (como string, lista, tupla ou dicionário) e retorna uma lista de resultados.

Sintaxe:

**map (função, iterável [, iterável2, iterável3, ... iterávelN]) -> objeto de mapa**

Considere a seguinte função quadrada simples, podemos chamar a função de mapa com a lista de números para obter a lista de resultados, como mostrado abaixo.

**def** square(x): **return** x \* x  
  
  
num = 10  
d = 0  
sqrList = map(square, range(num))  
**while** d < num:  
 *# 0 1 4 9 16 25 36 49 64 81* print(next(sqrList), end=**' '**)  
 d = d + 1

No exemplo acima, a função map() se aplica a cada elemento da lista de números []. Isso retornará um objeto de mapa que é iterável e, portanto, podemos usar a função next() para percorrer a lista.

Programa para dobrar cada item em uma lista usando map()

lst = [1, 5, 4, 6, 8, 11, 3, 12]  
new\_lst = list(map(**lambda** x: x \* 2 , lst))  
  
*# [2, 10, 8, 12, 16, 22, 6, 24]*print(new\_lst)

Outro exemplo do map()

*# Gerar uma lista da string # Imperativo*string = **'Python'**lista = [] *# estado inicial***for** l **in** string:  
 lista.append(l) *# cada iteração gera um novo estado*print(lista) *# ['P', 'y', 't', 'h', 'o', 'n']  
  
# Gerar uma lista da string # Funcional*string = **lambda** x: x  
lista = list(map(str, string(**'Python'**))) *# atribuição a um novo objeto*print(lista) *# ['P', 'y', 't', 'h', 'o', 'n']*

## Map com expressão Lambda

A função map() passa cada elemento da lista para a função interna, uma função lambda ou uma função definida pelo usuário e retorna o objeto mapeado. O map() seguir map() é usado com a função lambda.

num = 10  
d = 0  
sqrList2 = map(**lambda** x: x \* x, range(num))  
**while** d < num:  
 *# 0 1 4 9 16 25 36 49 64 81* print(next(sqrList2), end=**' '**)  
 d = d + 1

## Map com função incorporada

No exemplo a seguir, é fornecida uma função interna pow () para mapear dois objetos de lista, um para cada parâmetro base e de índice. O resultado é uma lista que contém a potência de cada número em bases aumentadas para o número correspondente no índice.

bases = [10, 20, 30, 40, 50]  
index = [1, 2, 3, 4, 5]  
powers = list(map(pow, bases, index))  
*# [10, 400, 27000, 2560000, 312500000]*print(powers)

FILTER()

A função filter() chama a função especificada, que retorna boolen para cada item da iterável (lista) especificada.

Sintaxe:

**filter (função, iterável) -> objeto de filtro**

A função filter() também recebe dois argumentos, uma função e uma sequência (por exemplo, uma lista). Cada item da lista é processado pela função que retorna Verdadeiro ou Falso. Somente os itens que retornam True são armazenados em um objeto de filtro. Isso pode ser convenientemente convertido em uma sequência.

Observe a seguinte função isPrime() que retorna True se o parâmetro for um número primo, caso contrário, ele retornará False. Essa função é usada dentro de filter () junto com um objeto de intervalo que produz números entre 2 e 100. Somente números primos são coletados no objeto de lista resultante.

**def** isPrime(x):  
 **for** n **in** range(2, x):  
 **if** x % n == 0:  
 **return False  
 else**:  
 **return True**fltrObj = filter(isPrime, range(10))  
*# Nº Primos de 1-10: [3, 5, 7, 9]*print(**'Nº Primos de 1-10:'**, list(fltrObj))

Outro exemplo de filter:

*# Programa para filtrar apenas os itens pares de uma lista*lst = [1, 5, 4, 6, 8, 11, 3, 12]  
new\_lst = list(filter(**lambda** x: (x % 2 == 0) , lst))  
  
*# Saída: [4, 6, 8, 12]*print(new\_lst)

A função lambda também pode ser usada como um filtro. O programa a seguir retorna uma lista de números pares de um intervalo.

Filtrar com Expressão Lambda

pares = filter(**lambda** x: x % 2 == 0, range(10))  
*# [0, 2, 4, 6, 8]*print(list(pares))

REDUCE()

A função reduce() é definida no módulo functools. Como as funções de mapa e filtro, a função reduce() recebe dois argumentos, uma função e um iterável. No entanto, ele não retorna outro iterável, mas retorna um único valor.

A função de argumento é aplicada cumulativamente aos argumentos da lista da esquerda para a direita. O resultado da função na primeira chamada se torna o primeiro argumento e o terceiro item da lista se torna o segundo. Isso é repetido até que a lista se esgote.

No exemplo abaixo, a função mult() é definida para retornar o produto de dois números. Esta função é usada na função reduce() junto com um intervalo de números entre 1 e 4, que são 1,2 e 3. A saída é um valor fatorial de 3.

**import** functools  
  
**def** mult(x, y):  
 print(**"x="**, x, **" y="**, y)  
 **return** x \* y  
  
fact = functools.reduce(mult, range(1, 4))  
print(**'Factorial of 3: '**, fact)  
**"""  
x= 1 y= 2  
x= 2 y= 3  
Factorial of 3: 6  
"""**

O reduce() não é mais builtin no python 3, por isso o import abaixo

**from** functools **import** reduce  
  
**def** soma(x1, x2):  
 **return** x1 + x2  
  
*# Saída: 10*print(reduce(soma, [1, 2, 3, 4]))

Outro exemplo do reduce()

*# Calcula o fatorial de n***def** fat(n):  
 **return** reduce(**lambda** x, y: x \* y, range(1, n))  
  
*# Saída: 720*print(fat(7))

Zip

Esta função retorna uma lista de tuplas, onde a i-ésima tupla contém o i-ésimo elemento de cada um dos argumentos.

x = [1, 2, 3]  
y = [4, 5, 6]  
**for** t **in** zip(x, y):  
 print(t)  
**"""  
(1, 4)  
(2, 5)  
(3, 6)  
"""**

A função zip() retorna uma sequência de tuplas. Lista com tamanhos diferentes serão emparelhadas e a diferença entre elas será desconsiderada.

lst\_a = [6, 7, 8, 9]  
lst\_b = [1, 2, 3, 4, 5,10,11]  
  
**for** x **in** zip(lst\_a, lst\_b):  
 print(x)  
**"""  
(6, 1)  
(7, 2)  
(8, 3)  
(9, 4)  
"""**

A função zip() aponta para o objeto instanciado, para vermos a lista precisamos passá-la para a função interna list().

x = [1, 2, 3]  
y = [4, 5, 6]  
print(zip(x, y)) *# <zip object at 0x000001DFC92829C8>*print(list(zip(x, y))) *# [(1, 4), (2, 5), (3, 6)]*

RECURSÃO EM PYTHON

Uma função que se chama é uma função recursiva. Este método é usado quando um determinado problema é definido em termos de si mesmo. Embora isso envolva iteração, o uso de uma abordagem iterativa para resolver esse problema pode ser entediante. A abordagem recursiva fornece uma solução muito concisa para um problema aparentemente complexo.

O exemplo mais popular de recursão é o cálculo do fatorial. Matematicamente, o fatorial é definido como: n! = n \* (n-1)!

Usamos o próprio fatorial para definir o fatorial. Portanto, este é um caso adequado para escrever uma função recursiva. Vamos expandir a definição acima para o cálculo do valor fatorial de 5.

5! = 5 X 4!

5 X4 X 3!

5 X4 X 3 X 2!

5 X4 X 3 X 2 X 1!

5 X4 X 3 X 2 X 1

= 120

Embora possamos realizar esse cálculo usando um loop, sua função recursiva envolve chamá-lo sucessivamente, decrementando o número até atingir 1. A seguir, é uma função recursiva para calcular o fatorial.

**def** factorial(n):  
 **if** n == 1:  
 print(n)  
 **return** 1  
 **else**:  
*# 14 \* 13 \* 12 \* 11 \* 10 \* 9 \* 8 \* 7 \* 6 \* 5 \* 4 \* 3 \* 2 \* 1* print(n, **'\*'**, end=**' '**)  
 **return** n \* factorial(n - 1)  
  
*# 87178291200*print(factorial(14))

Quando a função fatorial é chamada, são feitas chamadas sucessivas para a mesma função, reduzindo o valor. As funções começam a retornar à chamada anterior depois que o argumento atinge 1. O valor de retorno da primeira chamada é cumulativo produto dos valores de retorno de todas as chamadas.

ASSERT

O Python fornece a declaração assert para verificar se uma determinada expressão lógica é verdadeira ou falsa. A execução do programa continua apenas se a expressão for verdadeira e gera o AssertionError quando for falso.

O código a seguir mostra o uso da declaração assert, que será exibida se o número digitado for maior que ou igual a 0. Números negativos resultam no cancelamento do programa após a exibição do erro Assertion.

num = int(input(**'Enter a number: '**))  
**assert** num >= 0  
print(**'You entered: '**, num)  
*# Se digitar 100 >> You entered: 100  
# Se digitar menor que 0*

File "18\_assert.py", line 2, in <module>

assert num >= 0

AssertionError

A instrução assert pode opcionalmente incluir uma sequência de mensagens de erro, que é exibida junto com o AssertionError. No código acima, você pode alterar a declaração assert para o seguinte e executar o código:

num = int(input(**'Enter a number: '**))  
**assert** num >= 0, **"Somente números positivos"**print(**'You entered: '**, num)

*# Se digitar menor que 0*

Traceback (most recent call last):

File "18\_assert.py", line 2, in <module>

assert num >= 0, "Somente números positivos"

AssertionError: Somente números positivos

O AssertionError também é uma exceção interna e pode ser manipulado usando a construção try ... except da seguinte maneira:

**try**:  
 num = int(input(**'Enter a number: '**))  
 **assert** (num >= 0), **"Somente números positivos"** print(**'You entered: '**, num)  
**except** AssertionError **as** msg:  
 print(msg)

Quando a entrada causa um AssertionError, o programa não será finalizado, como foi o caso anteriormente. Em vez disso, ele será tratado pelo bloco de exceção. A mensagem de erro na declaração assert será passada como argumento para o argumento de exceção msg, usando a palavra-chave as .

ESTRUTURAS DE DECISÃO

As estruturas de decisão permitem alterar o curso do fluxo de execução de um programa, de acordo com o valor (Verdadeiro/Falso de um teste lógico. Em Python temos as seguintes estruturas de decisão

**if < condição1> :**

**Bloco de comandos 1**

**elif < condição2> :**

**Bloco de comandos 2**

**Else :**

**Bloco de comandos default**

O comando if é utilizado quando precisamos decidir se um trecho do programa deve ou não ser executado. Ele é associado a uma condição, e o trecho de código será executado se o valor da condição for verdadeiro. Se houver diversas condições, cada uma associada a um trecho de código, utiliza-se o elif. Se nenhuma das condições tiver valor verdadeiro, executa o bloco de comandos default

valor = int(input(**"Qual sua idade?"**))  
  
**if** valor < 6:  
 print(**"Que coisa fofa!"**)  
**elif** valor < 18:  
 print(**"Você ainda não pode dirigir!"**)  
**elif** valor > 60:  
 print(**"Você está na melhor idade!"**)  
**else**:  
 print(**"Você é o cara!"**)

Continue

Uma variação do if else, mais enxuta:

num = int(input(**"Digite um numero inteiro: "**))  
d = 2  
**while** num > 1:  
 **if** num % d == 0:  
 print(d)  
 num = num / d  
 **continue** d = d + 1

Pass

Sem muita utilidade, serve como variável de teste e tem valor equiparado a null:

**for** num **in** range(1,6):   
 **if** num==3:  
 **pass  
 else**:  
 print (**"Num = {} "**.format(num))

ESTRUTURAS DE REPETIÇÃO

A Estrutura de repetição é utilizada para executar uma mesma sequência de comandos v árias vezes. A repetição está associada ou a uma condição, que indica se deve continuar ou não a ou a um a sequência de valores, que determina quantas vezes a sequência deve ser repetida. As estruturas de repetição são conhecidas também como laço s loops

Laço while

No laço while o trecho de código da repetição está associado a uma condição. Enquanto a condição tiver valor verdadeiro , o trecho é executado. Quando a condição passa a ter valor falso, a repetição termina.

**Sintaxe:**

**while condição>**

**<Bloco de comandos**

Exemplo:

senha = **"54321"**leitura = **" "  
while** (leitura != senha):  
 leitura = input(**"Digite a senha: "**)  
 **if** leitura == senha:  
 print(**'Acesso liberado'**)  
 **else**:  
 print(**'Senha incorreta. Tente novamente'**)

Outro exemplo: Encontrar a soma de 5 valores.

contador = 0  
somador = 0  
**while** contador < 5:  
 contador = contador + 1  
 valor = float(input(**'Digite o '** + str(contador) + **'° valor: '**))  
 somador = somador + valor  
  
print(**'Soma = '**, somador)

Laço for

O laço for é a estrutura de repetição mais utilizada em Python. Pode ser utilizado com uma sequência numérica (gerada com o comando range) ou associado a uma lista. O trecho de código da repetição é executado para cada valor da sequência numérica ou da lista.

**Sintaxe:**

**for <variável > in range (início , limite, passo)**

**<Bloco de comandos>**

**ou**

**for <variável> in <lista>**

**<Bloco de comandos>**

Exemplo encontrar a soma S = 1+4+7+10+13+16+19

S = 0  
**for** x **in** range(1, 20, 3):  
 S = S + x  
  
print(**'Soma = '**, S) *# Soma = 70*

Outro exemplo: As notas de um aluno estão armazenadas em uma lista. Calcular a média dessas notas.

Lista\_notas = [3.4, 6.6, 8, 9, 10, 9.5, 8.8, 4.3]  
soma = 0  
**for** nota **in** Lista\_notas:  
 soma = soma + nota  
 media = soma / len(Lista\_notas)  
  
print(**'Media: {:.2f}'**.format(media)) *# Media: 7.45*

FUNÇÕES

Funções são pequenos trechos de código reutilizáveis. Elas permitem dar um nome a um bloco de comandos e executar esse bloco, a partir de qualquer lugar do programa

Funções são definidas usando a palavra-chave **def** conforme sintaxe a seguir

**def <nome\_função> (<definição dos parâmetros>) :**

**<Bloco de comandos da função>**

**Obs.: A definição dos parâmetros é opcional**

**def** hello():  
 print(**"Ola Mundo!!!"**)  
  
*# Para usar a função, basta chama-la pelo nome:*hello()

Parâmetros e argumentos

Parâmetros são as variáveis que podem ser incluídas nos parênteses das funções. Quando a função é chamada são passados valores para essas variáveis. Esses valores são chamados argumentos. O corpo da função pode utilizar essas variáveis, cujos valores podem modificar o comportamento da função.

**def** maior(x, y):  
 **if** x > y:  
 print(**"0 valor "** + str(x) + **" é maior do que "** + str(y))  
 **else**:  
 print(**"0 valor "** + str(y) + **" é maior do que "** + str(x))  
  
maior(5, 1) *# 0 valor 5 é maior do que 1*

Escopo das variáveis

Toda variável utilizada dentro de uma função tem escopo local, isto é, ela não será acessível por outras funções ou pelo programa principal. Se houver variável com o mesmo nome fora da função, será uma outra variável, completamente independentes entre si

**def** soma(x, y):  
 total = x + y  
 print(**"Total soma = "**, total) *# Total soma = 8  
  
# Programa principal*total = 10  
soma(3, 5)  
print(**"Total principal = "**, total) *# Total principal = 10*

Para uma variável ser compartilhada entre diversas funções e o programa principal, ela deve ser definida como variável global. Para isto, utiliza-se a instrução global para declarar a variável em todas as funções para as quais ela deva estar acessível. O mesmo vale para o programa principal.

**def** soma\_global(x, y):  
 **global** total  
 total = x + y  
 print(**"Total soma = "**, total) *# Total soma = 8*

Retorno de valores

O comando **return** é usado para retornar um valor de uma função e encerrá-la. Caso não seja declarado um valor de retorno, a função retorna o valor **None** (que significa nada, sem valor)

**def** soma\_retorno(x, y):  
 **return** x + y  
  
s = soma\_retorno(3, 5)  
print(**"Total soma = "**, s) *# Total soma = 8*

Observações:

a) O valor da variável total, calculado na função soma, retornou da função e foi atribuído à variável s.

b) O comando após o return foi ignorado.

Múltiplos Retornos

É possível desenvolver função com múltiplos valores de saida, essa prática, resume o código, pois, o código fica empacotado.

**def** function():  
 varl = 37  
 var2 = 16  
 **return** varl, var2  
  
*# Perceber que, a function retorna uma tupla.*print(function()) *#(37, 16)*x, y = function()  
print(x) *# 37*print(y) *# 16*z = function()  
print(z) *#(37, 16)*

Valor padrão

É possível definir um valor padrão para os parâmetros da função. Neste caso, quando o valor é omitido na chamada da função, a variável assume o valor padrão.

**def** calcula\_juros(valor, taxa=10):  
 juros = valor \* taxa / 100  
 **return** juros  
  
s = calcula\_juros(300)  
print(**"Taxa juros = "**, s) *# Taxa juros = 30.0*

Funcões úteis

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| dir() | Lista atributos de um objeto |
| help() | Help interativo ou help(objeto), info. sobre objeto |
| type() | Retorna tipo do objeto |
| raw\_input() | Prompt de entrada de dados |
| int(), str(), float() | typecast |
| chr(), ord() | ASCII |
| max(), min() | Maior e menos de uma string, list ou tuple |

BIBLIOTECAS / MÓDULOS

As bibliotecas armazenam funções pré-definidas que podem ser utilizados em qualquer momento do programa. Em Python muitas bibliotecas são instaladas por padrão junto com o programa. Existem mais de 137.000 bibliotecas e 198.826 pacotes para Python, prontos para facilitar a programação dos desenvolvedores. Essas bibliotecas e pacotes destinam-se a uma variedade de soluções modernas. Para usar uma biblioteca, deve se utilizar o comando import:

**import** math  
print(math.factorial(6)) *# 720*

Pode se importar uma função específica da biblioteca:

**from** math **import** factorial **as** fat  
print(fat(8)) *# 40320*

A tabela a seguir, mostra algumas das bibliotecas de Python.

|  |  |
| --- | --- |
| **Bibliotecas** | **Função** |
| Math, cmath, decimal e random | Funções matemáticas |
| os, glob, shutils e subprocess | Sistema, Arquivos, diretórios |
| Tkinter, Turtle | Criar interfaces Gráficas |
| smtplib | e-mail |
| Time e datetime | Funções de tempo |
| threading | Threads |
| pickle e cPickle | Persistência |
| xml.dom, xml.sax, elementTree | XML |
| ConfigParser e optparse | Configuração |

Para obter mais informação sobre o módulo:

help(math)  
**"""  
NAME  
 math  
  
DESCRIPTION  
 This module provides access to the mathematical functions  
 defined by the C standard.  
  
FUNCTIONS  
 acos(x, /)  
 Return the arc cosine (measured in radians) of x.  
"""**

help(math.log) específica sobre uma função

help(math.log)  
**"""  
log(...)  
 log(x, [base=math.e])  
 Return the logarithm of x to the given base.  
   
 If the base not specified, returns the natural logarithm (base e) of x.  
"""**

Para listar todas funções do modulo: dir(nome\_do\_modulo):

import math

dir(math)

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

import cmath

dir(cmath)

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp', 'inf', 'infj', 'isclose', 'isfinite', 'isinf', 'isnan', 'log', 'log10', 'nan', 'nanj', 'phase', 'pi', 'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau']

Módulo statistics

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| mean (lista) | Média dos valores de lista |
| median(lista) | Mediana de lista (valores não precisam eslar ordenados) |
| harmonic\_mean(lista) | Média harmônica |
| mode (lista) | Moda de uma lista de valores discretos |
| median\_low(lista) | Quando a lista possui um número par de valores, retoma o menor dos dois valores que seriam usados para computar a mediana. Exemplo: para (1,2,3,4). retoma 2 |
| median\_high(lista) | Quando a lista possui um número par de valores, retoma o maior dos dois valores que senam usados para computar a mediana. Exemplo: para (1.2.3,4). retoma 4 |
| median\_grouped(lista) | Retoma a mediana de dados continuos agrupados, calculada como o 2° quattil, usando interpolação |
| stdev(lista) | Desvio padrão de amostra |
| pstdev(lista) | Desvio padrão de população |
| variance(lista) | Variância da amostra |
| pstdev(lista) | Variância da população |

Módulo random

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| random () | Retoma um numero real (float) na faixa [0.0, 1.0). Esta notação indica que 0.0 pode ser gerado, mas 1.0 não; |
| randint(Inicio, fim) | Retoma um número inteiro na faixa [início, fim) |
| choice(sequencia) | Sorteia um elemento de uma sequencia |
| seed(semente) | Utilizado para definir uma semente em experimentos reproduziveis. Quando a semente não é especificada. a mesma é atribuída conforme o timestamp do sistema (data e hora, com precisão de nanossegundos) |
| getstate (): | Captura o estado corrente do gerador de numeros aleatórios |
| setstate(e) | Utilizado para retomar o gerador para o estado e, previamente capturado pela tunc5o getstatel) |

Módulo math

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| fabs(x) | O valor absoluto de x |
| pow(x,y) | x\*\*y (xy) |
| sqrt(x) | A raiz quadrada de x |
| floor(x) | O maior inteiro menor ou igual a x |
| ceil(x) | O menor inteiro maior ou igual a x (arredonda x por excesso) |
| radians(x) | O valor em radianos do angulo x dado em graus |
| hypo(x,y) | hipotenusa dos catetos fornecidos |
| degrees(x) | O valor em graus do ângulo x dado em radianos |
| sin(x) | O seno de x (em radianos). |
| cos(x) | O cosseno de x (em radianos) |
| tan(x) | A tangente de x (em radianos) |
| acos(x) | O arco (em radianos) cujo cosseno é x |
| asin(x) | O arco (em radianos) cujo seno é x |
| atan(x) | O arco (em radianos) cuja tangente é x |
| exp(x) | Exponencial d x |
| log(x [, b]) | O logaritmo de x na base b. Se b não for especificado, assume b = e (logaritmo natural) |
| log10(x) | O logaritmo de x na base 10 (equivale a log(x, 10)) |
| e | Constante e = 2.7182818284590451 (Número Nepperiano) |
| pi | Constante π = 3.1415926535897931 |

Módulo turtle

|  |  |
| --- | --- |
| **Função** | **Descrição** |
| forward(x) ou turtle.fd(x) | Avançar x pixels |
| left(x) ou turtle.lt(x) | Virar a esquerda x graus |
| right(x) ou turtle.rt(x) | Virar para direita x graus |
| bgcolor('orange') | Alterar cor de fundo |
| turtle.pensize(x) | Largura da reta |
| pencolor(‘red’) | Cor da reta |
| shape(‘turtle’) | Seta tartaruga |
| speed(10) | Velocidade |

Além das bibliotecas padrão, existem também outras bibliotecas externas de alto nível disponíveis para Python. A tabela a seguir mostra algumas dessas bibliotecas.

|  |  |
| --- | --- |
| **Bibliotecas** | **Função** |
| urllib | Leitor RSS\* para uso na internet |
| numpy | Funções matemáticas mais avançadas |
| matlibplot | Construção de gráficos |
| PIL / Pillow | fork da PIL - Python Image Library Manipulação de imagens |
| feedparser | Analisador de feed universal, manipula feeds RSS 0.9x, RSS 1.0, RSS 2.0, CDF, Atom 0.3 e Atom 1.0 |
| OpenCV | processamento de imagens |
| Requests | rica biblioteca HTTP sob a licença Apache 2.0 |
| TensorFlow, Scikit | biblioteca para Machine Learning de código aberto e gratuita |
| Keras | biblioteca de rede neural profunda de código aberto |
| Scipy | usada para computação científica e técnica |
| Pandas | ciência de dados para manipular facilmente qualquer tipo de dado, como dados estruturados ou de séries temporais |
| PyGame | Criar jogos |

\*RSS é um padrão baseado na linguagem XML que informa aos leitores as novidades dos sites do seu interesse. É uma forma simples de blogs e sites de notícias divulgarem informações atualizadas sem que o leitor tenha que entrar no endereço do site/blog para ter acesso a essas informações. O usuário interessado em obter as atualizações de um determinado site, deve instalar um programa-leitor (também chamado de agregador) específico no computador, ou subscrever o Feed RSS diretamente no site que oferece a opção de Feeds. Feed significa "fonte" ou "alimentador".

Urllib

Um exemplo de uso da biblioteca urllib com leitor de RSS feedparser para o uso na internet. O exemplo abaixo faz o download do módulo feedparser depois faz a leitura do RSS da Últimas Notícias do site Terra.

