



Python

Módulo Avançado



Este material foi produzido como parte do projeto de colaboração entre a empresa Huawei Brasil e o Centro Estadual de Educação Tecnológica Paula Souza, representado pela Fatec de Sorocaba e Fatec de Jundiaí

- 2024 -

Sumário

Capítulo 14 Exceções - Aprofundamento	1
Capítulo 15 Expressões <code>yield</code> e Funções Geradoras	8
15.1 Função Geradora - Conceito	8
15.2 Funções geradoras e o uso de <code>yield</code> como comando	9
15.2.1 O que é <code>yield</code> ?	9
15.2.2 A função <code>next()</code>	11
15.3 Exemplos de aplicação.....	12
15.3.1 Criação de sequências infinitas.....	12
15.3.2 Filtragem de dados	14
15.3.3 Processamento intenso de dados.....	15
15.3.4 Leitura de grandes arquivos	16
15.4 Construindo Geradores com Expressões Geradoras	16
15.5 Usando <code>yield</code> como expressão.....	19
15.5.1 Método <code>.send()</code> de um gerador.....	19
15.5.2 Métodos <code>.throw()</code> e <code>.close()</code> de um gerador.....	22
Capítulo 16 Módulos e Pacotes	29
16.1 Conceito de Módulo	29
16.1.1 Vantagens no uso de Módulos	29
16.2 Conceitos de Pacote e Biblioteca	30
16.3 Criação de um Módulo	31
16.3.1 Uso interativo do módulo <code>utilidades.py</code> no <code>Idle</code>	32
16.4 Caminho de pesquisa dos Módulos.....	33
16.5 Visibilidade e Organização do código Python.....	35
16.5.1 Motivação	35
16.5.2 Namespace	35
16.5.3 Escopo e Tabelas de símbolos	43
16.6 Detalhes sobre o comando <code>import</code>	45
16.6.1 Comando <code>import</code> – forma 1.....	45
16.6.2 Comando <code>import</code> – Cuidado importante: evite colisão de nomes	46
16.6.3 Comando <code>import</code> – forma 1 generalizada	48
16.6.4 Comando <code>import</code> – forma 1 com apelido	48
16.6.5 Comando <code>import</code> – forma 2.....	49
16.6.6 Comando <code>import</code> – forma 2 com o coringa *	50

16.6.7 Comando <code>import</code> – forma 2 com apelido	51
16.7 Pacotes	51
16.7.1 Importação de módulo em pacote com apelido	53
16.8 Execução de um módulo como um script	54
Capítulo 17 Programação Orientada a Objetos	56
17.1 Orientação a objetos com Python	56
17.1.1 Conceito.....	56
17.1.2 Métodos Mágicos	57
17.1.3 Como definir uma classe em Python	58
17.1.4 Construtor de uma classe	59
17.1.5 Métodos existentes na classe.....	60
17.2 Combinação de classes com listas	62
17.3 Combinação de classes com dicionários.....	64
17.4 Quando usar classes? E quando não usar?.....	65
17.5 Membros públicos e não-públicos	66
17.6 Deformação de nomes (<i>name mangling</i>).....	67
17.7 Dicionário de Atributos.....	68
17.8 Modo de declaração de atributos em classes	70
17.9 Atributos de Instância e Atributos de Classe.....	71
17.9.1 Atributos de Instância.....	72
17.9.2 Atributos de Classe	74
17.10 Atributos e métodos dinâmicos	76
17.11 Atributos de classe, de instância e dinâmicos - síntese	77
17.12 Atributos gerenciados - Propriedades.....	78
17.12.1 Criação do setter – decorator <code>@attr_name.setter</code>	79
17.12.2 Criação do getter – decorator <code>@property</code>	79
17.13 Métodos Mágicos	80
17.14 Descritores.....	83
17.15 Herança e Hierarquia de classes.....	86
Capítulo 18 Uso de Python com SQLite 3	90
18.1 Sistemas Gerenciadores de Bancos de Dados	90
18.2 Bancos de dados relacionais.....	91
18.3 A linguagem SQL.....	92
18.4 Banco de dados SQLite	93
18.5 IDE para SQLite	93
18.6 Python com SQLite	94
18.6.1 Primeiro Banco de Dados	95
18.6.2 Síntese do primeiro exemplo.....	97

18.7 Gerando um banco de dados mais extenso	97
18.7.1 Comandos SQL parametrizados.....	99
18.8 Exclusão de tabela	100
18.9 Como acessar os dados inseridos usando Python.....	101
18.10 Inserção de novos registros a partir da leitura de dados via teclado.....	102
18.11 Filtros e ordenação de registros	104
18.12 Alteração de tabelas e atualização de dados	104
18.13 Atualização de dados a partir de um arquivo.....	106
18.14 Exclusão de registros de uma tabela	108

Capítulo 14

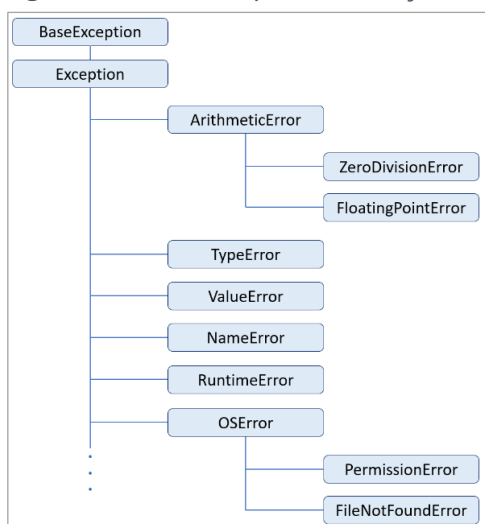
EXCEÇÕES - APROFUNDAMENTO

Exceções em Python são um mecanismo de tratamento de erros que podem ocorrer em um programa. Sobre esse assunto existem dois aspectos a serem considerados:

- Tratamento de exceção – é o que consideramos no nosso programa quando existe a possibilidade de ocorrência de erros em um trecho de código e que precisamos tratar para que o nosso programa não tenha sua execução interrompida. Este aspecto foi tratado no capítulo 6 do módulo básico;
- Levantamento de exceção – é o que precisamos implementar quando estamos escrevendo um código, uma situação de erro pode ocorrer e queremos gerar uma exceção para ser tratada em outro ponto do programa. Este é o assunto deste capítulo.

Tecnicamente falando, uma exceção é um objeto da classe `Exception`, ou de uma de suas classes herdeiras. Elas permitem rastrear informações sobre situações excepcionais e erros que venham a ocorrer durante a execução de algum código. O Python prevê uma série de exceções pré-definidas conhecidas pelo termo "exceções embutidas". A figura 14.1 mostra uma parte das exceções embutidas.

Figura 14.1 – Hierarquia de Exceções de Python



fonte: o Autor

A referência para exceções na documentação de Python pode ser acessada com este link
<https://docs.python.org/pt-br/3/library/exceptions.html>

Nesta documentação você encontrará todas as Exceções Embutidas previstas em Python

Já vimos como usar o comando `try-except` para tratar a ocorrência de uma exceção. Agora vamos aprender a criar exceções no nosso código. Para isso vamos utilizar como ponto de partida o programa resolvido 12.7 (apresentado no capítulo 12), no qual criamos uma função para verificar se um número é primo ou não. Ela é reproduzida a seguir.

Exercício Resolvido 12.7

reprodução

Enunciado: *Escreva um programa que verifique se um número inteiro lido é primo. Lembrando: um número primo é divisível apenas por 1 e por ele mesmo. A verificação do primo deve ser feita dentro de uma função.*

```
def Primo(V):
    '''Se V for primo retorna True, senão retorna False'''
    if V == 2:          # V é 2, portanto é primo
        return True
    elif V % 2 == 0:    # V é par maior que 2, portanto não é primo
        return False
    else:               # testa se V ímpar é primo
        raiz = pow(V, 0.5) # a raiz de V é o limite dos testes necessários
        i = 3
        while i <= raiz:
            if V % i == 0:
                return False # se for divisível retorna falso imediatamente
            i += 2
        return True # se chegar no final do laço então é primo

N = int(input('Digite um inteiro: '))
if Primo(N):
    print(f'{N} é primo')
else:
    print(f'{N} não é primo')

(reprodução da função apresentada no capítulo 12 sobre funções)
```

O motivo de reapresentá-la é destacar uma eventual falha quanto à forma de tratamento do parâmetro `V` recebido. A função é capaz de processar valores de `V` maiores ou iguais a 2. Qualquer valor de `V` menor que 2 não é levado em consideração, e isso pode representar um problema.

Imagine que a função tenha sido desenvolvida tempos atrás e por algum programador que não está mais na empresa e novos programadores precisem utilizá-la. Inadvertidamente esses novos programadores podem passar algum valor menor que 2 ao fazer a chamada da função e isso vai gerar um retorno `True`. Ou ainda pior, pode ocorrer de alguém tentar passar à função um número real, um string, ou qualquer outra coisa inválida para efeitos de verificação de números primos.

Uma função de verificação de números primos é um exemplo muito simples, mas se em seu lugar imaginarmos funções que realizem tarefas críticas em um software, fica clara a importância de se levantar uma exceção nos casos de falhas potenciais.

Levantar uma Exceção
é a ação de gerar uma exceção em um ponto do programa,
para ser tratada em outro ponto.

No exemplo 14.1 mostramos um caso concreto de levantamento de exceção aplicado a uma função que recebe um número inteiro e retorna se ele é par ou ímpar. A forma como a função e a parte principal do programa estão escritas propicia a ocorrência de erros no caso de o parâmetro passado não ser um número inteiro.

Exemplo 14.1



```
def Paridade(pValor: int) -> str: # usamos anotações para indicar os tipos
    if pValor % 2 == 0:           # esperados, mas as anotações não interferem
        return 'PAR'              # na execução da função
    else:
        return 'ÍMPAR'

n = input('Digite algo: ') # nesta linha não fizemos a conversão para int
r = Paridade(n)            # então o uso da função Paridade vai gerar um erro
print(f'{n} é {r}')
```

Digite algo: texto

```
Traceback (most recent call last):
  File "D:\Python\cap14_exemplo_14.1.py", line 8, in <module>
    r = Paridade(n)
    ~~~~~^~~~~
  File "D:\Python\cap14_exemplo_14.1.py", line 2, in Paridade
    if pValor % 2 == 0:
    ~~~~~^~~~~
TypeError: not all arguments converted during string formatting
```

Perceba acima que em função de passar um string para a função `Paridade()`, o interpretador Python levantou uma exceção `TypeError` que não foi tratada na parte principal. Esse programa pode ficar bem melhor se forem feitas as alterações mostradas a seguir:

Exemplo 14.1

solução reformulada

```
def Paridade(pValor: int) -> str:
    if type(pValor) != int:
        raise Exception('A função Paridade deve receber um int')
    if pValor % 2 == 0:
        return 'PAR'
    else:
        return 'ÍMPAR'

n = input('Digite algo: ')
r = Paridade(n)
print(f'{n} é {r}')
```

Digite algo: texto

```
Traceback (most recent call last):
  File "D:\Python\.venv\cap14_exemplo_14.1.py", line 10, in <module>
    r = Paridade(n)
    ~~~~~^~~~~
  File "D:\Python\.venv\cap14_exemplo_14.1.py", line 3, in Paridade
    raise Exception('A função Paridade deve receber um int')
Exception: A função Paridade deve receber um int
```

Nesta reformulação do exemplo 14.1 foram introduzidas no código da função estas linhas:

```
if type(pValor) != int:
    raise Exception('A função Paridade deve receber um int')
```

que são responsáveis pela geração de uma exceção da classe `Exception`. Ao executar a chamada da função passando como parâmetro qualquer objeto de classe diferente de `int` a exceção será levantada

com o uso do comando `raise`. Alternativamente, poderíamos substituir a exceção da classe `Exception`, por outra exceção mais específica, como a da classe `TypeError`, desta forma.

```
if type(pValor) != int:
    raise TypeError('A função Paridade deve receber um int')
```

Neste caso, o erro reportado será como abaixo.

```
Digite algo: texto
Traceback (most recent call last):
  File "D:\Python\.venv\cap14_exemplo_14.1.py", line 10, in <module>
    r = Paridade(n)
    ^^^^^^^^^^^
  File "D:\Python\.venv\cap14_exemplo_14.1.py", line 3, in Paridade
    raise TypeError('A função Paridade deve receber um int')
TypeError: A função Paridade deve receber um int
```

A palavra-chave `raise` do Python é usada para levantar as exceções. O comando `raise` gera uma indicação de erro e interrompe o fluxo natural de execução do programa, desviando-o para o manipulador de exceções do interpretador.

O uso do `raise` permite ao programador que personalize as verificações de situações de erro em potencial e antecipando-as para que não ocorram.

Agora vamos fazer uma revisão do exercício resolvido 12.7, com a inclusão de um tratamento de erro para situações em que o parâmetro `V` seja menor que 2.

Exercício Resolvido 14.1



Enunciado: Escreva um programa que verifique se um número inteiro lido é primo. Faça isso usando uma função e utilize exceções para controlar os erros.

O parâmetro da função deve ser um número inteiro maior ou igual a 2.

Se esse parâmetro for menor que dois use a exceção `ValueError`

Se esse parâmetro for de classe diferente de número inteiro use a exceção `TypeError`

```
def Primo(V):
    '''V deve ser um inteiro maior ou igual a 2.
    Se V for primo retorna True, senão retorna False'''
    if type(V) != int:
        raise TypeError('Tipo incorreto. V deve ser <int>')
    if V < 2:
        raise ValueError('Valor inválido. V deve ser maior que 1')
    if V == 2:
        # V é 2, portanto é primo
        return True
    elif V % 2 == 0:
        # V é par maior que 2, portanto é não primo
        return False
    else:
        # testa se V ímpar é primo
        raiz = pow(V, 0.5) # a raiz de V é o limite dos testes necessários
        i = 3
        while i <= raiz:
            if V % i == 0:
                return False # se for divisível retorna falso imediatamente
            i += 2
        return True # se chegar no final do laço então é primo
```

(implemente esta solução no problema 12.7 e faça seus testes)

No próximo exercício resolvido queremos implementar uma função que recebe uma lista ou uma tupla e retorna uma lista com os valores da lista recebida elevados ao quadrado.

Exercício Resolvido 14.2



Enunciado: Escreva uma função que recebe uma lista ou tupla com valores numéricos e retorne uma lista contendo o quadrado dos valores recebidos.

Utilize exceções para as situações de possível erro.

```
>>> def AoQuadrado(dados):
...     if not isinstance(dados, list) and not isinstance(dados, tuple):
...         raise TypeError(f'Lista ou Tupla esperados e você usou {type(dados)}')
...     if not all(isinstance(x, int) or isinstance(x, float) for x in dados):
...         raise ValueError(f'Os elementos de dados devem ser numéricos')
...     return [v**2 for v in dados]

>>> # Primeiro teste: o parâmetro não é lista ou tupla
>>> AoQuadrado('texto')
Traceback (most recent call last):
  File "<pyshell#97>", line 1, in <module>
    AoQuadrado('texto')
  File "<pyshell#86>", line 3, in AoQuadrado
    raise TypeError(f'Lista ou Tupla esperados e você usou {type(dados)}')
TypeError: Lista ou Tupla esperados e você usou <class 'str'>

>>> # Segundo teste: o parâmetro é lista ou tupla, mas há um elemento não numérico
>>> L = [2, 5, 8, 12, 15, 'texto']
>>> AoQuadrado(L)
Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    AoQuadrado(L)
  File "<pyshell#86>", line 5, in AoQuadrado
    raise ValueError(f'Os elementos de dados devem ser numéricos')
ValueError: Os elementos de dados devem ser numéricos

>>> # Terceiro teste: o parâmetro é lista com todos os dados numéricos
>>> L = [2, 5.3, 6.25, 8, 12, 15]
>>> AoQuadrado(L)
[4, 28.09, 39.0625, 64, 144, 225]

>>> # Quarto teste: o parâmetro é tupla com todos os dados numéricos
>>> T = (2, 5.3, 6.25, 8, 12, 15)
>>> AoQuadrado(T)
[4, 28.09, 39.0625, 64, 144, 225]
```

(exemplo interativo feito com IDE Idle)

Neste exercício utilizamos o método `.isinstance()` para verificar se o parâmetro `dados` é uma lista ou uma tupla. Caso não seja levantamos uma exceção `TypeError`. Se passar neste teste, na sequência verificamos todos os elementos da sequência `dados` para saber se todos são inteiros ou reais, caso contrário levantamos um `ValueError`.

Para a verificação de todos os elementos da lista foi utilizada a função `all()`, nesta construção:

```
all(isinstance(x, int) or isinstance(x, float) for x in dados)
```

Esta função aplica o teste duplo `isinstance(x, int) or isinstance(x, float)` a cada elemento da sequência `dados`. E se todos passarem nesse teste o retorno é `True`; caso contrário o retorno é `False`.

Exercício Proposto 14.1

Enunciado: *Escreva um programa que contenha uma função que recebe uma lista numérica e retorne a soma de seus valores se a lista não estiver vazia e se todos os elementos forem positivos.*

Caso a lista esteja vazia levante uma exceção Exception

Caso a lista contenha valores negativos levante uma exceção ValueError

Exercício Proposto 14.2

Enunciado: *Escreva um programa que solicite ao usuário que digite dois valores numéricos reais e apresente na tela a soma deles.*

Caso algum dos valores não seja número real levante uma exceção da classe TypeError. Faça uma validação independente para cada valor lido.

Sugestão *Implemente a leitura e validação do número real dentro de uma função. (opcional)*

Exercício Proposto 14.3

Enunciado: *Escreva um programa que permaneça em laço lendo nomes de pessoas e colocando-os em uma lista. Quando for fornecido o texto FIM o laço deve terminar. Apresente os nomes na tela um em cada linha.*

Caso algum nome seja um string vazio ou contenha caracteres não alfabéticos (algarismos e caracteres especiais) levante uma exceção da classe ValueError.

A execução do programa principal não pode ser interrompida pela exceção, ou seja, deve ser feito o tratamento das exceções e o usuário avisado que digitou um nome inválido, que não deve ser incluído na lista.

Exercício Proposto 14.4

Enunciado: *Escreva um programa que permaneça em laço até que seja digitado FIM. Dentro do laço leia códigos de barras de produtos no formato EAN13. Esse código de barras deve conter 13 dígitos numéricos e o último dígito é um dígito verificador.*

Dígitos verificadores são usados para validar o código digitado.

Caso o código EAN13 esteja vazio levante uma exceção Exception.

Caso o código EAN13 contenha caracteres inválidos (não numéricos) levante uma exceção TypeError.

Caso o dígito verificador digitado não coincida com o calculado levante uma exceção ValueError.

No programa principal faça o tratamento dos erros de modo que o programa não seja interrompido pela exceção e para cada classe de erro forneça uma mensagem apropriada.

Para cálculo do dígito verificador do código EAN13 veja a informação a seguir.

Informação de suporte para solução do ex. proposto 14.4 – Cálculo do D.V. do EAN13

fonte: <https://pt.wikipedia.org/wiki/EAN-13>

Cálculo do dígito verificador EAN 13

Tomando como base o EAN 13 de número: 789100031550-?

```
Multiplicam-se os dígitos do código por 1 e por 3, em sequência repetitiva de 1 e 3;
7 * 1 = 7
8 * 3 = 24
9 * 1 = 9
1 * 3 = 3
0 * 1 = 0
0 * 3 = 0
0 * 1 = 0
3 * 3 = 9
1 * 1 = 1
5 * 3 = 15
5 * 1 = 5
0 * 3 = 0

Somando o resultado das multiplicações encontra-se o total de 73.
O valor total da soma das multiplicações deve ser dividido por 10: (73/10 = 7.3)
Transforme o resultado em inteiro, "arredondando" o número para baixo. (7)
Some 1 ao resultado da divisão: (7+1 = 8)
Multiplique o resultado dessa soma por 10: (8*10 = 80)
Subtraia desse resultado o valor da soma inicial das multiplicações "73": (80 - 73 = 7)
Portanto, o dígito verificador é 7. Dessa forma, o código completo é: 7891000315507.
Se o resultado for um múltiplo de 10, o dígito verificador será 0.
```

Capítulo 15

EXPRESSÕES `yield` E FUNÇÕES GERADORAS

15.1 FUNÇÃO GERADORA - CONCEITO

Função Geradora (*generator function*) é um tipo especial de função capaz de implementar a estratégia conhecida como **Avaliação Preguiçosa** (*lazy evaluation*).

Na teoria das linguagens de programação o termo Avaliação Preguiçosa refere-se à estratégia de atrasar o cálculo de algum resultado até quando ele seja necessário.

Os benefícios da avaliação preguiçosa incluem:

- redução do consumo de memória armazenando apenas os dados necessários no momento e evitando armazenamento do que será necessário no futuro;
- redução do consumo de processador com o aumento do desempenho computacional, pois calcula apenas o que é necessário no momento;
- evita que eventual condição de erro futuro interfira em cálculos do presente momento, permitindo produção de resultados parciais;
- permite o cálculo eficiente de fluxos de dados (streaming) e tarefas de processamento em larga escala.

As principais desvantagens dessa estratégia são relacionadas à linguagem de programação adotada e ao processo de desenvolvimento do código quando ele ainda não está totalmente testado e homologado podendo, portanto, conter erros de programação. Nessas situações temos:

- implantação inadequada usando linguagem de programação sem suporte nativo para essa estratégia, exigindo codificação complexa, nem sempre bem-sucedida e com geração sobrecarga computacional (o que era para ficar mais eficiente, acaba ficando o oposto);
- complexidade de depuração, pois a avaliação tardia torna complexo o rastreamento de erros, o monitoramento dos valores contidos em variáveis e o acompanhamento da sequência de execução das instruções do programa;
- possibilidade de vazamento de memória no caso erros que levem ao descontrole quanto à alocação de memória e controle do conteúdo das variáveis do programa.

15.2 FUNÇÕES GERADORAS E O USO DE `yield` COMO COMANDO

Python é uma linguagem de programação multiparadigma, contendo elementos que suportam três paradigmas de programação, que são: o estruturado, o orientado a objetos e o funcional.

A avaliação preguiçosa que mencionamos na seção anterior está intimamente associada ao paradigma de programação funcional. Por Python suportar nativamente tal paradigma, fica bastante tranquilo aprendermos como usar esse recurso e neste capítulo vamos ver seus conceitos e alguns exemplos de aplicação.

Para começar precisamos conhecer o conceito de **Função Geradora** (*generator function*), que é um tipo especial de função capaz de retornar um objeto que pode ser usado como iterável em um laço `for`, do mesmo modo como usamos listas e tuplas.

No entanto, há uma diferença enorme: o objeto iterável gerado segue os princípios da avaliação preguiçosa. Com isso, não mantém todos os seus dados em memória, ou seja, não se trata de uma sequência previamente calculada e armazenada em memória. Com este recurso, cada elemento é gerado sob demanda apenas no momento em que será usado.

15.2.1 O QUE É `yield`?

A palavra reservada `yield` pode ser usada como comando ou como expressão. A seguir temos um exemplo para as duas formas:

```
def geradora():
    a = 10          # suponha a existência do objeto a
    yield a         # yield como comando - retorna o valor de a
# ou
    x = (yield a)   # yield como expressão - veremos depois na seção 15.5
```

Quando usado como comando, o `yield` substitui o comando `return`. Ambos retornam um valor e já sabemos que o `return` envia o retorno ao chamador e encerra a função.