**import** urllib.request  
**import** feedparser  
  
**with** urllib.request.urlopen(**"http://diveintomark.org/projects/feed\_parser/feedparser.py"**) **as** url:  
 arq = url.read()  
  
 *# AttributeError: 'bytes' object has no attribute 'read'  
 # open('feedparser.py', 'w').write(arq.read())* print(arq)  
  
terra = feedparser.parse(**'http://rss.terra.com.br/0,,EI1,00.xml'**)  
print(terra.values())

Numpy

Numpy é uma biblioteca matemática de código aberto para Python, com suporte a matrizes e vetores grandes, além de um grande número de funções matemáticas. O site oficial é http://numpy.scipy.org/ . É importante lembrar que Python possui uma biblioteca matemática padrão, chamada Math. Essa biblioteca precisa ser instalada antes de ser usada, não vem por padrão.

O exemplo abaixo mostra o uso da biblioteca Numpy na criação de arrays com valores aleatórios, e a manipulação desses valores.

**import** numpy **as** np  
  
a = np.random.randn(25) *# Cria um array com 25 valores aleatórios*len(a) *# Informa o tamanho do array*print(a) *# Mostra os valores do array*a = np.array([-0.97662319, -0.91427827, -0.17596519, -1.07721448, -0.64067605,  
 -1.21453417, -0.508037, 1.0928856, -0.4206412, -0.40750775,  
 -0.83259417, 0.41311712, -0.03811292, -0.41181049, 0.45493473,  
 0.5168261, 0.38207838, 1.27969391, -0.91343678, -0.89708382,  
 -0.94028084, 0.33992957, -1.73894293, 0.5228072, 0.57514934])  
  
print(**"============"**)  
print(a[0:10]) *# Mostra os valores do array, de 0 até 10*a = np.array([-0.97662319, -0.91427827, -0.17596519, -1.07721448, -0.64067605,  
 -1.21453417, -0.508037, 1.0928856, -0.4206412, -0.40750775])  
  
print(**"============"**)  
a = print(a[0:4] \* 2) *# Multiplicação dos valores 0 até 4 por 2*np.array([-1.95324639, -1.82855654, -0.35193039, -2.15442897])

Matplotlib

O Matplotlib é uma biblioteca Python que usa o Python Script para escrever gráficos e plotagens bidimensionais. Frequentemente, aplicações matemáticas ou científicas exigem mais do que eixos únicos em uma representação. Essa biblioteca nos ajuda a criar várias ao mesmo tempo. No entanto, você pode usar o Matplotlib para manipular diferentes características das imagens.

Recursos

* O Matplotlib pode criar números de qualidade realmente bons para publicação. Os números criados com o Matplotlib estão disponíveis em formatos impressos em diferentes plataformas interativas.
* Você pode usar o MatPlotlib com kits de ferramentas diferentes, como scripts em Python, IPython Shells, Jupyter Notebook e muitas outras interfaces gráficas do usuário.
* Várias bibliotecas de terceiros podem ser integradas aos aplicativos Matplotlib. Como seaborn, ggplot e outros kits de ferramentas de projeção e mapeamento, como o basemap.
* Uma comunidade ativa de desenvolvedores se dedica a ajudá-lo com qualquer uma de suas perguntas com o Matplotlib. Sua contribuição para o Matplotlib é altamente louvável.
* O bom é que você pode rastrear bugs, novos patches e solicitações de recursos na página do rastreador de issues do Github. É uma página oficial para apresentar diferentes questões relacionadas ao Matplotlib.

PIL/Pillow

PIL (Python Imaging Library) é uma biblioteca voltada para a manipulação de imagens com suporte para os formatos PNG, TIFF, BMP, EPS e GIF. Posteriormente, PIL, teve sua sequência pela biblioteca Pillow, que tem como objetivo principal dar suporte a Python 3 e facilitar a instalação desta.

Recursos

* Usando o Pillow, você pode não apenas abrir e salvar imagens, mas também influenciar o ambiente das imagens.
* O Pillow suporta muitos tipos de arquivos, como PDF, WebP, PCX, PNG, JPEG, GIF, PSD, WebP, PCX, GIF, IM, EPS, ICO, BMP e muitos outros.
* Com o Pillow, você pode criar facilmente miniaturas de imagens. As miniaturas carregam a maioria dos aspectos valiosos da sua imagem.
* O Pillow suporta uma coleção de filtros de imagem - FIND\_EDGES, DETALHE, LISO, BORRÃO, CONTORNO, SHARPEN, SMOOTH\_MORE e outros.
* O Pillow oferece grande apoio da comunidade.

O exemplo abaixo mostra o uso da biblioteca PIL para rotacionar uma imagem:

**from** PIL **import** Image *#Importar a biblioteca*im = Image.open(**"/home/exemplo.jpg"**) *#Abrir a imagem*im2 = im.rotate(45) *#Rotacionar a imagem em 45 graus*im2.save(**"/home/exemploRotacionado.jpg"**) *#Salvar imagem rotacionada*

CLASSES E OBJETOS

Python suporta a maioria das técnicas da programação orientada a objeto. A programação orientada a objetos e uma técnica de programação que organiza nossos programas em classes e objetos em vez de apenas funções e pode ser descrito como um conjunto de classes (pré-definidas e definidas) que possuem atributos e métodos, e que são instanciadas em objetos, durante a execução do programa.

A programação orientada a objetos facilita a escrita e a manutenção de nossos programas, utilizando classes e objetos. Classes são a definição de um novo tipo de dados que associa dados e operações em uma só estrutura.

Podemos entender um objeto em Python como a representação de um objeto do mundo real, escrita em uma linguagem de programação. Essa representação e limitada pela quantidade de detalhes que podemos ou queremos representar, uma abstração. Um objeto pode ser entendido como uma variável cujo tipo é uma classe, ou seja, um objeto e uma instancia de uma classe.

Ao trabalharmos com classes e objetos, assim como funções, precisamos representar em Python uma abstração do problema. Quando realizamos uma abstração, reduzimos os detalhes do problema ao necessário para soluciona-lo. Estamos construindo um modelo, ou seja, modelando nossas classes e objetos.

Sendo assim as estruturas de dados possuem atributos (os dados em si) e métodos (rotinas associadas aos dados). Tanto os atributos quanto os métodos são acessados usando ponto (.)

* Para mostrar um atributo: print objeto.atributo
* Para executar um método: objeto.metodo(argumentos)
* Mesmo um método sem argumentos precisa de parênteses: objeto.metodo()
* O ponto também é usado para acessar estruturas de módulos que foram importados pelo programa.

Vejamos, por exemplo, um aparelho de televisão. Podemos dizer que uma televisão tem uma marca e um tamanho de tela. Podemos também pensar no que podemos fazer com esse aparelho, por exemplo mudar de canal, liga-lo ou desliga-lo. Vejamos como escrever isso em Python:

**class** Televisao:  
 **def** \_\_init\_\_(self, min, max):  
 self.ligada = **False** self.canal = 2  
 self.cmin = min  
 self.cmax = max  
  
 **def** muda\_canal\_para\_baixo(self):  
 **if** self.canal - 1 >= self.cmin:  
 self.canal -= 1  
  
 **def** muda\_canal\_para\_cima(self):  
 **if** self.canal + 1 <= self.cmax:  
 self.canal += 1  
  
tv = Televisao(1, 15)  
print(tv.ligada) *## False*print(tv.canal) *## 2*tv\_sala = Televisao(1, 2)  
tv\_sala.ligada = **True**tv\_sala.canal = 4  
print(tv.canal) *## 2*print(tv\_sala.canal) *## 4*tv\_sala.muda\_canal\_para\_cima()  
tv\_sala.muda\_canal\_para\_cima()  
print(tv.canal) *## 2*tv.muda\_canal\_para\_baixo()  
print(tv.canal) *## 1*

Encapsulamento

Observe que como isolamos os detalhes do funcionamento da classe do resto do programa ao mudarmos o comportamento da classe Televisao quase nada no programa que a utiliza muda. Esse efeito é chamado de encapsulamento. Dizemos que uma classe deve encapsular ou ocultar detalhes de seu funcionamento o máximo possível. No caso dos métodos para mudar o canal, podemos incluir outros métodos e atributos sem alterar o resto do programa. Simplesmente solicitamos que o método realize seu trabalho, sem se preocupar com os detalhes internos de como ele realizara essa operação.

Linguagens clássicas orientadas a objetos, como C ++ e Java, controlam o acesso aos recursos de classe por palavras-chave public, private e protected. Membros privados de uma classe têm acesso negado do ambiente fora da classe. Eles podem ser manipulados somente de dentro da classe.

Membros públicos (geralmente métodos declarados em uma classe) são acessíveis de fora da classe. O objeto da mesma classe é necessário para invocar um método público. Esse arranjo de variáveis ​​de instância privada e métodos públicos garante o princípio do encapsulamento de dados.

Os membros protegidos de uma classe são acessíveis de dentro da classe e também estão disponíveis para suas subclasses. Nenhum outro ambiente tem acesso permitido a ele. Isso permite que recursos específicos da classe pai sejam herdados pela classe filho.

O Python não possui nenhum mecanismo que restrinja efetivamente o acesso a qualquer variável ou método de instância. O Python prescreve uma convenção de prefixar o nome da variável/método com sublinhado único ou duplo para emular o comportamento dos especificadores de acesso protegido e privado.

Todos os membros em uma classe Python são públicos por padrão. Qualquer membro pode ser acessado de fora do ambiente da classe. Você pode acessar os atributos da classe de funcionários e também modificar seus valores, conforme mostrado abaixo.

**class** Employee:  
 **def** \_\_init\_\_(self, name, sal):  
 self.name=name  
 self.salary=sal  
  
e1=Employee(**"Kiran"**,10000)  
print(e1.salary) *# 10000*e1.salary=20000  
print(e1.salary) *# 20000*

A convenção do Python de proteger uma variável de instância é adicionar um prefixo \_ (sublinhado único) a ele. Isso isso não impede que variáveis da instância acessem ou modifiquem a instância, ficando a cargo do programador a responsabilidade de NÃO acessar e modificar variáveis prefixadas com \_ de fora de sua classe.

**class** Protected\_Employee:  
 **def** \_\_init\_\_(self, name, sal):  
 self.\_name=name *# protected attribute* self.\_salary=sal *# protected attribute*e1=Protected\_Employee(**"Swati"**,10000)  
print(e1.\_salary) *# 10000*e1.\_salary=20000  
print(e1.\_salary) *# 20000*

Da mesma forma, um sublinhado duplo \_\_ prefixado a uma variável a torna privada. Dá uma forte sugestão de não tocá-lo de fora da classe. Qualquer tentativa de fazer isso resultará em um AttributeError:

**class** Private\_Employee:  
 **def** \_\_init\_\_(self, name, sal):  
 self.\_\_name=name *# private attribute* self.\_\_salary=sal *# private attribute*

e1=Private\_Employee(**"Bill"**,10000)  
print(e1.\_\_salary)  
**"""  
Traceback (most recent call last):  
 File "08.0.0\_Encapsulamento.py", line 29, in <module>  
 print(e1.\_\_salary)  
AttributeError: 'Private\_Employee' object has no attribute '\_\_salary'  
"""**

Python realiza a manipulação de nomes de variáveis privadas. Cada membro com sublinhado duplo será alterado para \_object.\_class\_\_variable. Se necessário, ele ainda pode ser acessado de fora da classe, mas a prática deve ser evitada.

e1=Private\_Employee(**"Bill"**,10000)  
print(e1.\_Private\_Employee\_\_salary) *# 10000*e1.\_Private\_Employee\_\_salary=20000  
print(e1.\_Private\_Employee\_\_salary) *# 20000*

Polimorfismo

As chamadas de função em Python são universais ou genéricas sem determinação de tipo. Por isso, sobrecarga não é suportada na linguagem. Graças à tipagem dinâmica, o polimorfismo de tipos acontece à todo momento!

class nome :

Utilizamos a instrução class para indicar a declaração de uma nova classe e : para iniciar seu bloco criando uma nova classe chamada Televisao. Quando declaramos uma classe, estamos criando um novo tipo de dados. Esse novo tipo define seus próprios métodos e atributos. Lembre-se dos tipos string e list. Esses dois tipos predefinidos do Python são classes. Quando criamos uma lista ou uma string, estamos instanciando ou criando uma instancia dessas classes, ou seja, um objeto. Quando definimos nossas próprias classes, podemos criar nossos próprios métodos e atributos.

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

Depois, definimos um método especial chamado \_\_init\_\_. Métodos nada mais são que funções associadas a uma classe. O método\_\_init\_\_será chamado sempre que criarmos objetos da classe Televisao, sendo por isso chamado de construtor (constructor) ou inicializador. Um método construtor é chamado sempre que um objeto da classe e instanciado. E o construtor que inicializa nosso novo objeto com seus valores-padrão. O método\_\_init\_\_recebe um parâmetro chamado self. Por enquanto, entenda self como o objeto televisão em si, o que ficara mais claro adiante.

self.atributo

Sempre que quisermos especificar atributos de objetos, devemos associa-los a self. Dizemos que self.ligada = False é um atributo do objeto, ou seja, está sendo atribuído o valor inicial False ao objeto televisão. Caso contrário, se escrevêssemos ligada = False, ligada seria apenas uma variável local do método\_\_init\_\_, e não um atributo do objeto.

objeto = classe()

Criamos um objeto tv utilizando a classe Televisao. Dizemos que tv e agora um objeto da classe Televisao ou que tv e uma instancia de Televisao. Quando solicitamos a criação de um objeto, o método construtor (\_\_init\_\_) de sua classe é chamado. Em seguida é criada outra instancia da classe Televisao chamada tv\_sala, em que é mudado o valor de ligada para True e o canal para 4. Observe que ao imprimirmos o canal de cada televisão temos valores independentes, pois tv e tv\_sala são dois objetos independentes, podendo cada um ter seus próprios valores, como duas televisões no mundo real.

def método(self)

Definimos os métodos da televisão utilizando def dentro da classe Televisao. Tudo que aprendemos com funções e também valido para métodos. A principal diferença e que um método e associado a uma classe e atua sobre um objeto.

Observe que não utilizamos \_ antes do nome do método, pois esse nome não e um nome especial do Python, mas apenas um nome escolhido por nós. Veja que passamos também um parâmetro self, que representa o objeto em si. O primeiro parâmetro do método e chamado self e representa a instância sobre a qual o método atuara. E por meio de self que teremos acesso aos outros métodos de uma classe, preservando todos os atributos de nossos objetos.

Observe que escrevemos diretamente self.canal -= 1, utilizando o atributo canal da televisão. Isso e possível porque criamos o atributo canal no construtor (\_\_init\_\_). É usando atributos que podemos armazenar valores entre as chamadas dos métodos.

objeto.métodos

Observe que escrevemos o nome do método após o nome do objeto, separando-os com um ponto, bem como que o método foi chamado da mesma forma que uma função, mas que na chamada não passamos nenhum parâmetro. Na realidade, o interpretador Python adiciona o objeto tv a chamada, utilizando-o como o self do método. E assim que o interpretador consegue trabalhar com vários objetos de uma mesma classe.

A grande vantagem de usar classes e objetos e facilitar a construção dos programas. Embora simples, você pode observar que não precisamos enviar o canal atual da televisão ao método muda\_canal\_para\_cima, simplificando a chamada do método. Esse efeito "memoria" facilita a configuração de objetos complexos, pois armazenamos as características importantes em seus atributos, evitando repassar esses valores a cada chamada. Na verdade, esse tipo de construção imita o comportamento do objeto no mundo real. Quando mudamos o canal da tv para cima ou para baixo, não informamos o canal atual para televisão!

Coletor de lixo

Um objeto continua existindo na memória enquanto existir pelo menos uma referência a ele. O interpretador Python possui um recurso chamado coletor de lixo (Garbage Collector) que limpa da memória objetos sem referências.

Python utiliza um mecanismo bem simples como coletor de lixo: contagem de referências. Isso significa que para cada objeto é mantido um registro do número de referências existentes para ele. Quando esse número chegar a zero, ou seja, quando o objeto não estiver mais sendo usado, ele é destruído. Quando o objeto é apagado, o método especial \_\_done\_\_() é evocado.

Exemplo de um banco

A classe Cliente e simples, tendo apenas dois atributos: nome e telefone. Digite o programa a seguir e salve-o em um arquivo chamado cliente.py.

Classe Clientes (cllentes.py):

**class** Cliente:  
 **def** \_\_init\_\_(self, nome, telefone):  
 self. nome = nome  
 self.telefone = telefone

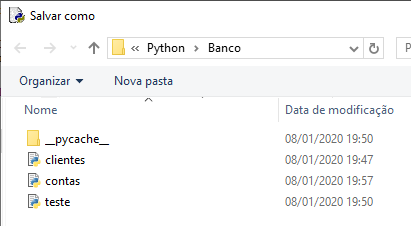
Como o programa do Banco Tatu vai ficar grande, vamos gravar cada classe em um arquivo .py separado. Em Python essa organização não e obrigatória, pois a linguagem permite que tenhamos todas as nossas classes em um só arquivo, se assim quisermos.

Vamos agora criar o arquivo teste.py, que vai simplesmente importar cUentes.py e criar dois objetos:

**from** clientes **import** Cliente  
  
joao = Cliente(**"Joao da Silva"**, **"777-1234"**)  
naria = Cliente(**"Maria da Silva"**, **"555-4321"**)  
  
print(joao.nome) *## Joao da Silva*

Observe que com poucas linhas de código conseguimos reutilizar a classe Cliente. Isso e possível porque clientes.py e teste, py estão no mesmo diretório. Essa separação permite que passemos a experimentar as novas classes no interpretador, armazenando o import e a criação dos objetos em um arquivo à parte. Assim poderemos definir nossas classes separadamente dos experimentos ou testes.

Para resolver o problema do Banco Tatu, precisamos de outra classe, Conta, para representar uma conta do banco com seus clientes e seu saldo; salve-o com o nome contas.py:

**class** Conta:  
 **def** \_\_init\_\_(self, clientes, numero, saldo=0):  
 self.saldo = 0  
 self.clientes = clientes  
 self.numero = numero  
 self.operacoes = []   
 self.deposito(saldo)   
 **def** resumo(self):  
 print(**f"CC Numero: {**self.numero**} Saldo: {**self.saldo**}"**)   
 **def** saque(self, valor):  
 **if** self.saldo >= valor:   
 self.saldo -= valor  
 self.operacoes.append([**"SAQUE"**, valor])   
 **def** deposito(self, valor):  
 self.saldo += valor  
 self.operacoes.append([**"DEPOSITO"**, valor])  
 **def** extrato(self):  
 print(**f"Extrato CC N° {**self.numero**}\n"**)  
 print(**f"Saldo Anterior: {**self.saldo**:10.2f}\n"**)  
 **for** o **in** self.operacoes:  
 print(**f"{**o[0]**:10s} {**o[1]**:10.2f}"**)   
 print(**f"\n Saldo: {**self.saldo**:10.2f}\n"**)

A classe Conta e definida recebendo clientes, número e saldo em seu construtor (\_\_init\_\_), onde em clientes esperamos uma lista de objetos da classe Cliente, número e uma string com o número da conta e saldo é um parâmetro opcional, tendo zero (0) como padrão. A listagem também apresenta os métodos resumo, saque e deposito. O método resumo exibe na tela o número da conta corrente e seu saldo. Saque permite retirar dinheiro da conta corrente, verificando se essa operação e possível (self .saldo >= valor), deposito simplesmente adiciona o valor solicitado ao saldo da conta corrente.

Modifique tambem o programa de testes para imprimir o extrato de cada conta:

cc\_joao = Conta(joao, 1, 100)  
cc\_maria = Conta(maria, 2)  
  
cc\_joao.resumo()  
cc\_joao.deposito(50)  
cc\_joao.resumo()  
cc\_joao.saque(100)  
cc\_joao.resumo()  
  
cc\_joao.extrato()  
  
cc\_maria.deposito(50)  
cc\_maria.extrato()

Precisamos de uma classe Banco para armazenar todas as nossas contas. Como atributos do banco, teríamos seu nome e a lista de contas. Como operações, considere a abertura de uma conta corrente e a listagem de todas as contas do banco. A classe Banco:

**class** Banco:  
 **def** \_\_init\_\_(self, nome):  
 self.nome = nome  
 self.clientes = []  
 self.contas = []   
 **def** abre\_conta(self, conta):  
 self.contas.append(conta)   
 **def** lista\_contas(self):  
 **for** c **in** self.contas:   
 c.resumo()

Agora, vamos criar os objetos para testar a classe Banco:

**from** clientes **import** Cliente  
**from** contas **import** Conta  
**from** bancos **import** Banco  
  
joao = Cliente(**"Joao da Silva"**, **"777-1234"**)  
cc\_joao = Conta(joao, 1, 100)  
tatu = Banco(**"Banco TATU"**)  
tatu.abre\_conta(cc\_joao)  
tatu.lista\_contas()

Herança

A orientação a objetos permite modificar nossas classes, adicionando ou modificando atributos e métodos, tendo como base outra classe. Para atrair novos clientes, o Banco Tatu começou a oferecer contas especiais aos clientes. Uma conta especial permite que possamos sacar mais dinheiro que atualmente disponível no saldo da conta, até um determinado limite. Vamos criar a classe ContaEspecial herdando o comportamento da classe Conta. Digite o programa a seguir no mesmo arquivo em que a classe Conta foi definida (contas.py):

**class** ContaEspecial(Conta):  
 def\_\_init\_\_(self, clientes, numero, saldo=0, limite=0):  
 Conta.\_\_init\_\_(self, clientes, nunero, saldo)  
 self.limite = limite  
 **def** saque(self, valor):  
 **if** self.saldo + self.limite >= valor:  
 self.saldo -= valor  
 self.operacoes.append([**"SAQUE"**, valor])

class subclasse(superclasse):

Definimos a classe ContaEspecial, mas observe que escrevemos Conta entre parênteses. Esse é o formato de declaração de classe usando herança, ou seja, e assim que declaramos a herança de uma classe em Python. A partir daqui, ContaEspecial é uma subclasse de Conta, herdando todos os métodos e atributos da classe Conta. Também dizemos que Conta e a superclasse de ContaEspecial.

subclasse.\_\_init\_\_

Toda vez que você utilizar herança, o método construtor da superclasse, no caso Conta, deve ser chamado. Observe que, nesse caso, passamos self para Conta.\_\_init\_\_. E assim que reutilizamos as definições já realizadas na superclasse, evitando ter de reescrever as atribuições de clientes, número e saldo. Chamar a inicialização da superclasse também tem outras vantagens, como garantir que modificações no construtor da superclasse não tenham de ser duplicadas em todas as subclasses.

self.limite

Criamos o atributo self.limite. Esse atributo será criado apenas para classes do tipo ContaEspecial. Observe que criamos o novo atributo depois de chamarmos Conta.\_\_init\_\_.

Observe também que não chamamos Conta.saque no método saque de ContaEspecial. Quando isso ocorre, estamos substituindo completamente a implementação do método por uma nova. Uma das grandes vantagens de utilizar herança de classes e justamente poder substituir ou complementar métodos já definidos.

Adicionamos o nome da classe ContaEspecial ao import. Veja que, praticamente, não mudamos nada, exceto o nome da classe, que agora e ContaEspecial, e nao Conta. Um detalhe importante para o teste e que adicionamos um parâmetro ao construtor, no caso 1000, como o valor de limite. Execute esse programa de teste e observe que, para a conta2, obtivemos um saldo negativo.

**from** clientes **import** Cliente  
**from** contas **import** Conta, ContaEspecial  
joao = Cliente(**"Joao da Silva"**, **"777-1234"**)  
maria = Cliente(**"Maria da Silva"**, **"555-4321"**)  
conta2 = ContaEspecial([maria, joao], 2, 500, 1000)   
conta2.deposito(300)  
conta2.saque(1500)  
conta2.extrato()

Utilizando herança, modificamos muito pouco nosso programa, mantendo a funcionalidade anterior e adicionando novos recursos. O interessante de tudo isso e que foi possível reutilizar os métodos que já havíamos definido na classe Conta. Isso permitiu que a definição da classe ContaEspecial fosse bem menor, pois lá especificamos apenas o comportamento que e diferente.

Quando você utilizar herança, tente criar classes nas quais o comportamento e as características comuns fiquem na superclasse. Dessa forma, você poderá definir subclasses enxutas. Outra vantagem de utilizar herança e que, se mudarmos algo na superclasse, essas mudanças serão também usadas pelas subclasses. Um exemplo seria modificarmos o método de extrato. Como em ContaEspecial não especificamos um novo método de extrato, ao modificarmos o método Conta.extrato, estaremos também modificando o extrato de ContaEspecial, pois as duas classes compartilham o mesmo método.

É importante notar que, ao utilizarmos herança, as subclasses devem poder substituir suas superclasses, sem perda de funcionalidade e sem gerar erros nos programas. O importante e que você conheça esse novo recurso e comece a utilizá-lo em seus programas. Lembre-se de que você não e obrigado a definir uma hierarquia de classes em todos os seus programas. Com o tempo, a necessidade de utilizar herança ficara mais clara.