O `yield` também envia o retorno de volta ao chamador, porém ele não encerra a função, apenas a suspende. Neste caso, a função permanece em memória com seu estado (valores dos objetos internos) mantido de modo a permitir novas chamadas. Quando a função é retomada, ela continua a execução imediatamente após a última execução do comando `yield`. Esse modo de execução permite que seu código produza uma série de valores ao longo do tempo, e em vez de calculá-los todos de uma só vez, cada valor é calculado em uma chamada.

Assim como o `return`, o uso do `yield` só é permitido dentro de funções. E quando uma função o contém, essa função é chamada de **função geradora**.

Vamos analisar essa ideia através de um exemplo. Importante ressaltar que o exemplo 15.1 é apenas um código de estudo, com fins didáticos. Ele serve para explicar o conceito e a dinâmica de implementação de uma função geradora, mas não é exatamente um exemplo de aplicação real.

Exemplo 15.1



```
>>> def gera_simples():
    yield 38
    yield 159
    yield 47
    yield 26
```



```
>>> gerador = gera_simples()
>>> for valor in gerador:
    print(valor)

38
159
47
26

>>> for valor in gera_simples: # cuidado, não esqueça os parênteses
    print(valor)
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    for valor in gera_simples:
TypeError: 'function' object is not iterable

>>> for valor in gera_simples(): # aqui está correto, com os parênteses colocados
    print(valor)

38
159
47
26
```

(exemplo interativo feito com IDE Idle)

A função deste exemplo 15.1 é uma função geradora. Dentro dela o comando `yield` foi usado 4 vezes para retornar algum valor inteiro. Cada vez que chega o momento de executar um comando `yield`, a função retorna o valor especificado e a função entra em estado de suspensão, permanecendo no aguardo de uma nova chamada.

Antes de realizar chamadas à função geradora precisamos atribuí-la a um objeto que passará a ser o **objeto gerador**. Em seguida, usamos o objeto gerador como iterável em um comando `for`.

Também é possível usar a função diretamente, porém não se esqueça de colocar os parênteses junto ao nome da função. Note que se você tentar usar a função geradora diretamente no comando `for` e esquecer dos parênteses, ocorrerá um erro de interpretação do código.

O resultado obtido é semelhante ao que obteríamos com o código a seguir:

```
>>> lista = [38, 159, 47, 26]
>>> for valor in lista:
>>>     print(valor)
```

Em um exemplo com tão poucos dados não há qualquer diferença de desempenho entre as duas alternativas. Observe, porém, que a lista de valores é muitíssimo pequena.

Imagine agora uma situação semelhante, porém com significativa mudança de quantidade, talvez com centenas de milhares ou dezenas de milhões de valores, e que a cada um estivesse associado algum processamento adicional. O consumo de recursos (processamento e memória) necessários para calcular e armazenar previamente todos os valores degradaria o desempenho do computador.

Em casos de grandes volumes de dados a serem processados o investimento de tempo e esforço em produzir uma função geradora será compensado com um processamento mais eficiente e com menor consumo de recursos do computador.

15.2.2 A FUNÇÃO `next()`

Uma outra forma de interagir com a função geradora é utilizar a função `next()`, que é uma função interna de Python. Em vez de usar um loop `for`, podemos usar a função `next()` diretamente no objeto gerador. Usá-la é especialmente útil para testar uma função geradora de forma interativa no Idle.

No exemplo 15.2 mostramos o uso do `next()`. E para deixar mais evidente a suspensão da função a cada `yield` fizemos uma alteração em seu código acrescentando um `print` antes de cada retorno.

Exemplo 15.2



```
>>> def gera_simples():
    print(' ...próximo valor retornado = 38')
    yield 38
    print(' ...próximo valor retornado = 159')
    yield 159
    print(' ...próximo valor retornado = 47')
    yield 47
    print(' ...próximo valor retornado = 26')
    yield 26

>>> gerador = gera_simples()
>>> next(gerador)
    ...próximo valor retornado = 38
    38
>>> next(gerador)
    ...próximo valor retornado = 159
    159
>>> next(gerador)
    ...próximo valor retornado = 47
    47
>>> next(gerador)
    ...próximo valor retornado = 26
    26
>>> next(gerador)
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    next(gerador)
StopIteration
```

(exemplo interativo feito com IDE Idle)

Neste exemplo podemos perceber com mais nitidez (em especial ao assistir ao vídeo) o comportamento do comando `yield`. Com a atribuição abaixo criamos um gerador, com o qual poderemos interagir usando o comando `next()`.

```
gerador = gera_simples()
```

O primeiro uso do comando `next()` provoca a primeira chamada ao gerador e isto faz com que o primeiro `print()` seja executado, seguido do `yield`. Este `yield` irá suspender o gerador e a prova disso é que o segundo `print()` não será executado. Esse segundo `print()` é executado apenas após a segunda chamada usando o `next()`. E assim por diante. A função geradora possui 4 comandos `yield`, o que faz com que seja possível usar a função `next()` 4 vezes. Depois que todos os `yield` forem executados, qualquer nova chamada gerará a exceção `StopIteration`. Quando isso ocorre dizemos que o gerador está esgotado ou exaurido.

Ao usar o `next()` pela quinta vez, ocorre o erro `StopIteration`, que é uma exceção natural gerada para sinalizar o fim de um iterador. Essa quinta execução excedeu a capacidade do gerador. Isso ocorre porque os geradores, assim como todos os iteradores, tem essa característica: eles se esgotam, ou seja, eles chegam a um fim. Os laços `for` são construídos de forma a verificar a ocorrência da exceção `StopIteration`. Na prática, isso quer dizer que o laço `for` encerra quando o iterador (ou gerador) se esgota.

É possível usar um gerador apenas uma vez. Após seu esgotamento não será possível voltar a usá-lo, a menos que uma nova atribuição `gerador = gera_simples()` seja feita.

15.3 EXEMPLOS DE APLICAÇÃO

15.3.1 CRIAÇÃO DE SEQUÊNCIAS INFINITAS

Este é um caso muito comum de uso de geradores: a criação de sequências infinitas.

Ao usar o termo infinito, a impressão que se tem é que faremos algo que ficará executando para sempre em um computador. Bem, não é essa a ideia real. Trata-se na verdade de escrever um código que tem potencial para ser infinito e quando formos utilizá-lo definiremos um limite.

No exemplo 15.3 implementamos uma função geradora com essa característica.

Exemplo 15.3



```
>>> def gerador_pares():
    a = 2
    while True:
        yield a
        a += 2

# primeira parte
gen = gerador_pares()
next(gen)
2
next(gen)
4
next(gen)
6
next(gen)
8

# segunda parte
>>> for _ in range(20):
>>>     print(next(gen), end = ' ')
10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
```

(exemplo interativo feito com IDE Idle)

Observe este código com atenção e perceba que ele produz números pares sucessivos. O código dentro da função tem um laço infinito `while True`.

```
a = 2
while True:
    yield a
    a += 2
```

Somando-se a isso o fato de que dentro do laço o comando `break` não é utilizado, garantimos que esse laço será infinito. Porém, dentro do laço é usado o comando `yield`, que ao ser executado suspende a função e retorna o valor que o objeto a contém no momento.

Depois de pronta a função passamos a usá-la através do gerador `gen`. No exemplo dividimos esse uso em duas etapas. Na primeira, com `next()` podemos constatar que ela funciona e podemos gerar tantos pares quanto quisermos. Na segunda parte usamos o gerador em conjunto com um comando `for` para gerar 20 elementos. E ao invés de 20 poderiam ser 50, 100 ou qualquer outra quantidade desejada.

A função geradora tem capacidade de gerar valores infinitamente. Mas o programa não ficará rodando infinitamente, pois o programa principal garantirá sua finitude.

Agora estude e reproduza o exercício resolvido 15.1 onde aplicamos essa mesma técnica para gerar uma sequência de números primos.

Exercício Resolvido 15.1



Enunciado: Escreva uma função geradora capaz de calcular e retornar números primos a partir do 2.

Escreva um programa para testá-la.

```
def gerador_primos():
    yield 2
    v = 3
    while True:
        raiz = v ** 0.5
        i = 3
        while i <= raiz and v % i != 0:
            i += 2
        if i > raiz:
            yield v
        v += 2

gen = gerador_primos()
Qtde = int(input('Digite a quantidade de primos: '))
for cont in range(Qtde):
    print(next(gen), end = ' ')
print('\n\nFim do Programa')

Digite a quantidade de primos: 22
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79

Fim do Programa
```

No próximo exercício resolvido, 15.2, faremos uma função geradora que retorna uma tupla contendo um número inteiro e seu fatorial. A função geradora é implementada contendo um laço infinito, sendo que em cada execução ela incrementa o objeto `num` e calcula o fatorial associado a esse novo `num`. Veja seu funcionamento rodando o código deste exercício.

Exercício Resolvido 15.2



Enunciado: Escreva uma função geradora capaz de calcular e retornar o fatorial de um número natural. Essa função deve retornar uma tupla com o próprio número natural e seu fatorial.

Escreva um programa para testá-la. Lembre-se que $0! = 1$

```
def funcao_fatorial():
    num, fat = 0, 1
    while True:
```

```
        yield num, fat
        num += 1
        fat *= num

N = int(input('Digite a quantidade: '))
gen = funcao_fatorial()
for i in range(N):
    print(next(gen))

print('\nFim do Programa')
Digite um valor: 12
(0, 1)
(1, 1)
(2, 2)
(3, 6)
(4, 24)
(5, 120)
(6, 720)
(7, 5040)
(8, 40320)
(9, 362880)
(10, 3628800)
(11, 39916800)

Fim do Programa
```

No programa principal um número inteiro é lido e usado como quantidade de tuplas a serem geradas usando a função geradora `funcao_fatorial()`.

Agora que você já testou esse programa considere fazer uma alteração. O objetivo dela é aprender o que acontece quando usamos uma função geradora infinita, sem incluir qualquer situação de restrição no programa principal. Por exemplo, se neste exercício forem executadas as linhas de código a seguir, teremos problemas:

```
for ret in funcao_fatorial():
    print(ret)
```

Ao executar esse laço `for` no programa principal, você vai provocar um laço infinito que gerará valores sem qualquer parada, até que um erro ocorra. Neste caso específico, o erro vai ocorrer quando o fatorial calculado for grande o bastante para estourar a capacidade de armazenamento de dígitos para os números inteiros do Python.

É claro que um programa não pode ficar gerando erros assim. Portanto, seja tenha cuidado ao criar funções geradoras infinitas. Você pode usá-las sempre que necessário, mas não se esqueça de fazer o controle no programa principal.

15.3.2 FILTRAGEM DE DADOS

É possível usar uma função geradora para implementar uma filtragem de dados. Neste tipo de uso o critério de filtragem é implementado dentro da função geradora, na qual além da filtragem pode-se implementar algum processamento adicional. No exemplo 15.4 a seguir, vamos filtrar uma lista de valores reais e aplicar a eles 10% de aumento.

Neste exemplo, vamos implementar um filtro no qual a função geradora recebe uma lista de preços (números reais) e dois limites, mínimo (`pmin`) e máximo (`pmax`). Para cada valor da lista, que esteja dentro do intervalo [`pmin`, `pmax`], a função retorna uma tupla contendo o valor original e o valor acrescido de 10%.

Exemplo 15.4

```
def filtro(dados, pmin, pmax):
    for valor in dados:
        if pmin <= valor <= pmax:
            yield valor, valor * 1.1

precos = [36.3, 174.19, 43.71, 108.32, 89.14]
lmin = int(input('Digite o mínimo da faixa: '))
lmax = int(input('Digite o máximo da faixa: '))
for valores in filtro(precos, lmin, lmax):
    print(valores)
print('\nFim do Programa')
Digite o mínimo da faixa: 80
Digite o máximo da faixa: 150
(108.32, 119.152)
(89.14, 98.054)

Fim do Programa
```

No programa principal ocorre a carga da lista de dados, bem como fazemos a leitura dos limites mínimo e máximo. Em seguida a função `filtro()` é usada como iterável para produzir o resultado final.

É claro que esta não é a única forma de solução, mas ela tem a vantagem de percorrer a lista original sem a necessidade de realizar qualquer duplicação da lista original.

15.3.3 PROCESSAMENTO INTENSO DE DADOS

Ao lidar com tarefas de pré-processamento de dados, o uso de `yield` é extremamente útil. em situações assim é possível realizar o processamento e retornar os resultados processados, um de cada vez, sem armazenar todo o conjunto de dados processados na memória. Conceitualmente será algo com o seguinte aspecto:

Exemplo 15.5**modelo conceitual**

```
def executa_proc(dados):
    algo = ...
    return algo

def processa(dados):
    for item in dados:
        item_processado = executa_proc(item)
        yield item_processado

D = [d1, d2, d3, d4, ...] # considere que d1, d2, d3, etc sejam grandes
for resultado in processa(D):
    salvar_bancodados(resultado)

(exemplo conceitual - necessita implementação específica)
```

Essa abordagem é benéfica ao lidar com grandes conjuntos de dados e reduz a sobrecarga do processador e o consumo de memória.

Um caso de uso típico para esse modelo conceitual acontece em web-scraping (raspagem de dados), que consiste em realizar a busca e extração de dados de várias páginas da internet. Você pode usar essa técnica para buscar e carregar páginas da web, uma de cada vez a partir de uma lista de endereços, e após essa carga, fazer a análise e extração das informações de interesse nela contida.

15.3.4 LEITURA DE GRANDES ARQUIVOS

Um caso de uso comum de geradores é trabalhar com fluxos de dados (streaming) ou arquivos grandes, com muitas linhas.

Nas áreas de BigData e Análise de Dados é comum trabalhar com arquivos muito grandes. Em geral são arquivos texto no formato CSV (*comma separated values*), nos quais os dados de uma linha são separados por vírgulas, ou algum outro caractere. Este formato é uma forma comum de compartilhar dados. Agora, e se você quiser contar o número de linhas em um arquivo CSV? O bloco de código abaixo mostra uma maneira de ler todas as linhas do arquivo:

```
arq = open(nome_arquivo)
linhas = arq.read().split("\n")
```

Esse código será satisfatório caso o tamanho do arquivo seja razoável. Porém, se ele contiver centenas de milhares de linhas isso pode ser um problema.

Em casos como esse pode ser bem mais eficiente executar a leitura linha a linha e uma função geradora pode ser uma alternativa interessante. No exemplo 15.6 implementamos um exemplo de uma aplicação assim.

Exemplo 15.6



```
def le_arquivo(nome_arq):
    for uma_linha in open(nome_arq, "r"):
        yield uma_linha.rstrip()          # o rstrip() remove o \n do final da linha

arquivo = input('Digite o nome do arquivo: ')
for linha in le_arquivo(arquivo):
    print(linha)
print('\nFim do Programa')

Digite o nome do arquivo: veiculos.txt
ELM8038;CELTA EL 1.0;2012;76320
LCA1E42;FOX TSI 1.6;2020;23900
FLB2297;PALIO FIRE 1.0;2016;68350
SCE9U33;COROLLA GLI 2.0;2022;43200
CMT7153;ONIX LT 1.0;2021;54600

Fim do Programa
```

A função geradora `le_arquivo()` abre o arquivo e retorna uma linha por vez ao programa principal, onde cada linha retornada poderá passar por qualquer processamento que seja necessário e adequado ao problema para o qual o programa esteja sendo desenvolvido.

15.4 CONSTRUINDO GERADORES COM EXPRESSÕES GERADORAS

Primeiro ponto: não confunda os termos **expressão geradora** com função geradora. Já vimos o que faz uma função geradora e já vimos que podemos criar objetos geradores a partir de uma função geradora quando escrevemos um código como este:

```
def funcao_geradora():
    ... (código da função)
    yield retorno

gen = funcao_geradora()          # aqui definimos o objeto gerador gen explícito
#ou
for dado in funcao_geradora():   # aqui definimos um objeto gerador implícito
```

Nesta seção vamos dar o próximo passo, apresentando o conceito de **expressão geradora**. Na documentação técnica oficial de Python o termo em inglês usado para esse conceito é *Generator Expression*. Porém, na literatura técnica em geral sobre esse assunto encontram-se também os termos *Generator Comprehension* e *Comprehension Expressions*.

Neste texto vamos padronizar o uso do termo em português "expressão geradora", que é encontrado nas traduções para português da documentação oficial de Python.

Uma expressão geradora também é referida na literatura técnica pelos termos em inglês

**Generator Comprehension e
Comprehension Expressions**

Este conceito está relacionado com a ideia de funções geradoras e também está relacionado com o conceito de list comprehension, visto no capítulo 13 do Módulo Intermediário deste curso.

Naquele capítulo vimos que list comprehension é uma forma inteligente de criar sequências de dados iteráveis que possam ser usados na implementação de algum algoritmo.

Uma expressão geradora permite fazer o mesmo, porém, consumindo uma quantidade muito menor de memória. imagine que queiramos produzir uma lista com números inteiros elevados ao quadrado, de 1 a 10. Podemos fazer isso de duas formas, mostradas nas linhas a seguir:

```
quadrados_lc = [num**2 for num in range(1, 11)] # list comprehension
quadrados_eg = (num**2 for num in range(1, 11)) # expressão geradora
```

Note que a única diferença sintática é o uso de colchetes `[]` para list comprehension e parênteses `()` para expressão geradora.

Ambos produzem a mesma sequência de valores e podem ser usados nos mesmos contextos. Veja o exemplo 15.7. Na parte 1 desse exemplo fazemos a geração das duas estruturas e as usamos em uma iteração com o comando `for`, produzindo o mesmo resultado com ambas.

Exemplo 15.7 – parte 1



```
>>> quadrados_lc = [num**2 for num in range(1, 11)]
>>> print(quadrados_lc)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] # quadrados_lc é uma lista
>>> for valor in quadrados_lc:
    print(valor, end=' ')
1 4 9 16 25 36 49 64 81 100

>>> quadrados_eg = (num**2 for num in range(1, 11))
>>> print(quadrados_eg)
<generator object <genexpr> at 0x000001763A308FB0> # quadrados_eg é um gerador
>>> for valor in quadrados_eg:
    print(valor, end=' ')
1 4 9 16 25 36 49 64 81 100

(exemplo interativo feito com IDE Idle)
```

Acima, usando `print()`, destacamos que `quadrados_eg` é um gerador e `quadrados_lc` é uma lista. Se você imaginar que a lista usa mais memória do que o gerador você estará certo, pois todos os

valores dessa precisam ser armazenados em memória. Será que existe alguma forma de mensurar essa memória consumida pelos dois objetos: `quadrados_eg` e `quadrados_lc`?

A resposta é sim e fazemos exatamente isso na parte 2 do exemplo 15.7. Mas para a parte 2 mudamos a quantidade de elementos de 10 para 10.000 (dez mil elementos). Com essa quantidade de elementos não faz sentido exibir todos em tela, mas vamos exibir quanto cada um consome de memória usando o método `sys.getsizeof()` do módulo `sys`.

Exemplo 15.7 – parte 2

o vídeo deste Exemplo contém as três partes

```
>>> quadrados_lc = [num**2 for num in range(1, 10001)]
>>> quadrados_eg = (num**2 for num in range(1, 10001))
>>> import sys
>>> sys.getsizeof(quadrados_lc)
85176
>>> sys.getsizeof(quadrados_eg)
200
```

(exemplo interativo feito com IDE Idle)

Veja a diferença na quantidade de memória consumida. A lista, com os 10 mil elementos consome mais de 85 mil bytes, enquanto a expressão geradora consome apenas 200 bytes.

Por outro lado, se a lista for menor que a memória disponível na máquina em uso, uma list comprehension poderá ser processada mais rapidamente do que a expressão geradora equivalente. Na parte 3 do exemplo 15.7 fazemos uma verificação de tempo de processamento usando o método `cProfile.run()`.

Exemplo 15.7 – parte 3

o vídeo deste Exemplo contém as três partes

```
>>> import cProfile
>>> cProfile.run('sum([num**2 for num in range(1, 10001)])') # list comprehension
4 function calls in 0.002 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
1      0.002    0.002    0.002    0.002  <string>:1(<module>)
1      0.000    0.000    0.002    0.002  {built-in method builtins.exec}
1      0.001    0.001    0.001    0.001  {built-in method builtins.sum}
1      0.000    0.000    0.000    0.000  {method 'disable' of
                                     '_lsprof.Profiler' objects}

>>> cProfile.run('sum((num**2 for num in range(1, 10001)))') # expressão geradora
10005 function calls in 0.006 seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
10001   0.003    0.000    0.003    0.000  <string>:1(<genexpr>)
1      0.000    0.000    0.006    0.006  <string>:1(<module>)
1      0.000    0.000    0.006    0.006  {built-in method builtins.exec}
1      0.003    0.003    0.006    0.006  {built-in method builtins.sum}
1      0.000    0.000    0.000    0.000  {method 'disable' of
                                     '_lsprof.Profiler' objects}
```

(exemplo interativo feito com IDE Idle)

Observe que o tempo de processamento do list comprehension foi 3 vezes mais rápido que o tempo da função geradora. Uma verificação de tempo como essa é dependente da máquina e dos programas que estão em execução no momento em que o teste for feito.

Agora imagine isso levado à casa dos muitos milhões de elementos – algo frequente de ocorrer quando se trabalha com Big Data – a escolha sobre usar um ou outro método deverá ser avaliada para cada caso levando em consideração esses dois aspectos:

- tempo necessário ao processamento;
- quantidade de memória disponível;

As expressões geradoras devem ser vistas como um recurso valioso em processamentos dessa natureza (grandes volumes de dados) e sempre que possível usadas em substituição às listas.

Lembre-se de que um list comprehension retorna uma lista completa, enquanto as expressões geradoras retornam geradores.

Por fim, lembre-se sempre que geradores funcionam da mesma forma, sejam eles construídos a partir de uma função ou de uma expressão. A sintaxe das duas são diferentes, mas o resultado é a criação de um gerador.

15.5 USANDO `yield` COMO EXPRESSÃO

Um aspecto avançado de `yield` é usá-lo como expressão, como na linha abaixo:

```
x = (yield dado)
```

A diferença parece sutil, mas é relevante e muito interessante. Uma função geradora com o `yield` neste formato produz um gerador que pode ser controlado a partir do chamador com as funcionalidades de **receber conteúdo**, **gerar uma exceção** e ser instruída a **finalizar a geração**.

Importante
É obrigatório o uso dos parênteses para definir o `yield` como expressão
`x = (yield dado)` ← parênteses obrigatórios

Assim, os objetos geradores que tenham uma expressão `yield` podem fazer uso dos seguintes métodos:

- `.send()` – envia um dado para o gerador;
- `.throw()` – gera uma exceção usando o gerador e finaliza;
- `.close()` – finaliza o gerador;

15.5.1 MÉTODO `.send()` DE UM GERADOR

Para apresentar o método `.send()` veremos o exemplo 15.8. Nesse exemplo a função geradora `fg()` é capaz de ter uma funcionalidade dupla: ela pode gerar uma sequência de números pares, ou uma sequência de ímpares. Tudo depende do valor contido no objeto `resto`.

- se o conteúdo de `resto` for 0 a sequência gerada será de números pares;
- se o conteúdo de `resto` for 1 a sequência gerada será de números ímpares;

Exemplo 15.8



```
def fg():
    resto = 0
    num = 2
    while True:
        if num % 2 == resto: # resto calculado é comparado com o objeto resto
```

```
        dado = (yield num)
        if dado is not None:
            resto = dado # troca o valor de resto
            num = 0 # reseta o valor de num
    num += 1

gen = fg()
print('Gera 5 pares')
for i in range(5):
    print(next(gen))

print('\nGera 5 ímpares')
ret = gen.send(1) # este método retorna o 1º valor da sequência
print(ret)
for i in range(4): # então precisamos gerar o próximos 4
    print(next(gen))

# tentativa de usar send com valor diferente de 0 ou 1
print('\nParâmetro incorreto')
ret = gen.send(2) # causa um laço infinito

print('\nFim do Programa')
```

Gera 5 pares
2
4
6
8
10

Gera 5 ímpares
1
3
5
7
9

Parâmetro incorreto
(Daqui para frente nada mais acontece. O programa está em laço infinito.
O programa não chega ao fim)

Vamos compreender a expressão `yield` contida neste exemplo. Quando construído da forma mostrada nas linhas a seguir, dentro da função geradora,

```
if dado is not None:
    dado = (yield num)
```

o objeto `dado` recebe o valor da expressão `yield` e a partir daí podem acontecer uma de duas coisas:

- se o chamado for feito com `next()` ou em um laço `for`, a expressão assume o valor `None`. É por esse motivo que devemos usar o `if` para perguntar se ele não é `None` e, caso não o usássemos, a função geradora ficará errada;
- se o chamado for feito com o método `.send()`, o valor passado como argumento será recebido pelo `yield` e será diferente de `None`, podendo ser atribuído a algum objeto;

Portanto, neste exemplo, ao usarmos o método `.send()` do modo abaixo, o `yield` receberá o valor passado, e esse valor será atribuído ao objeto `dado` dentro da função.

```
gen.send(1)
```

Para que a função gere as sequências corretamente deve ser passado o argumento 0 para sequência de pares e 1 para sequência de ímpares. Outros valores que sejam passados não estão previstos e fazem com que a função geradora entre em laço infinito.

É preciso corrigir isso e para fazê-lo acrescentamos uma verificação se `dado` é 0 ou 1, caso não seja uma exceção é levantada.

Exemplo 15.8 – correção do laço infinito

essa correção é mostrada no vídeo

```
>>> def fg():
    resto = 0
    num = 2
    while True:
        if num % 2 == resto: # resto calculado é comparado com o objeto resto
            dado = (yield num)
            if dado is not None:
                if dado not in (0, 1):
                    raise ValueError(f'Esperado 0 ou 1 - passado {dado}')
                resto = dado # troca o valor de resto
                num = 0 # reseta o valor de num
            num += 1

>>> gen = fg()
>>> for i in range(5):
    print(next(gen))
2
4
6
8
10
>>> gen.send(1)
1
>>> for i in range(4):
    print(next(gen))
3
5
7
9

>>> gen.send(2) # causará uma exceção e esgotará o gerador

Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    gen.send(2)
  File "<pyshell#36>", line 9, in fg
    raise ValueError(f'Esperado 0 ou 1 - passado {dado}')
ValueError: Esperado 0 ou 1 - passado 2

>>> next(gen) # mostra que o gerador está esgotado (causado pela exceção)
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    next(gen)
StopIteration
```

(destaques em negrito feitos pelo autor)

Quando uma exceção é levantada dentro de um gerador ele tem sua execução automaticamente esgotada. Isso pode ser visto acima quando fizemos com a chamada `next(gen)` após a exceção causada pelo envio do valor 2 feito com o método `.send()`.

Para poder voltar a usar o gerador é preciso reconstruí-lo com uma nova atribuição `gen = fg()`.

Cuidado importante

É preciso ter atenção com um detalhe relevante quanto ao uso do método `.send()`. Não é permitido que esse método seja usado imediatamente após a criação do gerador, ou seja, antes que o gerador seja usado pelo menos uma vez. Com isso uma tentativa de usar o código abaixo resultará em erro.

```
gen = função_geradora()
gen.send(10)           # vai gerar erro
```

Para eliminar o erro deve-se usar o gerador pelo menos uma vez, antes do `.send()`, por exemplo, deste modo:

```
gen = função_geradora()
next(gen)
gen.send(10)           # ok, vai funcionar corretamente
```

15.5.2 MÉTODOS `.throw()` E `.close()` DE UM GERADOR

Estes dois métodos são usados para interromper um gerador.

- O método `.throw(...)` faz a interrupção do gerador lançando uma exceção. Ele é útil em situações em que podemos precisar capturar e tratar essa exceção lançada. Este método deve receber um objeto `Exception` como parâmetro. Após o uso deste método qualquer tentativa de usar o gerador gerará um `StopIteration`;
- O método `.close()` faz a interrupção do gerador de modo silencioso, simplesmente colocando-o no estado `StopIteration`.

Vamos exemplificar os dois métodos, com essa função geradora que produz múltiplos de 10.

```
def simples():
    num = 10
    while True:
        yield num
        num += 10
```

Exemplo de uso do método `.throw()`:

Exemplo 15.9



```
>>> gen = simples()
>>> for _ in range(5):
    print(next(gen))
10
20
30
40
50

>>> gen.throw(Exception('Gerador terminado pelo usuário'))
Traceback (most recent call last):
  File "<pyshell#80>", line 1, in <module>
    gen.throw(Exception('Gerador terminado pelo usuário'))
  File "<pyshell#71>", line 4, in simples
    yield num
```

```

Exception: Gerador terminado pelo usuário

>>> next(gen)
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    next(gen)
StopIteration

>>> for valor in gen:
    print(valor)
# nada acontece

# outra possibilidade de uso
>>> gen = simples()
>>> for valor in gen:
    print(valor)
    if valor > 40:
        gen.throw(ValueError('Todos os valores foram gerados'))
10
20
30
40
50
Traceback (most recent call last):
  File "<pyshell#95>", line 4, in <module>
    gen.throw(ValueError('Todos os valores foram gerados'))
  File "<pyshell#71>", line 4, in simples
    yield num
ValueError: Todos os valores foram gerados
(exemplo interativo feito com IDE Idle)

```

Observe que após o uso do método `.throw()` não é mais possível usar a função `.next()` e haverá uma sinalização da situação de `StopIteration`. Também não é possível usar um iterador `for`, mas neste caso nada acontece porque o `for` verifica se o gerador está em estado de `StopIteration` e, caso esteja, nada faz.

Exemplo de uso do método `.close()`.

Vamos agora fazer um exemplo de uso do método `.close()`. Nesse exemplo faremos exatamente o mesmo que no exemplo 15.8, apenas substituindo o `.throw()` pelo `.close()`.

Exemplo 15.10

Teste este código no Idle

```

>>> gen = simples()
>>> for _ in range(5):
    print(next(gen))
10
20
30
40
50

>>> gen.close() # encerra o gerador silenciosamente
>>> next(gen)
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    next(gen)
StopIteration

>>> for valor in gen:
    print(valor)
# nada acontece

```

```
# outra possibilidade de uso
>>> gen = simples()
>>> for valor in gen:
    print(valor)
    if valor > 40:
        gen.close()

10
20
30
40
50
>>> next(gen)
Traceback (most recent call last):
  File "<pyshell#81>", line 1, in <module>
    next(gen)
StopIteration
```

(exemplo interativo feito com IDE Idle)

Com este exemplo podemos perceber que após o uso do método `.close()` o gerador não pode ser mais usado pois foi colocado no estado de `StopIteration`.

No exercício resolvido 15.3 vamos alterar a função geradora de números fatoriais criada no exercício resolvido 15.2, incluindo a possibilidade de o número base para cálculo do fatorial ser resetado usando o método `.send()`.

Exercício Resolvido 15.3



Enunciado: Reescreva a função geradora do exercício resolvido 15.2 de modo que ela possa receber um inteiro através de uma expressão `yield`. Esse inteiro deve ser usado para resetar o valor inicial para o qual calcularemos o fatorial.

No programa principal leia a quantidade de tuplas a serem geradas e faça um laço para gerar várias sequências e assim verificar se os retornos da função geradora estão corretos.

```
def funcao_fatorial():
    num, fat = 0, 1
    while True:
        i = (yield num, fat)
        num += 1
        fat *= num
        if i is not None:
            num, fat = i, 1
            for a in range(1, num+1):
                fat *= a

qtde = int(input('Digite a quantidade de fatoriais: '))
gen = funcao_fatorial()
next(gen)

prim = int(input('\nDigite valor inicial para a próxima sequência: '))
while prim >= 0:
    r = gen.send(prim)
    fatoriais = [r]
    for _ in range(qtde-1):
        fatoriais.append(next(gen))
    print(f'Sequência de fatoriais iniciando em {prim}!')
    print(fatoriais)

    prim = int(input('\nDigite valor inicial para a próxima sequência: '))
print('\nFim do Programa')

Digite a quantidade de fatoriais: 5
```

```
Digite valor inicial para a próxima sequência: 0
Sequência de fatoriais iniciando em 0!
[(0, 1), (1, 1), (2, 2), (3, 6), (4, 24)]

Digite valor inicial para a próxima sequência: 1
Sequência de fatoriais iniciando em 1!
[(1, 1), (2, 2), (3, 6), (4, 24), (5, 120)]

Digite valor inicial para a próxima sequência: 2
Sequência de fatoriais iniciando em 2!
[(2, 2), (3, 6), (4, 24), (5, 120), (6, 720)]

Digite valor inicial para a próxima sequência: 3
Sequência de fatoriais iniciando em 3!
[(3, 6), (4, 24), (5, 120), (6, 720), (7, 5040)]

Digite valor inicial para a próxima sequência: 4
Sequência de fatoriais iniciando em 4!
[(4, 24), (5, 120), (6, 720), (7, 5040), (8, 40320)]

Digite valor inicial para a próxima sequência: -1

Fim do Programa
```

No próximo exercício resolvido, 15.4, criamos uma função geradora que recebe dois parâmetros de entrada e os utiliza internamente, mostrando que essa também é uma opção possível em funções geradoras.

Exercício Resolvido 15.4

Teste este código no PyCharm

Enunciado: Escreva um programa para implementar um gerador capaz de gerar números aleatórios dentro de um determinado intervalo.

```
import random
def gerador_aleatorio(ini, fim):
    while True:
        yield random.randint(ini, fim)

# Programa principal
inicio = int(input('Digite o início da faixa: '))
final = int(input('Digite o final da faixa: '))
qtde = int(input('Digite quantos valores quer gerar: '))
# Cria o gerador
gen = gerador_aleatorio(inicio, final)
for _ in range(qtde):
    print(next(gen), end=' ')

print('\n\nFim do Programa')
Digite o início da faixa: 30
Digite o final da faixa: 200
Digite quantos valores quer gerar: 8
79 72 149 57 141 147 30 149

Fim do Programa
```

Neste exercício resolvido 15.5 faremos uma função geradora recursiva capaz de gerar todas as permutações de elementos dentro de uma lista.

Exercício Resolvido 15.5

Teste este código no PyCharm

Enunciado: Escreva um programa para implementar uma função geradora que gera todas as permutações dos caracteres de um string. O string deve ser lido como entrada de dados no programa principal.

```
def gera_permutacoes(texto):
    if len(texto) <= 1:
        yield texto
    else:
        for i in range(len(texto)):
            caractere_atual = texto[i]
            demais_caracteres = texto[:i] + texto[i+1:]
            for permutacao in gera_permutacoes(demais_caracteres):
                yield caractere_atual + permutacao

lista = input('Digite um texto: ')
for l in gera_permutacoes(lista):
    print(l)
print('\nFim do Programa')
```

Digite um texto: ABC
ABC
ACB
BAC
BCA
CAB
CBA

Fim do Programa

Com este exercício resolvido, vemos que a função `gera_permutacoes()` faz chamadas a si mesma, mostrando que a recursividade também pode ser utilizada em funções geradoras.

Essa chamada recursiva tem o objetivo de reduzir em um elemento o tamanho do `texto` a ser permutado. A ideia central é que a permutação de um `texto` de tamanho `N` pode ser visto como a concatenação: `texto[i] + texto[sem o caractere i]`.

Nessa recursividade o caso recursivo ocorre quando o objeto `texto` tem comprimento maior que 1 e o caso base quando esse tamanho é menor ou igual a 1.

Agora vamos passar a um novo exemplo.

Média móvel é um conceito de média dinâmica que se altera a cada novo valor acrescentado a um conjunto de valores. Geralmente é usada para demonstrar o que está acontecendo em um processo, enquanto o mesmo ainda está em andamento.

Por exemplo, se você mandar copiar um arquivo muito grande para um pendrive, o sistema operacional pode te mostrar a média de bytes que está sendo transferida por segundo enquanto a cópia ocorre. E a partir dessa média, estimar quanto tempo falta para terminar a cópia. Se você tiver outros programas rodando enquanto faz a cópia, outros fatores podem afetar o desempenho do sistema e a média móvel, calculada a cada 200 milissegundos (por exemplo), sofrerá variações.

O programa a seguir faz esse tipo de cálculo usando uma função geradora capaz de receber dados através do método `.send()`.

Exercício Resolvido 15.6



Enunciado: Escreva um programa que implemente uma função geradora capaz de calcular a média móvel de uma sequência de números. A média móvel é calculada e apresentada a cada valor que é fornecido.

Utilize uma função geradora que contenha uma expressão `yield` e use o método `.send()` para enviar os valores para a função. Exiba os valores com 3 casas decimais.

```
def media_movel():
    total = qtde = 0
    while True:
        dado = (yield total / qtde if qtde > 0 else 0)
        if dado is not None:
            total += dado
            qtde += 1

gen = media_movel() # cria o gerador
next(gen)           # inicializa o gerador
valor = input('Digite um valor (ou FIM para sair): ')
while valor.upper() != 'FIM':
    resultado = gen.send(float(valor)) # envia o valor
    print(f'média móvel atual = {resultado:.3f}\n')
    valor = input('Digite um valor (ou FIM para sair): ')

print(f'\nValor final da média = {resultado:.3f}')

print('\nFim do Programa')
```

Digite um valor (ou FIM para sair): 6.0
média móvel atual = 6.000

Digite um valor (ou FIM para sair): 3.5
média móvel atual = 4.750

Digite um valor (ou FIM para sair): 8.9
média móvel atual = 6.133

Digite um valor (ou FIM para sair): 4.4
média móvel atual = 5.700

Digite um valor (ou FIM para sair): fim

Valor final da média = 5.700

Fim do Programa

Exercício Proposto 15.1

Enunciado: *Altere o exercício resolvido 15.1 fazendo com que a função geradora se torne mais eficiente ao armazenar em uma lista todos os números primos calculados previamente.*

Exercício Proposto 15.2

Enunciado: *Altere o exercício resolvido 15.1 fazendo com que a função geradora possa ser resetada com o uso do método .send().*

Esse método deve passar um argumento inteiro que deve ser usado como novo elemento inicial na geração dos números primos. Garanta também que valores menores que 2 não possam ser usados como valor inicial.

Exercício Proposto 15.3

Enunciado: *Altere a função geradora do exercício resolvido 15.3 fazendo uma validação do valor passado com o método .send(). Não permita que os valores passados sejam menores que 0, levantando uma exceção quando isso acontecer. Escreva o programa principal para testá-la*

Exercício Proposto 15.4

Enunciado: *Escreva uma função geradora capaz de calcular e retornar os números da sequência de Fibonacci. Escreva um programa para testá-la.*

Exercício Proposto 15.5

Enunciado: *Escreva um programa que implemente um gerador que gere a sequência Collatz para um determinado número de entrada. Esta sequência é conhecida por convergir para o valor 1.*

A sequência de Collatz é definida da seguinte forma:

O valor inicial N é um inteiro positivo qualquer;

Se N for par o próximo valor é calculado dividindo N por 2;

Se N for ímpar o próximo valor é calculado com a expressão: $3 \cdot N + 1$;

A geração da sequência termina quando gerar o valor 1.

Exemplo 1: *para N = 18 a sequência será composta pelos números a seguir:*

18 9 28 14 7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

Exemplo 2: *para N = 85 a sequência será composta pelos números a seguir:*

85 256 128 64 32 16 8 4 2 1

Exercício Proposto 15.6

Enunciado: *Escreva uma função geradora capaz de listar todos os arquivos contidos em uma hierarquia de pastas de um disco rígido. Escreva o programa principal para testá-la.*

Dica: *Use o módulo os da biblioteca interna de Python.*

Capítulo 16

MÓDULOS E PACOTES

16.1 CONCEITO DE MÓDULO

Desde o início deste curso temos trabalhado, ora de forma interativa com o ambiente Idle para testar os elementos existentes em Python; ora escrevendo um programa maior e mais completo na forma de script com o PyCharm.

Quando usamos o modo interativo, ao fechar o programa, tudo o que foi digitado é perdido. Ao contrário, quando trabalhamos com scripts, os programas ficam salvos no armazenamento do computador e podemos utilizá-lo sempre que necessário.

Os scripts podem se tornar cada vez maiores à medida que acrescentamos funcionalidades a um programa e nestes casos entram em cena o Módulos de Python. À medida que um programa vai se tornando maior, é uma boa prática dividi-lo em arquivos menores, com os propósitos de facilitar a organização e futuras manutenções. A ideia essencial é usar arquivos separados para conter funções que podem ser usadas em vários programas diferentes. Isso evita a necessidade de repetir a função no código de programas distintos.

Esses arquivos que contêm parte do código são chamados de **Módulos** (*modules*). Elementos existentes em um módulo podem ser importados em outros módulos com o comando `import` e com certeza você deve lembrar que neste curso já usamos algumas vezes esse comando.

Assim, **programação modular** (ou modularização) refere-se ao processo de dividir um programa grande e complexo em subprogramas ou módulos separados, menores e mais gerenciáveis. Esses módulos menores podem então ser reunidos para criar um aplicativo maior. É algo semelhante aos brinquedos de blocos nos quais blocos básicos são usados para construir um brinquedo maior e completo.

16.1.1 VANTAGENS NO USO DE MÓDULOS

Simplicidade

Ao dividir seu projeto de software em módulos, ao invés de abordar o problema como um todo você vai concentrar-se em um módulo por vez. Considerando que em um módulo você foca em uma parte menor do problema, então terá uma tarefa específica que pode ser desenvolvida com mais objetividade e menor

incidência de erros.

Capacidade de manutenção

Em um bom projeto modular, cada módulo terá determinados limites lógicos para resolver diferentes partes do problema maior. Se os módulos forem projetados de uma forma que haja pouca interdependência, há uma boa probabilidade de que modificações em um único módulo não causem impacto em outras partes do programa. Isso torna mais viável para uma equipe de muitos programadores trabalhar de forma colaborativa em um aplicativo grande.

Reutilização

A funcionalidade definida em um único módulo pode ser facilmente reutilizada por outras partes da aplicação, desde que haja uma interface definida adequadamente. Isso elimina a necessidade de duplicar código.

Escopo

Os módulos normalmente definem um "**namespace**" separado, o que ajuda a evitar duplicidade de identificadores (nomes) de objetos em diferentes áreas de um programa – essa duplicidade de nomes também é chamada de **colisão de nomes**. Dê uma nova olhada no capítulo 1, seção 1.8, do Ebook do módulo básico deste curso: reveja o **Zen do Python** – um de seus postulados é: "*Namespaces* são uma grande ideia - vamos fazer mais deles!". Neste capítulo vamos falar mais sobre o que é um *namespace*.

16.2 CONCEITOS DE PACOTE E BIBLIOTECA

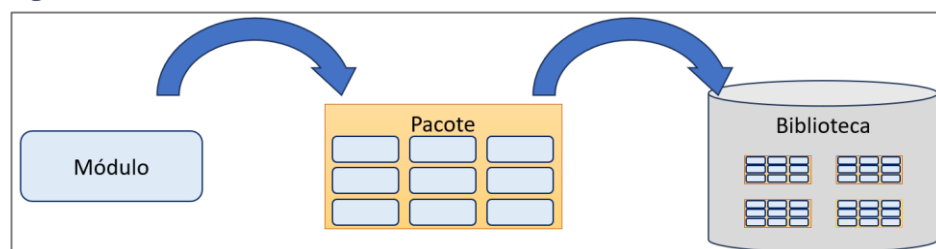
Se você entendeu o que é um módulo, entender pacote será bem simples: em Python um **Pacote** (**package**) é um grupo de módulos. O conceito é esse e mais adiante veremos como criar pacotes.

Alguns autores se referem a módulos e pacotes usando o termo "biblioteca". Não há nada errado em usar esse termo e isso ocorre porque nas linguagens de programação mais antigas esse é um termo historicamente usado. De forma geral pode-se pensar que módulos prontos para uso constituem uma biblioteca à disposição do programador.

Em Python, no entanto, precisamos compreender corretamente a diferença entre módulo e pacote porque isso afeta a forma como os organizamos e salvamos no disco do computador e também como faremos sua importação nos programas.

Para evitar confusão neste texto, vamos sempre usar o termo "módulo" quando nos referirmos a um arquivo a ser importado. O termo "pacote" será usado quando precisarmos nos referir a um grupo de módulos. Por fim, o termo "biblioteca" será usado para grupos constituídos por vários pacotes.

Figura 16.1 – Módulos, Pacotes e Bibliotecas



fonte: o Autor

16.3 CRIAÇÃO DE UM MÓDULO

Há três diferentes formas de módulos em Python:

1. Módulos embutidos (***built-in modules***), que estão contidos dentro do interpretador;
2. Módulos que são escritos em linguagem C e podem ser carregados no Python dinamicamente em tempo de execução do interpretador;
3. Módulos escritos em linguagem Python; neste curso nosso interesse é por este modo.

Os módulos podem conter quaisquer elementos Python, tais como: objetos, funções e classes. Neste capítulo vamos ver módulos com objetos e funções. No próximo capítulo veremos as classes.

O mais interessante dos módulos escritos em Python é que eles são extremamente simples de construir. Tudo o que você precisa fazer é criar um arquivo que contenha um código Python correto e salvá-lo em um arquivo com extensão .py em algum diretório (pasta). Nenhuma sintaxe especial é necessária.

Atenção
Nos nomes dos módulos use apenas
letras minúsculas, algarismos e underline.

Veja o exemplo 16.1 a seguir. Escreva este código e salve-o com o nome "utilidades.py" em um diretório de sua escolha. Usaremos o diretório "C:\CursoPython".

Exemplo 16.1 – Parte 1: módulo utilidades.py



```
""" Este é o módulo utilidades
Ele contém objetos e funções que podem ser úteis aos nossos programas.
É recomendável que haja um docstring indicando o conteúdo do módulo"""

texto = 'Este é o módulo utilidades.py'

meses = ['jan', 'fev', 'mar', 'abr', 'mai', 'jun',
        'jul', 'ago', 'set', 'out', 'nov', 'dez']

def paridade(valor):
    """Retorna PAR ou ÍMPAR conforme o valor passado"""
    if valor % 2 == 0:
        return 'PAR'
    else:
        return 'ÍMPAR'

def primo(valor):
    """Se valor for primo retorna True, senão retorna False"""
    if valor == 2:
        return True
    elif valor % 2 == 0:
        return False
    else:
        raiz = pow(valor, 0.5)
        i = 3
        while i <= raiz:
            if valor % i == 0:
                return False
            i += 2
        return True
```

(Isto é um módulo. Ele deve ser escrito e salvo dentro de algum diretório)

Como você pode ver neste código, o módulo `utilidades.py` possui quatro elementos:

- o string `texto`;
- a lista `meses`;
- a função `paridade()` que retorna um string indicando se um número é par ou ímpar;
- a função `primo()` que retorna `True` se um número é primo ou `False`, caso contrário;

Note que nesse arquivo não existe o que seria a "parte principal" do programa. Isso ocorre porque esse código é um módulo que será importado em um outro programa (que será o principal). No entanto, é possível que um módulo tenha também uma parte principal, mas isso será abordado mais adiante.