Desenvolvendo uma classe para controlar listas

Nos exemplos anteriores, ao usarmos a lista de clientes como uma lista simples, não fizemos nenhuma verificação quanto a duplicidade de valores. Por exemplo, uma lista de clientes em que os dois clientes são a mesma pessoa seria aceita sem problemas. Outro problema de nossa lista simples e que ela não verifica se os elementos são objetos da classe Cllente. Vamos modificar isso construindo uma nova classe, chamada ListaUnica:

**class** ListaUnica:  
 **def** \_\_init\_\_(self, elem\_class):  
 self.lista = []  
 self.elem\_class = elen\_class   
 **def** \_\_len\_\_(self):  
 **return** len(self.lista)   
 **def** \_\_iter\_\_(self):  
 **return** iter(self.lista)   
 **def** \_\_getitem\_\_(self, p):  
 **return** self.lista(p)   
 **def** indiceValido(self, i):  
 **return** i >= 0 **and** i < len(self.lista)   
 **def** adiciona(self, elem):  
 **if** self.pesquisa(elem) == -1:   
 self. lista. append(elem)   
 **def** remove(self, elem):  
 self.lista.remove(elem)  
 **def** pesquisa(self, elem):  
 self.verifica\_tipo(elem)  
 **try**:  
 **return** self.lista.index(elem)  
 **except** ValueError:   
 **return** -1   
  
 **def** verifica\_tipo(self, elem):  
 **if not** lsinstance(type(elem), self.elem\_class):   
 **raise** TypeError(**"Tipo invalido"**)   
 **def** ordena(self, chave=**None**):  
 self. lista.sort(key=chave)

Classes Abstratas Base ( \_\_nome\_\_ )

Python possui vários métodos mágicos, métodos especiais que tem o formato \_\_nome\_\_. O método \_\_init\_\_, usado em nossos construtores, é um método magico, \_\_len\_\_, \_\_getitem\_\_ e \_\_iter\_\_ também. Esses métodos permitem dar outro comportamento a nossas classes e usa-las quase que como classes da própria linguagem. A utilização desses métodos mágicos não é obrigatória, mas possibilita uma grande flexibilidade para nossas classes.

O método \_\_len\_\_, que é responsável por retornar o número de elementos a partir de self .lista. A implementação do método \_\_Iter\_\_ chama a função iter do Python com nossa lista interna self.lista. A intenção de utilizarmos esses métodos e de esconder alguns detalhes de nossa classe ListaUnica e evitar o acesso direto a self.lista. O método \_\_getitem\_\_, recebe o valor de um índice (p) e retorna o elemento correspondente de nossa lista.

|  |  |  |
| --- | --- | --- |
| **Método** | **Descrição** | **Exemplo:** |
| \_\_str\_\_ | É invocado quando o objeto é invocado como str | def \_\_str\_\_(self):  return 'is my class' |
| \_\_call\_\_ | É invocado quando o objeto é invocado como função. | def \_\_call\_\_(self):  return 'Hello World!' |
| \_\_init\_\_ | É utilizado para inicializar a classe. | def \_\_init\_\_(self, name):  self.name = name |
| \_\_float\_\_ | É invocado quando o objeto é invocado como float | def \_\_float\_\_(self):  return 1.11111 |
| \_\_lt\_\_ | Menor que (less than) |  |
| \_\_le\_\_ | Menor ou igual (less or equal) |  |
| \_\_eq\_\_ | Igual (equals) | def \_\_eq\_\_(self, a):  print("Chamou \_\_eq\_\_")  print(a.numero) |
| \_\_ne\_\_ | Não igual (not equal) |  |
| \_\_ge\_\_ | Maior ou igual (greather or equal) |  |
| \_\_gt\_\_ | - Maior que (greather than) |  |
| \_\_sub\_\_ | Chamado usando '-' | def \_\_sub\_\_(self,v):  return vetor(self.x-v.x, self.y-v.y) |
| \_\_add\_\_ | Chamado usando '+' | def \_\_add\_\_(self,v):  return vetor(self.x+v.x, self.y+v.y) |
| \_\_repr\_\_ | Chamado quando objeto é impresso | def \_\_repr\_\_(self):  return “vetor(%s,%s)”%(str(self.x),str(self.y)) |
| \_\_cmp\_\_ | Chamado ao fazer comparações | def \_\_cmp\_\_(self, other):      if self.valor\_dec > other.valor\_dec:          return 1      elif self.valor\_dec < other.valor\_dec:          return -1      else:          return 0 |
| \_\_len\_\_ | retorna o comprimento do objeto (se for um container) |  |
| \_\_getitem\_\_(self, key) | para containers, retorna o elemento correspondente à chave key; |  |

Em O criamos um objeto com a classe ListaUnica que acabamos de importar. Veja que passamos int como parâmetro no construtor, indicando que essa lista deve apenas conter valores do tipo int (inteiros). Assim, ao tentarmos adicionar um parâmetro de outro tipo, obteremos uma exceção com mensagem de erro tipo inválido (TypeError). Sempre que um elemento e adicionado (método adiciona), uma verificação do tipo do novo elemento é feita pelo método verifica\_tipo, que gera uma exceção, caso o tipo não seja igual ao tipo passado ao construtor. Veja que o método adiciona também realiza uma pesquisa para verificar se um elemento igual já não faz parte da lista e só realiza a adição caso um elemento igual não tenha sido encontrado. Dessa forma, podemos garantir que nossa lista conterá apenas elementos do mesmo tipo e sem repetições.

DECORADOR

O decorador @property nos permite definir propriedades facilmente sem chamar a função property() manualmente. Antes de aprender sobre o decorador @property, vamos entender o que é um decorador.

No Python, a função é um objeto de primeira ordem. Isso significa que pode ser passado como argumento para outra função. Também é possível definir uma função dentro de outra função. Essa função é chamada de função aninhada. Além disso, uma função pode retornar outra função.

Um decorador é uma função que recebe outra função como argumento. O comportamento da função de argumento é estendido pelo decorador sem realmente modificá-lo.

A função típica de decorador será semelhante a abaixo.

**def mydecoratorfunction(some\_function):  
 def wrapper\_function():  
 some\_function() # call some\_function  
 return wrapper\_function**

No exemplo acima, a função some\_function é uma função cujo comportamento queremos estender. Portanto, teremos que escrever uma função personalizada como mydecoratorfunction(), que aceita a função some\_function() como argumento. A função wrapper\_function() é uma função interna na qual podemos escrever código adicional para estender o comportamento da função some\_function, antes ou depois de chamá-lo. E, finalmente, o wrapper\_funtion() deve ser retornado. Dessa maneira, o Python inclui funções de decorador. Além disso, podemos definir nossa própria função decoradora para estender o comportamento de uma função sem modificá-la.

Agora, vamos dar um exemplo simples para demonstrar como definir uma função decoradora personalizada e seu uso. Considere a seguinte função simples.

**def** display(str): print(str)  
  
*# função a ser decorada passada como argumento***def** displaydecorator(funcao):  
 *# wrap the some\_function e estende seu comportamento* **def** display\_wrapper(str):  
 *# write code para estender o comportamento de some\_function()* print(**'Saída:'**, end=**" "**)  
 funcao(str)  
 *# retorno da wrapper\_function* **return** display\_wrapper  
  
out = displaydecorator(display)  
out(**'Olá Mundo'**) *# Saída: Olá Mundo*

Como você pode ver, 'Saída' é anexado ao resultado da função display(). A função displaydecorator() é usada para modificar o comportamento da função display() sem modificá-lo.

Python inclui a sintaxe @[decorator\_function\_name] para especificar uma função decoradora. Podemos especificar @displaydecorator para a função display() para indicar que a função display() é decorada com a função displaydecorator(), após aplicar o decorador @displaydecorator no exemplo acima, podemos chamar diretamente a função display() para obter o comportamento estendido, conforme mostrado abaixo.

@displaydecorator  
**def** display(str): print(str)  
  
display(**'Olá Mundo'**) *# Saída: Olá Mundo*

## @property Decorator

O @property decorator é um decorador interno em Python para a [função property()](https://translate.googleusercontent.com/translate_c?depth=1&hl=pt-BR&rurl=translate.google.com.br&sl=en&sp=nmt4&tl=pt&u=https://www.tutorialsteacher.com/python/property-function&xid=17259,15700023,15700186,15700191,15700259,15700271,15700302&usg=ALkJrhidIeMkj5eRbXnuy5zhilcxd1Apiw) . O código a seguir usa o decorador interno @property para definir a propriedade name na classe person.

**class** person:  
 **def** \_\_init\_\_(self):  
 self.\_\_name = **''** *# Sogrecarga de método name separando   
 # o get(name(self)) do set(name(self, value))* @property  
 **def** name(self):  
 **return** self.\_\_name  
  
 @name.setter  
 **def** name(self, value):  
 self.\_\_name = value  
  
p = person()  
p.name = **'Steve'**print(p.name)

A classe de person acima inclui dois métodos com o mesmo nome name() , mas com um número diferente de parâmetros. Isso é chamado de sobrecarga de método. A função name(self) é marcada com o decorador @property, que indica que o método name(self) é um método getter e o nome da propriedade é apenas o nome do método, neste name caso. Agora, name(self, value) está atribuindo um valor ao atributo privado \_\_name . Portanto, para marcar esse método como um método setter para a propriedade name, o decorador @name.setter é aplicado. É assim que podemos definir uma propriedade e seus métodos getter e setter.

Use @[nome da propriedade].setter para chamar um método setter e @[nome da propriedade].deleter para chamar o método deleter.

@name.deleter  
**def** name(self, value):  
 print(**'Deleting..'**)  
 **del** self.\_\_name

*# Se não implementar o decorador @name.deleter  
# ocorre o erro AttributeError: can't delete attribute***del** p.name

O deleter seria chamado quando você excluir a propriedade usando a palavra-chave del, como mostrado abaixo.

**del** p.name  
  
*# TypeError: name() missing 1 required positional argument: 'value'*print(p.name)

@classmethod decorator

O decorador @classmethod pode ser aplicado a qualquer método de uma classe. Este decorador nos permitirá chamar esse método usando o nome da classe em vez do objeto. @classmethod é aplicado no método showcount() . O método showcount() possui um parâmetro cls , que se refere à classe person. Agora, podemos chamar o método showcount() usando o nome da classe, como mostrado abaixo.

**class** person:  
 totalObjects = 0  
  
 **def** \_\_init\_\_(self):  
 person.totalObjects = person.totalObjects + 1  
  
 @classmethod  
 **def** showcount(cls):  
 print(**"Total objects: "**, cls.totalObjects)  
  
p1 = person()  
p2 = person()  
person.showcount() *# Total objects: 2*

No entanto, o mesmo método também pode ser chamado usando um objeto.

p1.showcount () *# Total objects: 2*

@staticmethod decorator

O método @staticmethod é um decorador interno em Python que define um método estático. Um método estático não recebe nenhum argumento de referência, seja ele chamado por uma instância de uma classe ou pela própria classe. A seguinte notação é usada para declarar um método estático em uma classe. @staticmethod é aplicado ao método greet() . Portanto, o método greet() pode ser chamado usando o nome da classe person.greet() , assim como o objeto.

**class** person:  
 @staticmethod  
 **def** greet(): print(**"Hello!"**)  
  
person.greet() *# Hello!*p1 = person()  
p1.greet() *# Hello!*

Mesmo que person.greet() e p.greet() sejam chamadas válidas, o método estático recebe referência de nenhuma delas. Portanto, ele não tem argumentos - neither self nor cls.

Decoratores internos

Vejamos outro exemplo de decorador apartir de uma classe com métodos mágicos (especiais). A classe Nome é utilizada para guardar nomes de pessoas e manter uma chave de pesquisa. Grave o programa a seguir como nome.py:

**class** Nome:  
 **def** \_\_init\_\_(self, nome):  
 **if** nome **is None or not** nome.strip():  
 **raise** ValueError(**"Nome não pode ser nulo nem em branco"**)  
 self.nome = nome  
 self.chave = nome.strip().lower()  
  
 **def** \_\_str\_\_(self):  
 **return** self.nome  
  
 **def** \_\_repr\_\_(self):  
 **return f"<Classe {**type(self).\_\_name\_\_**} em 0x{**id(self)**:x} Nome: {**self.nome**} Chave: {**self.chave**}>"  
  
 def** \_\_eq\_\_(self, outro):  
 print(**"\_\_eq\_\_ Chamado"**)  
 **return** self.nome == outro.nome  
  
 **def** \_\_lt\_\_(self, outro):  
 print(**"\_\_lt\_\_ Chamado"**)  
 **return** self.nome < outro.nome

Na classe Nome, definimos o método \_\_init\_\_ de torma a gerar uma exceção caso a string passada como nome seja nula (None) ou em branco. Veja que implementtamos também o metodo mágico \_\_str\_\_, que é chamado ao imprimirmos um objeto da classe Nome. Dessa forma, podemos configurar a saída de nossas classes com mais liberdade e sem mudar o comportamento esperado de uma classe normal em Python.

Vejamos o resultado desse programa no interpretador:

**from** nome **import** Nome  
A = Nome(**"Nilo"**)  
print(A) *# Nilo*

**"""  
B = Nome(" ")  
Traceback (most recent call last):  
 File "teste\_nome.py", line 5, in <module>  
 B = Nome(" ")  
 File "nome.py", line 4, in \_\_init\_\_  
 raise ValueError("Nome não pode ser nulo nem em branco")  
ValueError: Nome não pode ser nulo nem em branco  
"""  
"""  
C = Nome(None)  
Traceback (most recent call last):  
 File "teste\_nome.py", line 13, in <module>  
 C = Nome(None)  
 File "nome.py", line 4, in \_\_init\_\_  
 raise ValueError("Nome não pode ser nulo nem em branco")  
ValueError: Nome não pode ser nulo nem em branco  
"""**

Em A, criamos um objeto da classe Nome, veja que o resultado foi o nome Nilo, que é o valor retornado pelo nosso método \_\_str\_\_. Em B e C testamos a inicialização de um objeto com valores inválidos. Veja que uma exceção do tipo ValueError foi gerada, impedindo a criação do objeto com o valor inválido. O objetivo desse tipo de validação é garantir que o valor usado na classe seja válido. Podemos implementar regras mais complexas dependendo do objetivo de nosso programa.

Vamos continuar a descobrir os outros metodos especiais no interpretador:

print(repr(A)) *# <Classe Nome em 0x12c6474d0c8 Nome: Nilo Chave: nilo>*print(A == Nome(**"Nilo"**)) *# \_\_eq\_\_ Chamado True*print(A != Nome(**"Nilo"**)) *# \_\_eq\_\_ Chamado False*print(A < Nome(**"Nilo"**)) *# \_\_lt\_\_ Chamado False*print(A > Nome(**"Nilo"**)) *# \_\_lt\_\_ Chamado False*

Quando usamos a função repr temos a representação do objeto A, ou seja, a forma que é usada pelo interpretador para mostrar o objeto fora da função print. Veja que a saída foi gerada pelo metodo\_\_repr\_\_da classe Nome.

Utilizamos o operador == para verificar se o objeto A é igual a outro objeto. Em Python, o comportamento padrão é que == retorne True se os dois obje­tos forem o mesmo objeto, ou False caso contrário. No entanto, o objeto A e Nome(“Nilo") são claramente duas instâncias diferentes da classe Nome. Veja que o resultado é True e que nosso método\_\_eq\_\_ foi chamado. Isso acontece porque\_\_eq\_\_ (equal) é o método especial utilizado para comparações de igualdade (==) de nossos objetos. Em nossa implementação, retornamos True se o conteúdo de self.nome for igual nos dois objetos, independentemente de serem o mesmo objeto ou não. Veja também que, embora não tenhamos definido o metodo\_\_neq\_\_(!=, not equal, não igual), o método\_\_eq\_\_também foi chamado.

Esse comportamento vem da implementação padrão desses métodos, em que\_\_eq\_\_ foi negado para implementar o \_\_neq\_\_ inexistente. Veremos depois uma forma de declarar todos os operadores de comparação apenas implementando o \_\_eq\_\_ e \_\_It\_\_ (less than, menor que).

Usamos o operador < (menor que) para comparar os dois objetos. Veja que o método especial \_\_It\_\_ foi chamado nesse caso. Nossa implementação do método \_\_It\_\_ utiliza a comparação de self.nome para decidir a ordem dos obje­tos. Em seguida, o operador > (maior que) retorna o valor correto, e tambem chama o método\_\_It\_\_, uma vez que o método\_\_gt\_\_ {greater than, maior que) não foi implementado, de forma análoga ao que aconteceu entre \_\_eq\_\_ e \_\_neq\_\_.

Agora veja o que acontece quando tentamos utilizar os operadores >= (maior ou igual) e <= (menor ou igual):

**"""  
print(A >= Nome("Nilo"))  
Traceback (most recent call last):  
 File "teste\_nome.py", line 29, in <module>  
 print(A >= Nome("Nilo"))  
TypeError: '>=' not supported between instances of 'Nome' and 'Nome'  
"""  
"""  
print(A <= Nome("Nilo"))  
Traceback (most recent call last):  
 File "teste\_nome.py", line 37, in <module>  
 print(A <= Nome("Nilo"))  
TypeError: '<=' not supported between instances of 'Nome' and 'Nome'  
"""**

Isso acontece porque os operadores >= e <= chamam os métodos \_\_ge\_\_ {greater than or equal, maior ou igual) e \_\_le\_\_ (less than or equal, menor ou igual), respectivamente, e ambos não foram implementados.

Como algumas dessas relações não funcionam como o esperado, existe uma forma mais simples de implementar esses métodos especiais, apenas com a implementação de\_\_eq\_\_e de\_\_It\_\_. Vamos usar um recurso chamado decora­tors (adornos ou decoradores):

**from** functools **import** total\_ordering  
  
@total\_ordering  
**class** Nome:  
 **def** \_\_init\_\_(self, nome):  
 **if** nome **is Nome or not** nome.strip():  
 **raise** ValueError(**"Nome não pode ser nulo nem é branco"**)  
 self.nome = nome  
 self.chave = **Nome**.CriaChave(nome)  
  
 **def** \_\_str\_\_(self):  
 **return** self.nome  
  
 **def** \_\_repr\_\_(self):  
 **return f"<Classe {**type(self).\_\_name\_\_**} em 0x{**id(self)**:x} Nome: {**self.nome **} Chave: {** self.chave **}>"  
  
 def** \_\_eq\_\_(self, outro):  
 print(**"\_\_eq\_\_Chanado"**)  
 **return** self.nome == outro.nome  
  
 **def** \_\_lt\_\_(self, outro):  
 print(**"\_\_It\_\_Chanado"**)  
 **return** self.nome < outro.nome  
  
 @staticmethod  
 **def** CriaChave(nome):  
 **return** nome.strip().lower()

O primeiro decorador @total\_ordering é definido no módulo functools, por isso tivemos de importa-lo no inicio de nosso programa. Ele é responsavel por implementar, ou seja, gerar o código responsavel pela implementação de todos os métodos de comparação especiais, a parrir de\_\_eq\_\_e de\_\_It\_\_. Dessa forma,

\_\_neq\_\_sera a negação de\_\_eq\_\_;

\_\_gt\_\_, a negação de\_\_It\_\_;

\_\_le\_\_, a combinação de\_\_It\_\_com\_\_eq\_\_;

\_\_ge\_\_, a combinação de\_\_gt\_\_com\_\_eq\_\_,

Implementado, assim, todos os operadores de comparacao (==, !=, >, <, >=, <=).

Vejamos o resultado no interpretador:

from nome import Nome

print(A == Nome(**"Nilo"**)) *# \_\_eq\_\_Chamado True*print(A != Nome(**"Nilo"**)) *# \_\_eq\_\_Chamado False*print(A > Nome(**"Nllo"**)) *# \_\_It\_\_Chamado False*print(A < Nome(**"Nilo"**)) *# \_\_It\_\_Chamado False*print(A <= Nome(**"Nilo"**))  
**"""  
\_\_It\_\_Chamado  
\_\_eq\_\_Chamado  
True  
"""**print(A >= Nome(**"Nilo"**)) *# \_\_It\_\_Chamado True*

Outro exemplo @staticmethod decorator

Observe que utilizamos um outro decorador @staticnethod antes da definição do método CriaChave. Veja tambem que o metodo CriaChave não possui o parâmetro self. Esse decorador cria um metodo estático, isto é, um metodo que pode ser chamado apenas com o nome da classe, não necessitando de um objeto para ser chamado. Vejamos no interpretador:

print(A.CriaChave(**"X"**)) *# x*print(Nome.CriaChave(**"X"**)) *# x*

Tanto a chamada de CriaChave com o objeto A quanto a chamada com o nome da classe em Nome.CriaChave funcionaram como esperado. Métodos estáticos são utilizados para implementar métodos que não acessam ou que não precisam acessar as propriedades de uma instância da classe. No caso do método CriaChave, apenas o parâmetro nome e necessário para seu funcionamento. O decorador @staticnethod é necessário para informar ao interpretador não passar o parâmetro self automaticamente, como faz nos métodos normais, não estáticos. A vantagem de definirmos métodos estáticos e agrupar certas funções em nossas classes.

Dessa forma, fica claro que CriaChave funciona no contexto de nomes. Se fosse definido como uma função fora de uma classe, sua implementacção poderia parecer mais genérica do que realmente é. Dessa forma, métodos estáticos ajudam a manter o programa coeso e a manter o contexto da implementação do método.

O programa seguinte introduziu um problema de consistência de dados. Veja que, ao construirmos o objeto em \_\_init\_\_, verificamos o valor de nome e também atualizamos a chave de pesquisa. O problema é que podemos alterar nome sem qualquer validação e deixar chave em um estado inválido. Vejamos isso no interpretador:

print(repr(A)) *# <Classe Nome em 0x1bda6204688 Nome: Nilo Chave: nilo>*A.nome = **"Teste"**print(repr(A)) *# <Classe Nome em 0x180d1cc3208 Nome: Teste Chave: nilo>*A.chave = **"TST"**print(repr(A)) *# <Classe Nome em 0x180d1cc3208 Nome: Teste Chave: TST>*

Outro exemplo @property decorator

Veja que, depois da construção do objeto, o nome e a chave estão corretos, pois essa operação é garantida pela nossa implementação de\_\_init\_\_. No entanto, um acesso a A. nome não atualizara o valor da chave, que continua com o valor configurado pelo construtor. O mesmo acontece com o atributo chave que pode ser alterado sem qualquer ligação com o nome em si. Na programação orientada a objetos, uma classe é responsável por gerenciar seu estado interno e manter a consistência do objeto entre chamadas de métodos. Para garantir que nome e chave sempre sejam atualizados corretamente, vamos fazer uso de um recurso do Python que utiliza métodos para realizar a atribuição e a leitura de valores. Esse recurso é chamado de propriedade.

O programa seguinte utilizara um outro recurso, uma convenção especial de nomenclatura de nossos métodos para esconder os atributos da classe e impedir o acesso direto a eles de fora da classe:

**from** functools **import** total\_ordering  
  
@total\_ordering  
**class** Nome:  
 **def** \_\_init\_\_(self, nome):  
 self.nome = nome  
  
 **def** \_\_str\_\_(self):  
 **return** self.nome  
  
 **def** \_\_repr\_\_(self):  
 **return f"<Classe {**type(self).\_\_name\_\_**} em 0x{**id(self)**:x} Nome: {**self.\_\_nome**} Chave: {**self.\_\_chave**}>"  
  
 def** \_\_eq\_\_(self, outro):  
 **return** self.nome == outro.nome  
  
 **def** \_\_lt\_\_(self, outro):  
 **return** self.nome < outro.nome  
  
 @property  
 **def** nome(self):  
 **return** self.\_\_nome  
 @nome.setter  
 **def** nome(self, valor):  
 **if** valor **is** Nome **or not** valor.strip():  
 **raise** ValueError(**"Nome não pode ser nulo nem é branco"**)  
 self.\_\_nome = valor  
 self.\_\_chave = Nome.CriaChave(valor)  
  
 @property  
 **def** chave(self):  
 **return** self.\_\_chave  
  
 @staticmethod  
 **def** CriaChave(nome):  
 **return** nome.strip().lower()

O programa utiliza outros decoradores como @property e @nome.setter. Esses decoradores modificam o método logo abaixo deles, transformando-os em propriedades. O primeiro decorador, @property, transforma o método none na propriedade nome; dessa forma, sempre que escrevermos objeto. nome, estaremos chamando esse método, que retorna self.\_\_nome. Já o segundo decorador, @nome.setter, transforma o método nome na propriedade usada para alterar o valor de\_\_none.