Nos módulos é usual, recomendável e considerado uma boa prática que em seu início haja um *docstring* (comentário de múltiplas linhas) indicando qual é a finalidade do módulo. Também para cada função recomenda-se colocar um *docstring* esclarecendo o que a função faz.

Depois de escrito o módulo é hora de testá-lo. Para isso vamos usar a interatividade do Idle e depois escrevemos um programa em arquivo à parte no qual faremos a importação e uso do módulo.

16.3.1 USO INTERATIVO DO MÓDULO `utilidades.py` NO IDLE

Após escrever o módulo do exemplo 16.1, abra o Idle e tente importá-lo.

Exemplo 16.1 – Parte 2: uso do módulo `utilidades` no Idle

[veja o vídeo 16.1](#)

```
import utilidades
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import utilidades
ModuleNotFoundError: No module named 'utilidades'
(exemplo interativo feito com IDE Idle)
```

Provavelmente você obterá a mensagem de erro mostrada acima. A importação falhou porque o interpretador Python não conseguiu localizar o módulo. Precisamos informar ao interpretador onde encontrá-lo. Há diferentes formas de fazê-lo e na seção 16.4 isso será visto em detalhes. Para fazermos testes vamos alterar o diretório corrente do Idle. Isso pode ser feito usando um método do módulo `os`. No caso, vamos usar `os.chdir()` para alterar o diretório corrente no Idle, fazendo com que o Idle "enxergue" o módulo.

Exemplo 16.1 – Parte 3: uso do módulo `utilidades` no Idle

[veja o vídeo 16.1](#)

```
>>> import os      # importa o módulo os (operating system)
>>> os.getcwd()    # consulta o diretório corrente
'C:\\Users\\sergi\\AppData\\Local\\Programs\\Python\\Python312'
>>> os.chdir('C:\\CursoPython') # altera o diretório corrente
>>> os.getcwd()
'C:\\CursoPython'
>>> import utilidades # agora conseguimos importar o módulo
>>> utilidades.texto  # acesso ao string texto
'Este é o módulo utilidades.py'
>>> utilidades.meses  # acesso à lista meses
['jan', 'fev', 'mar', 'abr', 'mai', 'jun', 'jul', 'ago', 'set', 'out', 'nov',
 'dez']
>>> utilidades.paridade(17) # acesso à função paridade()
'ÍMPAR'
>>> utilidades.primo(17)   # acesso à função primo()
True
(exemplo interativo feito com IDE Idle)
```

Outro uso possível para o nosso módulo utilidades é importá-lo em um programa como mostrado a seguir. É importante que você salve o arquivo desse programa no mesmo diretório onde salvou o módulo. Se não fizer isso, ocorrerá erro na importação.

Exemplo 16.1 – Parte 4: programa que usa o módulo utilidades

teste este código no PyCharm

```
import utilidades

x = int(input('Digite um inteiro: '))
while x != 0:
    parid = utilidades.paridade(x) # chamada à função do módulo (é necessário
    xPrimo = utilidades.primo(x)   # usar o <nome módulo>.<nome elemento>)
    print(f'{x} é {parid}')
    if xPrimo:
        print(f'{x} é primo')
    else:
        print(f'{x} não é primo')
    x = int(input('Digite um inteiro: '))

print('\nFim do Programa')
Digite um inteiro: 317
317 é ÍMPAR
317 é primo

Digite um inteiro: 225
225 é ÍMPAR
225 não é primo

Digite um inteiro: 86
86 é PAR
86 não é primo

Digite um inteiro: 0

Fim do Programa
```

Este programa importa o módulo `utilidades`. A partir dessa importação, todos os elementos do módulo – funções, classes, objetos, etc – passam a estar disponíveis ao programa, através da notação qualificada com o nome do módulo. Ou seja, a maneira correta de realizar a chamada é escrever a notação `<nome módulo>.<nome elemento>`, como mostrado acima no exemplo. Com isto você já pode começar a criar seus próprios módulos, e terá condição de organizar seus programas mais complexos dividindo-os em unidades lógicas menores e juntando-as em um programa principal através da importação.

16.4 CAMINHO DE PESQUISA DOS MÓDULOS

Como você viu acima, não basta que o módulo exista. Ele também deve ser encontrado pelo interpretador Python. Se não encontrar, o interpretador gera um erro, como foi visto no vídeo do exemplo 16.1.

Quando o interpretador Python executa um `import`, ele procura o módulo nos seguintes locais:

- O diretório a partir do qual o script de entrada foi executado ou o diretório corrente se o interpretador estiver sendo executado interativamente;
- A lista de diretórios contidos na variável de ambiente `PYTHONPATH`, se estiver definida. O formato para `PYTHONPATH` vai variar conforme o sistema operacional que você estiver usando;
- Uma lista de diretórios configurados no momento da instalação do Python;

O caminho de pesquisa resultante está acessível na variável Python `.path`, que é obtida de um módulo chamado `.path`. Experimente usar o `sys.path` como mostrado no exemplo 16.2.

Exemplo 16.2



```
>>> import sys
>>> for c in sys.path:
>>>     print(c)
C:\Users\sergi\AppData\Local\Programs\Python\Python312\Lib\idlelib
C:\Users\sergi\AppData\Local\Programs\Python\Python312\python312.zip
C:\Users\sergi\AppData\Local\Programs\Python\Python312\DLLs
C:\Users\sergi\AppData\Local\Programs\Python\Python312\Lib
C:\Users\sergi\AppData\Local\Programs\Python\Python312
C:\Users\sergi\AppData\Local\Programs\Python\Python312\Lib\site-packages
(exemplo interativo feito com IDE Idle)
```

Os caminhos mostrados neste exemplo são os que estão configurados na máquina onde este texto foi escrito. No caso, trata-se de uma máquina com sistema operacional MS-Windows.

Lembre-se

O conteúdo exato de `sys.path` depende do ambiente e da instalação.
O texto acima com certeza será diferente no seu computador.

Para ter certeza de que o seu módulo será encontrado pelo interpretador adote uma dessas ações:

- Colocar o módulo no mesmo diretório do programa principal ou, caso esteja usando o Idle interativo colocá-lo no diretório atual;
- Modificar a variável de ambiente `PYTHONPATH` para incluir o diretório onde o módulo está localizado ou colocar o módulo em um dos diretórios já contidos na variável `PYTHONPATH`, antes de iniciar o interpretador;
- Colocar o módulo em um dos diretórios configurados durante a instalação do Python, ao qual você pode ou não ter acesso de gravação, dependendo do sistema operacional;
- Colocar o arquivo do módulo em qualquer diretório de sua escolha e então modificar o atributo `sys.path`, que é uma lista, fazendo a inclusão desse diretório com `.append()`.

Nós usaremos esta última alternativa porque ela é bem prática e independe de qual sistema operacional esteja em uso. Como o nosso módulo está no diretório `C:\CursoPython` então basta executar o código mostrado no exemplo 16.3.

Exemplo 16.3



```
>>> import utilidades
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    import utilidades
ModuleNotFoundError: No module named 'utilidades'

>>> import sys
>>> sys.path.append('C:\CursoPython')      # acrescentando o caminho à sys.path
>>> import utilidades                      # a importação funcionou
>>> utilidades.primo(3)
True
(exemplo interativo feito com IDE Idle)
```

16.5 VISIBILIDADE E ORGANIZAÇÃO DO CÓDIGO PYTHON

Antes de prosseguirmos com o nosso aprendizado sobre módulos precisamos compreender conceitos de Python relativos à visibilidade e organização do nosso código.

16.5.1 MOTIVAÇÃO

Tudo o que fizemos neste curso até este momento mostra a relevância e a importância dos objetos em Python. Os objetos estão por toda parte e, em última análise, tudo o que o seu programa Python cria e atua é um objeto.

Identificador ou Nome ?

Em vários momentos deste texto usamos um desses dois termos.
Ambos se referem à mesma coisa: o nome de um objeto.

Então, quando escrevemos

```
texto = 'algum texto aqui'
```

```
valor = 115.38
```

As duas palavras – `texto` e `valor` – são o que designamos como
identificador ou nome.

Uma simples instrução de atribuição cria um identificador e faz referência a um objeto que é criado na memória do computador. Uma instrução como `txt = 'algo'` cria um nome simbólico `txt` que se refere ao objeto que contém o string `'algo'`.

Em um programa de qualquer complexidade, você criará dezenas, centenas ou até milhares desses nomes, cada um levando para um objeto específico em memória. O ponto que queremos ressaltar aqui é sobre como o Python gerencia todos esses nomes. Como isso é feito?

A resposta são os conceitos de **Namespace** e **Escopo**

16.5.2 NAMESPACE

Namespace é uma coleção de diferentes objetos criados em memória sendo que cada um está associado a um nome. Esses nomes são únicos em determinado momento, ou seja, não pode haver dois ou mais nomes iguais nesse momento.

Em termos práticos você pode pensar em um namespace como um dicionário no qual a chave é o nome do objeto e o valor é o próprio objeto. Cada par de chave-valor mapeia um nome para seu objeto correspondente, que já está criado e, portanto, presente na memória do computador naquele momento.

Quando um programa Python está em execução há quatro categorias de namespaces a serem considerados.

O namespace built-in

Este namespace contém os nomes de todos os objetos embutidos do interpretador Python. Tais objetos estão disponíveis sempre que o Python está em execução e é possível listá-los com o comando `dir(__builtins__)` como mostrado no exemplo a seguir.

A função `dir()` é usada para listar os nomes definidos dentro de algum escopo. Seu retorno é uma lista de strings organizada em ordem alfabética. Usada sem argumentos, ela retorna a lista de nomes no escopo local atual. Por outro lado, se for usada com um argumento, ela tentará retornar uma lista de

atributos válidos para esse argumento. No exemplo, por uma questão de espaço, estamos exibindo apenas parte do resultado, mas observe bem o que é retornado pela `dir(__builtins__)`:

Exemplo 16.4



```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeError', 'BufferError',
 ...
 'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray',
 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex',
 'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate',
 'eval', 'exec', 'exit', 'filter', 'float', 'format', 'frozenset',
 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
 'int', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod',
 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Esta lista é extensa então reproduzimos apenas parte dela. Observe os nomes acima, você reconhecerá vários deles que já usou.

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Acima destacamos em negrito funções que já foram usadas diversas vezes neste curso, como `print()`, `id()`, `type()`, `len()`, `min()`, `max()` e a própria função `dir()` que usaremos bastante neste capítulo.

O Python cria este namespace quando é inicializado e ele permanece existente até que o interpretador termine. Este namespace é único, ou seja, não existirão dois deles em uma execução do interpretador.

O namespace global

O **namespace global** contém quaisquer nomes definidos no âmbito do programa principal. Mas este não é o único namespace global que existe. Cada módulo que seja carregado durante a execução do programa também terá seu namespace global.

No caso do programa principal, o interpretador cria seu namespace global quando o corpo principal do código é iniciado. Esse namespace permanece existente até que o interpretador termine. No caso de um módulo, seu namespace global é criado apenas quando o módulo é carregado com o comando `import` e permanece existente até que o programa termine.

Exemplo 16.4 - continuação



```
>>> import utilidades
>>> dir(utilidades)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'meses', 'paridade', 'primo', 'texto']
```

(exemplo interativo feito com IDE Idle)

Nesta continuação do exemplo 16.4 fazemos a importação do módulo `utilidades` e listamos com `dir()` os objetos de seu namespace.

Essa categoria global de namespace às vezes pode induzir um raciocínio incorreto no estudante de Python. Ao usar o termo "global" pode-se passar a impressão de que ele está associado ao programa principal e que deve existir apenas um namespace global. Mas não é esse o conceito. Todo programa e todo módulo terá seu namespace global.

Ficará mais simples de entender depois que virmos as próximas duas categorias.

O namespace local

Sempre que estamos trabalhando com funções, o interpretador cria um novo namespace sempre que uma função é executada. Esse é o **namespace local** e ele tem validade para a função apenas enquanto ela está em execução. Quando a função terminar seu namespace e quaisquer objetos que tenham sido criados dentro dele são destruídos. O exemplo 16.5 mostra a relação entre os três namespace: built-in, global e local.

Exemplo 16.5



```
def soma(a, b):
    print('\nParte 3 - Escopo local - função soma')
    print(dir())
    return a + b

def diferenca(a, b):
    print('\nParte 3 - Escopo local - função diferenca')
    print(dir())
    return a - b

print('Parte 1 - Exibição do namespace built-in')
print(dir(__builtins__))
v1 = 26
v2 = 15
print('\nParte 2 - Namespace global do programa - exceto elementos com "__"')
for s in dir():
    if '__' not in s:
        print(s, end=', ')
print()
rs = soma(v1, v2)
rd = diferenca(v1, v2)
print(f'\n{v1} + {v2} = {rs}')
print(f'{v1} - {v2} = {rd}')
```

Parte 1 - Exibição do namespace built-in
['ArithmeticError', 'AssertionError', 'AttributeError', ... (truncado pelo autor)]

Parte 2 - Namespace global do programa - exceto elementos com "__"
diferenca, soma, v1, v2,

Parte 3 - Escopo local - função soma
['a', 'b']

Parte 3 - Escopo local - função diferenca
['a', 'b']

26 + 15 = 41
26 - 15 = 11

As linhas em negrito são um destaque feito pelo autor

Na execução deste exemplo estão presentes as três categorias de namespace – built-in, global, e local. Assim que o programa inicia são exibidos os nomes presentes no namespace built-in que é o mais externo.

Em seguida os objetos `v1` e `v2` são criados através da atribuição de valores e em conjunto com as funções `soma()` e `diferença()` eles estão contidos no namespace global do programa.

Em um terceiro nível estão os namespaces locais das funções `soma()` e `diferença()`. Note que os dois possuem os objetos `a` e `b`, porém não há risco de colisão de nomes porque esses objetos só existem quando a função está sendo executada. Dados que ambas não são executadas ao mesmo tempo, não há risco de conflito.

Compreendendo melhor os namespaces local e global

No que diz respeito ao código interno de uma função existe um conhecimento relacionado aos namespaces local e global que todo programador Python deve conhecer e dominar. Considere o seguinte:

- Todos os objetos globais estão disponíveis "para leitura" dentro da função;
- Toda atribuição feita dentro da função criará um objeto local;
- Se a função precisar atribuir um valor a um objeto global, então a diretiva **global** deve ser usada;

Essas afirmações são sempre verdadeiras e o exemplo a seguir é uma ilustração delas.

Exemplo 16.6



```
def funcao1():
    print('...início da funcao1')
    if criterio == 'alterar':
        valor = 999
    print(f'valor dentro de funcao1 = {valor}')
```

```
def funcao2():
    print('...início da funcao2')
    global valor # faz referência ao objeto global valor
    if criterio == 'alterar':
        valor = 999
    print(f'valor dentro de funcao2 = {valor}')
```

```
criterio = 'alterar'
valor = 10
print(f'pgm principal valor = {valor} (antes funcao1)')
funcao1()
print(f'pgm principal valor = {valor} (apos funcao1)')
print()
print(f'pgm principal valor = {valor} (antes funcao2)')
funcao2()
print(f'pgm principal valor = {valor} (apos funcao2)')
```

```
pgm principal valor = 10 (antes funcao1)
...início da funcao1
valor dentro de funcao1 = 999
pgm principal valor = 10 (apos funcao1)
```

```
pgm principal valor = 10 (antes funcao2)
...início da funcao2
valor dentro de funcao2 = 999
pgm principal valor = 999 (apos funcao2)
```

No programa principal são criados dois objetos: `criterio` que contém o texto 'alterar' e valor que contém o inteiro 10. O objeto `criterio` é usado apenas "para consulta", ou seja, não estamos interessados em alterar seu conteúdo, mas precisamos verificar seu conteúdo em uma condição. Já o objeto `valor` tem seu conteúdo alterado dentro das funções.

Como o programa roda normalmente, fica claro que o objeto `criterio` está disponível dentro da função. E como ele não foi criado localmente a única conclusão é que se trata do objeto pertencente ao programa principal, o objeto no namespace global.

Agora vamos analisar o que ocorre com o objeto `valor`, cujo conteúdo precisa ser alterado. Dentro de `funcao1()` fizemos a atribuição `valor = 999`. Essa atribuição provoca a criação de um novo objeto local totalmente desassociado do objeto global de mesmo nome. Com isto, a atribuição feita dentro de `funcao1()` não teve qualquer efeito sobre o objeto global `valor`.

Por outro lado, dentro de `funcao2()` incluímos a declaração `global valor`. Isso cria um vínculo da função com o objeto global para efeitos de alterações de conteúdo. Essa declaração garante que qualquer atribuição feita ao nome `valor` irá afetar diretamente o objeto global.

O namespace não-local (nonlocal)

Inicialmente, cuidado! Poderíamos usar o termo não-local em português (em inglês são comuns os termos *non-local* namespace ou *enclosing* namespace) mas não imagine que esse termo seja uma negação de local e, portanto, global. Em Python namespace não-local é um conceito a mais nesse assunto.

Para evitar confusão com termos vamos adotar o termo **namespace nonlocal** para esse conceito.

Um namespace nonlocal só existirá nas situações em que tenhamos funções aninhadas. Isso mesmo, uma função criada dentro de outra função. Veja o exemplo 16.7. Neste código temos a função `soma()`. Além disso, dentro de `soma()` foi criada a função aninhada `calcula()`.

Exemplo 16.7

[Teste este código no PyCharm](#)

```
def soma(a, b):  
  
    def calcula(d1, d2):  
        print('\nEscopo local - função calcula dentro da função soma')  
        print(dir())  
        return d1 + d2  
  
    print('\nEscopo local - função soma')  
    print(dir())  
    return calcula(a, b) # faz uma chamada à função aninhada calcula()  
  
v1 = 26  
v2 = 15  
r = soma(v1, v2)  
print(f'\nsoma de {v1} com {v2} = {r}')
```

```
Escopo local - função soma  
['a', 'b', 'calcula']  
  
Escopo local - função calcula dentro da função soma  
['d1', 'd2']  
  
soma de 26 com 15 = 41
```

Quando o programa principal chama `soma()`, Python cria um namespace local para `soma()`.

Da mesma forma, quando `soma()` chama `calcula()`, `calcula()` obtém seu próprio namespace separado. O namespace criado para `calcula()` é o namespace local. Nessa situação o namespace local de `soma()` é o namespace nonlocal para `calcula()`. Assim, dentro de uma função aninhada, além de seu namespace local, estará disponível o namespace da função que a contém, que é o nonlocal.

O exercício resolvido 16.1 ilustra a relação entre os namespaces global, nonlocal e local.

Observação sobre a solução do exercício resolvido 16.1
A solução apresentada é bem mais complicada do que precisaria ser. Porém, o objetivo são os namespaces, não escrever o melhor programa.

Vamos apresentar duas versões para a solução deste enunciado e a versão 1 **está incorreta quanto ao resultado fornecido**. Note que o programa é executado normalmente, sem qualquer ocorrência de erro, porém o resultado produzido está incorreto. Veja a seguir:

Exercício Resolvido 16.1 – Versão 1: solução incorreta



Enunciado: Escreva um programa que exiba na tela o código, o preço de custo e o preço de venda de produtos, sabendo que a margem bruta de lucro aplicada pela empresa é de 16% para a maioria dos produtos. Alguns produtos podem ter essa margem reduzida e nesses casos o código do produto tem seu primeiro dígito trocado. Quando o primeiro dígito for '8' a margem é 12% e quando esse dígito for '9' a margem é 10%.

Dados para testes

Código	Preço de custo	Preço de venda
1280	100.00	116.00
8280	100.00	112.00
9280	100.00	110.00

```
def valor_venda(codigo, val_custo):

    def altera_margem(): # função aninhada que altera o valor da margem
        if codigo[0] == '8':
            margem = 12/100
        elif codigo[0] == '9':
            margem = 10/100

    margem = 16/100 # estabelece a margem em 16%
    altera_margem() # altera valor da margem conforme o código
    val_venda = val_custo * (1 + margem) # calcula valor de venda
    return val_venda # e retorna

PcCusto = 100
CodProd = '1280'
PcVenda = valor_venda(CodProd, PcCusto)
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')

CodProd = '8280'
PcVenda = valor_venda(CodProd, PcCusto)
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')

CodProd = '9280'
PcVenda = valor_venda(CodProd, PcCusto)
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')

Produto 1280: compra = 100.00 e venda = 116.00
Produto 8280: compra = 100.00 e venda = 116.00
Produto 9280: compra = 100.00 e venda = 116.00
```

Neste resultado os preços de venda estão todos com 16% de margem, mesmo para os códigos com dígito inicial '8' e '9'. O programa principal fez três chamadas à função `valor_venda()` e em cada uma passou os parâmetros corretos. Dentro dessa função o objeto `margem` foi carregado com 16% e em seguida a função aninhada `altera_margem()` foi chamada.

Vamos nos concentrar nessa função aninhada. Perceba que em seu código ela utiliza os objetos `codigo` e `margem`, sem que eles tenham sido definidos dentro dela.

Por quê, então, não ocorreu erro no interpretador?

Resposta simples → não ocorre erro porque esses objetos estão disponíveis para `altera_margem()`.