Dessa forma, quando escrevermos objeto.nome = valor, esse método será chamado para efetuar as alterações. Observe que copiamos a verificação de tipo e a criação da chave do método \_\_init\_\_ para o método marcado por @nome.setter. Assim, chamamos o método ao escrevermos: self.nome = nome, uma vez que self.nome é agora uma propriedade. Outro detalhe importante é que acrescentamos duas sublinhas (\_\_) antes do nome dos atributos nome e chave, fazendo-os \_\_none e \_\_chave, dessa forma, ficam escondidos quando acessados de fora da classe. Esse "esconder" é apenas um detalhe da implementacao do Python, que modifica o nome desses atributos de forma a torna-los inacessiveis (name mangling). No entanto, seus nomes foram apenas transformados pela adição de \_ e do nome da classe, fazendo com que\_\_chave se torne \_Nome\_\_chave. Mas essa proteção é sunciente para garantir que nome e chave estejam sincronizadas e que nossa classe funcione como esperado. Em Python, preste muita atencjao ao utilizar nomes que comecam com \_ ou \_\_. Esses símbolos indicam que esses atributos não devem ser acessados, exceto pelo código da propria classe. Não confundir os atributos protegidos, cujo nome começa por \_\_, com o nome dos métodos mágicos especiais que já vimos, que comecam e terminam por\_\_. Podemos ver que \_\_none e \_\_chave continuam acessiveis dentro do codigo da classe por self.\_\_none e self.\_\_chave.

Vamos experimentar o novo codigo no interpretador:

**from** nome\_decorador\_property **import** Nome  
  
A = Nome(**"Nilo"**)  
  
print(repr(A)) *# <Classe Nome em 0x1bda6204688 Nome: Nilo Chave: nilo>*A.nome = **"Teste"**print(repr(A)) *# <Classe Nome em 0x2183c4d3208 Nome: Teste Chave: teste>  
  
# A.chave = "TST" # AttributeError: can't set attribute***"""  
Porém, ainda é possível ter acesso ao nome e chave,   
com o truque de usar o nome completo após a escondida   
do Python (name mangling): \_nonedaclasse\_\_atributo,   
gerando: \_Nome\_\_chave. Entenda esse tipo de acesso como   
uma curiosidade e que, quando um programador marcar um   
atributo com \_\_. esta dizendo: não utilize esse atributo   
fora da classe, salvo se tiver certeza do que esta fazendo.  
"""***# Mas ainda é possível driblar esta estrutura na seguinte maneira*A.\_Nome\_\_nome = **"Nilo menezes"**A.\_Nome\_\_chave = **"TST"**print(repr(A)) *# <Classe Nome em 0x180d1cc3208 Nome: Teste Chave: TST>*

Exemplo de uma agenda

Podemos visualizar nossa agenda como uma lista de nomes e telefones, nossa agenda deverá suportar vários tele­fones, e cada telefone tera um tipo especifico (celular, trabalho etc.). Vamos comegar pela classe que vai gerenciar os tipos de telefone. Vejamos o codigo:

**from** functools **import** total\_ordering  
  
@total\_ordering  
**class** TipoTelefone:  
 **def** \_\_init\_\_(self, tipo):  
 self.tipo = tipo  
  
 **def** \_\_str\_\_(self):  
 **return** self.tipo  
  
 **def** \_\_eq\_\_(self, outro):  
 **if** outro **is None**:  
 **return False  
 return** self.tipo == outro.tipo  
  
 **def** \_\_lt\_\_(self, outro):  
 **return** self.tipo < outro.tipo

O programa implementa nosso tipo de telefone. Veja que implementamos o metodo\_\_str\_\_para exibir o nome do tipo entre parenteses, e o metodo \_\_eq\_\_ e\_\_It\_\_para ativar a comparação por tipo. Nesse exemplo, não usamos propriedades, pois na classe TipoTelefone não estamos fazendo qualquer validação. A grande vantagem de utilizar propriedades é que podemos, mais tarde, se necessário, transformar self.tipo em uma propriedade, sem alterar o código que utiliza a classe.

Observe que, no metodo\_\_eq\_\_, incluimos uma verificação se a outra parte for None. Essa verificação é necessaria porque nossa futura agenda utilizara None quando não soubermos o tipo de um telefone. Assim, self.tipo == outro.tipo falharia, pois None não possui um atributo chamado tipo. O teste se outro for None foi feito com o operador is do Python. Isso é importante, pois, se utilizarmos ==, estaremos chamando recursivamente o método \_\_eq\_\_, que é chamado pelo operador de comparação (==). A classe Telefone sera implementada pelo programa a seguir:

**class** Telefone:  
 **def** \_\_init\_\_(self, numero, tipo=**None**):  
 self.numero = numero  
 self.tipo = tipo  
  
 **def** \_\_str\_\_(self):  
 **if** self.tipo **is not None**:  
 tipo = self.tipo  
 **else**:  
 tipo = **""  
 return f"{**self.numero**} {**tipo**}"  
  
 def** \_\_eq\_\_(self, outro):  
 **return** self.numero == outro.numero **and** (  
 (self.tipo == outro.tipo) **or** (  
 self.tipo **is None or** outro.tipo **is None**))  
  
 @property  
 **def** numero(self):  
 **return** self.\_\_numero  
  
 @numero.setter  
 **def** numero(self, valor):  
 **if** valor **is None or not** valor.strip():  
 **raise** ValueError(**"Número não pode ser None ou em branco"**)  
 self.\_\_numero = valor

No método \_\_eq\_\_, implementamos a verificação de forma a ignorar um Telefone sem TipoTelefone, ou seja, um telefone em que o tipo é None. Dessa forma, se compararmos dois objetos de Telefone e um ou ambos possuirem o tipo valendo None, o resultado da comparação será decidido apenas se o número for idêntico. Se as duas instâncias de Telefone possuirem um tipo valido (diferente de None), então o tipo fará parte da comparação, sendo iguais apenas se os números e os tipos forem iguais.

Agora que temos as classes Nome, Telefone e TipoTelefone, podemos passar a pensar em como organizar os objetos dessas classes. Podemos ver nossa agenda como uma grande lista, em que cada elemento é um composto de Nome com uma lista de objetos do tipo Telefone. Vamos chamar cada entrada em nossa agenda de DadoAgenda e implementar uma classe para agrupar instâncias das duas classes:

**from** nome **import** Nome  
**from** telefone **import** Telefone  
**from** listaunica **import** ListaUnica  
  
**class** Telefones(ListaUnica):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(Telefone)  
  
**class** DadoAgenda:  
 **def** \_\_init\_\_(self, nome):  
 self.nome = nome  
 self.telefones = Telefones()  
   
 @property  
 **def** nome(self):  
 **return** self.\_\_nome  
   
 @nome.setter  
 **def** nome(self, valor):  
 **if not** isinstance(valor, Nome):  
 **raise** TypeError(**"nome deve ser instância da classe Nome"**)  
 self.\_\_nome = valor  
  
 **def** pesquisaTelefone(self,telefone):  
 posicao = self.telefones.pesquisa(Telefone(telefone))  
 **if** posicao == -1:  
 **return None  
 else**:  
 **return** self.telefone

Como cada objeto de DadoAgenda pode conter vários telefones, criamos uma classe Telefones que herda da classe ListaUnica seu comportamento. Dessa forma self.nome de DadoAgenda é uma instancia da classe Nome, e self. telefones uma ins­tancia de Telefones. Veja que adicionamos uma propriedade nome para facilitar o acesso ao objeto de self.\_\_nome e fazer as verificações de tipo necessárias. Adicio­namos também um método pesquisaTelefone que transforma uma string em um objeto da classe Telefone, sem tipo. Esse objeto é entaão utilizado pelo método pesquisa, vindo da implementação original de ListaUnica que retorna a posição do objeto na lista, ou -1, caso ele não seja encontrado. Veja que o método pesquisaTelefone retorna uma instância de Telefone se ela for encontrada na lista ou None, caso contrário. Vejamos agora a classe Agenda:

**from** nome **import** Nome  
**from** tipo\_telefone **import** TipoTelefone  
**from** listaunica **import** ListaUnica  
**from** dado\_agenda **import** DadoAgenda  
  
**class** TiposTelefone(ListaUnica):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(TipoTelefone)  
  
**class** Agenda(ListaUnica):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(DadoAgenda)  
 self.tiposTelefone = TiposTelefone()  
  
 **def** adicionaTipo(self, tipo):  
 self.tiposTelefone.adiciona(TipoTelefone(tipo))  
  
 **def** pesquisaNome(self, nome):  
 **if** isinstance(nome, str):  
 nome = Nome(nome)  
 **for** dados **in** self.lista:  
 **if** dados.nome == nome:  
 **return** dados  
 **else**:  
 **return None  
  
 def** ordena(self):  
 super().ordena(**lambda** dado: str(dado.nome))

O programa é uma listagem parcial, apenas para a explicação da classe Agenda. O programa completo precisa importar as definições de classe feitas anteriormente. Observe que a classe Agenda é uma subclasse de ListaUnica configurada para aceitar somente objetos do tipo DadoAgenda. Veja que defmimos uma classe TiposTelefone, também herdando de ListaUnica, para manter a lista de tipos de telefone. Em nossa agenda, os tipos de telefone são pre-configurados na classe Agenda. Para facilitar o trabalho de inclusão de novos tipos, incluimos o método adicionaTipo, que prepara uma string, transformando-a em objeto de TipoTelefone e incluindo-o na lista de tipos validos. Outro método a notar é pesquisaNone, que recebe o nome a pesquisar como objeto da classe Nome ou como string. Veja que utilizamos a verificação do tipo do parâmetro nome para transforma-lo em objeto de Nome, caso necessário, deixando nossa classe funcionar com strings ou com objetos do tipo None.

A função então pesquisa na lista interna de dados e retorna o objeto, caso encontrado. È importante notar a diferença dessa função de pesquisa e outras funções de pesquisa que definimos anteriormente. Em pesquisaNome, o objeto é retornado ou None, caso não seja encontrado. Vamos utilizar as referências retornadas por pesquisaNome para editar os valores da instância de DadoAgenda diretamente, sem a necessidade de reinseri-los na lista. Isso ficara mais claro quando analisarmos o programa da agenda completo. Para finalizar, o método ordena cria uma função para extrair a chave de ordenação de DadoAgenda; nesse caso, o nome.

Antes de passarmos para o programa da agenda, vamos construir uma classe Menu para exibir o menu principal:

**class** Menu:  
 **def** \_\_init\_\_(self):  
 self.opcoes = [[**"Sair"**, **None**]]  
  
 **def** adicionaopcao(self, nome, funcao):  
 self.opcoes.append([nome, funcao])  
  
 **def** exibe(self):  
 print(**"===="**)  
 print(**"Menu"**)  
 print(**"====\n"**)  
 **for** i, opcao **in** enumerate(self.opcoes):  
 print(**f"[{**i**}] - {**opcao[0]**}"**)  
 print()  
  
 @staticmethod  
 **def** nulo\_ou\_vazio(texto):  
 **return** texto **is None or not** texto.strip()  
  
 @staticmethod  
 **def** valida\_faixa\_inteiro\_ou\_branco(pergunta, inicio, fim):  
 **while True**:  
 **try**:  
 entrada = input(pergunta)  
 **if** Menu.nulo\_ou\_vazio(entrada):  
 **return None** valor = int(entrada)  
 **if** inicio <= valor <= fim:  
 **return** valor  
 **except** ValueError:  
 print(**f"Valor inválido, favor digitar entre {**inicio**} e {**fim**}"**)  
  
 @staticmethod  
 **def** valida\_faixa\_inteiro(pergunta, inicio, fim, padrao=**None**):  
 **while True**:  
 **try**:  
 entrada = input(pergunta)  
 **if** Menu.nulo\_ou\_vazio(entrada) **and** padrao **is not None**:  
 entrada = padrao  
 valor = int(entrada)  
 **if** inicio <= valor <= fim:  
 **return** valor  
 **except** ValueError:  
 print(**f"Valor inválido, favor digitar entre {**inicio**} e {**fim**}"**)  
  
 **def** execute(self):  
 **while True**:  
 self.exibe()  
 escolha = Menu.valida\_faixa\_inteiro(**"Escolha uma opção: "**,  
 0, len(self.opcoes) - 1)  
 **if** escolha == 0:  
 **break** self.opcoes[escolha][1]()

Nosso menu é bem simples, adicionando a opção para "Sair" como valor padrão. O método exibe mostra o menu na tela, percorrendo a lista de opções. Jé o método execute mostra continuamente o menu, pedindo uma escolha e cha-mando o método correspondente. Vejamos como utilizar a classe Menu na inicialização da classe AppAgenda no programa completo:

**from** agenda **import** Agenda  
**from** menu **import** Menu  
**from** nome **import** Nome  
**from** telefone **import** Telefone  
**from** dado\_agenda **import** DadoAgenda  
**import** sys  
**import** pickle  
**import** os.path  
  
**class** AppAgenda:  
 @staticmethod  
 **def** pede\_nome():  
 **return** input(**"Nome: "**)  
  
 @staticmethod  
 **def** pede\_telefone():  
 **return** input(**"Telefone: "**)  
  
 @staticmethod  
 **def** mostra\_dados(dados):  
 print(**f"Nome: {**dados.nome**}"**)  
 **for** telefone **in** dados.telefones:  
 print(**f"Telefone: {**telefone**}"**)  
 print()  
  
 @staticmethod  
 **def** mostra\_dados\_telefone(dados):  
 print(**f"Nome: {**dados.nome**}"**)  
 **for** i, telefone **in** enumerate(dados.telefones):  
 print(**f"{**i**} - Telefone: {**telefone**}"**)  
 print()  
  
 @staticmethod  
 **def** pede\_nome\_arquivo():  
 **return** input(**"Nome do arquivo: "**)  
  
 **def** \_\_init\_\_(self):  
 self.agenda = Agenda()  
 self.agenda.adicionaTipo(**"Celular"**)  
 self.agenda.adicionaTipo(**"Residencia"**)  
 self.agenda.adicionaTipo(**"Trabalho"**)  
 self.agenda.adicionaTipo(**"Fax"**)  
 self.menu = Menu()  
 self.menu.adicionaopcao(**"Novo"**, self.novo)  
 self.menu.adicionaopcao(**"Altera"**, self.altera)  
 self.menu.adicionaopcao(**"Apaga"**, self.apaga)  
 self.menu.adicionaopcao(**"Lista"**, self.lista)  
 self.menu.adicionaopcao(**"Arquiva"**, self.arquiva)  
 self.menu.adicionaopcao(**"Le arquivo"**, self.le)  
 self.menu.adicionaopcao(**"Ordena"**, self.ordena)  
 self.ultimo\_nome = **None  
  
 def** pede\_tipo\_telefone(self, padrao=**None**):  
 **for** i, tipo **in** enumerate(self.agenda.tiposTelefone):  
 print(**f" {**i**} - {**tipo**} "**, end=**None**)  
 t = Menu.valida\_faixa\_inteiro(**"Tipo: "**, 0, len(self.agenda.tiposTelefone) - 1, padrao)  
 **return** self.agenda.tiposTelefone[t]  
  
 **def** pesquisa(self, nome):  
 dado = self.agenda.pesquisaNome(nome)  
 **return** dado  
  
 *# opção 1* **def** novo(self):  
 novo = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(novo):  
 **return** nome = Nome(novo)  
 **if** self.pesquisa(nome) **is not None**:  
 print(**"Nome ja existe!"**)  
 **return** registro = DadoAgenda(nome)  
 self.menu\_telefones(registro)  
 self.agenda.adiciona(registro)  
  
 *# opção 2* **def** altera(self):  
 **if** len(self.agenda) == 0:  
 print(**"Agenda vazia, nada a alterar"**)  
 **return** nome = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(nome):  
 **return** p = self.pesquisa(nome)  
 **if** p **is not None**:  
 print(**"\nEncontrado:\n"**)  
 AppAgenda.mostra\_dados(p)  
 print(**"Digite enter caso nao queira alterar o nome"**)  
 novo = AppAgenda.pede\_nome()  
 **if not** Menu.nulo\_ou\_vazio(novo):  
 p.nome = Nome(novo)  
 self.menu\_telefones(p)  
 **else**:  
 print(**"Nome não encontrado!"**)  
  
 *# opção 3* **def** apaga(self):  
 **if** len(self.agenda) == 0:  
 print(**"Agenda vazia, nada a apagar"**)  
 nome = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(nome):  
 **return** p = self.pesquisa(nome)  
 **if** p **is not None**:  
 print(p)  
 self.agenda.remove(p)  
 print(**f"Apagado. A agenda agora possui apenas: {**len(self.agenda)**} registros"**)  
 **else**:  
 print(**"Nome não encontrado."**)  
  
 *# opção 4* **def** lista(self):  
 print(**"\nAgenda"**)  
 print(**"-"** \* 60)  
 **for** e **in** self.agenda:  
 AppAgenda.mostra\_dados(e)  
 print(**"-"** \* 60)  
  
 *# opção 5* **def** arquiva(self):  
 **if** self.ultimo\_nome **is not None**:  
 print(**f"Último nome utilizado foi '{**self.ultimonome**}'"**)  
 print(**"Digite enter caso queira utilizar o mesmo nome"**)  
 nome\_arquivo = AppAgenda.pede\_nome\_arquivo()  
 **if** Menu.nulo\_ou\_vazio(nome\_arquivo):  
 **if** self.ultimonome **is not None**:  
 nome\_arquivo = self.ultimonome  
 **else**:  
 **return  
 with** open(nome\_arquivo, **"wb"**) **as** arquivo:  
 pickle.dump(self.agenda, arquivo)  
  
 *# opção 6* **def** le(self, nome\_arquivo=**None**):  
 **if** nome\_arquivo **is None**:  
 nome\_arquivo = AppAgenda.pede\_nome\_arquivo()  
 **if** Menu.nulo\_ou\_vazio(nome\_arquivo) **or not** os.path.isfile(nome\_arquivo):  
 **return  
 with** open(nome\_arquivo, **"rb"**) **as** arquivo:  
 self.agenda = pickle.load(arquivo)  
 self.ultimo\_nome = nome\_arquivo  
  
 *# opção 7* **def** ordena(self):  
 self.agenda.ordena()  
 print(**"\nAgenda ordenada\n"**)  
  
 **def** menu\_telefones(self, dados):  
 **while True**:  
 print(**"\nEditando telefones\n"**)  
 AppAgenda.mostra\_dados\_telefone(dados)  
 **if** len(dados.telefones) > 0:  
 print(**"\n[A] - alterar\n[D] - apagar\n"**, end=**""**)  
 print(**"[N] - novo\n[S] - sair\n"**)  
 operacao = input(**"Escolha uma operação: "**)  
 operacao = operacao.lower()  
 **if** operacao **not in** [**"a"**, **"d"**, **"n"**, **"s"**]:  
 print(**"Operação inválida. Digite A, D, N ou S"**)  
 **continue  
 if** operacao == **'a' and** len(dados.telefones) > 0:  
 self.altera\_telefones(dados)  
 **elif** operacao == **'d' and** len(dados.telefones) > 0:  
 self.apaga\_telefone(dados)  
 **elif** operacao == **'n'**:  
 self.novo\_telefone(dados)  
 **elif** operacao == **"s"**:  
 **break** *# opção N* **def** novo\_telefone(self, dados):  
 telefone = AppAgenda.pede\_telefone()  
 **if** Menu.nulo\_ou\_vazio(telefone):  
 **return  
 if** dados.pesquisaTelefone(telefone) **is not None**:  
 print(**"Telefone ja existe"**)  
 tipo = self.pede\_tipo\_telefone()  
 dados.telefones.adiciona(Telefone(telefone, tipo))  
  
 *# opção A* **def** altera\_telefones(self, dados):  
 t = Menu.valida\_faixa\_inteiro\_ou\_branco(**"Digite a posição do "  
 "número a alterar, enter para sair: "**, 0, len(dados.telefones) - 1)  
 **if** t **is** Nome:  
 **return** telefone = dados.telefones[t]  
 print(**f"Telefone: {**telefone**}"**)  
 print(**"Digite enter caso não queira alterar o número"**)  
 novotelefone = AppAgenda.pede\_telefone()  
 **if not** Menu.nulo\_ou\_vazio(novotelefone):  
 telefone.nunero = novotelefone  
 print(**"Digite enter caso não queira alterar o tipo"**)  
 telefone.tipo = self.pede\_tipo\_telefone(self.agenda.tiposTelefone.pesquisa(telefone.tipo))  
  
 *# opção D* **def** apaga\_telefone(self, dados):  
 t = Menu.valida\_faixa\_inteiro\_ou\_branco(**"Digite a posição do "  
 "número a apagar, enter para sair: "**, 0, len(dados.telefones) - 1)  
 **if** t **is None**:  
 **return** dados.telefones.remove(dados.telefones[t])  
  
 **def** execute(self):  
 self.menu.execute()  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 app = AppAgenda()  
 **if** len(sys.argv) > 1:  
 app.le(sys.argv[1])  
 app.execute()

A classe AppAgenda é nossa aplicacao agenda em si. Veja a implementação de\_\_init\_\_, na qual criamos uma instância de Agenda para conter nossos dados e populamos os tipos de telefone que queremos trabalhar. Também criamos uma instância de Menu e adicionamos as opções. Observe que passamos o nome da opção e o método correspondente a cada escolha. O atributo self.ultimonome é usado para guardar o nome usado na última leitura do arquivo.

Na agenda, note que dividimos o tratamento de nomes e telefones. Como podemos ter vários telefones, outro menu com opções para o gerenciamento de telefones aparece. Voce pode ver os detalhes na implementação do método menu\_telefones. Em nossa agenda, adotamos o seguinte comportamento quando um valor não muda, digitamos apenas Enter. Em todas as opções, você deve perceber o tratamento de entradas em branco, até criamos uma função de suporte para isso: nulo\_ou\_vazio.

Em todos os métodos de edição, vocâ pode perceber que utilizamos apenas os métodos fornecidos pela classe ListaUnica. Veja também que as pesquisas retornam os objetos, e não apenas a posição deles na lista.

Os métodos grava\_arquivo e le foram modificados para utilizar o modulo pickle do Python. Como nossa agenda agora possui vários telefones, cada um com um tipo, a utilização de arquivos simples se tornaria muito trabalhosa. O módulo pickle fornece métodos que gravam um objeto ou uma lista de objetos em disco. No método grava\_arquivo, utilizamos a função pickle.dump, que grava no arquivo o objeto agenda inteiro. No método le, utilizamos pickle.load para recriar nossa agenda a partir do arquivo em disco. A utilização do pickle facilita muito a tarefa de serializar os dados em um arquivo, ou seja, representar os objetos de forma que possam ser reconstruidos, caso necessario (desserialização).

Como última modificação, adicionamos a possibilidade de passar o nome do arquivo da agenda como parâmetro. Caso o nome seja passado, o arquivo é lido antes de apresentar o menu. Observe tambem que o método execute tem um loop infinito que roda até sairmos do menu principal.

CONCORRÊNCIA

GIL (Global Interpreter Locker)

* Mecanismo de exclusão mútua do interpretador;
* Apenas uma thread pode executar instruções por vez;
* Threading
  + Presente na biblioteca padrão;
  + Classes de alto nível de abstração;
  + Módulo thread, que implementa as rotinas de baixo nível e geralmente não é usado diretamente;

Stackless Python

* Concorrência baseada em tasklets (green thread);
* Também utiliza canais síncronos para comunicação e sincronização de tasks;
* Multiprocessing
  + Paralelismo através de múltiplos processos Python;
  + Comunicação principalmente através de envio de mensagens;
  + Maior overhead de sincronização;
* Green threads
  + Threads no user space;
  + Representam uma abstração às threads do sistema operacional(kernel space);

EXCEÇÕES

O tratamento de erros em Python é realizado através de blocos utilizando as palavras-chaves try e except:

meu\_dicionario = {**'chave.a'**: 1, **'chave.b'**: 2}  
  
**try**:  
 print(meu\_dicionario[**'chave\_c'**])  
**except** KeyError:  
 print(**"Chave não encontrada!"**)

Para levantar exceções em Python, a palavra reservada raise deve ser utilizada:

**def** verifica\_numero\_primo(numero):  
 **if** numero < 0:  
 **raise** ValueError(**'O valor deve ser maior ou igual a zero!'**)  
  
verifica\_numero\_primo(-1)

No Python, as palavras-chave else e finally também podem ser usadas junto com as cláusulas try e except. Enquanto o bloco de exceção é executado se a exceção ocorrer dentro do bloco de tentativa, o bloco else é processado se o bloco de tentativa for considerado livre de exceção.

**try**:  
 print(**"try block"**)  
 x=int(input(**'Enter a number: '**))  
 y=int(input(**'Enter another number: '**))  
 z=x/y  
**except** ZeroDivisionError:  
 print(**"except ZeroDivisionError block"**)  
 print(**"Division by 0 not accepted"**)  
**else**:  
 print(**"else block"**)  
 print(**"Division = "**, z)  
**finally**:  
 print(**"finally block"**)  
 x=0  
 y=0  
print (**"Out of try, except, else and finally blocks."** )

SyntaxError

*# Sem fechamento de string "*a = **"5**

**for** e **in** [1,2,3] *# SyntaxError: invalid syntax(sem dois pontos:)* print(e)

Um dos tipos mais comuns. Um erro de sintaxe acontece quando o interpretador não consegue ler o que você escreveu. Em outras palavras, seu programa está mal formatado, normalmente com erros de digitação ou símbolos esquecidos. Sempre que ocorrer um erro de sintaxe, verifique:

1. A linha onde ocorreu um erro {line), mas não se esqueça de também olhar as linhas anteriores, caso não ache o erro.
2. Se você fechou todas as aspas que abriu, o mesmo valendo para parênteses.
3. Os dois pontos após o while, for, if, else, definições de função, métodos e classes, para indicar o início de um novo bloco.
4. Se você não trocou letras minúsculas por maiúsculas e vice-versa.
5. Se você digitou corretamente todos os nomes.

Indentation Error

*# IndentationError: unindent does not match any outer indentation level*x = 8  
**while** x < 16:  
 print(x)  
 x = x + 1

Nesse caso, observe que a linha x = x + 1 não está alinhada nem com o bloco do print nem com o while. Todas as linhas de um mesmo bloco devem ser alinhadas na mesma coluna.

Uma variação muito difícil de perceber desse erro e quando misturamos espaços em branco com tabulações (tabs). Configure seu editor de textos para substituir tabs por espaços em branco ou vice-versa. Jamais misture tabulações e espaços em branco em seus programas em Python.

KeyError

*# KeyError: 'sequro'*mensalidades = {**"carro"**: 586, **"casa"**: 1506}  
print(mensalidades[**"seguro"**])

A exceção KeyError ocorre quando acessamos ou tentamos acessar um dicionário usando uma chave que não existe. No exemplo anterior, o dicionário mensalidades contém as chaves carro e casa. Na linha da função print, tentamos acessar mensalidades["seguro"]. Nesse caso, "seguro" e a chave que causa a mensagem de erro, uma vez que ela não pertence ao dicionário. Atenção ao trabalhar com chaves string, pois Python diferencia letras minúsculas de maiúsculas.

NameError

*# NameError: name 'x' is not defined***while** x < 0:  
 print(x)  
 x = x + 1

Aqui a variável x é utilizada em while, mesmo antes de ser iniciada. Toda variável em Python precisa ser inicializada antes de ser utilizada. Lembre-se de que, para inicializarmos uma variável, devemos atribuir um valor inicial. No caso, poderíamos escrever x = 0 antes da linha de while para inicializar a variável x com zero, resolvendo o erro.

Ao receber esse erro, verifique também se escreveu corretamente o nome da variável. Python, leva em consideração letras minúsculas e maiúsculas e também nomes acentuados são também diferentes de nomes sem acentos.

ValueError

*# ValueError: invalid literal for int() with base 10: 'abc'*int(**"abc"**)

A exceção ValueError pode acontecer por diversas causas. Uma delas e a impossibilidade de converter um valor com as funções int ou float:

Ela pode ocorrer se, por exemplo, o valor retornado pela função input for inválido. Essa exceção também ocorre quando procuramos uma string que não existe, como mostrado a seguir.

*# ValueError: substring not found*s = **"Alo Mundo"**s.index(**"rei"**)

TypeError

Essa exceção acontece se tentamos chamar uma função usando mais parâmetros do que ela recebe. No exemplo a seguir, a função float foi chamada com dois parâmetros: 35 e 4. Lembre-se de que números em Python devem set escritos no formato americano, separando a parte inteira da parte decimal de um numero com ponto e não com virgula.

*# TypeError: float expected at most 1 arguments, got 2*float(35, 4)

TypeError também ocorre quando trocamos o tipo de um índice. No exemplo a seguir, tentamos utilizar a string "marrom" como índice de uma lista. Lembre-se de que listas só podem ser indexadas por números inteiros. Dicionários, por sua vez, permitem índices do tipo string.

*# TypeError: list indices must be integers or slices, not tuple*s= []  
s[**"amarelo"**, **"vermelho"**, **" verde"**]  
s[**"marrom"**]

IndexError

*# IndexError: string index out of range*s = **"ABCDE"**s[19]

IndexError indica que um valor inválido de índice foi utilizado. No exemplo a seguir, a string s contém apenas cinco elementos, podendo ter índices de 0 a 4.

TabError

Esse erro acontece quando misturamos tabs (caractere gerado quando você pressiona a tecla tab) e espaços para fazer os recuos e avanços em nossos programas. Esse erro é praticamente invisível, mas pode ser corrigido por seu editor de testes. Procure uma opção que converta todos os caracteres de tab por espaços para resolve-lo.

Criando Exceções

Para facilitar o gerenciamento de exceções em nossos programas, podemos criar novos tipos de exceção para diferenciar os erros gerados pela nossa aplicação dos gerados pela linguagem ou suas bibliotecas. Para criar nossas proprias exceções, vamos utilizar herança para estender a classe Exception do Python.

**class** NovaException(Exception):  
 **pass  
  
def** lancador():  
 **raise** NovaException  
**try**:  
 lancador()  
**except** NovaException **as** n:  
 print(**"Uma exceção do tipo NovaException foi lançada"**)

**try**:  
 *# IndexError: string index out of range* s = **"ABCDE"** s[19]  
**except** KeyError:  
 Excep.lancador() *# Uma exceção do tipo NovaException foi lançada*

Veja que NovaException e uma subclasse de Exception. Ao definirmos nossas próprias exceções, estas devem derivar da classe Exception. Observe que podemos udlizar essa nova exceção como todas as outras em Python, por exemplo, em um bloco try except. No caso, NovaException não adiciona nenhum método ou atributo a classe Exception, atuando apenas como um novo tipo, por isso, utilizamos apenas o pass em sua definição. A instrução pass pode ser utilizada em Python para indicar o flm de um bloco que seria vazio. Ela também pode ser usada para definir funções vazias quando ainda estamos escrevendo nosso programa e queremos apenas satisfazer a sintaxe da linguagem, para mais tarde modificar ou adicionar conteúdo e remover pass.

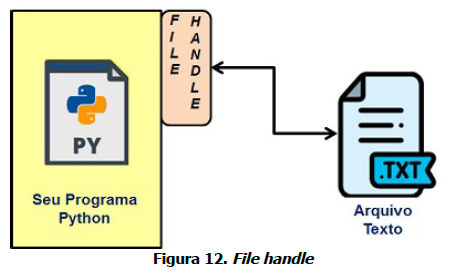
Ao desenvolver novos programas, é recomendado que você crie um tipo próprio de exceção para servir de super classe a todas as exceções do programa. Essas exceções não precisam substituir as previamente definidas pela linguagem, mas definir um significado próprio ao programa.

|  |  |
| --- | --- |
| **Exception** | **Descrição** |
| AssertionError | Gerado quando a instrução assertiva falha. |
| AttributeError | Gerado na atribuição de atributo ou referência falhar. |
| EOFError | Gerado quando a função input () atinge a condição de fim de arquivo. |
| FloatingPointError | Gerado quando uma operação de ponto flutuante falha. |
| GeneratorExit | Gerado quando o método close () de um gerador é chamado. |
| ImportError | Gerado quando o módulo importado não foi encontrado. |
| IndexError | Gerado quando o índice de uma sequência está fora do intervalo. |
| KeyError | Gerado quando uma chave não é encontrada em um dicionário. |
| KeyboardInterrupt | Gerado quando o usuário pressiona a tecla de interrupção (Ctrl + c ou delete). |
| MemoryError | Gerado quando uma operação fica sem memória. |
| NameError | Gerado quando uma variável não é encontrada no escopo local ou global. |
| NotImplementedError | Criado por métodos abstratos. |
| OSError | Gerado quando uma operação do sistema causa um erro relacionado ao sistema. |
| OverflowError | Gerado quando o resultado de uma operação aritmética é muito grande para ser representado. |
| ReferenceError | Gerado quando um proxy de referência fraco é usado para acessar um referente de coleta de lixo. |
| RuntimeError | Gerado quando um erro não se enquadra em nenhuma outra categoria. |
| StopIteration | Gerado pela função next () para indicar que não há mais itens a serem retornados pelo iterador. |
| SyntaxError | Gerado pelo analisador quando um erro de sintaxe é encontrado. |
| IndentationError | Gerado quando há um recuo incorreto. |
| TabError | Gerado quando o recuo consiste em guias e espaços inconsistentes. |
| SystemError | Gerado quando o intérprete detecta um erro interno. |
| SystemExit | Gerado pela função sys.exit (). |
| TypeError | Gerado quando uma função ou operação é aplicada a um objeto de um tipo incorreto. |
| UnboundLocalError | Gerado quando é feita referência a uma variável local em uma função ou método,  mas nenhum valor foi vinculado a essa variável. |
| UnicodeError | Gerado quando ocorre um erro de codificação ou decodificação relacionado ao Unicode. |
| UnicodeEncodeError | Gerado quando ocorre um erro relacionado ao Unicode durante a codificação. |
| UnicodeDecodeError | Gerado quando ocorre um erro relacionado ao Unicode durante a decodificação. |
| UnicodeTranslateError | Gerado quando ocorre um erro relacionado ao Unicode durante a tradução. |
| ValueError | Gerado quando uma função obtém um argumento do tipo correto, mas valor inadequado. |
| ZeroDivisionError | Gerado quando o segundo operando de uma operação de divisão ou módulo é zero. |

ARQUIVOS TEXTO: FILE HANDLE

Toda vez que um programa Python precisa acessar um arquivo – seja para importá-lo por inteiro para a memória ou para processá-lo sequencialmente (linha por linha) – é necessário, antes de tudo, usar a função open() para comandar a sua abertura.

A função open() estabelece uma “conversa” entre o Python e o sistema operacional do computador (Windows, Mac, Linux). Melhor explicando: abrir um arquivo, significa pedir ao sistema operacional para encontrar o endereço de localização do arquivo no HD, pen drive, cartão SD, enfim, no dispositivo em que ele esteja armazenado. Ao encontrar o endereço do arquivo, o sistema operacional retornará uma coisa chamada file handle para o programa Python. O file handle não é a mesma coisa que conteúdo do arquivo, ou seja, ele não consiste nos dados de fato. Na verdade, ele é uma espécie de ferramenta que lhe permite “manejar” os dados do arquivo.



Mas, antes de qualquer coisa, é muito importante deixar claro que existem bibliotecas que oferecem maneiras bem mais práticas para trabalhar bases de dados texto, especialmente arquivos estruturados no formato CSV. Um exemplo é a biblioteca ‘pandas’.

De qualquer forma, consideramos importante que você primeiro conheça e aprenda a utilizar o jeito padrão do Python para lidar com arquivos por dois motivos: (i) a técnica baseada em file handle é, muitas vezes, a alternativa mais eficiente para realizar o acesso sequencial a arquivos, um processo onde apenas uma linha por vez é carregada para a memória. Em muitas aplicações reais, onde a base de dados a ser trabalhada é muito volumosa e não cabe em memória, o acesso sequencial acaba até mesmo sendo o único método de acesso possível; (ii) o file handle é simples e adequado para lidar com arquivos separados por colunas.

Embora esse formato não seja muito popular nos dias de hoje, ele ainda é utilizado em algumas aplicações (por exemplo, é adotado por muitos países para armazenar dados de censos demográficos).

manipulador = open(arquivo, modo)

Sendo:

* Arquivo é uma string que indica o nome / caminho do arquivo a ser aberto no sistema de arquivos. Se o arquivo não estiver no mesmo diretório onde o script se encontra, devemos informar o caminho completo como parâmetro da função open().
* Modo é opcional, sendo que o padrão é t (modo de texto). Se não especificarmos o segundo parâmetro, a forma padrão leitura ('r') será utilizada.

|  |  |
| --- | --- |
| **Modo** | **Tipo de acesso** |
| r | Abre o arquivo para leitura ou o cria, caso não exista. O cursor é posicionado no inicio do arquivo. |
| r+ | Abre o arquivo para leitura e escrita ou o cria, caso não exista. O cursor é posicionado no início do arquivo. |
| w | Se o arquivo existir, ele é aberto para escrita e o seu tamanho é reduzido para zero (conteúdo do arquivo é apagado). SE não existir, ele é criado na hora.  O cursor é posicionado no início do arquivo. Se o conteúdo do arquivo for editado também será reescrito. |
| w+ | Se o arquivo já existir, ele tem tamanho reduzido para zero (conteúdo do arquivo é apagado) e é aberto para escrita e leitura. Se não existir, um novo arquivo é criado.  O cursor é posicionado no início do arquivo. |
| a | Abre o arquivo para escrita. Caso não exista, ele é criado e o cursor é posicionado no final do arquivo, As linhas que forem escritas estarão sempre no final do arquivo, mesmo se funções como fseek(6) ou similares forem utilizadas. |
| a+ | Abre o arquivo para leitura e escrita. Caso não exista, ele é criado e o cursor é posicionado no final do arquivo, As linhas que forem escritas estarão sempre no final do arquivo, mesmo se funções como fseek(6) ou similares forem utilizadas. |
| "ab", "rb", "r+b", "wb", "w+b" | "ab", "rb", "r+b", "wb", "w+b" - Abre arquivos para trabalhar com entrada e saída no modo binário, para plataformas Windows e Macintosh |

Se tentarmos abrir um arquivo para leitura que não existe, um erro será lançado.

*# FileNotFoundError: [Errno 2] No such file or directory: 'nome-errado.text'*f = open(**'nome-errado.text'**, **'r'**)

Se tentarmos abrir um arquivo para escrita que não existe, então ele será criado, porém, se ele já existir, todo seu conteúdo será apagado no momento em que abrimos o arquivo.

**for** line **in** open(**"myfile.txt"**):  
 print(line, end=**""**)

O problema com esse código é que ele deixa o arquivo aberto por um período indeterminado após a execução da parte do código. Isso não é um problema em scripts simples, mas pode ser um problema para aplicativos maiores. Devemos sempre fechar o arquivo aberto.

arquivo.close()

Os objetos file possuem um atributo closed que retorna True se o arquivo estiver fechado, e retorna False se o arquivo ainda estiver aberto. Desta forma, é possível verificar se um arquivo foi realmente fechado ou não. Veja o exemplo a seguir:

arquivo = open(**'nome\_errado.txt'**, **'r'**)  
conteudo\_arquivo = arquivo.read()  
  
print(**'Arquivo fechado? {}'**.format(arquivo.closed))  
**if not** arquivo.closed:  
 arquivo.close()  
 print(**"Arquivo fechado? {}"**.format(arquivo.closed))

Também é possível fechar um arquivo automaticamente. A instrução with permite que objetos como arquivos sejam usados de uma maneira que garanta que eles sempre sejam limpos de forma rápida e correta.

**with** open(**"myfile.txt"**) **as** f:  
 **for** line **in** f:  
 print(line, end=**""**)

Podemos alterar o ponteiro de arquivo por meio do método seek(), que recebe como argumento a posição desejada – um offset (deslocamento) em bytes.

Para voltarmos ao início do arquivo, usamos a posição zero, como seek(0). Caso você queira ir até o oitavo byte, por exemplo, use seek(8). No geral, o número do byte corresponde ao número do caractere no arquivo, exceto quando são usados caracteres que ocupam mais de um byte, como as codificações de idiomas asiáticos em UTF-8.

Também podemos descobrir a posição atual dentro do arquivo com o método tell().

Veja os exemplos a seguir, nos quais utilizamos um arquivo de nome arq01.txt:

arquivo = open(**'nome\_errado.txt'**, **'r'**)  
  
print(**'Posição no arquivo: {}'**.format(arquivo.tell())) *# 0*print(arquivo.read()) *# conteudo de arquivo*print(**'Posição no arquivo: {}'**.format(arquivo.tell())) *# 19*print(arquivo.read()) *# ""*arquivo.seek(8)  
print(**'Posição no arquivo: {}'**.format(arquivo.tell())) *# 8*print(arquivo.read()) *# de arquivo*arquivo.close()

O terceiro parâmetro é opcional e nele especificamos a codificação do arquivo.

arquivo = open(**'nome\_errado.txt'**, **'r'**, encoding=**"utf8"**)

Gravar linhas no arquivo

O Python fornece o método writelines() para salvar o conteúdo de um objeto de lista em um arquivo. Como o caractere de nova linha não é gravado automaticamente no arquivo, ele deve ser fornecido como parte da cadeia. Caso houver conteúdo no arquivo o comando writelines() sobreescreve.

lines = [**"Hello world.\n"**, **"Welcome to TutorialsTeacher.\n"**]  
arquivo = open(**'nome\_errado.txt'**, **'w'**, encoding=**"utf8"**)  
  
arquivo.writelines(lines)  
arquivo.close()

Lendo de um arquivo

Três métodos diferentes são fornecidos para ler dados do arquivo.

1. readline (): lê os caracteres desde a posição atual de leitura até um caractere de nova linha.
2. read (caracteres): lê o número especificado de caracteres, começando na posição atual.
3. readlines (): lê todas as linhas até o final do arquivo e retorna um objeto de lista.

arquivo = open(**'nome\_errado.txt'**, **'r'**, encoding=**"utf8"**)  
  
line = arquivo.readline()  
print(line)  
arquivo.close()

Como você pode ver, temos que abrir o arquivo no modo 'r'. O método readline () retornará uma primeira linha e, em seguida, apontará para a segunda linha no arquivo. Para ler todas as linhas de um arquivo, use o loop while, como mostrado abaixo.

line = arquivo.readline()  
**while** line != **''**:  
 print(line)  
 line = arquivo.readline()  
arquivo.close()

### File Iterator

O objeto de arquivo possui um iterador embutido. O programa a seguir lê o arquivo especificado, linha por linha, até que StopIteration seja gerado, ou seja, o final do arquivo é atingido.

arquivo = open(**'nome\_errado.txt'**, **'r'**, encoding=**"utf8"**)  
  
**while True**:  
 **try**:  
 line = next(arquivo)  
 print(line)  
 **except** StopIteration:  
 **break**arquivo.close()

Use o loop for para ler um arquivo facilmente.

**for** line **in** arquivo:  
 print(line)

Todos os métodos de objeto de arquivo são fornecidos abaixo:

|  |  |
| --- | --- |
| **Method** | **Descrição** |
| file.close() | Fecha o arquivo. |
| file.flush() | Libera o buffer interno. |
| next(file) | Retorna a próxima linha do arquivo toda vez que é chamado. |
| file.read([size]) | Lê em um número especificado de bytes do arquivo. |
| file.readline() | Lê uma linha inteira do arquivo. |
| file.readlines() | Lê até EOF e retorna uma lista contendo as linhas. |
| file.seek(offset, from) | Define a posição atual do arquivo. |
| file.tell() | Retorna a posição atual do arquivo |
| file.write(str) | Grava uma string no arquivo. Não há reembolso. |

Sistemas e editores atuais vêm configurados com a codificação UTF-8, que abrange a maioria dos idiomas existentes.

Unicode

Unicode é um padrão adotado mundialmente que possibilita com que todos os caracteres de todas as linguagens escritas utilizadas no planeta possam ser representados em computadores. A missão do Unicode é apresentada de forma clara no web site do Unicode Consortium18 (entidade responsável pela sua gestão):

Unicode fornece um número único para cada caractere,

* Não importa a plataforma,
* Não importa o programa,
* Não importa a linguagem.

O padrão Unicode é capaz de representar não somente as letras utilizadas pelas linguagens mais familiares para nós ocidentais, como Inglês, Espanhol, Francês e o nosso Português, mas também letras e símbolos utilizados em qualquer outra linguagem: Russo, Japonês, Chinês, Hebreu, etc. Além disso, inclui símbolos de pontuação, símbolos técnicos e outros caracteres que podem ser utilizados em texto escrito.

No padrão Unicode, cada diferente letra ou símbolo de cada alfabeto utilizado no mundo é mapeado para um diferente code point. O code point é um código no formato U + número em hexadecimal. O exemplo abaixo mostra os códigos das letras que compõem a palavra “BRASIL” (em maiúsculo).

B - U+0042; R - U+0052; A - U+0041; S - U+0053; I - U+0049; L - U+004C

É muito importante mencionar que as letras maiúsculas possuem code points diferentes das letras minúsculas. Por exemplo: o code point da letra “A” é U+0041, enquanto o da letra “a” é U+0061, o code point de “Ç” é U+00C7 e o de “ç” é U+00E7. Outra observação importante é que os primeiros 127 code points (até U+007F) são compatíveis com os códigos utilizados na antiga tabela ASCII, a primeira criada com objetivo de padronizar a codificação dos caracteres. Estes 127 code points representam, basicamente, os códigos associados aos números, letras maiúsculas e minúsculas sem acento e símbolos de pontuação mais comuns.

O aplicativo Mapa de caracteres (charmap) do Windows pode ser utilizado para consulta à tabela Unicode. Para acessá-lo, basta ir para o Prompt de Comando e digitar charmap. Na Figura 14, o Mapa de caracteres informa o code point associado à letra Á (“A” maiúsculo com acento agudo).



Encodings

Do que foi apresentado até agora, podemos entender que o Unicode nada mais é do que uma enorme tabela que associa um número único (code point) para cada diferente letra ou símbolo dos alfabetos de todo o mundo. Mas como esses code points podem ser armazenados em um arquivo texto ou na memória do computador? É aí que entram em cena os encodings.

Um encoding é um esquema de armazenamento dos code points dos caracteres que compõem as strings na memória do computador. Existem vários deles: UTF-8, ISO-8859-1 (apelidado de Latin-1), UCS-2, ANSI (ou Windows-1252), etc.

Cada encoding utiliza uma técnica distinta para lidar com os códigos Unicode19. O encoding UTF-8, por exemplo, é capaz de representar qualquer caractere Unicode. Para conseguir isso, utiliza uma técnica onde uma quantidade de 1 a 6 bytes pode ser utilizada para representar cada caractere. Ou seja, o UTF-8 não trabalha com uma representação em tamanho fixo. Os code points de 0 a 127 são armazenados com 1 byte. Porém, os demais podem ser armazenados em memória com tamanho de 2 a 6 bytes. O UTF-8 é completo (armazena qualquer caractere Unicode) e usa uma técnica de armazenamento que pode ser considerada “sofisticada”. Outros encodings, como ISO-8859-1 e ANSI são menos completos do que o UTF-8, e preferem trabalhar apenas com um subconjunto da tabela Unicode (ex: apenas com caracteres das linguagens ocidentais). Em compensação, podem fazer uso de técnicas de armazenamento mais simples que, além disso, conseguem representar strings gastando um número menor de bytes.

Mas por que é tão importante que um cientista de dados saiba o que é Unicode, encoding, padrão UTF-8, padrão Latin-1, padrão ANSI, etc., etc.? É porque em muitas situações práticas, torna-se necessário avisar ao Python (na verdade, a qualquer linguagem de programação) a codificação de um arquivo para que seja possível abri-lo. Melhor detalhando: quando um arquivo contém apenas os caracteres convencionais (code points de 0 a 127, ou seja, letras sem acento, números, sinais de pontuação mais comuns, tab, espaço em branco, “\n”, etc.) não importa a sua codificação, pois todos os encodings utilizam o mesmo esquema interno para armazená-los. Mas se o arquivo contém algum caractere especial, mesmo que seja uma inocente cedilha ou uma letra “a” acentuada, o Python não conseguirá processá-lo se você não informar o encoding correto, pois os diferentes encodings usam técnicas distintas para armazenar caracteres especiais.

Leitura de Arquivo Separado por Colunas – Tamanho Fixo.

O arquivo apresentado abaixo, armazena informações de duas variáveis: “n1”, uma variável numérica, que inicia na coluna 1 e possui tamanho fixo de 4 colunas; e “c1”, uma variável do tipo caractere, iniciando na coluna 5 e com tamanho fixo de 5 colunas.

1001aaaaa

1002bbbbb

1003ccccc

1004ddddd

1005eeeee

A seguir, apresenta-se um programa que abre o arquivo, realiza a leitura linha por linha (acesso sequencial) e captura as duas variáveis.

arquivo = open(**'nome\_errado.txt'**, **'r'**, encoding=**"utf8"**)  
**for** linha **in** arquivo:  
 n1 = linha[:4]  
 c1 = linha[4:]  
 print(n1, c1)  
**"""  
1001 aaaaa  
  
1002 bbbbb  
  
1003 ccccc  
  
1004 ddddd  
  
1005 eeeee  
"""**

Ao executar o exemplo, você deve ter percebido que uma linha em branco foi impressa entre cada resultado produzido por print(n1, c1). Isto ocorreu porque o último caractere de uma linha lida do arquivo é na verdade o caractere “\n”, ele é invisível para nós, mas existe no arquivo! Para evitar a impressão do “\n”, basta efetuar a seguinte modificação no programa:

c1 = linha[4:9]

Desta maneira, “c1” receberá a substring compreendida entre o quarto e o oitavo caractere da variável “linha” (não esqueça que o fatiamento sempre vai do primeiro índice até o último menos um).

Outra coisa importante de ser mencionada é que no acesso sequencial a gente só consegue “andar pra frente” no arquivo. Uma vez que uma linha seja alcançada, não há como voltar para a linha anterior … a não ser que você feche e abra o arquivo novamente, e implemente outro comando for para varrer o arquivo até chegar na linha desejada.