```
def altera_margem():
    if codigo[0] == '8':
        margem = 12/100
    elif codigo[0] == '9':
        margem = 10/100
```

Resposta não tão simples → não ocorre erro por dois motivos diferentes:

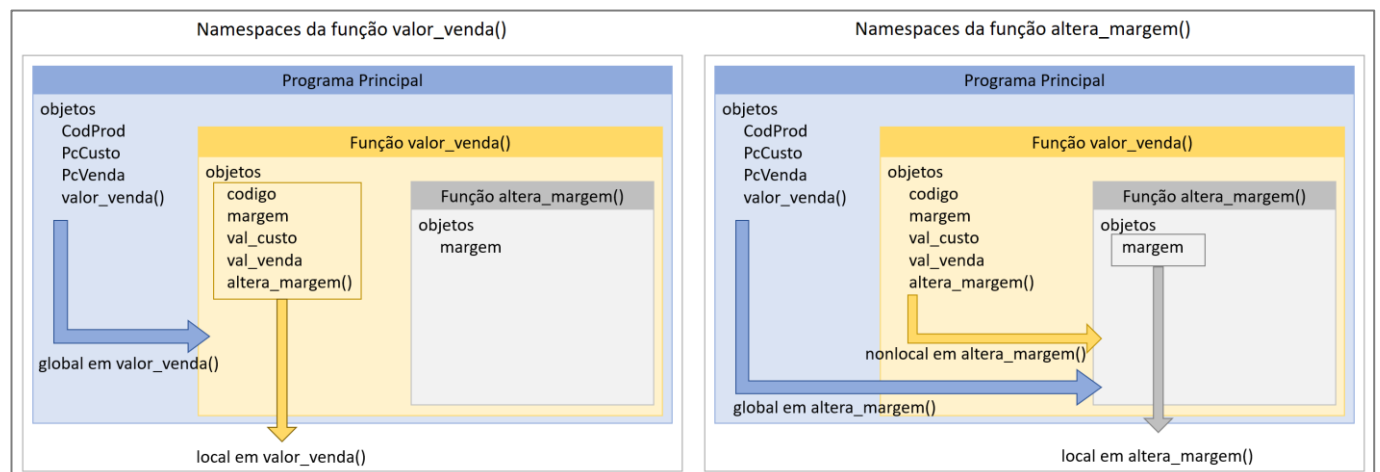
- não ocorre erro relacionado ao objeto `margem` porque ele está sendo atribuído dentro da função e, portanto, ele é um objeto local que será criado no momento da atribuição;
- não ocorre erro relacionado ao objeto `codigo` porque, embora não seja definido dentro de `altera_margem()` ele existe na função `valor_venda()`. É como se os objetos de `valor_venda()` fossem globais para `altera_margem()` e a eles damos o nome de não-locais (usaremos o termo `nonlocal`), pois o termo `global` é reservado para objetos do programa principal.

A figura 16.2 ilustra a situação que ocorre nesse programa.

Na situação à esquerda temos o "ponto de vista" da função `valor_venda()` (amarela) e para essa função os objetos da área amarela são locais e os da área azul são globais.

Na situação à direita temos o "ponto de vista" da função `altera_margem()` (cinza) onde os objetos da área cinza são locais, os da área amarela são não-locais e os da área azul são globais.

Figura 16.2 – Ilustração de namespaces global, nonlocal e local para a Versão 1 da solução do exemplo 16.7



fonte: o Autor

Agora que conhecemos a relação local – nonlocal – global vamos entender o erro do exercício resolvido 16.1 versão 1.

Observe na figura que o ambiente da função `valor_venda()` (área amarela) contém um objeto com o nome `margem`; e o ambiente da função `altera_margem()` (área cinza) também contém um objeto com esse nome, que é criado no momento da atribuição do valor.

Esse é o motivo pelo qual essa versão do programa está produzindo os resultados errados. Quando a função `altera_margem()` é chamada ela deveria alterar o objeto `margem` da área amarela e não criar um novo objeto local.

Para resolver esse problema é preciso que a função `altera_margem()` faça a alteração diretamente no objeto `margem` que pertence à função `valor_venda()`.

O acesso aos objetos do namespace não-local se faz pela diretiva **nonlocal**.

Exercício Resolvido 16.1 – Versão 2: **solução correta**

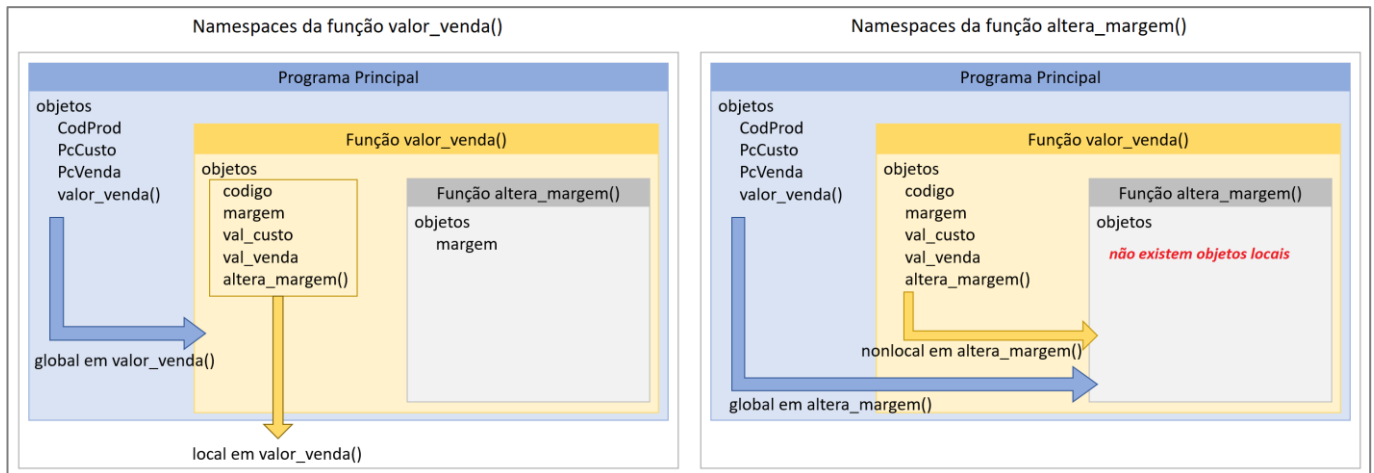
A apresentação das duas versões está no mesmo vídeo

```
def valor_venda(codigo, val_custo):  
  
    def altera_margem(): # função aninhada que altera o valor da margem  
        nonlocal margem # essa declaração deve ser incluída  
        if codigo[0] == '8':  
            margem = 12/100  
        elif codigo[0] == '9':  
            margem = 10/100  
  
    margem = 16/100 # estabelece a margem em 16%  
    altera_margem() # altera valor da margem conforme o código  
    val_venda = val_custo * (1 + margem) # calcula valor de venda  
    return val_venda  
  
PcCusto = 100  
CodProd = '1280'  
PcVenda = valor_venda(CodProd, PcCusto)  
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')  
CodProd = '8280'  
PcVenda = valor_venda(CodProd, PcCusto)  
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')  
CodProd = '9280'  
PcVenda = valor_venda(CodProd, PcCusto)  
print(f'Produto {CodProd}: compra = {PcCusto:.2f} e venda = {PcVenda:.2f}')  
Produto 1280: compra = 100.00 e venda = 116.00  
Produto 8280: compra = 100.00 e venda = 112.00  
Produto 9280: compra = 100.00 e venda = 110.00
```

Após essa alteração a realidade do programa é outra e está ilustrada na figura 16.3 onde está indicado que na função `altera_margem()` não existem objetos locais.

De fato, após a alteração nenhum novo objeto é criado dentro dessa função e ela apenas utiliza os objetos que estão em seu namespace não-local.

Figura 16.3 – Ilustração de namespaces global, nonlocal e local para a Versão 2 da solução do exemplo 16.7



fonte: o Autor

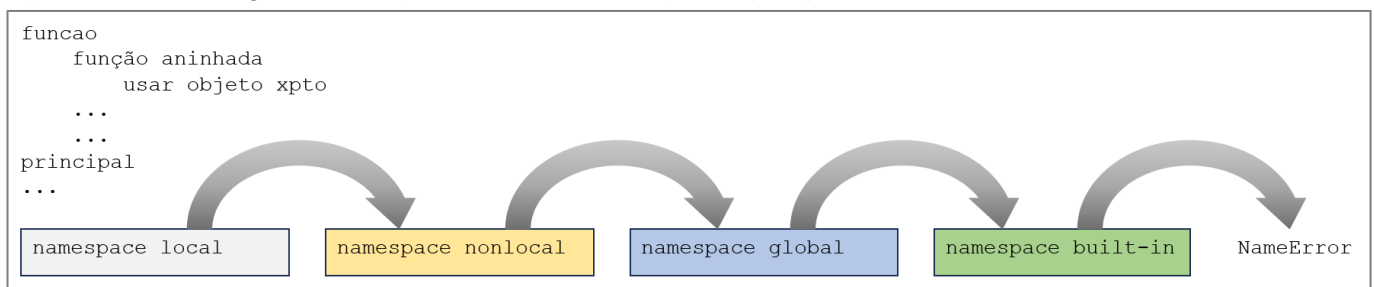
16.5.3 ESCOPO E TABELAS DE SÍMBOLOS

No capítulo 12 sobre funções já falamos brevemente sobre escopo. Vamos agora nos aprofundar.

Agora que já conhecemos o conceito de namespace vamos tratar do conceito de escopo. Muitos autores que escrevem sobre a linguagem Python tratam esses dois conceitos como sinônimos. Não chegam a estar errados pois os dois conceitos estão intimamente associados. Porém existe uma diferença sutil que acreditamos que vale a pena conhecer e que vamos apresentar agora.

- Como vimos um **namespace** é um dicionário que contém os pares chave-valor relacionando nomes de objetos com os objetos em memória, visíveis e prontos para uso. Quando você faz uma atribuição, como `obj = 1`, você está modificando um namespace. Quando você faz uma referência, como `print(obj)`, o Python examina uma lista de namespaces para tentar encontrar um objeto que tenha o nome `obj` como chave do dicionário.
- O **escopo** define quais namespaces serão pesquisados e em que ordem. O escopo de qualquer referência sempre começa no namespace local e se move para os níveis acima até atingir o namespace built-in, o nível mais alto criado na inicialização do interpretador Python.

Figura 16.4 – Ilustração de Escopo - ordem de namespaces pesquisados



fonte: o Autor

Assim, quando dizemos que o objeto `xpto` está no namespace de uma função, queremos dizer que ele foi criado localmente dentro da função. Quando dizemos que `xpto` está no escopo da função,

queremos dizer que `xpto` está no namespace da função **ou** em qualquer um dos namespaces externos nos quais o namespace da função está atualmente aninhado.

Em síntese: o que é **Escopo**?

Diz respeito à visibilidade dos elementos de um programa Python.

Se um nome está disponível ao interpretador, então dizemos que o nome está dentro do escopo atual.

Sempre que você define uma função, você cria um novo namespace e um novo escopo. O namespace é o novo dicionário de armazenamento de pares nome-objeto. O escopo é a cadeia implícita de namespaces que começa nesse novo namespace e, em seguida, segue o caminho descrito através de quaisquer namespaces externos (escopos externos), até o namespace global (o escopo global) e por fim o built-in do interpretador.

A pesquisa de objetos em um escopo é implementada através das **tabelas de símbolos**. Essas tabelas são as estruturas de dados em forma de dicionário que mencionamos quando apresentamos os namespaces. O interpretador Python constrói em memória essas tabelas e as mantém dinamicamente atualizadas à medida que elementos – objetos, funções, classes, etc – sejam criados, importados, alterados ou destruídos.

No exemplo 16.8 mostramos como investigar a disponibilidade de objetos, ou seja, investigar namespaces para saber se o objeto está visível e, portanto, dentro do escopo.

Os termos **Namespace** e **Escopo**

podem ser usados quase indistintamente, não porque signifiquem a mesma coisa,

mas porque eles se sobrepõem muito no tocante ao uso.

Essa investigação pode ser feita com a função `dir()` e no exemplo a usamos diversas vezes para ver o que ocorre à medida que interagimos com o interpretador Python. Sugerimos que antes de prosseguir na leitura desse texto, assista o vídeo desse exemplo, analise o código e reproduza-o no Idle. Ao fazer isso você compreenderá com mais facilidade as explicações que se seguem.

Nas partes de 1 a 3 do exemplo 16.4 usamos `dir()` sem argumentos para listar os objetos existentes no namespace atual (onde `dir()` está sendo usado). Quando essa função é usada com um argumento, ela mostrará os objetos existentes no namespace do argumento que foi passado.

A parte 1 registra a situação em que o Idle acabou de ser aberto nenhum objeto foi criado. Assim, nesse momento o namespace do interpretador contém apenas os objetos criados automaticamente em sua inicialização. Você pode perceber que há um grupo de nomes que iniciam e terminam com dois caracteres underline no formato `__nome__`. Não se preocupe com isso agora. Mais adiante vamos falar sobre eles e as situações em que são usados.

Na parte 2 do exemplo criamos os objetos `a`, `b` e `txt`. Na sequência usamos novamente a função `dir()`. Perceba que esses identificadores foram incluídos na listagem de objetos disponíveis. Isso quer dizer que esses objetos agora fazem parte namespace e, portanto, do escopo.

Na parte 3 importamos o módulo `sys` (necessário para alterar `sys.path`) e em seguida importamos nosso módulo `utilidades`. Usando `dir()` pela terceira vez vemos que o namespace global agora contém esses dois módulos.

Exemplo 16.8



Parte 1

```
>>> dir() # Namespace global de um interpretador recém iniciado
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__']
```

Parte 2

```
>>> a = 236 # criação de alguns objetos
>>> b = 17.3
>>> txt = 'algum texto'
>>> dir() # os objetos criados estão presentes na tabela
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'a', 'b', 'txt']
```

Parte 3

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> import utilidades
>>> dir() # agora 'sys' e 'utilidades' também estão incluídos no namespace global
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'a', 'b', 'sys', 'txt', 'utilidades']
```

Parte 4

```
>>> dir(utilidades) # inspeção do namespace interno ao módulo utilidades
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', 'meses', 'paridade', 'primo', 'texto']
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Por fim, na parte 4 usamos `dir(utilidades)`. Ao acrescentar um nome como argumento estamos solicitando a listagem dos elementos internos ao módulo, ou seja, queremos conhecer quais elementos existem dentro do módulo. Esse conjunto de elementos é o namespace do módulo `utilidades` e ali estão presentes o objetos `texto` e `meses`, também as funções `paridade()` e `primo()`.

Faça testes adicionais com a função `dir()`. Experimente usá-la com os objetos `a`, `b` e `txt`. Você verá que serão listados seus elementos internos.

16.6 DETALHES SOBRE O COMANDO `import`

Agora que já nos aprofundamos bastante nos conceitos de namespace e escopo temos condições de entender em detalhes as implicações do uso do comando `import`.

16.6.1 COMANDO `import` – FORMA 1

Vamos nos referir com frequência ao código que fará a chamada do módulo e para facilitar a redação do texto vamos usar a palavra "chamador" para nos referir ao local do código onde o comando `import` é executado.

Temos usado a forma mais básica de importação que é essa mostrada na linha a seguir. Vamos chamá-la de forma 1.

```
import <nomedomódulo>
```

Este tipo de execução torna o módulo disponível para o chamador, porém não torna o conteúdo do módulo diretamente disponível. Isso significa que essa forma de chamada insere o módulo no namespace do chamador, mas não insere seus elementos internos.

Notação com ponto

Também chamada de **qualificação**, é a notação usada na escrita do código quando não temos acesso direto a um elemento que pertence a um módulo.

Nessa notação usamos o nome do módulo como prefixo no formato `<módulo>.<elemento>` para acessar os elementos internos do módulo

A consequência disso é que o módulo estará no escopo do chamador, mas para usar seus elementos internos será necessário realizar as chamadas usando a **notação com ponto**. Veja no exemplo 16.9 que após a importação do módulo `utilidades` não conseguimos acessar diretamente seus objetos internos. Porém, ao usarmos o nome do módulo como prefixo esse acesso é possível.

Exemplo 16.9



```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> import utilidades

>>> texto
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    texto
NameError: name 'texto' is not defined.
>>> meses
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    meses
NameError: name 'meses' is not defined

>>> utilidades.texto
'Este é o módulo utilidades.py'
>>> utilidades.meses
['jan', 'fev', 'mar', 'abr', 'mai', 'jun', 'jul', 'ago', 'set', 'out', 'nov',
 'dez']

(exemplo interativo feito com IDE Idle)
```

Assim, após a importação do módulo usando essa forma 1 seus elementos internos estão acessíveis apenas com o uso da notação com ponto – `<módulo>.<objeto>`, pois apenas o nome do módulo foi acrescentado ao escopo do chamador.

16.6.2 COMANDO `import` – CUIDADO IMPORTANTE: EVITE COLISÃO DE NOMES

Atenção

O comando `import` sobrescreve qualquer elemento de mesmo nome pré-existente no escopo onde o comando é usado

Esteja atento ao usar o comando `import`, pois ele vai sobrescrever qualquer elemento de mesmo nome pré-existente. Veja o exemplo:

Exemplo 16.10



Parte 1

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> utilidades = 'Este objeto é um string'
>>> type(utilidades)
<class 'str'>
>>> print(utilidades)
Este objeto é um string
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'sys', 'utilidades']
```

Parte 2

```
>>> # Agora vamos importar o módulo utilidades
>>> import utilidades
>>> type(utilidades)
<class 'module'>
>>> print(utilidades)
<module 'utilidades' from 'C:\\CursoPython\\utilidades.py'>
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'sys', 'utilidades']
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Na parte 1 deste exemplo fizemos a preparação do ambiente e criamos um objeto chamado `utilidades` contendo um texto. Ou seja, é um objeto string e como pode ser visto ele está inserido no namespace do interpretador.

O primeiro comando da parte 2 é a importação do módulo `utilidades`. Você pode ver nos comandos subsequentes que o identificador `utilidades` continua inserido no escopo do interpretador, porém agora categorizado como `<class 'module'>`, ou seja, o mesmo nome agora está associado a um elemento totalmente diferente. O objeto de mesmo nome existente anteriormente foi descartado e seu conteúdo está perdido. E isso é feito normalmente pelo Python, sem qualquer aviso da existência do objeto anterior.

Esta situação que acabamos de exemplificar pode representar um problema significativo no tocante à organização do código de um sistema computacional, em especial aos de grande porte, que podem conter dezenas ou centenas de módulos e milhares de objetos, funções e classes. Quando uma situação como essa acontece dizemos que ocorreu uma Colisão de Nomes.

Colisão de Nomes

Esta é uma situação a ser evitada sempre.
Acontece quando o mesmo nome (identificador)
é usado para diferentes elementos de um software.

Em Python há algumas medidas que podem ser adotadas para evitá-la.

16.6.3 COMANDO `import` – FORMA 1 GENERALIZADA

É possível fazer a importação de múltiplos módulos em um único comando `import` bastando colocar os nomes dos módulos separados por vírgulas.

```
import <modulo1>, <modulo2>, <modulo3>
```

No exemplo 16.7 três módulos foram importados simultaneamente, `sys`, `os` e `math`. Em seguida a função `dir()` e elementos de cada um dos módulos são utilizados para mostrar que os três estão presentes no escopo do chamador. Abra o Idle e reproduza esse exemplo e você vai constatar seu funcionamento.

Exemplo 16.11

[Teste este código no Idle](#)

```
>>> import sys, os, math
>>> dir()
['_annotations_', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'math', 'os', 'sys']
>>> # Faça os testes abaixo. Perceba que os módulos estão disponíveis.
>>> sys.path.append('C:\CursoPython')
>>> os.getcwd()
'C:\\Users\\sergi\\AppData\\Local\\Programs\\Python\\Python312'
>>> math.pi
3.141592653589793
```

(exemplo interativo feito com IDE Idle – destaques em negrito feitos pelo autor)

16.6.4 COMANDO `import` – FORMA 1 COM APELIDO

É muito comum que os programadores Python façam a importação de um módulo adotando um apelido para ele. Em inglês o termo usado é "*alias*" e será comum você encontrá-lo nas documentações de Python em língua inglesa. A importação com apelido é feita com o uso da cláusula `as`. Isso nada mais é do que escolher usar um nome alternativo para o módulo, durante o ciclo de vida do programa em execução.

```
import <modulo> as <apelido>
```

Quando se usa esta opção é o apelido que será inserido na tabela de símbolos, como pode ser visto no exemplo 16.12. Internamente haverá um vínculo do apelido com o nome físico do módulo e qualquer referência ao apelido acionará os elementos do módulo real. Com essa opção, o nome físico do módulo não estará inserido no namespace do chamador, apenas o apelido.

Exemplo 16.12

[Teste este código no Idle](#)

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> import utilidades as util
>>> dir()
['_annotations_', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'sys', 'util']

>>> util.paridade(12) # usamos o módulo utilidades através do apelido util
'PAR'
>>> utilidades.paridade(12) # note que o nome utilidades não está disponível
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    utilidades.paridade(12)
NameError: name 'utilidades' is not defined

>>> print(util)
<module 'utilidades' from 'C:\\CursoPython\\utilidades.py'>
```

(exemplo interativo feito com IDE Idle – destaques em negrito feitos pelo autor)

Importar um módulo com apelido não causa mudança no nome físico do módulo, apenas atribui um nome temporário em tempo de execução. Isso pode ser visto quando fazemos `print(util)`, veja acima o que é exibido nesse print: o nome físico do módulo (caminho + nome do arquivo gravado no disco).

O uso de apelidos pode ser útil para evitar conflitos de nomes, melhorar a legibilidade do código ou fornecer um nome mais curto ou mais significativo para um módulo ou componente importado. Na seção 16.6.2 alertamos sobre a questão da substituição de um nome em uma importação. O uso de apelidos ajuda a contornar essa questão.

Por exemplo, considere que em seu software você tem o módulo `codigo_barras.py` no qual vem trabalhando há algum tempo e já contém funções e classes que usa bastante. No entanto, você precisa agora implementar um tipo de código de barras novo e descobre na comunidade Python que já existe um módulo que faz isso e, melhor, que seu uso é livre. Você então baixa esse módulo feito por terceiros e descobre que ele tem o mesmo nome que o seu módulo. Não é viável alterar o nome físico de um dos módulos, e você passa a usar um apelido para o novo módulo, evitando a colisão de nomes. Simples assim.

16.6.5 COMANDO `import` – FORMA 2

A segunda forma de uso do `import` é aquela em que importamos um ou mais elementos do módulo.

```
from <modulo> import <elemento1>, <elemento2>, ... <elementoN>
```

Esta forma permite que elementos individuais internos ao módulo sejam importados diretamente para a tabela de símbolos do chamador. Esta forma fará a inserção dos elementos listados diretamente no namespace do chamador e por consequência esse objeto estará diretamente disponível em seu escopo. Veja o exemplo 16.13:

Exemplo 16.13



```
>>> import sys
>>> sys.path.append('C:\CursoPython')

>>> from utilidades import paridade

>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'paridade', 'sys']
>>> print(paridade)
<function paridade at 0x0000016D2894F4C0>

>>> paridade(19)
'ÍMPAR'
```

(exemplo interativo feito com IDE Idle – destaques em negrito feitos pelo autor)

Ao usar essa segunda forma o método `paridade` foi inserido na tabela de símbolos tornando-se disponível para uso direto sem a necessidade do prefixo com o nome do módulo.

Esteja atento ao usar esta forma de importação para que não ocorram colisões de nomes. Lembre-se que se um nome já existir no namespace e ocorrer uma importação de objeto com o mesmo nome haverá a substituição desse nome anterior como mostrado a seguir, no exemplo 16.14.

Exemplo 16.14



```
# Parte 1 - preparação do ambiente
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> texto = '... string pré-existente ...' # um string
>>> paridade = 90 # um inteiro
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'paridade', 'sys', 'texto']
>>> print(texto)
... string pré-existente ...
>>> print(paridade)
90

# Parte 2 - importação com substituição de elementos existentes
>>> from utilidades import paridade, texto # importa 2 elementos de utilidades
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'paridade', 'sys', 'texto']
>>> print(texto)
# é string, mas mudou o conteúdo
Este é o módulo utilidades.py
>>> print(paridade)
# é uma função
<function paridade at 0x000001A5325D8FE0>
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Assim, com esse exemplo, fica clara a ocorrência da colisão. Os objetos `paridade` e `texto` foram substituídos no namespace do chamador após o `import`.

16.6.6 COMANDO `import` – FORMA 2 COM O CORINGA *

É possível usar o caractere coringa * na forma 2 do `import`, assim:

```
from <módulo> import *
```

O uso do coringa causará a importação de todos os elementos contidos no módulo, com exceção daqueles cujos nomes iniciam com underline "_". O exemplo a seguir é dividido em duas partes. Na parte 1 investigamos elementos do módulo `utilidades` para evidenciar a existência de vários elementos que começam com underline.

Exemplo 16.15



```
# Parte 1 - Inspeção dos elementos do módulo utilidades
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> import utilidades
>>> for s in dir(utilidades):
...     print(s)
__builtins__
__cached__
__doc__
__file__
__loader__
__name__
__package__
__spec__
meses
paridade
primo
texto
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Na segunda parte usamos o coringa * para mostrar que isto colocará os nomes de todos os objetos do módulo no namespace do chamador, com exceção de qualquer um que comece com o caractere de underline (_).