Leitura de Arquivo Separado por Colunas – Tamanho Variável.

Considere agora o arquivo que contém o código e o nome de alguns produtos. Apesar de se tratar de um arquivo separado por colunas, ele é diferente do arquivo mostrado na subseção anterior, pois o comprimento de cada uma de suas linhas é variável (os nomes de cada produto possuem comprimento diferente).

1001Leite

1002Biscoito

1003Café

1004Torradas

1005Chá

Para que o arquivo seja processado corretamente, o macete é utilizar a função len(), que retorna o comprimento de uma string (cuidado, pois esse tamanho inclui o chato do “\n”, uma vez que estamos lidando com linhas de arquivo).

c1 = linha[4:len(linha)-1]

Caso você esteja trabalhando com um arquivo não muito volumoso, poderá importá-lo inteiramente para a memória através do método read(), que é oferecido pelo objeto file handle. Quando um arquivo é lido dessa forma, todos os seus caracteres (incluindo todos os sem vergonhas dos “\n”!) são armazenados em um único “stringão”, como mostra o exemplo a seguir. Obviamente, esse processo não pode ser realizado para arquivos muito grandes.

conteudo = arquivo.read()  
print(conteudo)

Arquivos CSV

Também podemos realizar a leitura de arquivos CSV (comma-separated values – valores separados por vírgula) e de outros tipos de arquivos baseados em caracteres delimitadores utilizando o Python padrão, como veremos nos próximos exemplos.

O arquivo “CEPS.csv” armazena informações sobre faixas de CEPs utilizadas em estados da região Sudeste. O primeiro valor corresponde ao CEP inicial, o segundo é o CEP final, e o último o nome da UF. A primeira linha do arquivo contém o cabeçalho, ou seja, a descrição das variáveis.

cep\_ini, cep\_fim, nome\_uf

20000000,28999999,Rio de Janeiro

29000000,29999999,Espírito Santo

30000000,39999999,Minas Gerais

01000000,19999999,São Paulo

Para realizar leitura deste arquivo CSV separando as variáveis de forma correta, podemos fazer uso do método split(), disponibilizado automaticamente para qualquer variável do tipo string. Este método quebra uma string em uma lista de palavras, bastando para tal que o caractere delimitador seja passado como parâmetro. Um exemplo é apresentado no código a seguir, que abre o arquivo “CEPS.csv” para acesso sequencial, “pula” a linha de cabeçalho e imprime o CEP inicial e o CEP final de cada UF.

f = open(**'ceps.csv'**, **'r'**, encoding=**"utf8"**)  
aux = 0 *# auxiliar para permitir que cabeçalho seja ignorado***for** linha **in** f:  
 **if** (aux > 0): *# ignora a linha de cabeçalho* linha = linha[:len(linha) - 1] *# remove o tremendamente chato do "\n"* lstPalavras = linha.split(**","**)  
 cep\_ini = lstPalavras[0]  
 cep\_fim = lstPalavras[1]  
 uf = lstPalavras[2]  
 print(uf + **" -> CEPS de "** + cep\_ini + **" a "** + cep\_fim)  
 aux = aux + 1

Módulo CSV

Mostramos que é possível acessar arquivos CSV utilizando a função open() em conjunto com o método de string split(). No entanto, a standard library oferece o módulo ‘csv’, que tem a interessante capacidade de associar linhas de arquivos CSV a listas e dicionários.

A utilização do módulo csv será demonstrada através de exemplos que utilizam o arquivo “paises.csv”. Este arquivo armazena uma pequena base de dados que possui 10 observações e 5 variáveis: sigla do país, nome do país, continente (‘A’=América ou ‘E’=Europa), extensão territorial (em km2) e tamanho da população.

sigla;nome;continente;extensao;populacao

BRA;Brasil;A;8515767;204450649

CUB;Cuba;A;109890;11389562

FRA;França;E;549190;64395345

HUN;Hungria;E;93030;9855023

ITA;Itália;E;301340;59797685

MEX;México;A;1964380;127017224

NOR;Noruega;E;323780;5210967

PER;Peru;A;1285220;31376670

PRT;Portugal;E;92090;10349803

URY;Uruguai;A;176220;3431555

**import** csv  
**with** open(**'paises.csv'**,**'rt'**) **as** f:  
 meu\_csv = csv.reader(f, delimiter=**';'**)  
 **for** linha **in** meu\_csv:  
 print(linha)  
**"""  
['sigla', 'nome', 'continente', 'extensao', 'populacao']  
['BRA', 'Brasil', 'A', '8515767', '204450649']  
['CUB', 'Cuba', 'A', '109890', '11389562']  
['FRA', 'FranÃ§a', 'E', '549190', '64395345']  
['HUN', 'Hungria', 'E', '93030', '9855023']  
['ITA', 'ItÃ¡lia', 'E', '301340', '59797685']  
['MEX', 'MÃ©xico', 'A', '1964380', '127017224']  
['NOR', 'Noruega', 'E', '323780', '5210967']  
['PER', 'Peru', 'A', '1285220', '31376670']  
['PRT', 'Portugal', 'E', '92090', '10349803']  
['URY', 'Uruguai', 'A', '176220', '3431555']  
[]  
"""**

O programa possui apenas 5 linhas, que são explicadas a seguir:

* + import csv :

Importa o módulo ‘csv’.

* + with open('C:/CursoPython/paises.csv','rt') as f:

Abre o arquivo para leitura (parâmetro ‘rt’), associando-o ao file handle “f”.

* + meu\_csv = csv.reader(f, delimiter=';')

Cria o objeto “meu\_csv”, do tipo csv.reader. Esse tipo de objeto “entende” o que é um arquivo CSV e permite a iteração sobre as linhas do mesmo. Veja que dois parâmetros foram passados: o file handle (“f”) e o delimitador (“;”).

* + for linha in meu\_csv:

print(linha)

Varre todo o arquivo CSV de forma sequencial, da primeira à última linha.

Com o uso do pacote “csv”, cada linha do arquivo é carregada como uma lista em vez de uma string podendo escolher as variáveis de interesse. Isto facilita bastante o processamento do arquivo, pois cada variável fica associada a um índice específico. Em nosso exemplo, “sigla” fica com o índice 0, “nome” com o índice 1, etc., como mostra o programa abaixo, que imprime apenas a sigla e a população de cada país, além de não imprimir o cabeçalho.

**with** open(**'paises.csv'**, **'rt'**, encoding=**"utf8"**) **as** f:  
 meu\_csv = csv.reader(f, delimiter=**';'**)  
 i = 0;  
 **for** linha **in** meu\_csv:  
 **if** i > 0 **and** len(linha) > 3: *# para ignorar o cabeçalho* print(linha[0] + **' -> população = '** + linha[4])  
 i = i + 1  
**"""  
BRA -> população = 204450649  
CUB -> população = 11389562  
FRA -> população = 64395345  
HUN -> população = 9855023  
ITA -> população = 59797685  
MEX -> população = 127017224  
NOR -> população = 5210967  
PER -> população = 31376670  
PRT -> população = 10349803  
URY -> população = 3431555  
"""**

Com o uso do método DictReader() é possível estruturar cada linha de um arquivo CSV em um dicionário em vez de uma lista. Com o uso deste módulo:

* A linha de cabeçalho é interpretada e importada de maneira automática;
* Cada variável pode ser referenciada pelo seu nome e não pela sua posição.

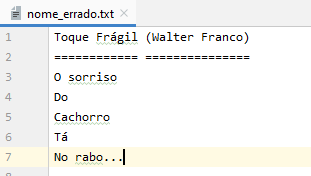
**with** open(**'paises.csv'**, **'rt'**, encoding=**"utf8"**) **as** f:  
 meu\_csv = csv.DictReader(f, delimiter=**';'**)  
 **for** linha **in** meu\_csv:  
 print(linha[**"sigla"**] + **' -> populacao = '** + linha[**"populacao"**])

Gravando Arquivos

Para gravar um arquivo, você deve abri-lo utilizando o modo “w” como segundo parâmetro da função open(). Quando a gravação terminar, é necessário utilizar a função close() para fechar o arquivo.

arquivo = open(**'nome\_errado.txt'**, **'w'**, encoding=**"utf8"**)  
msg1 = **"O sorriso\n"** + **"Do\n"** + **"Cachorro\n"** + **"Tá\n"** + **"No rabo...\n"**arquivo.write(**"Toque Frágil (Walter Franco)\n"**)  
arquivo.write(**"============ ==============="**)  
arquivo.write(**"\n"**)  
arquivo.write(msg1)  
arquivo.close()

Após a execução, o programa terá gerado um arquivo com seis linhas.



Lendo e gravando em um arquivo binário  
  
A função open () abre um arquivo em formato de texto por padrão. Para abrir um arquivo em formato binário, adicione 'b' ao parâmetro mode. Portanto, o modo "rb" abre o arquivo em formato binário para leitura, enquanto o modo "wb" abre o arquivo em formato binário para gravação. Diferentemente dos arquivos no modo de texto, os arquivos binários não são legíveis por humanos. Quando abertos usando qualquer editor de texto, os dados são irreconhecíveis.  
  
O código a seguir armazena uma lista de números em um arquivo binário. A lista é primeiro convertida em uma matriz de bytes antes da gravação. A função interna bytearray () retorna uma representação em bytes do objeto.

f=open("binfile.bin","wb")

num=[5, 10, 15, 20, 25]

arr=bytearray(num)

f.write(arr)

f.close()

Para ler o arquivo binário acima, a saída do método read () é convertida em uma lista usando a função list ().

f=open("binfile.bin","rb")

num=list(f.read())

print (num)

f.close()

DIRETÓRIOS

Módulo OS

O módulo OS é uma interface que permite interagir com o sistema operacional executando serviços ou operações básicas, sem a necessidade de comandos específicos das diferentes plataformas ou tipos de Sistema Operacional e suas muitas versões.

Um exemplo básico de aplicação é a definição e mudança do diretório de trabalho, ou ainda ações para criar, renomear ou apagar diretórios.

A maioria dos métodos apresentados a seguir utilizam o caminho completo para endereçar um arquivo ou mesmo o diretório, visando facilitar uma boa opção é definir o diretório de trabalho como sendo o local onde estão os arquivos a serem manipulados.

Criar um diretório de trabalho temporário para cada instância do programa rodando, também é uma boa prática pois arquivos temporários podem ser criados independente de quantas instâncias estejam rodando simultaneamente.

O método getcwd() não espera receber nenhum parâmetro e retorna uma string com a indicação do caminho completo do diretório de trabalho atualmente ativo, conforme as definições do sistema operacional corrente. Estas definições referentes as variáveis de ambiente do sistema operacional podem ser modificadas por meio do módulo SYS.

**import** os  
  
*# C:\Users\x\_kat\Documents\Projetos\ProjectTestEnv\09\_Arquivos\_Diretorios*print(os.getcwd())

Para mudar o diretorio de trabalho, deverá ser feito utilizando o método chdir passando como parâmetro uma string do novo caminho completo. Quando passamos ".." para chdir ou qualquer outra função que manipule arquivos e diretórios, estamos nos referindo ao diretorio pai ou de nivel supe­rior.

*# Repare que as barras tem que estar ao contrário   
# ou indicar r de raw string senão dá erro de escape  
# SyntaxError: (unicode error) 'unicodeescape'   
# codec can't decode bytes in position 2-3: truncated \UXXXXXXXX escape*os.chdir(**r"C:\Users\x\_kat\Desktop\Loterias"**)  
os.chdir(**"C:/Users/x\_kat/Desktop/Loterias"**)

A listagem do conteúdo de um diretório pode ser obtida por meio do método listdir que espera um caminho completo e devolve uma lista.

print(os.listdir(os.getcwd()))  
*# ['Pasta1', 'Pasta2', 'Pasta3']*

Para criar um novo diretório o módulo OS possui o método mkdir que funciona independente do sistema operacional. Porém caso seja necessário criar uma árvore de diretórios completa pode ser utilizado o método makedirs. Da mesma forma que existem métodos para criação de diretórios existem dois métodos semelhantes para apagar rmdir e removedirs, podem ser utilizados para apagar um único diretório e o segundo para toda a árvore.

**if** (os.path.exists(**"PastaTST"**)):  
 os.rmdir(**"PastaTST"**)  
**if** (os.path.exists(**"PastaTST\_new"**)):  
 os.rmdir(**"PastaTST\_new"**)  
**if** (os.path.exists(**"PastaTST1/SubTST/SubSubTST"**)):  
 os.removedirs(**"PastaTST1/SubTST/SubSubTST"**)  
  
os.mkdir(**"PastaTST"**)  
os.makedirs(**"PastaTST1/SubTST/SubSubTST"**)  
*# ['Pasta1', 'Pasta2', 'Pasta3', 'PastaTST1', 'PastaTST\_new']*print(os.listdir(os.getcwd()))

Você pode mudar o nome de um diretorio ou arquivo, ou seja, renomea-lo usando a função rename. A função rename também pode ser utilizada para mover arquivos, bastando especificar o mesmo nome em outro diretorio:

os.rename(**"PastaTST"**, **"PastaTST\_new"**)

Função os.walk

A função os.walk facilita a navegação em uma árvore de diretórios. Imagine que deseje percorrer todos os diretórios a partir de um diretório inicial, retornando o nome do diretório sendo visitado (raiz), os diretórios encontrados dentro do diretório sendo visitado (diretórios) e uma lista de seus arquivos (arquivos). Você deve executar essa função passando o diretório inicial a visitar a todos os subdiretorios recursivamente.

**import** os  
**import** sys

cur\_dir = sys.argv[1] **if** len(sys.argv) > 1 **else '.'  
for** raiz, diretorios, arquivos **in** os.walk(cur\_dir):  
 print(**"\nCaminho:"**, raiz)  
 **for** d **in** diretorios:  
 print(**f" {**d**}/"**)  
 **for** f **in** arquivos:  
 print(**f" {**f**}"**)  
 print(**f"{**len(diretorios)**} diretorio(s), {**len(arquivos)**} arquiwo(s)"**)

A grande vantagem da função os.walk é que ela visita automaticamente todos os subdiretorios dentro do diretório passado como parâmetro tazendo-o repetidamente até navegar a árvore de diretórios completa. Combinando a função os.walk as funções de manipulação de diretórios e arqui­vos que ja conhecemos, você pode escrever programas para manipular árvores de diretórios completas

módulo SHUTIL

Quando é necessário mover ou efetuar a cópia de um arquivo é recomendado utilizar o módulo shutil que possui as mesmas funcionalidades do módulo OS porém um conjunto de recursos extras que foram estendidos aos métodos padrão.

O método copyfile recebe como parâmetros o caminho completo do dado que será copiado seguido pelo caminho completo do arquivo destino. Todos as pastas do caminho devem existir para que a operação possa ser concluída com êxito.

**import** shutil  
**import** os  
  
shutil.copyfile(**"ceps.csv"**, **r"C:\Users\x\_kat\Desktop\Teste"**)  
print(os.listdir(os.getcwd()))

módulo GLOB

O módulo GLOB ajuda encontrar os arquivos dentro de um diretório por meio filtros com uso de caracteres curingas que podem ser utilizados para filtrar a lista a ser gerada. O método glob recebe o caminho completo e o conjunto de caracteres que será utilizado no filtro.

Dica: Utilizar o filtro entre asteriscos (ex: "modis") para filtrar todos os arquivos com a palavra modis, o métido é sensível a tipografia, ou seja, faz diferença entre maiúscula e minúscula.

**import** glob  
  
lst = glob.glob(**"\*ceps\*"**)  
print(lst) *# ['ceps.csv']*

Módulo path

Quando estamos depurando o programa ou preparando para que deixe de ser um script específico e ganhe a possibilidade de ser reutilizado em outros computadores, precisamos nos preocupar com os caminhos dos arquivos e diretórios. Para facilitar o desenvolvimento e deixar o código independente do Sistema Operacional o módulo path possui uma grande variedade de métodos para auxiliar nestas atividades.

Os problemas de endereçamento de arquivos e diretórios acontecem sempre quando trocamos entre sistema operacional, pois cada sistema tern suas próprias caracteristicas, como "/" no Linux e no Mac OS X para separar o nome dos diretorios e "\" no Windows. No Linux o endereço do arquivo pode ser representado pela string:

'/Users/fabianomorelli/Documents/dados\_exemplo/foc\_sev\_20161010.bin'

Este mesmo endereço no windows pode ser representado pela seguinte string: 'c:\fabianomorelli\Documents\dados\_exemplo\foc\_sev\_20161010.bin'

Note que existe uma diferença na direção da barra separadora das pastas além do fato que em windows temos a representação da unidade de disco que está sendo utilizada para armazenar o arquivo.

|  |  |
| --- | --- |
| **Método** | **Descrição** |
| abspath | Utiliza o caminho do diretório de trabalho para criar uma string com o caminho completo até o arquivo que foi referenciado. |
| basename | retorna apenas a ultima parte do caminho |
| dirname | retorna o caminho a esquerda da ultima barra |
| join | Junta dois ou mais partes de um caminho inserindo o separador quando necessário, pois nunca é adicionado o separador final, pois o método não diferencia se estamos informando o nome de um diretório ou de um arquivo. Para forçar a formação da string terminada com o separador o último elemento indicado deve ser uma string vazia. |
| split | Método que corta a cadeia de caracteres em duas partes. Normalmente a primeira parte representada pelo caminho com todos os diretórios e a segunda parte que representa o nome do arquivo ou a última parte da string sem separador. |
| splitext | É um método que também separa a cadeia de caracteres em duas partes, sendo a primeira parte o caminho completo e nome do arquivo, seguido pela extensão do arquivo na segunda parte. |
| splitdrive | No Windows, serve para separar a letra do drive do caminho em si. A função retorna uma tupla, em que a letra do drive é o primeiro elemento; e o restante do caminho, o segundo. |
| isdir | retornam True se o nome passado for um diretorio |
| isfile | retornam True se o nome passado for um arquivo |
| exists | verificar se um diretorio ou arquivo ja existe |
| getsize | retorna o tamanho do arquivo em bytes |
| getctime | retorna a data e hora de criacao |
| getmtime | retorna a data e hora de modificacao |
| getatime | retorna a data e hora de acesso |

**import** os  
  
*# C:\Users\x\_kat\Documents\Projetos\ProjectTestEnv\09\_Arquivos\_Diretorios\ceps.csv*print(os.path.abspath(**"ceps.csv"**))  
*# ProjectTestEnv*print(os.path.basename(**r'\Users\x\_kat\Documents\Projetos\ProjectTestEnv'**))  
*# \Users\x\_kat\Documents\Projetos*print(os.path.dirname(**r'\Users\x\_kat\Documents\Projetos\ProjectTestEnv'**))  
*# \Users\x\_kat\Documents\Projetos\ProjectTestEnv\XXXXX\YYYYY*print(os.path.join(**r'\Users\x\_kat\Documents\Projetos\ProjectTestEnv'**, **'XXXXX'**, **'YYYYY'**))  
*# ('\\Users\\x\_kat\\Documents\\Projetos\\ProjectTestEnv\\09\_Arquivos\_Diretorios', 'ceps.csv')*print(os.path.split(**r'\Users\x\_kat\Documents\Projetos\ProjectTestEnv\09\_Arquivos\_Diretorios\ceps.csv'**))  
*# ('\\Users\\x\_kat\\Documents\\Projetos\\ProjectTestEnv\\09\_Arquivos\_Diretorios\\ceps', '.csv')*print(os.path.splitext(**r'\Users\x\_kat\Documents\Projetos\ProjectTestEnv\09\_Arquivos\_Diretorios\ceps.csv'**))

help(os.path)

O modulo os.path traz várias outras funções que vamos utilizar para obter mais informações sobre os arquivos em disco. As duas primeiras sao isdir e isfile, que retornam True se o nome passado for um diretorio ou um arquivo

respectivamente:

print(os.path.isfile(**"ceps.csv"**)) *# True*print(os.path.isdir(**"ceps.csv"**)) *# False*print(os.path.isdir(**"09\_Arquivos\_Diretorios"**)) *# False*print(os.path.isdir(**r"\Users\x\_kat\Documents\Projetos"**)) *# True*

Execute o programa e veja que imprimimos apenas os nomes de diretorios e arquivos, sendo que adicionamos uma barra no final dos nomes de diretorio.

**import** os

**for** a **in** os.listdir(**"."**):  
 **if** os.path.isdir(a):  
 print(**f"{**a**}/"**)  
 **elif** os.path.isfile(a):   
 print(a)

Temos também outras funções que retornam mais informaçõees sobre arquivos e diretórios como seu tamanho e datas de modificação, criação e acesso:

**import** os  
**import** time

dir = **"ceps.csv"**print(**f'Nome: {**dir**}'**)  
print(**f'Tamanho: {**os.path.getsize(dir)**}'**) *# 153  
# Sun Mar 22 19:23:59 2020*print(**f'Criado: {**time.ctime(os.path.getctime(dir))**}'**)  
*# Sun Mar 22 19:24:11 2020*print(**f'Modificado: {**time.ctime(os.path.getmtime(dir))**}'**)  
*# Sun Mar 22 19:24:11 2020*print(**f'Acessado: {**time.ctime(os.path.getatime(dir))**}'**)

Observe que chamamos time.ctime para transformar a data e hora retornadas por getmtime, getatime e getctime em string. Isso é necessário porque o valor retornado é expresso em segundos e precisa ser corretamente convertido para ser exibido.

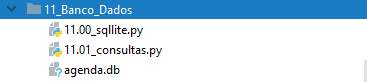
BANCO DE DADOS

SQLite

O SQLite é um gerenciador de banco de dados leve e completo, muito utilizado e presente mesmo em telefones celulares. Uma de suas principais características é não precisar de um servidor dedicado, sendo capaz de se iniciar a partir de seu programa. Vejamos um programa Python que cria um banco de dados, uma tabela e um registro:

**import** sqlite3  
  
conexao = sqlite3.connect(**"agenda.db"**)  
cursor = conexao.cursor()  
  
**try**:  
 cursor.execute(**'''  
 create table agenda(  
 nome text,  
 telefone text)  
 '''**)  
**except** sqlite3.OperationalError:  
 **None**cursor.execute(**'''  
insert into agenda (nome, telefone) values(?, ?)  
'''**, (**"Nilo"**, **"7788-1432"**))  
conexao.commit()  
cursor.close()  
conexao.close()

A primeira coisa a fazer e informar que ualizaremos um banco SQLite. Depois do import, várias funções e objetos que acessam o banco de dados se tornam disponiveis ao seu programa. Foi criado o banco de dados em sqlite3.connect("agenda.db"). A conexao com o banco de dados se assemelha a manipulação de um arquivo, é a operação análoga a abrir um arquivo. O nome do banco de dados que estamos criando será gravado no arquivo agenda.db.



A extensão .db é apenas uma convenção, mas é recomendado diferenciar o nome do arquivo de um arquivo normal, principalmente porque todos os seus dados serão guardados nesse arquivo. A grande vantagem de um banco de dados é que o registro de informações e toda a manutenção dos dados são feitos automaticamente para você com comandos SQL.

Cursores sao objetos utilizados para enviar coman­dos e receber resultados do banco de dados. Um cursor é criado para uma conexão, chamando-se o método cursor(). Uma vez que obtivemos um cursor, nós podemos enviar comandos ao banco de dados. O primeiro deles e criar uma tabela chamada agenda para guardar nomes e telefones.

Os seguintes métodos do objeto cursor são úteis.

|  |  |
| --- | --- |
| **Method** | **Description** |
| execute() | Executa a consulta SQL em um parâmetro de string |
| executemany() | Executa a consulta SQL usando um conjunto de parâmetros na lista de tuplas |
| fetchone() | Busca a próxima linha do conjunto de resultados da consulta. |
| fetchall() | Busca todas as linhas restantes do conjunto de resultados da consulta. |
| callproc() | Chama um procedimento armazenado. |
| close() | Fecha o objeto do cursor. |

O comando SQL usado para criar uma tabela é create table. Esse comando precisa do nome da tabela a criar; nesse exemplo, agenda e uma lista de campos entre parenteses. Nome e telefone sao nossos campos e text é o tipo. Embora em Python não precisemos declarar o tipo de uma variavel, a maioria dos bancos de dados exige um tipo para cada campo. No caso do SQLite, o tipo não é exigido, mas vamos continuar a usa-lo para que você não tenha problemas com outros bancos, e para que a noção de tipo comece a fazer sentido. Um campo do tipo text pode armazenar dados como uma string do Python.

Utilizamos o método execute de nosso cursor para enviar o comando ao banco de dados. Observe que escrevemos o comando em varias linhas, usando apostrofos triplos do Python. A linguagem SQL não exige essa formatação, embora ela deixe o comando mais claro e simples de entender. Voce poderia ter escrito tudo em uma so linha e mesmo utilizar uma string simples do Python.

O comando insert precisa do nome da tabela, na qual vamos inserir os dados, e também do nome dos campos e seus respectivos valores. into faz parte do comando insert e é escrito antes do nome da tabela. O nome dos campos é escrito logo a seguir, separados por virgula e, dessa vez, não precisamos mais informar o tipo dos campos, apenas a lista de nomes. Os valores que vamos inserir na tabela são especificados também entre parênteses, mas na segunda parte do comando insert que começa após a palavra values. Em nosso exemplo, a posição de cada valor foi marcada com interrogações, uma para cada campo. A ordem dos valores é a mesma dos campos; logo, a primeira interrogação se refere ao campo nome; a segunda, ao campo telefone.

A linguagem SQL permite que escrevamos os valores diretamente no comando, como uma grande string, mas esse tipo de sintaxe não é recomendado, por ser inseguro e facilmente utilizado para gerar um ataque de seguranca de dados chamado SQLInjection. Voce não precisa se preocupar com isso agora, principalmente porque, ao utilizarmos as interrogações, estamos utilizando parâmetros que evitam esse tipo de problema. Podemos entender as interrogações como um equivalente das másca­ras de string do Python, mas que utilizaremos com comandos SQL.

Utilizamos o método execute para executar o comando insert, mas, dessa vez, passamos os dados logo após o comando. No exemplo, "Nilo" e "7788-1432" vão substituir a primeira e a segunda interrogação quando o comando for executado. É importante notar que os dois valores foram passados como uma tupla. Uma vez que o comando é executado, os dados são enviados para o banco de dados, mas ainda não estão gravados definitivamente. Isso acontece, pois esta­mos usando uma transação. Transações serão apresentadas com mais detalhes em outra seção; por enquanto, considere o comando commit como parte das operações necessárias para modificar o banco de dados.

Antes de terminarmos o programa, fechamos (close) o cursor e a conexao com o banco de dados. Execute o programa e verifique se o arquivo agenda.db foi criado. Se você executar o programa uma segunda vez, um erro será gerado com a mensagem:

Traceback (most recent call last):

File " 11.00\_sqllite.py", line 10, in <module>

sqlite3.OperationalError: table agenda already exists

Esse erro acontece porque a tabela agenda já existe. Se você precisar executar o programa novamente, apague o arquivo agenda.db. Lembre-se de que todos os dados estão nesse arquivo e, ao apagá-lo, tudo é perdido. Você pode apagar esse arquivo sempre que quiser reinicializar o banco de dados.

Vejamos agora como ler os dados que gravamos no banco de dados, vamos fazer uma consulta (query). O programa a seguir realiza a consulta e mostra os resultados na tela, mas utilizando a clausula with. Uma das vantagens de utilizarmos with é que criamos um bloco onde um objeto é tido como válido. Se algo acontecer dentro do bloco, como uma exceção, a estrutura with garante que o método close será chamado. Na realidade, with chama o método \_\_exit\_\_ no fim do bloco e funciona muito bem com arquivos e conexões de banco de dados. Infelizmente, cursores não possuem o método \_\_exit\_\_, obrigando-nos a chamar manualmente o método close, ou a importar o módulo contextlib, que oferece a função closing que adapta um cursor com um método \_\_exit\_\_, que chama close.

**import** sqlite3  
  
**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 **with** closing(conexao.cursor()) **as** cursor:  
  
cursor.execute(**"select \* from agenda"**)  
 resultado = cursor.fetchone()  
 print(**f"Nome: {**resultado[0]**}\nTelefone: {**resultado[1]**}"**)

**"""  
Nome: Nilo  
Telefone: 7788-1432  
"""**

O comando select, em sua forma mais simples, utiliza uma lista de campos e uma lista de tabelas. Em nosso exemplo, a lista de campos foi substituida por \* (asterisco). O asterisco representa todos os campos da tabela sendo consultada, nesse caso nome e telefone. A palavra from é utilizada para separar a lista de cam­pos da lista de tabelas.

Para acessar os resultados do comando select, devemos utilizar o metodo fetchone de nosso cursor. Esse método retorna uma tupla com os resultados de nossa consulta ou None, caso a tabela esteja vazia.

A tupla retornada possui a mesma ordem dos campos de nossa consulta, nesse caso nome e telefone. Assim, resultado[0] é o carnpo nome, e resultado[1] é o campo telefone. Usamos uma f-string em Python e uma máscara que usa os campos da tupla resultado.

Vejamos agora como incluir os outros telefones de nossa agenda. O programa seguinte apresenta o metodo executemany. A principal diferença entre executemany e execute é que executemany trabalha com uma lista de tuplas chamada dados. Cada elemento da lista é uma tupla com dois valores, exatamente como fizemos no programa anterior.

dados = [(**"Joao"**, **"98901-0109"**),  
 (**"Andre"**, **"98902-8900"**),  
 (**"Maria"**, **"97891-3321"**)]  
  
cursor.executemany(**'''  
insert into agenda (nome, telefone) values(?, ?)  
'''**, dados)

Com os dados inseridos pelo programa, nossa agenda deve ter agora 4 registros. Vejamos como imprimir o conteiido de nossa tabela, usando o mesmo comando SQL, mas, dessa vez, utilizamos o metodo fetchall de nosso cursor para retornar uma lista com os resultados de nossa consulta. Fetchall retorna uma lista de tuplas. Cada elemento dessa lista é uma tupla contendo todos os campos retornados pela consulta. Uma vez que temos a lista resultado, utilizamos um simples for para trabalhar com cada registro.

resultados = cursor.fetchall()  
**for** registro **in** resultados:  
 print(**f"Nome: {**registro[0]**}\nTelefone: {**registro[1]**}"**)

O método fetchall retorna None caso o resultado da consulta seja vazio. Veremos isso em outros exemplos. Para consultas pequenas, contendo poucos registros como resultado, o método fetchall é muito interessante e fácil de utilizar. Para consultas maiores, em que mais de 100 registros são retornados, outros métodos de obter os resultados da consulta podem ser mais interessantes. Esses métodos evitam a criação de uma longa lista que pode ocupar uma grande quanridade de memória e demorar muito tempo para executar.

O programa seguinte mostra o método fetchone sendo unlizado dentro de uma estrutura de repetição while. Como não sabemos quantos registros serão retornados, utilizamos um while True, que é interrompido quando o método fetchone retorna None, significando que todos os resultados da consulta já foram obtidos. Você pode traduzir fetch como obter/trazer; logo, fetchone seria obter um resultado e fetchall obter todos os resultados. A vantagem de fetchone nesse caso é que imprimimos o resultado da consulta tão logo obtemos um e mantemos a impressão a medida que outros resultados forem chegando. Esse tempo de entrega é um conceito importante a perceber, uma vez que os dados vêm do banco de dados para o nosso programa. Essa transferência é controlada pelo banco de dados, responsável por executar nossa consulta e gerar os resultados.

**while True**:  
 resultado = cursor.fetchone()  
 **if** resultado **is None**:  
 **break** print(**f"Nome: {**resultado[0]**}\nTelefone: {**resultado[1]**}"**)

Consultando registros

Vamos utilizar uma variação do comando select para mostrar apenas alguns registros, implementando uma seleção de registros com base em uma pesquisa. Pesquisas em SQL são feitas com a cláusula where. Vejamos o comando SQL que seleciona todos os registros da agenda, cujo nome seja igual a "Nilo". Veja que apenas acrescentamos a cláusula where após o nome da tabela. O critério de seleção ou de pesquisa deve ser escrito como uma expressão, no caso nome = "Nilo".

cursor.execute(**"select \* from agenda where nome = 'Nilo'"**)

Veja que escrevemos 'Nilo' entre apóstrofos. Aqui, poderiamos trocar os apóstrofos por aspas ou ainda usar aspas triplas:

cursor.execute(**'select \* from agenda where nome = "Nilo"'**)  
cursor.execute(**"""select \* from agenda where nome = "Nilo" """**)

Agora imagine que invés do parâmetro de filtro “Nilo” fosse passado pelo usuário por uma função input o parâmetro: X" or "i"="1. O comando SQL resultante seria:

select \* from agenda where none = "X" or "1"="1"

Esse tipo de vulnerabilidade é um exemplo de SQLInjection, um tipo de erro que é muito grave e pode ficar muito tempo em nossos programas sem ser percebido. Os operadores relacionais “or”, “and” e “not” funcionam exatamente como em Python, e também pode ser usado em expressões SQL, podendo o valor digitado pelo usuário introduzir elementos que não desejamos. Para evitar esse tipo de ataque, sempre utilize parâmetros em consultas:

**import** sqlite3  
**from** contextlib **import** closing  
  
nome = input(**"Nome a selecionar: "**)  
**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 **with** closing(conexao.cursor()) **as** cursor:  
cursor.execute(**"select \* from agenda where nome = ?"**, (nome,))  
x = 0  
 **while True**:  
 resultado = cursor.fetchone()  
 **if** resultado **is None**:  
 **if** x == 0 :  
 print(**"Nada encontrado"**)  
 **break** print(**f"Nome: {**resultado[0]**}\nTelefone: {**resultado[1]**}"**)  
 x += 1

Agora utilizamos um parâmetro, como fizemos antes para inserir nossos regis­tros. Um detalhe importante é que escrevemos (nome,), repare a vírgula apos nome. Esse detalhe é importante, pois o segundo parâmetro do método execute é uma tupla, e, em Python, tuplas com apenas um elemento são escritas com uma vir­gula após o primeiro valor. Veja também que utilizamos a variável x para contar quantos resultados obtivemos. Como o método fetchone retorna None quando todos os registros foram recebidos, verificamos se x == 0, para saber se algo já havia sido obtido anteriormente ou se devemos imprimir uma mensagem dizendo que nada foi encontrado. Em Python, x == 0 produz o mesmo resultado de apenas x. Dessa forma, você poderia ter escrito a linha como if x: em vez de if x == 0. Isso acontece, pois, valores diferentes de 0 no Python são considerados como verdadeiros e None, 0, "", {}, [] como falsos.

Atualizando registros

Vamos alterar o registro com o telefone de "Nilo" para "12345-6789":

**import** sqlite3  
**from** contextlib **import** closing  
  
**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 **with** closing(conexao.cursor()) **as** cursor:  
 cursor.execute(**'update agenda set telefone = "12345-6789" where nome = "Nilo"'**)  
  
 print(**"Registros alterados: "**, cursor.rowcount)  
 **if** cursor.rowcount == 1:  
 conexao.commit()  
 cursor.execute(**'select \* from agenda where nome = "Nilo"'**)  
 resultado = cursor.fetchone()  
 print(**f"Nome: {**resultado[0]**}\nTelefone: {**resultado[1]**}"**)  
 **else**:  
 conexao.rollback()  
 print(**"Alteracoes abortadas"**)

A cláusula where funciona como no comando select, ou seja, ela avalia uma expressão lógica que, quando verdadeira, inclui o registro na lista de registros a modiflcar. A segunda parte do comando update é a cláusula set. Essa cláusula é usada para indicar o que fazer nos registros selecionados pela expressão do where. No exemplo, set telefone = "12345-6789" muda o conteúdo do campo telefone para "12345-6789". Não se esqueça de que, após modificar o banco de dados, precisamos chamar o método commit, como fizemos ao inserir os registros. Caso nos esqueçamos, as alterações serão perdidas.

A propriedade rowcount é muito interessante para confirmarmos o resultado de comandos de atualização, como update. Essa propriedade não funciona com select, retornando sempre -1. Utilizamos o valor de rowcount para decidir se as alterações deveriam ser registradas ou ignoradas. Como já sabemos, o método commit grava as alterações. O método rollback faz o inverso, abortando as alterações e deixando o banco de dados como antes. Os métodos commit e rollback fazem o controle de transa­cts do banco de dados. Podemos entender uma transação como um conjunto de operações que deve ser executado completamente. Isso significa operações que não fazem sentido, salvo se realizadas em um só grupo.

Apagando registros

O comando delete apaga registros com base em um critério de seleção, especificado na cláusula where que já conhecemos. Por exemplo, apague da tabela agenda todos os registros com nome igual a "Maria":

**import** sqlite3  
**from** contextlib **import** closing  
  
**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 **with** closing(conexao.cursor()) **as** cursor:  
 cursor.execute(**'delete from agenda where nome = "Maria"'**)  
  
 print(**"Registros alterados: "**, cursor.rowcount)  
 **if** cursor.rowcount == 1:  
 conexao.commit()  
 **else**:  
 conexao.rollback()  
 print(**"Alteracoes abortadas"**)  
  
Simplificando o acesso sem cursores

A interface de banco de dados do Python nos permite executar alguns comandos utilizando diretamente o objeto da conexão, sem criarmos explicitamente um cursor. Essa utilização simplificada funciona muito bem com SQLite, mas não faz parte da inter­face padrão de banco de dados do Python, a DB-API 2.0. Ao utilizar cursores, você obedece a DB-API 2.0 que é implementada por outros bancos de dados, simplificando a migração de seu código para outros bancos de dados, como o MySQL ou MariaDB.

**import** sqlite3  
  
**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 **for** registro **in** conexao.execute(**'select \* from agenda'**):  
 print(**f"Nome: {**registro[0]**}\nTelefone: {**registro[1]**}"**)

Acessando os campos como em um dicionário

Acessar os campos por posicjlo nem sempre e tao facil. Em Python, usando SQLite, podemos acessa-los pelo nome, adicionando a instrução conexao.row\_factory = sqlite3.Row

**with** sqlite3.connect(**"agenda.db"**) **as** conexao:  
 conexao.row\_factory = sqlite3.Row  
 **for** registro **in** conexao.execute(**'select \* from agenda'**):  
 print(**f"Nome: {**registro[**'NOME'**]**}\nTelefone: {**registro[**'telefone'**]**}"**)

Dessa forma, registro pode ser acessado como se fosse um dicionário, no qual o nome do campo e usado como chave. Outra facilidade que essa linha traz é que as chaves são aceitas independentemente se escrevermos o nome dos campos em maiúsculas ou minúsculas.

Gerando uma chave primária

Ate agora, trabalhamos apenas com campos normais, ou seja, campos que contém dados. Conforme nossas tabelas crescem, trabalhar com os dados pode não ser a melhor solução, e precisaremos acrescentar campos para identificar cada registro de maneira única. Uma alternativa oferecida pelo SQLite e a geração automática de chaves. Nesse caso, o banco se encarrega de criar números únicos para cada registro.

O programa a seguir cria o banco de dados brasil.db, a tabela esta­dos e também inclui o nome e a população de todos os estados brasileiros.

**import** sqlite3  
**from** contextlib **import** closing  
  
dados = [[**"São Paulo"**, 43663672], [**"Minas Gerais"**, 26593366],  
 [**"Rio de Janeiro"**, 16369178], [**"Bahia"**, 15044127],  
 [**"Rio Grande do Sul"**, 11164050], [**"Parana"**, 10997462],  
 [**"Pernanbuco"**, 9208511], [**"Ceara"**, 8778575],  
 [**"Pará"**, 7969655], [**"Maranhão"**, 6794298],  
 [**"Santa Catarina"**, 6634250], [**"Goias"**, 6434052],  
 [**"Paraiba"**, 3914418], [**"Espirito Santo"**, 3838363],  
 [**"Amazonas"**, 3807923], [**"Rio Grande do Norte"**, 3373969],  
 [**"Alagoas"**, 3300938], [**"Piauí"**, 3184165],  
 [**"Mato Grosso"**, 3182114], [**"Distrito Federal"**, 2789761],  
 [**"Mato Grosso do Sul"**, 2587267], [**"Sergipe"**, 2195662],  
 [**"Rondônia"**, 1728214], [**"Tocantins"**, 1478163],  
 [**"Acre"**, 776463], [**"Amapá"**, 734995], [**"Roraina"**, 488072]]  
  
**with** sqlite3.connect(**"brasil.db"**) **as** conexao:  
 conexao.row\_factory = sqlite3.Row  
 **with** closing(conexao.cursor()) **as** cursor:  
 **try**:  
 cursor.execute(**'''  
 create table estados(  
 id integer primary key autoincrement,   
 nome text,   
 populacao integer  
 )'''**)  
 **except** sqlite3.OperationalError:  
 **None** cursor.executemany(**'''  
 insert into estados(nome, populacao) values(?,?)  
 '''**, dados)  
  
 conexao.commit()

O valor do campo id sera gerado automaticamente. Uma vez que temos a popu­lação dos estados, vamos fazer uma consulta para listar os estados em ordem alfabetica. Ao executar o próximo programa, observe os valores gerados no campo id, no caso, valores numéricos de 1 a 27.

**import** sqlite3  
**from** contextlib **import** closing  
  
**with** sqlite3.connect(**"brasil.db"**) **as** conexao:  
 conexao.row\_factory = sqlite3.Row  
 print(**f"{'Id':3s} {'Estado':20s} {'População':12s}"**)  
 print(**"="** \* 37)  
 **for** estado **in** conexao.execute(**"select \* from estados order by nome"**):  
 print(**f"{**estado[**'id'**]**:3d} {**estado[**'nome'**]**:20s} {**estado[**'populacao'**]**:12d}"**)  
**"""  
Id Estado População   
=====================================  
 25 Acre 776463  
 17 Alagoas 3300938  
 26 Amapá 734995  
"""**

A grande diferença é que estamos utilizando a clausula order by para ordenar os resultados de nossa consulta; nesse caso, pelo campo nome. Podemos também modificar a ordenação para que os estados sejam impressos pela populacao, crescente ou decreste, usando a consulta:

select \* from estados order by populacao

select \* from estados order by populacao desc

Alterando a tabela

Vamos acrescentar mais alguns campos a nossa tabela de estados. Um campo para a região do Brasil e outro para a sigla do estado. Em SQL, o comando utilizado para alterar os campos de uma tabela e o alter table.

alter table estados add sigla text alter table estados add regiao text

O comando alter table do SQLite é limitado se comparado com outros bancos de dados. Em outros bancos, pode-se alterar vários campos com um só alter table, mas, no SQLite, somos obrigados a alterar um campo de cada vez. Por isso, planeje suas tabelas com cuidado e, caso precise realizar grandes mudanças, prefira criar uma outra tabela com as alterações e copiar os dados da tabela antiga. Execute o programa para alterar a tabela estados e adicionar os campos sigla e região:

**import** sqlite3  
  
**with** sqlite3.connect(**"brasil.db"**) **as** conexao:  
 conexao.execute(**"""alter table estados add sigla text"""**)  
 conexao.execute(**"""alter table estados add regiao text"""**)  
  
 dados = [[**"SP"**, **"SE"**, **"São Paulo"**], [**"MG"**, **"SE"**, **"Minas Gerais"**],  
 [**"PJ"**, **"SE"**, **"Rio de Janeiro"**], [**"BA"**, **"NE"**, **"Bahia"**],  
 [**"RS"**, **"S"**, **"Rio Grande do Sul"**], [**"PR"**, **"S"**, **"Parana"**],  
 [**"PE"**, **"NE"**, **"Pernanbuco"**], [**"CE"**, **"NE"**, **"Ceara"**],  
 [**"PA"**, **"N"**, **"Pará"**], [**"MA"**, **"NE"**, **"Maranhão"**],  
 [**"SC"**, **"S"**, **"Santa Catarina"**], [**"GO"**, **"CO"**, **"Goias"**],  
 [**"PB"**, **"NE"**, **"Paraiba"**], [**"ES"**, **"SE"**, **"Espirito Santo"**],  
 [**"AM"**, **"N"**, **"Amazonas"**], [**"RN"**, **"NE"**, **"Rio Grande do Norte"**],  
 [**"AL"**, **"NE"**, **"Alagoas"**], [**"PI"**, **"NE"**, **"Piauí"**],  
 [**"MT"**, **"CO"**, **"Mato Grosso"**], [**"DF"**, **"CO"**, **"Distrito Federal"**],  
 [**"MS"**, **"CO"**, **"Mato Grosso do Sul"**], [**"SE"**, **"NE"**, **"Sergipe"**],  
 [**"RO"**, **"N"**, **"Rondônia"**], [**"TO"**, **"N"**, **"Tocantins"**],  
 [**"AC"**, **"N"**, **"Acre"**], [**"AP"**, **"N"**, **"Amapá"**], [**"RR"**, **"N"**, **"Roraina"**]]  
  
 conexao. executemany(**"""update estados  
 set sigla = ?,  
 regiao = ?  
 where nome = ?"""**, dados)  
  
 conexao.commit()

Agora nosso banco de dados possui uma tabela estados com a população, sigla e região de cada estado. Esses novos campos permitirão utilizarmos funções de agregado da linguagem SQL: count, min. max, avg e sum

Agrupando dados

Vejamos como realizar um grupo simples e exibir quantos registros fazem parte desse grupo, usando a função count. A clausula SQL que indica agrupamento é group by, seguida do nome dos campos que compõem o grupo. Essa consulta com grupo só pode conter os campos utilizados para compor a chave do grupo e funções de agru­pamento de dados, como min (mínimo), max (máximo), avg (média), sum (soma) e count (contagem).

Um exemplo concreto como nosso banco de dados é agrupar os estados por regiao. Esse comando utiliza a cláusula group by regiao para especificar a chave de grupo. Dessa forma, todos os registros que pertencem a mesma regiao são agrupados. Assim, conseguimos calcular a população mínima, máxima, média e total de cada região e também para o Brasil.

Podemos tambem filtrar os resultados após o agrupamento, usando a clausula having. Para entender a diferença entre where e having, imagine que where é executada antes do agrupamento, selecionando os registros que farão parte do resul­tado. A clausula having avalia o resultado do agrupamento e decide quais farão parte do resultado final. Por exemplo, podemos escolher apenas as regioes com mais de 5 estados. Como a quantidade de estados por região so e conhecida após o agrupamento (group by), essa condição deve aparecer em uma cláusula having.

Veja que o programa completo utilizando having, retorna 2 registros, uma vez que apenas as regiões Norte N e Nordeste (NE) possuem mais de 5 estados.

**import** sqlite3  
print(**"Região Estados População Mínima Máxima Média Total(soma)"**)  
print(**"====== ======= ========= ========== ========= ==========="**)  
**with** sqlite3.connect(**"brasil.db"**) **as** conexao:  
 **for** regiao **in** conexao.execute(**"""  
 select regiao,   
 count(\*),   
 min(populacao),  
 max(populacao),   
 avg(populacao),   
 sum(populacao) as tpop   
 from estados group by regiao   
 having count(\*) > 5 order by tpop desc  
 """**):  
 print(**"{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:12,}"**.format(\*regiao))  
**"""  
Região Estados População Mínima Máxima Média Total(soma)  
====== ======= ========= ========== ========= ===========  
NE 9 2,195,662 15,044,127 6,199,407 55,794,663  
N 7 488,072 7,969,655 2,426,212 16,983,485  
"""**

Trabalhando com datas

Embora o SQLite trabalhe com datas, o tipo DATE não é suportado diretamente, gerando uma certa confusão entre datas e strings. Vamos criar uma tabela feriados e inserir algumas datas. Observe que escrevemos as datas no formato ISO 8601, ou seja: ANO-MES-DIA. Nesse formato, a data do Natal (25/12/2018) é escrita como 2018-12-25. Escrever as datas nesse formato é uma caracteristica do SQLite. Sempre escreva suas datas no formato ISO ao trabalhar com esse gerenciador de banco de dados. Observe que utilizamos o tipo date (data) na coluna data. Modifique o ano de 2018 para o ano corrente, caso necessário.

**import** sqlite3  
  
feriados = [[**"2018-01-01"**, **"Confraternizagao Universal"**],  
 [**"2018-04-21"**, **"Tiradentes"**],  
 [**"2018-05-01"**, **"Dia do trabalhador"**],  
 [**"2018-09-07"**, **"Independencia"**],  
 [**"2018-10-12"**, **"Padroeira do Brasil"**],  
 [**"2018-11-02"**, **"Finados"**],  
 [**"2018-11-15"**, **"Proclanacao da Repiiblica"**],  
 [**"2018-12-25"**, **"Natal"**]]  
**with** sqlite3.connect(**"feriados.db"**) **as** conexao:  
 conexao.execute(**"create table feriados("  
 "id integer primary key autoincrement, "  
 "data date, "  
 "descricao text)"**)  
  
 conexao.executemany(**"insert into feriados(data,descricao) values (?,?)"**, feriados)  
 conexao.commit()

**for** feriado **in** conexao.execute(**"select \* from feriados"**):  
 print(feriado)  
**"""  
(1, '2018-01-01', 'Confraternizagao Universal')  
(2, '2018-04-21', 'Tiradentes')  
(3, '2018-05-01', 'Dia do trabalhador')  
"""**

Acessamos o campo data como fazemos até então, sem algum procedimento especial. Veja que, ao imprimirmos a tupla feriado com o resultado de nossa seleção, o campo data foi impresso como uma string qualquer. Nada impede que utilizemos strings para representar datas, como fizemos ao criar o banco de dados, mas campos datas são mais interessantes, pois podemos facilmente consultar o dia da semana e também realizar operações com datas.