Exemplo 16.15 – segunda parte

```
# Parte 2 - Importação com coringa *
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> from utilidades import *
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'meses', 'paridade', 'primo', 'sys', 'texto']
(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)
```

Não é recomendável fazer importação com coringa em larga escala, em softwares de grande porte, com muitos módulos com vários elementos internos. Isso é um tanto perigoso porque você está inserindo nomes em massa no namespace. A menos que você conheça muito bem todos os módulos e seus elementos e tenha certeza de que não haverá colisão de nomes, você terá uma boa chance de substituir inadvertidamente nomes existentes. O que causará falhas severas na execução do software como um todo.

No entanto, essa sintaxe é bastante útil quando você está apenas usando o interpretador em ambiente interativo, para fins de estudos e testes de módulos, porque ela fornece acesso rápido a tudo o que um módulo tem a oferecer, sem muita digitação.

16.6.7 COMANDO **import** – FORMA 2 COM APELIDO

Por fim, na importação do elemento de um módulo também pode ser feita a atribuição de um apelido ao elemento.

```
from <modulo> import <elemento> as <apelido>
```

Ao usar esta forma o elemento será importado e o apelido especificado será usado como nome no namespace do chamador.

Exemplo 16.16

[Teste este código no Idle](#)

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> from utilidades import paridade as pdd
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
 '__package__', '__spec__', 'pdd', 'sys']
(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)
```

16.7 PACOTES

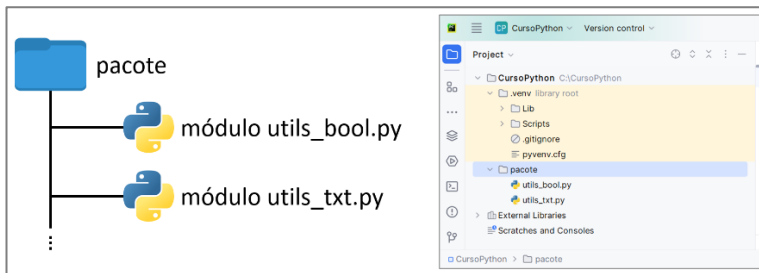
Suponha que você desenvolveu uma aplicação muito grande que inclui muitos módulos. À medida que o número de módulos aumenta, pode-se tornar difícil gerenciá-los e controlá-los se forem mantidos em um único local. Isto é especialmente verdade se eles tiverem nomes ou funcionalidades semelhantes. Em casos assim é conveniente haver um meio de agrupá-los e organizá-los.

Os pacotes permitem uma estruturação hierárquica do namespace do módulo usando notação com ponto. Da mesma forma que os módulos ajudam a evitar colisões entre nomes de objetos, os pacotes ajudam a evitar colisões entre nomes de módulos.

Criar um pacote Python é bastante simples, pois utiliza a estrutura hierárquica de arquivos do sistema operacional que você estiver usando no seu computador. Em outras palavras, para criar um pacote basta criar uma pasta e salvar os arquivos .py do Python dentro dela. Simples assim.

A figura 16.5 ilustra esse arranjo. A pasta `pacote` foi criada dentro da pasta `C:\CursoPython` e dentro dela foram salvos os dois arquivos Python mostrados no exemplo 16.17

Figura 16.5 – Pacote de módulos



fonte: o Autor

Exemplo 16.17 – módulo `utils_bool.py`



```
texto = 'Neste módulo as funções retornam um booleano'

def paridade(valor):
    """Se valor for par retorna True, senão retorna False"""
    if valor % 2 == 0:
        return True
    else:
        return False

def primo(valor):
    """Se valor for primo retorna True, senão retorna False"""
    if valor == 2:
        return True
    elif valor % 2 == 0:
        return False
    else:
        raiz = pow(valor, 0.5)
        i = 3
        while i <= raiz:
            if valor % i == 0:
                return False
            i += 2
        return True
```

(módulo)

Exemplo 16.17 – módulo `utils_txt.py`

continuação

```
texto = 'Neste módulo as funções imprimem um texto'

def paridade(valor):
    """imprime PAR ou ÍMPAR conforme o valor passado"""
    if valor % 2 == 0:
        print(f'{valor} é PAR')
    else:
        print(f'{valor} é ÍMPAR')
```

```
def primo(valor):
    """imprime PRIMO ou NÃO-PRIMO conforme o valor passado"""
    if valor == 2:
        print(f'{valor} é PRIMO')
    elif valor % 2 == 0:
        print(f'{valor} é NÃO-PRIMO')
    else:
        raiz = pow(valor, 0.5)
        resto = i = 3
        while i <= raiz and resto != 0:
            resto = valor % i
            i += 2
        if resto != 0:
            print(f'{valor} é PRIMO')
        else:
            print(f'{valor} é NÃO-PRIMO')
```

(módulo)

Exemplo 16.17 – programa principal

continuação

```
import pacote.utils_bool
import pacote.utils_txt

a = 17
print('Uso das funções do módulo utils_bool')
r = pacote.utils_bool.primo(a)
print(f'{a} é primo? {r}')
r = pacote.utils_bool.paridade(a)
print(f'{a} é par? {r}')

print('\nUso das funções do módulo utils_txt')
pacote.utils_txt.primo(a)
pacote.utils_txt.paridade(a)
```

```
Uso das funções do módulo utils_bool
17 é primo? True
17 é par? False
```

```
Uso das funções do módulo utils_txt
17 é PRIMO
17 é ÍMPAR
```

O vídeo do exemplo 16.17 mostra a criação do pacote, dos dois módulos e do programa principal.

No programa principal os módulos do pacote são importados usando a notação com ponto <pacote>.<módulo>. A partir daí as funções internas ao módulo podem ser acessadas desde que a notação com ponto seja utilizada.

16.7.1 IMPORTAÇÃO DE MÓDULO EM PACOTE COM APELIDO

No exemplo anterior pode-se perceber que os nomes podem ficar bastante extensos ao usar a notação com pontos, como destacado a seguir:

```
pacote.utils_txt.primo(a)
pacote.utils_txt.paridade(a)
```

Para tornar a escrita do código mais compacta é possível usar apelidos para os módulos em pacotes.

A forma do comando import é esta:

```
from <pacote>.<módulo> as <apelido>
```

A partir da importação feita o acesso aos elementos do módulo será possível através da forma:

```
<apelido>.<elemento>
```

Isto está ilustrado no exemplo 16.18.

Exemplo 16.18



```
import pacote.utils_bool as ub
import pacote.utils_txt as ut

a = 17
print('Uso das funções do módulo utils_bool')
r = ub.primo(a)
print(f'{a} é primo? {r}')
r = ub.paridade(a)
print(f'{a} é par? {r}')

print('\nUso das funções do módulo utils_txt')
ut.primo(a)
ut.paridade(a)
```

Uso das funções do módulo utils_bool
17 é primo? True
17 é par? False

Uso das funções do módulo utils_txt
17 é PRIMO
17 é ÍMPAR

Observe que essa notação com apelido torna a escrita do programa bem mais compacta e legível, além de permitir que cada módulo do pacote tenha um apelido apropriado escolhido pelo programador.

16.8 EXECUÇÃO DE UM MÓDULO COMO UM SCRIPT

Não há nada que impeça que um módulo seja executado como um script. Em outras palavras, um módulo pode conter um código principal. Caso esse código principal esteja presente ele será executado no momento em que for feita a importação.

Exemplo 16.19



```
# Código do módulo cap16_operacoes.py
def soma(*args):
    r = 0
    for x in args:
        r = r + x
    return r

def multi(*args):
    r = 1
    for x in args:
        r = r * x
    return r

print('Início do módulo') # esta é a parte principal do módulo

# Código do programa que usa o módulo cap16_operacoes.py
import cap16_operacoes as op

print('soma:', op.soma(2, 4, 6, 8))
print('multi:', op.multi(2, 4, 6, 8))
```

Início do módulo
soma: 20
multi: 384

Ao executar o programa principal veja que a importação do módulo causou a execução do comando `print('Inicio do módulo')` fazendo com que a mensagem seja exibida na tela de execução do programa.

Esse recurso existe para que tarefas de configuração e inicialização do módulo possam ser executados no momento da carga do módulo

Exercício Proposto

Enunciado: *Um exemplo da situação em que módulos se aplicam é o caso da função `ExibeLista()` que foi usada na solução de alguns exercícios propostos do capítulo 12 deste curso. Abra os arquivos dos exercícios propostos 12.4, 12.11 e 12.13 e observe-os com atenção. Você verá que essa função está presente nos três programas.*

Se tivéssemos criado um módulo contendo essa função, poderíamos importá-lo e usá-la sem a necessidade de reescrever seu código.

Escreva a função `ExibeLista()` em um módulo e depois altere os programas dos exercícios propostos 12.4, 12.11 e 12.13 para usar o módulo.

Capítulo 17

PROGRAMAÇÃO ORIENTADA A OBJETOS

17.1 ORIENTAÇÃO A OBJETOS COM PYTHON

17.1.1 CONCEITO

Python suporta o paradigma de programação orientada a objetos por meio da criação de classes. As classes fornecem um meio eficaz e elegante de agrupar dados e funcionalidades em um elemento. Após criar uma classe o programador pode usá-la para construir objetos baseados nessa classe, assim podemos dizer que o código de uma classe é o modelo para criação de instâncias em memória, que são os objetos.

Esse paradigma fornece uma maneira inteligente de definir elementos de código que podem ser reutilizados conforme a necessidade dos desenvolvedores. Por exemplo, durante este curso já usamos diversas vezes elementos `int`, `float`, `string`, `list` e `dict` que são classes disponíveis no Python e com elas criamos objetos que foram usados para resolver vários exercícios.

Classe e Objetos

Classe é um modelo usado para criar Objetos.
Objeto é uma instância de uma Classe.

A Classe é uma entidade lógica, um código de programação.
O Objeto é uma entidade física na memória, durante a execução do programa.

Uma analogia comum para explicar a relação entre classe e objeto é imaginar que a classe seja uma forma de bolo e os objetos são os bolos que podem ser feitos com essa forma. A forma terá características próprias como seu tamanho e formato e os bolos produzidos com ela seguirão tais características. Cada bolo produzido terá o formato e o tamanho dessa forma, mas também poderá ter características próprias com variações de sabor e textura decorrentes dos ingredientes usados e da sequência de preparo da massa.

A forma, assim como a classe, define aspectos gerais e cada bolo produzido é uma instância específica que terá um uso e um tempo de duração.

Levando essa ideia para programação veja os objetos abaixo. Todos eles são da mesma forma – a classe `list` – mas cada um tem seus próprios ingredientes – os dados que armazenam.

```
L1 = [12, 50, 89] # lista com inteiros
L2 = ['Oliveira', 'Silva', 'Grigoletto', 'Munhoz'] # lista com sobrenomes
L3 = [(3.7, 9.95), (14.32, 8.56)] # lista com tuplas
L4 = [] # lista vazia
```

Na Programação Orientada a Objetos (POO), os membros de uma classe são:

- **Atributo:** usado para se referir a um dado necessário ao objeto de uma determinada classe. Em Python, atributos são objetos definidos dentro da classe e são eles que armazenam todos os dados necessários para o funcionamento da classe. O conjunto de valores armazenados nos atributos definem o **estado do objeto**.
- **Método:** usado para se referir a uma função implementada dentro da classe. Os métodos conferem funcionalidades e comportamentos ao objeto. Essas funções normalmente realizam operações usando os dados armazenado nos atributos do objeto.

O objetivo maior da POO é produzir classes totalmente funcionais para modelar elementos existentes no mundo real. Por exemplo, você pode usar classes para criar objetos que emulem pessoas, produtos, veículos, notas fiscais ou quaisquer outros objetos que seu software necessite.

Em Python, o corpo de uma determinada classe funciona como um namespace onde residem atributos e métodos.

Só se pode acessar esses atributos e métodos através da classe ou de seus objetos.

17.1.2 MÉTODOS MÁGICOS

Se você chegou até aqui já deve ter lido o capítulo 15 sobre módulos e pacotes. Nesse capítulo, quando usamos a função `dir()`, foram exibidas na tela diversas funções cujos nome possuem dois underlines no início e dois no final como: `.__annotations__()`, `.__builtins__()`, `.__doc__()`, `.__name__()`, etc.

Esse estilo de nome pode parecer estranho aos iniciantes em Python, pois inicia e termina com dois underlines. Bem, acostume-se com isso, pois são muito comuns nessa linguagem.

Eles são conhecidos como Métodos Mágicos e existem para que os programadores possam realizar tarefas especiais e personalizar o comportamento dos objetos.

Métodos Mágicos (Magic Methods)

Em Python são os métodos que iniciam e terminam com dois caracteres underline.

Esses métodos conferem grande flexibilidade e robustez à programação dos objetos Python.

Também são conhecidos como Dunder Methods (dunder = contração de "double underline")

Neste capítulo vamos trabalhar com alguns desses métodos mágicos.

17.1.3 COMO DEFINIR UMA CLASSE EM PYTHON

As classes são definidas através da palavra-chave `class`, seguida do nome da classe. Elas podem ser definidas diretamente no programa principal ou em um módulo. Geralmente são definidas em módulos.

O exemplo 17.1 mostra o esquema mínimo necessário para criar e usar uma classe.

Exemplo 17.1



```
>>> class MinhaClasse:
...     # corpo da classe
...     pass      # este comando não faz nada, mas garante a existência da classe

>>> obj = MinhaClasse()  # o objeto obj é criado com base em MinhaClasse
>>> type(obj)
<class '__main__.MinhaClasse'>
>>> print(obj)
<__main__.MinhaClasse object at 0x000002D95F15FD40>
```

(exemplo interativo feito com IDE Idle)

No corpo da classe você define os atributos e métodos conforme necessário – para que sejam definidos de modo inteligente e funcional é preciso um bom levantamento de necessidades e planejamento prévios.

Agora vamos criar uma classe que possua atributos e funcionalidades. Observe com atenção o exemplo 17.2. No exemplo primeiramente definimos a classe `Retangulo`. Em seguida, na parte principal do programa, ela é usada para criar os objetos `r1` e `r2`.

Exemplo 17.2



```
class Retangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calc_area(self):
        return self.base * self.altura

    def exibe(self):
        print(f'retângulo {self.base} x {self.altura}')
        print(f' área = {self.calc_area()}')

print('Início do programa principal')
r1 = Retangulo(12, 5)
print('r1: ', end='')
r1.exibe()
print('r2: ', end='')
r2 = Retangulo(3.5, 9.0)
r2.exibe()

r2.base = 9.5      # alteração no atributo base de r2
r2.altura = 16.3   # alteração no atributo altura de r2
print('\nAcessos individuais a atributos e métodos')
print(f'Medidas do retângulo r2: {r2.base} x {r2.altura}')
area = r2.calc_area()
print(f'Área do retângulo r2 = {area}')

Início do programa principal
```

```
r1: retângulo 12 x 5
    área = 60
r2: retângulo 3.5 x 9.0
    área = 31.5

Acessos individuais a atributos e métodos
Medidas do retângulo r2: 9.5 x 16.3
Área do retângulo r2 = 154.85
```

Vamos agora usar esse exemplo para aprofundar um pouco mais no entendimento das classes.

17.1.4 CONSTRUTOR DE UMA CLASSE

No paradigma da POO existe o conceito de **método construtor de classe**. Vamos entender: para um objeto existir na memória do computador ele precisa ser construído e quem faz essa tarefa é o construtor de classe que é executado uma única vez para cada objeto criado.

Esse método tem as finalidades de: alocar memória para o objeto, executar quaisquer tarefas necessárias à configuração elementos dentro do objeto, bem como inicializar atributos com valores passados pelo chamador.

Em linguagens como C++ e Java o programador deve escrever esse método, o qual terá o mesmo nome da classe. Em Python é um pouco diferente, pois nós não escrevemos o método construtor. O interpretador Python cuida disso para nós.

O processo de instanciação do Python é acionado sempre que fazemos a criação de uma nova instância, ou seja, um objeto. Este processo é executado em duas etapas separadas que pode ser descrito da seguinte forma:

- **Criação** da nova instância da classe: para isso usamos o método `.__new__()` que é responsável pela criação e retorno de um novo objeto vazio. Depois da criação do objeto vazio ele poderá ser referenciado através do identificador `self`;
- **Inicialização** da nova instância: para isso há o método especial `.__init__()` que usa o novo objeto recém-criado e inicializa seus atributos com os argumentos que foram passados.

self, o que é isso?

Em Python, `self` é o identificador (nome) usado no código da classe com o propósito de se referir ao próprio objeto (instância da classe) e prover o acesso aos seus atributos e métodos.

Todos os métodos de uma classe precisam ter o **self** como primeiro argumento.

Retornando ao exemplo 17.2, vemos que o construtor da classe `Retangulo` foi usado duas vezes quando escrevemos essas linhas:

```
r1 = Retangulo(12, 5)
r2 = Retangulo(3.5, 9.0)
```

Essa chamada realiza a construção do objeto e como parte do processo o método `.__init__()` será executado, sendo que ele tem esta sintaxe:


```
def __init__(self, base, altura):    # cabeçalho
    self.base = base                # atribuição ao atributo self.base
    self.altura = altura             # atribuição ao atributo self.altura
```

Em seu cabeçalho, além do parâmetro obrigatório `self`, temos os parâmetros necessários no momento da criação da classe. Cada parâmetro recebido é atribuído ao atributo correspondente. Assim, `self.base` e `self.altura` são atributos que recebem o valor inicial contido no parâmetro.

17.1.5 MÉTODOS EXISTENTES NA CLASSE

Os outros dois métodos da classe `Retangulo` realizam tarefas específicas para essa forma geométrica retângulo:

`.calc_area()` – calcula e retorna a área do retângulo;

`.exibir()` – imprime os dados da classe;

Depois que os objetos `r1` e `r2` foram construídos eles podem ser usados livremente pelo programador, sendo que o acesso a seus membros – sejam atributos ou métodos – podem ser feitos com o uso da notação com ponto: `<objeto>.<elemento>`. O restante do código do exemplo 17.2 nos mostra esse uso.

```
r1.exibe()           # uso do método .exibe() com a instância r1
r2.exibe()           # uso do método .exibe() com a instância r2

r2.base = 9.5        # alteração no atributo base de r2
r2.altura = 16.3      # alteração no atributo altura de r2
print('\nAcessos individuais a atributos e métodos')
print(f'Medidas do retângulo r2: {r2.base} x {r2.altura}')
area = r2.calc_area()
print(f'Área do retângulo r2 = {area}')
```

No código acima os destaques em negrito mostram os momentos em que é feito um acesso a algum membro da classe através da notação com ponto. Note que é possível fazer alterações no conteúdo de atributos – alteramos `base` e `altura` do objeto `r2`; bem como executamos os métodos `.exibe()` e `.calc_area()`.

Exercício Resolvido 17.1



Enunciado: Escreva um programa que permaneça em laço lendo a base e a altura de retângulos e exibindo na tela sua área, com 3 casas decimais. Isso deve ser feito com o uso de uma classe `Retangulo` contida em um módulo à parte do programa principal. O laço termina quando for digitado "FIM".

```
# módulo m_exresolvido_17_1.py

class Retangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calc_area(self):
        return self.base * self.altura

    def exibe(self):
        print(f'retângulo {self.base} x {self.altura}')
        print(f' área = {self.calc_area():.3f}')
```

```

"""Programa principal
usa o módulo m_exresolvido_17_1 que está no pacote cap17"""
from cap17.m_exresolvido_17_1 import Retangulo

ret = Retangulo(0,0)
s = input('Digite dois reais (base altura): ')
while s.upper() != 'FIM':
    valores = s.split()
    ret.base = float(valores[0])
    ret.altura = float(valores[1])
    ret.exibe()
    s = input('Digite dois reais (base altura): ')

print('\nFim do Programa')

Digite dois reais (base altura): 3.3 6
retângulo 3.3 x 6.0
    área = 19.800
Digite dois reais (base altura): 4.1 9.2
retângulo 4.1 x 9.2
    área = 37.720
Digite dois reais (base altura): 12 25.1
retângulo 12.0 x 25.1
    área = 301.200
Digite dois reais (base altura): FIM

Fim do Programa

```

Neste exercício resolvido aproveitamos a classe `Retangulo` criada no exemplo 17.2, separando-a em um módulo que foi colocado dentro de um pacote denominado `cap16`, para melhor organização de todos os códigos deste capítulo. No programa principal fizemos a importação desse módulo e usamos a classe `Retangulo` conforme pode ser visto acima e no vídeo deste exercício.

Agora vamos fazer uma melhoria na classe `Retangulo`. Veja o exercício resolvido 17.2

Exercício Resolvido 17.2

Teste este código no PyCharm

Enunciado: *Faça uma alteração no código da classe `Retangulo` de modo que se um dos valores de base e altura for menor ou igual a zero, uma exceção `ValueError` seja levantada.*

Ajuste o programa principal para tratar essa exceção.

```

# módulo m_exresolvido_17_2.py
class Retangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calc_area(self):
        return self.base * self.altura

    def exhibe(self):
        if self.base <= 0:
            raise ValueError('Base: valor numérico maior que zero esperado')
        if self.altura <= 0:
            raise ValueError('Altura: valor numérico maior que zero esperado')
        print(f'retângulo {self.base} x {self.altura}')
        print(f'    área = {self.calc_area():.3f}')

```

```

"""Programa principal
usa o módulo m_exresolvido_16_2 que está no pacote cap16"""
from cap17.m_exresolvido_17_2 import Retangulo

```

```

ret = Retangulo(0,0)
s = input('\nDigite dois reais (base altura): ')
while s.upper() != 'FIM':
    valores = s.split()
    try:
        ret.base = float(valores[0])
        ret.altura = float(valores[1])
        ret.exibe()
    except ValueError as e:
        print(f'Erro! {e}')
    s = input('Digite dois reais (base altura): ')

print('\nFim do Programa')

```

Digite dois reais (base altura): 3.3 5.6
 retângulo 3.3 x 5.6
 área = 18.480

Digite dois reais (base altura): -3.3 10.0
 Erro! Base: valor numérico maior que zero esperado

Digite dois reais (base altura): 4.2 0.0
 Erro! Altura: valor numérico maior que zero esperado

Digite dois reais (base altura): FIM

Fim do Programa

A novidade neste exercício resolvido 17.2 são essas linhas inseridas no método `.exibe()` da classe `Retangulo`. Elas fazem a verificação do valor e se necessário levantam uma exceção `ValueError`.