Vamos modificar nossa conexão com o SQLite de forma a solicitar o prccessamento dos tipos de campo em nossas consultas. Ao solicitarmos a conexão devemos passar detect\_types=sqlite3.PARSE\_DECLTYPES como parâmetro. Vejamos o resultado com essa modificação:

**(1, datetime.date(2018, 1, 1), 'Confraternizagao Universal')  
(2, datetime.date(2018, 4, 21), 'Tiradentes')  
(3, datetime.date(2018, 5, 1), 'Dia do trabalhador')**

Veja que os valores do campo data agora são objetos da classe datetime.date. Isso evita termos de fazer a conversão manualmente de string para datetime.date. Vejamos um pouco o que podemos fazer com os objetos do módulo datetime.

**import** sqlite3  
**import** datetime  
**from** dateutil.relativedelta **import** relativedelta  
  
**with** sqlite3.connect(**"feriados.db"**, detect\_types=sqlite3.PARSE\_DECLTYPES) **as** conexao:  
 **for** feriado **in** conexao.execute(**"select \* from feriados"**):  
 print(feriado)  
  
 hoje = datetime.date.today()  
 hoje = hoje - relativedelta(years=2)  
 hoje60dias = hoje + datetime.timedelta(days=60)  
 conexao.row\_factory = sqlite3.Row  
 **for** feriado **in** conexao.execute(**"select \* from feriados "  
 "where data >= ? and data <= ?"**, (hoje, hoje60dias)):  
 print(**f"{**feriado[**'data'**]**:%d/%m} {**feriado[**'descricao'**]**}"**)  
**"""  
21/04 Tiradentes  
01/05 Dia do trabalhador  
"""**

Voltamos a utilizar row\_factory para acessarmos os campos por nome, como em um dicionario. Foi utilizado com a máscara "%d/%m" para exi­bir apenas o dia e o mês. O programa utiliza objetos do modulo datetine, hoje é formado pela data atual (datetime.date.today()) e subtraiu-se 2 anos com relativedelta(years=2). Em hoje60dias, utilizamos um objeto do tipo datetime.timedelta para acrescentar 60 dias a data atual. Com esses dois objetos date, podemos utilizar a cláusula where do SQLite para selecionar os feriados entre hoje e hoje60dias. Os objetos das classes timedelta e datetime.datetime são bastante úteis, caso você precise realizar operações com datas e guardar a hora com a data (datetime).

Convertendo a agenda para utilizar um banco de dados

Converter a agenda para um banco de dados nos levara a enfrentar um problema de mapeamento entre objetos e os bancos de dados relacionais, como o SQLite. Um dos maiores problemas nesse tipo de mapeamento é manter os dados entre nosso programa e o banco de dados sincronizados. Existem bibliotecas inteiras escritas apenas para resolver esse tipo de problema, usando o que é chamado de Mapeamento Objeto Relacional (Object-Relational Mapping, ORM). Em nossa agenda, temos uma lista de registros, cada um com um nome e uma lista de telefones. Cada telefone possui um tipo pre-cadastrado.

Primeiro, vamos criar uma subclasse de ListaUnica para controlar os registios apagados de nossas listas. Depois, criaremos metodos em uma outra classe. cha-mada DBAgenda, responsavel por manter o banco de dados e executar as operações da agenda. Uma mudanca nessa nova versao da agenda e que canegamos o registro apenas quando precisamos carregar. Ao voltarmos ao menu principal, todas as mudanc.as ja estarao salvas no banco de dados. faTpnAi as opcoes Le e Grava inuteis.

**from** listaunica **import** ListaUnica  
**from** nome **import** Nome  
**from** tipo\_telefone **import** TipoTelefone  
**from** telefone **import** Telefone  
  
**class** DBListaUnica(ListaUnica):  
 **def** \_\_init\_\_(self, elem\_class):  
 super().\_\_init\_\_(elem\_class)  
 self.apagados = []  
  
 **def** remove(self, elem):  
 **if** elem.id **is not None**:  
 self.apagados.append(elem.id)  
 super().remove(elem)  
  
 **def** limpa(self):  
 self.apagados = []  
  
  
**class** DBNome(Nome):  
 **def** \_\_init\_\_(self, nome, id\_=**None**):  
 super().\_\_init\_\_(nome)  
 self.id = id\_  
  
**class** DBTipoTelefone(TipoTelefone):  
 **def** \_\_init\_\_(self, id\_, tipo):  
 super().\_\_init\_\_(tipo)  
 self.id = id\_  
  
**class** DBTelefone(Telefone):  
 **def** \_\_init\_\_(self, numero, tipo=**None**, id\_=**None**, id\_nome=**None**):  
 super().\_\_init\_\_(numero, tipo)  
 self.id = id\_  
 self.id\_nome = id\_nome  
  
**class** DBTelefones(DBListaUnica):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(DBTelefone)  
  
**class** DBTiposTelefone(ListaUnica):  
 **def** \_\_init\_\_(self):  
 super().\_\_init\_\_(DBTipoTelefone)  
  
**class** DBDadoAgenda:  
 **def** \_\_init\_\_(self, nome):  
 self.nome = nome  
 self.telefones = DBTelefones()  
  
 @property  
 **def** nome(self):  
 **return** self.\_\_nome  
  
 @nome.setter  
 **def** nome(self, valor):  
 **if not** isinstance(valor, DBNome):  
 **raise** TypeError(**"nome deve ser uma instância da classe DBNome"**)  
 self.\_\_nome = valor  
  
 **def** pesquisaTelefone(self, telefone):  
 posicao = self.telefones.pesquisa(DBTelefone(telefone))  
 **if** posicao == -1:  
 **return None  
 else**:  
 **return** self.telefones[posicao]

A classe DBListaUnica herda de nossa classe ListaUnica e sua principal função é manter uma lista de id apagados. Isso nos permitirá apagar os elementos de nossas listas e, numa fase seguinte, apaga-los do banco de dados. A classe DBListaUnica so pode trabalhar com classes que possuam um atributo id.

As classes DBNome e DBTelefone derivam de Nome e Telefone respectivamente. A princi­pal diferença é que agora elas incluem o atributo id. Veja que esse atributo é um parâmetro opcional, pois, ao criarmos nossos objetos, eles não terão ainda suas chaves primárias. Além disso, usaremos o lato de que objetos sem id provavelmente acabaram de ser criados e precisam ser inseridos no banco de dados. Já na classe DBDadosAgenda modihcamos o tipo da lista de telefones de Telefones para DBTelefones. A classe DBTelefones é uma derivação de DBListaUnica que aceita apenas ele­mentos do tipo DBTelefone. Fizemos o mesmo entre DBTipoTelefone e DBTiposTelefones.

Até agora, hzemos apenas a mudança dos tipos, em preparação para trabalhar com o banco de dados. A principal mudança foi o novo atributo id que acrescentamos em todas as nossas classes. E o valor desse campo que utilizaremos em nossas consultas, alteraçõs e removes.

O programa a seguir apresenta a classe DBAgenda, que substituira a classe Agenda. A classe DBAgenda mantem o banco de dados em sincronia com as classes e objetos em memória, sendo responsável por todas as operações com o banco. Esse tipo de construção em camadas evita termos de escrever o código de manipulação e criação do banco na classe AppAgenda.

**import** os  
**import** sqlite3  
**import** db\_listaunica **as** dbl  
  
BANCO = **"""  
create table tipos(id integer primary key autoincrement,  
 descricao text);   
create table nomes(id integer primary key autoincrement,  
 nome text);   
create table telefones(id integer primary key autoincrement,  
 id\_nome integer,  
 numero text,  
 id\_tipo integer);  
  
insert into tipos(descricao) values ("Celular");  
insert into tipos(descricao) values ("Fixo");  
insert into tipos(descricao) values ("Fax");  
insert into tipos(descricao) values ("Casa");  
insert into tipos(descricao) values ("Trabalho");  
"""  
  
class** DBAgenda:  
 **def** \_\_init\_\_(self, banco):  
 self.tiposTelefone = dbl.DBTiposTelefone()  
 self.banco = banco  
 novo = **not** os.path.isfile(banco)  
 self.conexao = sqlite3.connect(banco)  
 self.conexao.row\_factory = sqlite3.Row  
 **if** novo:  
 self.cria\_banco()  
 self.carregaTipos()  
  
 **def** carregaTipos(self):  
 **for** tipo **in** self.conexao.execute(**"select \* from tipos"**):  
 id\_ = tipo[**'id'**]  
 descricao = tipo[**"descricao"**]  
 self.tiposTelefone.adiciona(dbl.DBTipoTelefone(id\_, descricao))  
  
 **def** cria\_banco(self):  
 self.conexao.executescript(BANCO)  
  
 **def** pesquisaNome(self, nome):  
 **if not** isinstance(nome, dbl.DBNome):  
 **raise** TypeError(**"nome deve ser do tipo DBNome"**)  
 achado = self.conexao.execute(**"""select count(\*)  
 from nomes where nome = ?"""**,  
 (nome.nome,)).fetchone()  
 **if** achado[0] > 0:  
 **return** self.carrega\_por\_nome(nome)  
 **else**:  
 **return None  
  
 def** carrega\_por\_id(self, id):  
 consulta = self.conexao.execute(  
 **"select \* from nomes where id = ?"**, (id,))  
 **return** carrega(consulta.fetchone())  
  
 **def** carrega\_por\_nome(self, nome):  
 consulta = self.conexao.execute(  
 **"select \* from nomes where nome = ?"**, (nome.nome,))  
 **return** self.carrega(consulta.fetchone())  
  
 **def** carrega(self, consulta):  
 **if** consulta **is None**:  
 **return None** novo = dbl.DBDadoAgenda(dbl.DBNome(consulta[**"nome"**], consulta[**"id"**]))  
 **for** telefone **in** self.conexao.execute(  
 **"select \* from telefones where id\_nome = ?"**, (novo.nome.id,)):  
 ntel = dbl.DBTelefone(telefone[**"numero"**], **None**,  
 telefone[**"id"**], telefone[**"id\_nome"**])  
 **for** tipo **in** self.tiposTelefone:  
 **if** tipo.id == telefone[**"id\_tipo"**]:  
 ntel.tipo = tipo  
 **break** novo.telefones.adiciona(ntel)  
 **return** novo  
  
 **def** lista(self):  
 consulta = self.conexao.execute(**"select \* from nomes order by nome"**)  
 **for** registro **in** consulta:  
 **yield** self.carrega(registro)  
  
 **def** novo(self, registro):  
 **try**:  
 cur = self.conexao.cursor()  
 cur.execute(**"insert into nomes(nome) values (?)"**, (str(registro.nome),))  
 registro.nome.id = cur.lastrowid  
 **for** telefone **in** registro.telefones:  
 cur.execute(**"""insert into telefones(numero,  
 id\_tipo, id\_nome) values (?,?,?)"""**,  
 (telefone.numero, telefone.tipo.id, registro.nome.id))  
 telefone.id = cur.lastrowid  
 self.conexao.commit()  
 **except** Exception:  
 self.conexao.rollback()  
 **raise  
 finally**:  
 cur.close()  
  
 **def** atualiza(self, registro):  
 **try**:  
 cur = self.conexao.cursor()  
 cur.execute(**"update nomes set nome=? where id = ?"**,  
 (str(registro.nome), registro.nome.id))  
 **for** telefone **in** registro.telefones:  
 **if** telefone.id **is None**:  
 cur.execute(**"""insert into telefones(numero,  
 id\_tipo, id\_nome)  
 values (?,?,?)"""**,  
 (telefone.numero, telefone.tipo.id, registro.nome.id))  
 telefone.id = cur.lastrowid  
 **else**:  
 cur.execute(**"""update telefones set numero=?,  
 id\_tipo=?, id\_nome=?  
 where id = ?"""**,  
 (telefone.numero, telefone.tipo.id, registro.nome.id, telefone.id))  
 **for** apagado **in** registro.telefones.apagados:  
 cur.execute(**"delete from telefones where id = ?"**, (apagado,))  
 self.conexao.commit()  
 registro.telefones.limpa()  
 **except** Exception:  
 self.conexao.rollback()  
 **raise  
 finally**:  
 cur.close()  
  
 **def** apaga(self, registro):  
 **try**:  
 cur = self.conexao.cursor()  
 cur.execute(**"delete from telefones where id\_nome = ?"**, (registro.nome.id,))  
 cur.execute(**"delete from nomes where id = ?"**, (registro.nome.id,))  
 self.conexao.commit()  
 **except** Exception:  
 self.conexao.rollback()  
 **raise  
 finally**:  
 cur.close()

A primeira coisa que a classe DBAgenda faz e verificar se o banco de dados ja existe. Essa verificação é feita antes do pedido de conexão no método \_\_init\_\_.

Para saber se o banco já existe, utilizamos a função os.path.isfile(banco). Caso o arquivo não exista, ele será criado pelo método cria\_banco, chamado logo após a conexão com o banco de dados. Veja que guardamos o objeto conexão como um atributo de DBAgenda.

O método cria\_banco é muito simples, utilizando o método executescript da conexão. Esse método executa vários comandos de uma só vez. Para simplificar a criação do banco, escrevemos o código que cria todas as tabelas e popula a tabela tipos na variável global BANCO. A execução de vários comandos só é possivel porque eles estão separados por ;.

O método carrega\_tipos realiza a leitura de todos os tipos de telefone no banco de dados e os guarda na lista tiposTelefone. Observe o cuidado em manter os id. Esse valor será utilizado depois para obter o tipo correto de cada telefone.

O método pesquisanome também traz novidades. Nele, executamos uma consulta usando a função count(\*). Se um registro for encontrado, o método carrega\_por\_nome é chamado para transformar o resultado de nossa consulta em uma coleção de objetos. Veja tam­bém que utilizamos a cláusula where para pesquisar na tabela nomes diretamente, uma vez que não carregamos todos os registros do banco de dados para a memária.

No método carrega, o resultado de nossa consulta é transformado em um objeto DBDadoAgenda, após criarmos uma instância de DBNone com os campos nome e id vindos de nossa consulta. Esse tipo de acesso é possivel, pois registramos self.conexao.row\_factory = sqlite3.Row no método \_\_init\_\_. Uma vez que o nome foi carregado, utilizamos seu id como id\_nome em nossa próxima consulta, que carregara os valores de telefone.

Observe o cuidado durante a leitura dos telefones e a transformacao do resultado em DBTelefone. Como o tipo ainda não foi carregado e temos todos eles em memória, fazemos uma pesquisa em self.TiposTelefone para converter o id em uma instância de TipoTelefone. É esse tipo de mapeamento que não é tão simples de fazer e que é bastante trabalhoso pela grande quantidade de código necessária para mante-lo. E nessas horas que um ORM ajuda. Uma vez que tudo foi convertido, novo e retornado com todos os dados pre-carregados.

O método lista executa uma consulta total da tabela nomes. Porém, para evitar a criação de uma grande lista, utilizamos a instrução yield do Python que retorna cada valor carregado, um de cada vez. Na realidade. o método lista retorna um gerador que pode ser utilizado em um for do Python. Isso evita termos de carregar e converter todos os valores antes de ter os primeiros resultados na tela.

No método novo, convertemos um objeto do tipo DBDadoAgenda em registros das tabelas nomes e telefones. Essa conversão é realizada dentro de uma transação, dai o porque de utilizarmos explicitamente um cursor (cur). Realizar essa operação dentro de uma transaçãao permitira melhorar a consistência do banco de dados, pois, se um erro acontecer antes de completarmos todas as operações, o estado do banco de dados será revertido ao estado anterior ao inicio de nossas operações. Como o id de nomes é gerado automaticamente, veja que utilizamos a propriedade lastrowid de nosso cursor, logo após a execuçãoo do insert. Dessa forma, podemos utilizar o novo id para popular a tabela de telefones. Realizamos então o mesmo processo a cada novo telefone, atribuindo o valor de lastrowid ao id do telefone.

Já o método atualiza é bem mais complexo. Durante a atualização de um registro, precisamos atualizar o campo nome na tabela de nomes. Em nossa solução caseira, não temos como saber se nome foi alterado ou não. Poderiamos alterar a classe para saber quando o nome foi alterado e executar o update apenas quando necessário. Essa é outra caracteristica das bibliotecas de ORM que trazem esse tipo de funcionalidade em suas classes. Para mantermos a agenda o mais simples possivel, nome sempre será atualizado. Para cada telefone, fazemos a verificação do valor de telefone.id. Se um telefone ja possui um valor em id, provavelmente ele já está registrado no banco de dados e precisa ser atualizado. Caso ainda não possua um valor em id, provavelmente se trata de um telefone inserido durante a alteração. Dessa forma, escolhemos entre fazer um insert ou um update na tabela telefones.

Por último, verificamos se a lista de apagados possui uma lista de id a apagar. Essa lista foi construída pela classe DBListaUnica, em que guardamos o id de cada elemento apagado. Essa mudançaa foi necessária, pois, diferente dos novos regis­tros e dos registros alterados, os registros apagados são removidos de nossa lista. Se não mantivermos a lista dos id apagados, ficaremos sem saber quais telefo­nes foram removidos, e nosso banco ficara inconsistente. Para cada id na lista de apagados, executamos um delete. Logo após, terminamos a transação com um commit para nos assegurarmos de que todas as operações foram registradas no banco de dados e, so então, apagamos a lista de telefones removidos com o método limpa.

O método apaga é um pouco mais simples. Nele, utilizamos a mesma estrutura de proteção e uma transação. O interessante é que, primeiro, apagamos os tele­fones e, depois, os nomes. Da forma que geramos nosso banco de dados, essa ordem não importa, uma vez que não estamos utilizando os recursos de integridade referencial do banco.

O programa da agenda e apresentado por inteiro:

**import** sys  
**import** pickle  
**import** os.path  
**import** db\_agenda **as** DBA  
**import** db\_listaunica **as** DBL  
**from** agenda **import** Agenda  
**from** menu **import** Menu  
**from** nome **import** Nome  
**from** telefone **import** Telefone  
**from** dado\_agenda **import** DadoAgenda  
  
**class** AppAgenda:  
 @staticmethod  
 **def** pede\_nome():  
 **return** input(**"Nome: "**)  
  
 @staticmethod  
 **def** pede\_telefone():  
 **return** input(**"Telefone: "**)  
  
 @staticmethod  
 **def** mostra\_dados(dados):  
 print(**f"Nome: {**dados.nome**}"**)  
 **for** telefone **in** dados.telefones:  
 print(**f"Telefone: {**telefone**}"**)  
 print()  
  
 @staticmethod  
 **def** mostra\_dados\_telefone(dados):  
 print(**f"Nome: {**dados.nome**}"**)  
 **for** i, telefone **in** enumerate(dados.telefones):  
 print(**f"{**i**} - Telefone: {**telefone**}"**)  
 print()  
  
 @staticmethod  
 **def** pede\_nome\_arquivo():  
 **return** input(**"Nome do arquivo: "**)  
  
 **def** \_\_init\_\_(self, banco):  
 self.agenda = DBA.DBAgenda(banco)  
 self.menu = Menu()  
 self.menu.adicionaopcao(**"Novo"**, self.novo)  
 self.menu.adicionaopcao(**"Altera"**, self.altera)  
 self.menu.adicionaopcao(**"Apaga"**, self.apaga)  
 self.menu.adicionaopcao(**"Lista"**, self.lista)  
 self.ultimo\_nome = **None  
  
 def** pede\_tipo\_telefone(self, padrao=**None**):  
 **for** i, tipo **in** enumerate(self.agenda.tiposTelefone):  
 print(**f" {**i**} - {**tipo**} "**, end=**None**)  
 t = Menu.valida\_faixa\_inteiro(**"Tipo: "**, 0, len(self.agenda.tiposTelefone) - 1, padrao)  
 **return** self.agenda.tiposTelefone[t]  
  
 **def** pesquisa(self, nome):  
 **if** isinstance(nome, str):  
 nome = DBL.DBNome(nome)  
 dado = self.agenda.pesquisaNome(nome)  
 **return** dado  
  
 *# opção 1* **def** novo(self):  
 novo = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(novo):  
 **return** nome = DBL.DBNome(novo)  
 **if** self.pesquisa(nome) **is not None**:  
 print(**"Nome ja existe!"**)  
 **return** registro = DBL.DBDadoAgenda(nome)  
 self.menu\_telefones(registro)  
 self.agenda.novo(registro)  
  
 *# opção 2* **def** altera(self):  
 **if** self.conta\_lista() == 0:  
 print(**"Agenda vazia, nada a alterar"**)  
 **return** nome = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(nome):  
 **return** p = self.pesquisa(nome)  
 **if** p **is not None**:  
 print(**"\nEncontrado:\n"**)  
 AppAgenda.mostra\_dados(p)  
 print(**"Digite enter caso nao queira alterar o nome"**)  
 novo = AppAgenda.pede\_nome()  
 **if not** Menu.nulo\_ou\_vazio(novo):  
 p.nome.nome = novo  
 self.menu\_telefones(p)  
 **else**:  
 print(**"Nome não encontrado!"**)  
  
 *# opção 3* **def** apaga(self):  
 **if** self.conta\_lista() == 0:  
 print(**"Agenda vazia, nada a apagar"**)  
 **return** nome = AppAgenda.pede\_nome()  
 **if** Menu.nulo\_ou\_vazio(nome):  
 **return** p = self.pesquisa(nome)  
 **if** p **is not None**:  
 print(p)  
 self.agenda.apaga(p)  
 print(**f"Apagado. A agenda agora possui apenas: {**self.conta\_lista()**} registros"**)  
 **else**:  
 print(**"Nome não encontrado."**)  
  
 **def** conta\_lista(self):  
 cont = 0  
 **for** e **in** self.agenda.lista():  
 cont = cont +1  
 **return** cont  
  
 *# opção 4* **def** lista(self):  
 print(**"\nAgenda"**)  
 print(**"-"** \* 60)  
 **for** e **in** self.agenda.lista():  
 AppAgenda.mostra\_dados(e)  
 print(**"-"** \* 60)  
  
 **def** menu\_telefones(self, dados):  
 **while True**:  
 print(**"\nEditando telefones\n"**)  
 AppAgenda.mostra\_dados\_telefone(dados)  
 **if** len(dados.telefones) > 0:  
 print(**"\n[A] - alterar\n[D] - apagar\n"**, end=**""**)  
 print(**"[N] - novo\n[S] - sair\n"**)  
 operacao = input(**"Escolha uma operação: "**)  
 operacao = operacao.lower()  
 **if** operacao **not in** [**"a"**, **"d"**, **"n"**, **"s"**]:  
 print(**"Operação inválida. Digite A, D, N ou S"**)  
 **continue  
 if** operacao == **'a' and** len(dados.telefones) > 0:  
 self.altera\_telefones(dados)  
 **elif** operacao == **'d' and** len(dados.telefones) > 0:  
 self.apaga\_telefone(dados)  
 **elif** operacao == **'n'**:  
 self.novo\_telefone(dados)  
 **elif** operacao == **"s"**:  
 **break** *# opção N* **def** novo\_telefone(self, dados):  
 telefone = AppAgenda.pede\_telefone()  
 **if** Menu.nulo\_ou\_vazio(telefone):  
 **return  
 if** dados.pesquisaTelefone(telefone) **is not None**:  
 print(**"Telefone ja existe"**)  
 tipo = self.pede\_tipo\_telefone()  
 dados.telefones.adiciona(DBL.DBTelefone(telefone, tipo))  
  
 *# opção A* **def** altera\_telefones(self, dados):  
 t = Menu.valida\_faixa\_inteiro\_ou\_branco(**"Digite a posição do "  
 "número a alterar, enter para sair: "**, 0, len(dados.telefones) - 1)  
 **if** t **is** Nome:  
 **return** telefone = dados.telefones[t]  
 print(**f"Telefone: {**telefone**}"**)  
 print(**"Digite enter caso não queira alterar o número"**)  
 novotelefone = AppAgenda.pede\_telefone()  
 **if not** Menu.nulo\_ou\_vazio(novotelefone):  
 telefone.nunero = novotelefone  
 print(**"Digite enter caso não queira alterar o tipo"**)  
 telefone.tipo = self.pede\_tipo\_telefone(self.agenda.tiposTelefone.pesquisa(telefone.tipo))  
  
 *# opção D* **def** apaga\_telefone(self, dados):  
 t = Menu.valida\_faixa\_inteiro\_ou\_branco(**"Digite a posição do "  
 "número a apagar, enter para sair: "**, 0, len(dados.telefones) - 1)  
 **if** t **is None**:  
 **return** dados.telefones.remove(dados.telefones[t])  
  
 **def** execute(self):  
 self.menu.execute()  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 app = AppAgenda(**"agenda\_sqlite.db"**)  
 **if** len(sys.argv) > 1:  
 app.le(sys.argv[1])  
 app.execute()

Na classe AppAgenda, modificamos as opções do menu e os tipos usados nas pesquisas. Veja que, com a utilização da classe DBAgenda, conseguimos isolar as operações de banco de dados da classe AppAgenda. Como não temos operações de leitura e gravação, o nome do banco de dados deve ser passado obrigatoriamente no código comando, no caso definiu-se o nome para agenda\_sqlite.db.