```

if self.base <= 0:
    raise ValueError('Base: valor numérico maior que zero esperado')
if self.altura <= 0:
    raise ValueError('Altura: valor numérico maior que zero esperado')

```

No programa principal, por sua vez usamos o comando `try-except` para interceptar a exceção levantada e exibir a mensagem para o usuário. Mais adiante vamos aprimorar esse tratamento de erro usando uma técnica mais aprimorada disponível nas classes Python.

17.2 COMBINAÇÃO DE CLASSES COM LISTAS

Uma vez que uma classe seja criada é possível escrever um programa que utilize os objetos dessa classe em combinação com listas. Ao inserir vários objetos da sua classe personalizada em uma lista você tem a possibilidade de criar interessantes coleções de dados.

Vamos criar uma classe capaz de conter dados de veículos que estejam à venda em uma loja. As informações que queremos manipular são: placa, modelo, ano e quilometragem. Os dados estão disponíveis em um arquivo do tipo CSV no qual cada linha é um veículo. Eles deverão ser lidos, carregados na classe que vamos criar e essa classe deve ser inserida em uma lista. A figura 17.1 mostra a estrutura do arquivo que será usado como entrada de dados para os testes.

Figura 17.1 – Conteúdo do arquivo `veículos.txt`

```

ELM8038;CELTA EL 1.0;2012;76320
LCA1E42;FOX TSI 1.6;2020;23900
FLB2297;PALIO FIRE 1.0;2016;68350
SCE9U33;COROLLA GLI 2.0;2022;43200
CMT7153;ONIX LT 1.0;2021;54600

```

fonte o Autor

Primeiramente vamos criar a classe contendo atributos para os dados necessários. Além dos quatro atributos vamos criar também o método `.exibe()` que poderá ser usado para exibição dos dados do veículo na tela.

Exemplo 17.3 – classe Veiculo



```
# módulo m_exemplo_17_3.py

class Veiculo:
    def __init__(self, placa, modelo, ano, km):
        self.placa = placa
        self.modelo = modelo
        self.ano = ano
        self.km = km

    def exibe(self):
        print(f'Veículo placa {self.placa}')
        print(f'    {self.modelo}, ano: {self.ano} - km: {self.km}')
```

(módulo)

Com a classe pronta, podemos escrever o programa principal que ficará da seguinte forma:

Exemplo 17.3 – programa principal

continuação

```
from cap17.m_exemplo_17_3 import Veiculo

# leitura e carga dos dados
LstV = [] # lista de veículos
for s in open('veiculos.txt', 'r'):
    s = s.split(';')
    v = Veiculo(
        s[0], # placa
        s[1], # modelo
        int(s[2]), # ano
        int(s[3]) # km
    )
    LstV.append(v) # inclui o objeto v na lista

# apresentação dos dados na tela
for v in LstV:
    v.exibe()
print('\nFim do Programa')
```

Veículo placa ELM8038
CELTA EL 1.0, ano: 2012 - km: 76320
Veículo placa LCA1E42
FOX TSI 1.6, ano: 2020 - km: 23900
Veículo placa FLB2297
PALIO FIRE 1.0, ano: 2016 - km: 68350
Veículo placa SCE9U33
COROLLA GLI 2.0, ano: 2022 - km: 43200
Veículo placa CMT7153
ONIX LT 1.0, ano: 2021 - km: 54600

Fim do Programa

Observe esse código e vamos entender o que está sendo feito: o trecho a seguir usa os dados contidos no string `s` que vieram da leitura do arquivo. O método `.split()` faz a separação das partes contidas na linha lida, usando o caractere `' '` como separador. Em seguida o objeto da classe `Veiculo` é criado.

```
s = s.split(';')
v = Veiculo(
    s[0], # placa
    s[1], # modelo
    int(s[2]), # ano
    int(s[3]) # km
)
```

Na criação do objeto foi usado o construtor da classe com os quatro argumentos que vieram do arquivo. O ano e a quilometragem foram convertidos para inteiros ao serem passados para a classe.

Com os dados já carregados o passo seguinte foi fazer a exibição em tela com este código:

```
for v in LstV:
    v.exibe()
```

Como os elementos da lista `LstV` são da classe veículos, então basta criar uma iteração com essa lista e cada objeto atribuído a `v` contém o método `.exibe()`. Basta executá-lo e os dados serão exibidos em tela, no formato especificado dentro da classe.

17.3 COMBINAÇÃO DE CLASSES COM DICIONÁRIOS

Na seção anterior vimos que é possível combinar classes com listas. De modo análogo, também é possível combinar classes com dicionários.

No exemplo 17.4 implementamos uma nova solução para o exemplo anterior, trocando a lista `LstV` pelo dicionário `DictV`. Neste caso precisamos escolher alguma chave para o dicionário e a placa do veículo é a candidata perfeita, uma vez que não há dois veículos com a mesma placa.

Exemplo 17.4



```
from cap17.m_exemplo_17_3 import Veiculo

# leitura e carga dos dados
DictV = {} # dicionário de veículos
for s in open('veiculos.txt', 'r'):
    s = s.split(';')
    v = Veiculo(
        s[0], # placa - será usada como chave para o dicionário (veja abaixo)
        s[1], # modelo
        int(s[2]), # ano
        int(s[3]) # km
    )
    DictV[s[0]] = v # inclui objeto v no dicionário usando s[0] (placa) como chave

# apresentação dos dados na tela
for v in DictV.values():
    v.exibe()
print('\nFim do Programa')
```

```
Veículo placa ELM8038
    CELTA EL 1.0, ano: 2012 - km: 76320
Veículo placa LCA1E42
    FOX TSI 1.6, ano: 2020 - km: 23900
Veículo placa FLB2297
    PALIO FIRE 1.0, ano: 2016 - km: 68350
Veículo placa SCE9U33
    COROLLA GLI 2.0, ano: 2022 - km: 43200
Veículo placa CMT7153
    ONIX LT 1.0, ano: 2021 - km: 54600

Fim do Programa
```

Se você comparar esta solução com a anterior verá que as mudanças são bem pequenas. A leitura do arquivo e a criação do objeto da classe `Veiculo` são a mesma coisa nas duas soluções.

A primeira diferença acontece na inserção do objeto `Veiculo` no dicionário, que é feita com a linha:

```
DictV[s[0]] = v # inclui o objeto v no dicionário
```

Como o elemento `s[0]` contém a placa do veículo, então ele foi usado como chave. E o valor atribuído a essa chave é `v`, o objeto criado com a classe `Veiculo`.

Replique e execute esses dois exemplos, 17.3 e 17.4, faça seus testes e constate como é prático combinar classes personalizadas com listas e dicionários.

17.4 QUANDO USAR CLASSES? E QUANDO NÃO USAR?

As classes Python são ferramentas muito flexíveis e poderosas que você pode usar em vários cenários. Por causa disso, algumas pessoas tendem a abusar delas e tentar resolver todos os seus problemas de programação usando-as.

No entanto, às vezes usar uma classe não é a melhor solução. Há situações em que algumas funções são suficientes para resolver um determinado problema.

A linguagem Python é multiparadigma, ou seja, ela permite que adotemos diferentes abordagens de programação, tais como: programação procedural, orientação a objetos e até mesmo programação funcional. Assim, você deve ter em mente que em Python você pode escolher não usar classes quando elas não forem necessárias.

Em linhas gerais evite usar classes em duas situações:

- Quando você **apenas precisa armazenar dados**: nestes casos use tuplas, listas ou dicionários, que são especialmente projetados para armazenar dados. Então, eles podem ser a melhor solução se você não precisar de funcionalidades associadas a esses dados;
- Quando você **apenas precisa executar uma tarefa**: nestes casos escreva uma função. Se sua classe tiver um, dois, enfim, uns poucos métodos e não fortemente relacionados a dados, então talvez você não precise de uma classe. Em vez disso, use funções comuns.

Posteriormente, se mais métodos aparecerem e estiverem fortemente vinculados a dados, você poderá alterar seu código e criar uma classe.

Você encontrará várias situações em que talvez não seja necessário ou uma boa abordagem usar classes em seu código. Por exemplo:

- Um programa pequeno e simples que não requer estruturas de dados ou lógica complexas. Em casos assim usar classes pode ser um exagero.
- Um programa para situações críticas em termos de desempenho e performance. As classes adicionam sobrecarga ao seu programa, especialmente quando você precisa criar muitos objetos. Isso pode afetar o desempenho geral do seu código.
- Em sistemas legados (programas antigos que ainda estão em uso). Se um sistema antigo usado em produção não usa classes, então não é uma boa ideia introduzi-las em eventuais

manutenções. Isso quebrará o estilo de codificação usado e poderá gerar uma quebra de consistência no código.

Não é porque Python suporta orientação a objetos que precisamos adotar o paradigma. Precisamos conhecer o paradigma e suas ferramentas e em cada projeto usar discernimento e bom senso para decidir. E sempre lembrar de um dos princípios de Python: **simples é melhor que complexo**.

17.5 MEMBROS PÚBLICOS E NÃO-PÚBLICOS

Linguagens como Java e Object Pascal possuem um recurso que permite fazer a distinção quanto à visibilidade de atributos e métodos. São os chamados modificadores de acesso. Nessas linguagens os membros de uma classe podem ser **públicos**, **protegidos** e **privados**. Os públicos são totalmente visíveis fora da classe; os privados apenas são visíveis dentro da classe; e os protegidos são visíveis na estrutura hierárquica das classes, mas invisíveis fora dessa estrutura.

Pois bem, em Python isso não existe.

E talvez programadores que já conhecem essas outras linguagens e estão acostumados com os modificadores de acesso, vão considerar isso muito estranho. Mas vamos entender como Python trata a visibilidade dos membros de uma classe.

Python tem uma convenção de nomenclatura bem estabelecida que deve ser usada para declarar se um membro de classe, atributo ou método, pode ser usado fora da classe ou não. Essa convenção de nomenclatura consiste em adicionar um sublinhado inicial ao nome do membro. Então, teremos:

- **Membro Público:** inicie seu nome com uma letra;
- **Membro Não-Público:** inicie seu nome com um caractere underline "_";

Nas convenções de Python, os membros públicos constituem a parte da interface oficial da classe. Essa interface será normalmente utilizada pelos programadores que fazer uso da classe nos seus programas. Por outro lado, os membros não-públicos não se destinam a fazer parte dessa interface. Isso significa que você não deve usar membros não públicos fora da classe que os define.

Os membros não públicos existem para dar suporte à implementação interna da classe. Por exemplo, atributos não-públicos podem armazenar dados temporários e métodos não-públicos podem realizar parte de uma tarefa, mas não a tarefa toda. E seus detalhes de implementação só interessam a quem desenvolveu a classe, não a quem usa a classe.

Além disso, todo pacote de classes evolui ao longo do tempo com o lançamento de versões com melhorias e novos recursos. Nessas novas versões os membros públicos serão mantidos, mas nada garante que os membros não-públicos ainda estarão lá e se você usá-los seu código "vai quebrar".

Outro aspecto é que você pode descobrir a existência de um determinado membro não-público e sentir-se tentado a usá-lo. Mas você não saberá o que ele faz, porque não haverá documentação para ele. Os desenvolvedores de classes não tem a obrigação de documentar membros não-públicos, quase nunca o fazem e com isso você não terá referências. O exemplo 17.5 ilustra uma situação dessas.

Neste exemplo você precisa usar uma lista. Você a cria e faz uma inspeção usando a função `dir()` e descobre um membro não público com o nome `__sizeof__` e supõe que esse método irá retornar o

tamanho da lista. Ao usá-lo você recebe de retorno um valor completamente fora do esperado. Com isso constata que não está fazendo certo. Use um elemento público da linguagem como `len(lista)` e você terá o resultado que necessita.

Exemplo 17.5



```
>>> lista = [34, 5, 16, 41, 27] # lista com 5 inteiros
>>> dir(lista)
['_add_', '__class__', '__class_getitem__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getstate__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__',
 '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
 '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
 'sort']
>>> print(lista.__sizeof__()) # você tenta usar __sizeof__()
88 # e não entende o valor que ele retorna
>>> lista = [34, 5, 16, 41, 27, 63, 19, 44] # em seguida aumenta a lista
>>> print(lista.__sizeof__()) # tenta usá-lo novamente
104 # e continua a não entender o que ele faz

>>> print(len(L)) # É bem mais simples usar algum elemento público
8
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Em síntese, um membro não-público é sinalizado através do caractere underline no início de seu nome. Se inspecionar a classe, você saberá que esse membro existe, terá acesso a ele e até poderá usá-lo diretamente no seu programa. Porém, jamais deve fazer isso.

Explícito é melhor que implícito.

Esse é um dos princípios do Zen do Python.
Membros de uma classe podem ser Públicos ou Não-públicos,
mas todos estarão visíveis, ou seja explícitos.
Consequência: cabe ao programador trabalhar do jeito certo!

17.6 DEFORMAÇÃO DE NOMES (NAME MANGLING)

Esta é uma outra convenção relacionada a nomes de elementos Python. Deformação de nome é uma transformação automática de um nome feita pelo interpretador para os membros de uma classe que começam com dois caracteres underline "__".

Essa deformação do nome é o seguinte. Considere a classe `MinhaClasse` a seguir:

```
class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
    def __fazalgo(self):
        print(f'Valor = {self.valor} em método fazalgo')
    def exhibevalor(self):
        print(f'Valor = {self.valor} em método exhibevalor')
```

Além do `.__init__()` há outros dois métodos: `.__fazalgo()` e `.__exibevalor()`. Este último é um nome público e poderá ser usado normalmente.

O método `.__fazalgo()` é não-público e terá seu nome deformado com a inclusão de um underline inicial mais o nome da classe, assim:

```
.__fazalgo() → .__MinhaClasse__fazalgo()
```

No exemplo 17.6 mostramos o que ocorre:

Exemplo 17.6



```
>>> class MinhaClasse:
    def __init__(self, valor):
        self.valor = valor
    def __fazalgo(self):
        print(f'Valor = {self.valor} em método fazalgo')
    def exhibevalor(self):
        print(f'Valor = {self.valor} em método exhibevalor')
>>> dir(MinhaClasse)
['_MinhaClasse__fazalgo', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',
 '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', 'exibevalor']

>>> obj = MinhaClasse(25) # criamos um objeto de MinhaClasse
>>> obj.exibevalor()      # usamos o método público .exibevalor() sem problemas
Valor = 25 em método exhibevalor
>>> obj.__fazalgo()       # ao tentar usar __fazalgo() ocorre erro
Traceback (most recent call last):
  File "<pyshe11#253>", line 1, in <module>
    obj.__fazalgo()
AttributeError: 'MinhaClasse' object has no attribute '__fazalgo'

>>> obj._MinhaClasse__fazalgo() # ainda assim podemos acessar esse método
Valor = 25 em método fazalgo
(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)
```

Veja que a deformação do nome de `.__fazalgo()` realmente ocorreu (destaque em negrito). Como consequência dessa transformação, ao tentarmos usar esse método, o interpretador Python retorna um `AttributeError` – o método não foi encontrado.

Este comportamento interno oculta o nome, gerando uma ilusão de que o membro da classe se tornou privado, mas não é verdade. Eles ainda podem ser acessados através do nome deformado, conforme mostrado na linha final do exemplo acima.

17.7 DICIONÁRIO DE ATRIBUTOS

Toda classe e todo objeto Python possui um atributo de uso especial que é o `__dict__` que é o **dicionário de atributos**. Como você pode perceber, esse atributo é não-público, portanto, você não deve usá-lo nos seus programas desenvolvidos para implantação em produção e uso por parte de usuários finais.

Porém é um recurso muito útil quando estamos estudando uma classe e seus objetos, buscando entender seu funcionamento e conhecer seus membros.

O atributo `__dict__` é um dicionário e cada par chave-valor representa um membro. No caso de classes, os membros presentes em `__dict__` serão atributos e métodos da classe. No caso de objetos os membros presentes em `__dict__` serão os atributos associados ao objeto. A chave é o nome do membro e o valor é seu conteúdo no caso de atributos e seu id no caso de métodos.

Observe com atenção o exemplo 17.7 no qual foi criada a classe `Exemplo`, a partir da qual criamos o objeto `obj`. Em seguida usamos `__dict__` para inspecionar ambos.

Exemplo 17.7



```
>>> class Exemplo:
    def __init__(self, dado):
        self.dado = dado
    def calculo(self):
        a = 10
        self.b = 5
        return self.dado + a + self.b

>>> obj = Exemplo(23)
>>> obj.__dict__
{'dado': 23}
>>> print(obj.calculo())
38
>>> obj.__dict__
{'dado': 23, 'b': 5}

>>> for s in Exemplo.__dict__.items():
    print(s)
('__module__', '__main__')
('__init__', <function Exemplo.__init__ at 0x000001FE9B1D7EC0>)
('calculo', <function Exemplo.calculo at 0x000001FE9B1E0220>)
('__dict__', <attribute '__dict__' of 'Exemplo' objects>)
('__weakref__', <attribute '__weakref__' of 'Exemplo' objects>)
('__doc__', None)

>>> id(Exemplo.calculo)
2193035756064 # converta esse valor para hexadecimal e você obterá 1FE9B1E0220
(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)
```

Vamos analisar ponto a ponto o que ocorre acima.

1. Com este código criamos o objeto `obj`. O valor 23 é atribuído a `self.dado` no método `__init__()`.

```
obj = Exemplo(23)
```

2. Exibindo o `__dict__` constatamos a associação entre o nome `dado` e o valor 23.

```
obj.__dict__
{'dado': 23}
```

3. Executamos o método público `.calculo()`. Esse método usa um objeto local `a` que existirá apenas dentro do método e também usa o atributo `self.b` que é inserido no dicionário de atributos de `obj`. Com isso, podemos constatar que um atributo será criado apenas depois de usado uma primeira vez. Constatamos também que objetos não qualificados com `self` não são inseridos no dicionário de atributos.

```
print(obj.calculo())
38
obj.__dict__
```

```
{'dado': 23, 'b': 5}
```

4. Também é possível exibir o dicionário de atributos para uma classe. Fizemos isso usando um iterador e obtivemos o resultado abaixo mostrando os membros da classe `Exemplo`. Fazemos um destaque para os métodos declarados na classe `__init__()` e `.calcula()` que são mostrados como funções em conjunto com um número hexadecimal que representa o Id do método

```
for s in Exemplo.__dict__.items():
    print(s)
('__module__', '__main__')
('__init__', <function Exemplo.__init__ at 0x000001FE9B1D7EC0>)
('calcula', <function Exemplo.calcula at 0x000001FE9B1D7F60>)
('__dict__', <attribute '__dict__' of 'Exemplo' objects>)
('__weakref__', <attribute '__weakref__' of 'Exemplo' objects>)
('__doc__', None)
```

Nas seções a seguir vamos usar o dicionário de atributos para nos aprofundar no conhecimento sobre os atributos de objetos e classes.

17.8 MODO DE DECLARAÇÃO DE ATRIBUTOS EM CLASSES

Em outras linguagens orientadas a objetos, como C++ e Java, qualquer atributo deve formalmente declarado no corpo da classe. Em Python a criação de atributos é diferente.

Para um atributo existir basta que ele seja criado através de uma atribuição como essa:

```
self.atributo = <valor atribuído>
```

Essa atribuição pode ser feita **em qualquer método** da classe, embora, em geral, isso seja feito no método `__init__()`.

Volte ao exemplo 17.7 da seção anterior e observe que na classe `Exemplo` dois atributos são criados. O atributo `self.dado` é criado no método `__init__()` e o atributo `self.b` é criado no método `.calcula()`.

Vamos fazer agora um novo exemplo para demonstrar e reforçar esse aspecto da POO com Python e também fazer algumas considerações sobre consistência na criação de atributos. Considere a classe `Produto` criada no exemplo 17.8 a seguir.

Exemplo 17.8



```
>>> class Produto:
    def __init__(self, codbarras, nome):
        self.codbarras = codbarras
        self.nome = nome
        self.estq = 0
    def cad_custo(self, custo):
        self.custo = custo
    def exibe(self):
        print(f'Produto: {self.nome} ({self.codbarras})')
        print(f' preço de custo: R$ {self.custo}')

>>> prod = Produto('7893316002386', 'Arroz 1kg')
>>> prod.__dict__
{'codbarras': '7893316002386', 'nome': 'Arroz 1kg', 'estq': 0}
>>> prod.exibe()
Produto: Arroz 1kg (7893316002386)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
```

```
prod.exibe()
File "<pyshell#35>", line 10, in exhibe
    print(f' preço de custo: R$ {self.custo}')
AttributeError: 'Produto' object has no attribute 'custo'
>>> prod.cad_custo(7.23)
>>> prod.__dict__
{'codbarras': '7893316002386', 'nome': 'Arroz 1kg', 'estq': 0, 'custo': 7.23}
>>> prod.exibe()
Produto: Arroz 1kg (7893316002386)
preço de custo: R$ 7.23
```

(exemplo interativo feito com IDE Idle)

Observe que ao construir o objeto `prod` os três atributos dentro do método `.__init__()`, `self.codbarras`, `self.nome` e `self.estq` são devidamente criados. Os dois primeiros recebem valores iniciais passados como argumento, e o terceiro é criado com a atribuição do valor zero. Com o atributo `__dict__` constatamos que os três atributos realmente foram criados.

Em seguida, usamos o método `.exibe()` para mostrar os dados na tela, mas obtivemos um erro do tipo `AttributeError`, pois o atributo `self.custo` ainda não existe no nosso objeto.

Depois de usarmos o método `.cad_custo()` o atributo `self.custo` foi criado, de modo que a partir daí esse quarto atributo passa a constar de `__dict__` e o método `.exibe()` funciona corretamente.

Perceba então que essa classe tem um problema de projeto. E é um problema severo, uma vez que o método de exibição dos dados depende da existência do atributo `self.custo` que não será criado com os demais no momento da criação da instância da classe `Produto`. Com certeza essa não é uma boa prática de programação orientada a objetos em Python. Teria sido muito simples evitar esse problema se o atributo `self.custo` tivesse sido criado junto com os outros no `.__init__()`. E isso poderia ser feito tranquilamente através da atribuição do valor zero a ele assim como foi feito com o estoque.

Com isso mostramos que atributos podem ser criados em qualquer método da classe, porém convém tomar cuidado com esse recurso, fazendo um bom projeto de classes sempre com o objetivo de serem elas sejam corretas e consistentes.

17.9 ATRIBUTOS DE INSTÂNCIA E ATRIBUTOS DE CLASSE

Python suporta duas categorias de atributos em uma classe. A diferença entre as duas categorias está relacionada ao local da classe em que o atributo é criado. E essa diferença possui implicações na forma de uso desses tipos de atributo.

Atributos de instância armazenam dados vinculados a uma instância da classe, ou seja, um objeto. Eles são definidos dentro de qualquer método existente na classe;

Atributos de classe armazenam dados vinculados à classe e todos os objetos dessa classe compartilham seu conteúdo. Eles são definidos no corpo da classe, fora dos métodos existentes;

Vamos usar o exemplo a seguir para compreender os detalhes dessa diferença.

Exemplo 17.9



```
>>> class Attr:
    attrclasse = 9.99
    def __init__(self, num, txt):
        self.num = num
        self.txt = txt
    def exibedados(self):
        self.outro = 999
        print(f'attrclasse = {self.attrclasse}\n' +
              f'num = {self.num}\n' +
              f'txt = {self.txt}\n' +
              f'outro = {self.outro}')

>>> objA = Attr(10, 'texto do objeto A')
>>> objB = Attr(20, 'texto do objeto B')
>>> objA.exibedados()
attrclasse = 9.99
num = 10
txt = texto do objeto A
outro = 999
>>> objB.exibedados()
attrclasse = 9.99
num = 20
txt = texto do objeto B
outro = 999

>>> Attr.attrclasse
9.99
>>> objA.attrclasse
9.99
>>> objB.attrclasse
9.99
>>> Attr.attrclasse = 1989.95
>>> Attr.attrclasse
1989.95
>>> objA.attrclasse
1989.95
>>> objB.attrclasse
1989.95
```

(exemplo interativo feito com IDE Idle)

17.9.1 ATRIBUTOS DE INSTÂNCIA

Atributos de instância são vinculados a um objeto específico, de modo que o valor do atributo de instância estará anexado ao próprio objeto. É por isso que dizemos que o valor do atributo é específico da instância que o contém.

Os atributos de instância são definidos dentro dos métodos de instância, conforme vimos na seção anterior 17.8, os métodos são aqueles que recebem `self` como primeiro argumento e é neles que os atributos de instância são criados.

Lembre-se

Atributos de instância podem ser criados em qualquer método de instância, porém convém que eles sejam criados no método `__init__()`

No exemplo 17.10 definimos uma classe que contém três atributos de instância e em seguida a utilizamos para criar três objetos com diferentes dados

Exemplo 17.10



```
>>> class Musica:
    def __init__(self, titulo, artista, album, ano, genero):
        self.titulo = titulo
        self.artista = artista
        self.album = album
        self.ano = ano
        self.genero = genero
    def exhibir(self):
        s = '{}\n {} \n {} \n {}'
        print(s.format(self.titulo, self.artista,
                        self.album, self.ano, self.genero))

>>> m1 = Musica('Time', 'Pink Floyd', 'The dark side of the moon', 1973, 'Rock')
>>> m2 = Musica('Óculos', 'Paralamas do Sucesso', 'O passo do Lui', 1984, 'MPB')
>>> m3 = Musica('Evidências', 'Chitãozinho e Xororó', 'Cowboy do Asfalto', 1990,
                'Sertanejo')

>>> m1.titulo
'Time'
>>> m2.titulo
'Óculos'
>>> m3.titulo
'Evidências'
>>> m1.exibir()
Time
Pink Floyd
The dark side of the moon
1973
Rock
>>> m2.exibir()
Óculos
Paralamas do Sucesso
O passo do Lui
1984
MPB
>>> m3.exibir()
Evidências
Chitãozinho e Xororó
Cowboy do Asfalto
1990
Sertanejo
```

(exemplo interativo feito com IDE Idle)

Nesta classe são definidos 5 atributos de instância dentro do método `.__init__()` que são inicializados com os valores passados como argumentos ao construtor da classe `Musica()`.

Em seguida, criamos três instâncias `m1`, `m2` e `m3`, que são objetos da classe `Musica`, cada um contendo seu próprio conteúdo.

Note que para acessar um atributo de instância, dentro de um dos métodos da classe, é preciso usar a notação com a palavra-chave `self`, desta forma: `self.nome_do_atributo`. Essa notação foi usada nos dois métodos existentes na classe `Musica`.

E depois, fora da classe, para acessar qualquer um de seus atributos usa-se a notação `nome_do_objeto.nome_do_atributo`.

17.9.2 ATRIBUTOS DE CLASSE

Atributos de Classe são algo bem diferente dos atributos de instância vistos acima.

Se você já conhece programação orientada a objetos em C++ ou Java, então saiba que os atributos de classe em Python são semelhantes aos atributos estáticos dessas outras duas linguagens.

Os atributos de classe são definidos diretamente no corpo da classe, fora de qualquer método. Eles estão vinculados à própria classe e são compartilhados por todos os objetos (instâncias) dessa classe.

Todos os objetos criados a partir de uma classe compartilham os mesmos atributos de classe com seus valores originais. Por causa disso, se você alterar um atributo de classe, essa alteração afetará todos os objetos derivados.

Por exemplo, digamos que você deseja criar uma classe que mantenha uma contagem interna das instâncias que você criou. Nesse caso, você pode usar um atributo de classe como o `cont_instancias` criado da forma mostrada no exemplo 17.11.

Exemplo 17.11



```
>>> class Contador:
    cont_instancias = 0
    def __init__(self):
        Contador.cont_instancias += 1

>>> Contador.cont_instancias
0
>>> Contador()    # cria a primeira instância da classe Contador
<__main__.Contador object at 0x000001E3E47FFEF0>
>>> Contador()    # cria a segunda instância da classe Contador
<__main__.Contador object at 0x000001E3E47FFF50>
>>> Contador.cont_instancias
2
>>> obj = Contador() # cria a terceira instância da classe Contador
>>> Contador.cont_instancias
3
>>> obj.cont_instancias
3
```

(exemplo interativo feito com IDE Idle)

Esse objeto `cont_instancias` está definido no corpo da classe e dentro do método `__init__()` ele é inicializado com zero. A forma de fazer referência ao atributo de classe requer usar o nome da classe como qualificador da forma como está indicado na linha abaixo:

```
Contador.cont_instancias += 1    # acrescentamos 1 ao atributo cont_instancias
```

No exemplo 17.11 a classe foi instanciada 3 vezes de modo que o valor do atributo `cont_instancias` é 3 e ele pode ser acessado fora da classe, tanto através do nome da classe, como através de uma de suas instâncias.

No próximo exemplo, 17.12, usamos o atributo `__dict__` para inspecionar os atributos de classe.

Exemplo 17.12



```
# Parte 1 - Criação da classe e de instâncias dessa classe
>>> class Exemplo:
    fruta = 'Laranja'
```

```
        cor = 'Azul'
        def __init__(self, valor):
            self.valor = valor
# Criação do objeto 1
>>> obj1 = Exemplo(125)
>>> obj1.fruta
'Laranja'
>>> obj1.cor
'Azul'
>>> obj1.valor
125

# Criação do objeto 2
>>> obj2 = Exemplo(3.75)
>>> obj2.fruta
'Laranja'
>>> obj2.cor
'Azul'
>>> obj2.valor
3.75

# Parte 2 - Exploração dos atributos de classe
>>> Exemplo.fruta
'Laranja'
>>> Exemplo.cor
'Azul'
>>> Exemplo.valor # Ocorre erro porque .valor é atributo de instância
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    Exemplo.valor
AttributeError: type object 'Exemplo' has no attribute 'valor'

# Alteração nos atributos de classe
>>> Exemplo.fruta = 'Maçã'
>>> Exemplo.cor = 'Verde'

>>> obj1.fruta # todos os objetos da classe Exemplo refletem a alteração
'Maçã'
>>> obj2.fruta
'Maçã'
>>> obj1.cor
'Verde'
>>> obj2.cor
'Verde'

# Inspeção do __dict__ da classe Exemplo
>>> for s in Exemplo.__dict__:
    print(s)
(' __module__', ' __main__')
('fruta', 'Maçã')
('cor', 'Verde')
(' __init__', <function Exemplo.__init__ at 0x0000014AFEB6F2E0>)
(' __dict__', <attribute ' __dict__' of 'Exemplo' objects>)
(' __weakref__', <attribute ' __weakref__' of 'Exemplo' objects>)
(' __doc__', None)

# Inspeção do __dict__ das instâncias da classe Exemplo
>>> for s in obj1.__dict__.items():
    print(s)
('valor', 125) # apenas o atributo de instância está contido no objeto

>>> for s in obj2.__dict__.items():
    print(s)
('valor', 3.75) # apenas o atributo de instância está contido no objeto
(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)
```


Na primeira parte deste exemplo definimos a classe `Exemplo` que contém dois atributos de classe – `fruta` e `cor` – e um atributo de instância – `valor`. Em seguida, criamos dois objetos:

- `obj1` é criado com o valor 125 e nós o utilizamos para exibir os três atributos;
- `obj2` é criado com o valor 3.75 e nós o utilizamos para exibir os três atributos;
- no final usamos a própria classe `Exemplo` para exibir seus atributos de classe e tudo funciona. Porém, quando tentamos usar a classe para exibir o valor obtemos uma mensagem de erro. Isso ocorre porque `valor` é um atributo de instância e não pode ser acessado através do nome da classe.

Na segunda parte usamos a classe `Exemplo` para alterar os dois atributos de classe, atribuindo novos conteúdos aos atributos `fruta` e `cor`. Na sequência mostramos que os dois objetos – `obj1` e `obj2` – refletem essa alteração, passando a conter os novos valores atribuídos, pois atributos de classe são compartilhados por todas as instâncias.

Por fim, inspecionamos o dicionário de elementos da classe `Exemplo` e nele podemos verificar a presença dos dois atributos `fruta` e `cor`, constatando assim que ambos estão ligados à classe e não aos objetos.

17.10 ATRIBUTOS E MÉTODOS DINÂMICOS

Ainda fazendo referência ao exemplo 17.12 e dando continuidade ao mesmo vamos fazer uma alteração no atributo `fruta` do objeto `obj1`. Veja a seguir que na instância `obj1` **"parece que"** alteramos o conteúdo do atributo `fruta` de `Maçã` para `Limão`.

Exemplo 17.12

continuação

```
>>> obj1.fruta
'Maçã'
>>> obj1.fruta = 'Limão'
>>> obj1.fruta
'Limão'
>>> for s in obj1.__dict__.items():
    print(s)
('valor', 125)
(('fruta', 'Limão'))
```

(exemplo interativo feito com IDE Idle – destaques em negrito feitos pelo autor)

Mas não foi isso que aconteceu, pois a única forma de alterar um atributo de classe é através da classe. O que acabamos de fazer foi criar um **Atributo Dinâmico** na instância `obj1`.

Não é possível alterar um atributo de classe através de uma instância.

Se tentarmos fazer isso o que, na prática, faremos é criar um Atributo Dinâmico na instância.

Em outras palavras, a atribuição:

```
obj1.fruta = 'Limão'
```

criou um novo atributo de instância no objeto `obj1`, com o mesmo nome do atributo de classe existente na classe `Exemplo`. Ou seja, no objeto `obj1` você sobrescreveu o atributo de classe `fruta` com um objeto de instância dinâmico com o mesmo nome. Ao fazer isso não é mais possível acessar o atributo de classe `fruta` a partir da instância `obj1`.

Do que foi visto até aqui, extraímos um novo conhecimento sobre a linguagem Python. Ela possui o recurso de permitir que atributos e métodos sejam acrescentados de forma dinâmica tanto às classes, quanto aos objetos.

Este recurso permite agregar novos dados e funcionalidades ao software, permitindo realizar adaptações a classes existentes conforme necessidades específicas de um determinado projeto.

Por outro lado, devemos ter cuidado com esse recurso para evitar cometer o erro de, acidentalmente, criar um atributo de instância com o mesmo nome de um atributo de classe previamente existente. Para evitar fazer isso, sempre poderemos consultar o atributo `__dict__` da classe e das instâncias para verificar se não criamos uma sobreposição de nomes.

17.11 ATRIBUTOS DE CLASSE, DE INSTÂNCIA E DINÂMICOS - SÍNTESE

Em síntese, dentro da classe a forma de acesso aos atributos seguem as seguintes regras:

- para atributos de instância usa-se o qualificador `self`:

```
self.nome_do_atributo
```

- para atributos de classe usa-se o nome da classe como qualificador:

```
Classe.nome_do_atributo
```

E fora da classe as regras que se aplicam, são:

- para atributos de instância usa-se o nome do objeto como qualificador:

```
objeto.nome_do_atributo
```

- para atributos de classe usam-se ou o nome da classe ou o nome do objeto como qualificador, desde que neste último caso não seja para atribuição:

```
objeto.nome_do_atributo  
Classe.nome_do_atributo
```

Ao fazer uma atribuição como nas linhas a seguir, e supondo que `nome_do_atributo` ainda não exista, você estará criando um atributo dinâmico.

```
objeto.nome_do_atributo  
Classe.nome_do_atributo
```

17.12 ATRIBUTOS GERENCIADOS - PROPRIEDADES

A linguagem Python permite associar funcionalidades aos atributos e quando isso é feito estamos criando o que chamamos de **Atributos Gerenciados** ou **Propriedades**. Neste texto usaremos o termo propriedade para nos referir aos atributos gerenciados.

Para quem já conhece programação orientada a objetos com C++ ou Java sabe da existência do princípio do encapsulamento de atributos e do acesso através dos métodos getters e setters. Nessas linguagens temos que:

- método **getter**: existe para retornar o valor que está armazenado no atributo;
- método **setter**: existe para que um novo valor possa ser atribuído ao atributo;

A vantagem de usar essa abordagem é que dentro desses métodos pode-se fazer tarefas associadas com a manipulação dos atributos, por exemplo, validações de seu conteúdo.

Em Python, as propriedades assumem esse papel. O objetivo de usar esse recurso é garantir a consistência aos valores contidos nos atributos de um objeto.

Vamos ver isso na prática recorrendo à classe `Retangulo` escrita no exemplo 17.2 e para a qual desejamos implementar uma melhoria. Assim, criamos o exemplo 17.13. no qual queremos validar os valores atribuídos ao atributo `base` para que apenas números positivos sejam aceitos. Para termos um elemento de comparação não vamos alterar o código para o atributo `altura`.

Para validar o atributo `base` vamos implementá-lo como uma propriedade. Veja o código a seguir:

Exemplo 17.13



```
class Retangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    @property
    def base(self):
        return self._base

    @base.setter
    def base(self, valor):
        if not isinstance(valor, int | float) or valor <= 0:
            raise ValueError('Número positivo esperado')
        self._base = valor

    def calc_area(self):
        return self.base * self.altura

    def exhibe(self):
        print(f'retângulo {self.base} x {self.altura}')
        print(f' área = {self.calc_area()}')
```

(exemplo interativo feito com IDE Idle – destaques em negrito feitos pelo autor)

No método `__init__()` não é necessário fazer qualquer alteração. Ele continua recebendo os dois valores de `base` e `altura` e atribuindo-os aos atributos correspondentes.

Para transformar um atributo em uma propriedade devemos criar os métodos: getter e setter.

decorator

Um decorator em Python é
um modificador de comportamento de uma função

Note que ao criar os métodos getter e setter nossa classe terá dois métodos com o mesmo nome, no exemplo este nome é `base`. Porém, cada um é afetado pela aplicação do respectivo *decorator*.

E é isso mesmo. Essa é uma das formas de criação de getters e setters no Python.

Agora vamos detalhar e compreender o getter e o setter.

17.12.1 CRIAÇÃO DO SETTER – DECORATOR `@attr_name.setter`

O setter serve para receber um valor e armazená-lo na classe. Ele é criado com o *decorator* `@attr_name.setter`, onde `@attr_name` é o nome do atributo que queremos implementar. Obrigatoriamente o nome do método deve ser o nome do atributo e ele deve conter dois argumentos: o `self` e o `valor` que precisa ser recebido para armazenamento:

```
@base.setter
def base(self, valor):
    if not isinstance(valor, int | float) or valor <= 0:
        raise ValueError('Número positivo esperado')
    self._base = valor
```

Neste exemplo, dentro do setter o principal objetivo é armazenar o valor recebido no argumento não público `_base` (note a presença do underline).

Mas não é qualquer conteúdo que será aceito, apenas números positivos. Assim, antes de fazer a atribuição nós fazemos a validação do `valor` recebido: se for inválido levantamos uma exceção da classe `ValueError` com uma mensagem apropriada. O valor será inválido se acontecer uma de duas situações:

- `not isinstance(valor, int | float)`: verifica se `valor` é um objeto `int` ou `float`;
- `valor <= 0`: verifica se `valor` é menor ou igual a zero;

Caso ambas as condições acima sejam falsas, então o `valor` é válido e será atribuído a `_base`.

É importante não confundir o atributo não público `_base` com `base` (com e sem underline). Na forma como essa classe está escrita, não existe mais um atributo `base` (sem underline). O nome `base` existe na forma de getter e setter e o nome `_base` existe na forma de atributo.

17.12.2 CRIAÇÃO DO GETTER – DECORATOR `@property`

Para criar o getter devemos aplicar o *decorator* `@property`. Obrigatoriamente o nome do método deve ser o nome do atributo que queremos retornar, como mostrado nestas linhas:

```
@property
def base(self):
    return self._base
```

O nome do método é `base`, ele deve receber `self` como primeiro argumento e deve retornar o conteúdo armazenado no atributo não-público `_base`.

Agora que as alterações na classe já foram feitas precisamos testar seu funcionamento. Para isso o código a seguir é a continuação do exemplo 17.13 onde usamos a classe `Retangulo`.

Exemplo 17.13

continuação

```
>>> r1 = Retangulo(10, 5)
>>> r1.calc_area()
50
>>> r1.base = 9
>>> r1.calc_area()
45
# a altura aceita números negativos
>>> r1 = Retangulo(10, -5)
>>> r1.calc_area()
-50
>>> r1.altura = -9
>>> r1.calc_area()
-90

# a base não aceita números negativos
>>> r1.base = -9
Traceback (most recent call last):
  File "<pyshell#91>", line 1, in <module>
    r1.base = -9
  File "<pyshell#86>", line 12, in base
    raise ValueError('Número positivo esperado')
ValueError: Número positivo esperado

>>> r1 = Retangulo(-10, 5)
Traceback (most recent call last):
  File "<pyshell#92>", line 1, in <module>
    r1 = Retangulo(-10, 5)
  File "<pyshell#86>", line 3, in __init__
    self.base = base
  File "<pyshell#86>", line 12, in base
    raise ValueError('Número positivo esperado')
ValueError: Número positivo esperado
```

(exemplo interativo feito com IDE Idle)

A respeito deste código, é importante destacar que a validação da base é feita tanto na atribuição direta de um valor à propriedade, quanto na construção de um objeto. Assim, atribuições feitas como abaixo passam pela validação feita no `@base.setter`:

```
r1.base = -9 # a validação será feita e o valor negativo não será aceito
```

E a construção de novo objeto feito com a linha a seguir também passam pela validação feita no `@base.setter`

```
r1 = Retangulo(-10, 5) # neste caso a validação também será feita
```

O uso de getters e setters é um recurso amplamente usado em programação orientada a objetos e em Python essa é a forma moderna e mais atual de fazer sua implementação. Existem outras formas, mas por serem mais antigas e menos usadas atualmente, não serão abordadas neste texto. Caso queira conhecer mais sobre isso pesquise sobre a função `property`. Um ótimo ponto de partida para isso é este link da referência de Python docs <https://docs.python.org/3/library/functions.html#property>

17.13 MÉTODOS MÁGICOS

A esta altura do curso você já deve ter se acostumado a ver os integrantes de classe cujos nomes começam e terminam com duplo underline, como `__dict__`, `__init__` e tantos outros. Esses integrantes podem ser atributos como no caso do `__dict__` ou método como o `__init__`.

Os métodos mágicos são exatamente esses métodos que iniciam e terminam com duplo underline.

É óbvio que não há qualquer magia envolvida, mas eles ganharam essa denominação porque são métodos que realizam tarefas nos bastidores. A documentação oficial de Python afirma literalmente que "Método Mágico" é um termo informal para "Método Especial".

Eles existem para executar um código que um programador Python não sabe que está sendo executado, a menos que seja um profissional avançado que conhece e que seja capaz de escrever classes Python. Um programador que apenas utiliza as classes existentes não toma conhecimento daquilo que os métodos mágicos são capazes de realizar. Em termos práticos, você estudante deste curso, já usou mais métodos mágicos do que é capaz de imaginar, e nem sabia que eles estavam lá.

E isso é bom, pois se você foi capaz de usar tais recursos sem saber de sua existência, isso ocorreu porque Python está construído de uma maneira inteligente e funcional a tal ponto que permitiu essa situação. Vamos ver no exemplo a seguir quando você usou métodos mágicos sem saber:

Exemplo 17.14

Teste este código no Idle

```
>>> texto = 'Métodos Mágicos em uso'
>>> print(texto)                # nesta linha você usou o método .__str__()
Métodos Mágicos em uso
>>> texto                        # nesta linha você usou o método .__repr__()
'Métodos Mágicos em uso'
>>> len(texto)                  # nesta linha você usou o método .__len__()
22

>>> for x in dir(texto):        # vamos listar todos os elementos no "__" no nome
>>>     if '___' in x:
>>>         print(x)
__add__
__class__
__contains__
...
__le__
__len__
__lt__
__mod__
__mul__
__ne__
__new__
__reduce__
__reduce_ex__
__repr__
__rmod__
__rmul__
__setattr__
__sizeof__
__str__
__subclasshook__
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Neste exemplo, definimos o objeto `texto` que é da classe `str` e fizemos três operações com ele: exibimos seu conteúdo de duas formas e exibimos seu comprimento. Na prática o que aconteceu foi:

- exibição usando a função `print()`: nos casos de uso que envolvem a função `print()` o interpretador Python tem o comportamento padrão de usar o método mágico `.__str__()` para determinar o que deve ser exibido na tela;

- exibição usando diretamente o nome do objeto: este é um caso que se aplica aos ambientes interativos e o interpretador Python tem o comportamento padrão de usar o método `.__repr__()`, sendo que "repr" vem de "representação do objeto" – a documentação afirma que este método é destinado aos desenvolvedores e seu objetivo é permitir que o programador acesse e tenha conhecimento de detalhes da classe;
- exibição do tamanho usando a função `len()`: nos casos de uso que envolvem a função `len()` o interpretador Python tem o comportamento padrão de usar o método mágico `.__len__()` para determinar o tamanho. Esse tamanho retornado por `len()` sempre se refere à quantidade de elementos contidos em um objeto e se aplica a objetos compostos como strings, listas, tuplas, dicionários e conjuntos. Se você escrever uma classe que contenha elementos, então convém que faça a implementação do método `.__len__()` para seguir o padrão definido na linguagem.

Ainda no exemplo acima, no seu final, fizemos um código para exibir todos os elementos que contém duplo underline no nome. Dentre esses elementos há aqueles que são atributos e outros são métodos. Esses são os métodos mágicos implementados na classe `str`.

Cada classe terá seus próprios métodos mágicos, segundo específicas necessidades de implementação. Vamos ver isso de forma prática.

Exemplo 17.15



```
# módulo m_exemplo_17_15.py

class Veiculo:
    def __init__(self, placa, modelo, ano, km):
        self.placa = placa
        self.modelo = modelo
        self.ano = ano
        self.km = km

    def exibe(self):
        print(f'Veículo placa {self.placa}')
        print(f'    {self.modelo}, ano: {self.ano} - km: {self.km}')

    def __str__(self):
        s = f'Veículo placa {self.placa}\n'
        s += f'    {self.modelo}, ano: {self.ano} - km: {self.km}'
        return s

    def __repr__(self):
        s = 'instância da classe Veiculo,\n'
        s += '    carregada com os seguintes dados:\n'
        s += f'        {self.placa}\n'
        s += f'        {self.modelo}\n'
        s += f'        {self.ano}\n'
        s += f'        {self.km}'
        return s

    def __len__(self):
        return 1
```

(módulo)

No exemplo 17.15 mostramos a criação de uma classe na qual implementamos os três métodos mágicos `.__str__()`, `.__repr__()`, `.__len__()`. Essa classe é uma variação da classe `Veiculo`

apresentada no exemplo 17.3, que implementamos em um módulo denominado `m_exemplo_17_15.py`, dentro do pacote `cap17`. Em seguida, a utilizamos no `Idle`, para demonstrar seu funcionamento.

Exemplo 17.15 - continuação

```
>>> import sys
>>> sys.path.append('C:\CursoPython')

>>> from cap17.m_exemplo_17_15 import Veiculo    # importação da classe Veiculo
>>> v = Veiculo('ELM8038', 'CELTA EL 1.0', 2012, 76320)
>>> v      # esta chamada usa o método .__repr__() da classe Veiculo
instância da classe Veiculo,
carregada com os seguintes dados:
. ELM8038
. CELTA EL 1.0
. 2012
. 76320
>>> print(v)    # esta chamada usa o método .__repr__() da classe Veiculo
Veículo placa ELM8038
CELTA EL 1.0, ano: 2012 - km: 76320
>>> len(v)     # esta chamada usa o método .__len__() da classe Veiculo
1
(exemplo interativo feito com IDE Idle)
```

Agora que já conhecemos o conceito de métodos mágicos, vamos mostrar como eles podem ser usados para aprimorar a criação de getters e setters.

17.14 DESCRITORES

Outra forma de criar getters e setters é usando um elemento de Python conhecido como descritor. Os descritores são um recurso poderoso para criar atributos gerenciados.

Você deve ter percebido que aplicamos o conceito de atributos gerenciados apenas ao atributo `base` da classe `Retangulo`. Propositamente, deixamos o atributo `altura` de lado, para incluí-lo agora. Com essa inclusão vamos mostrar como os descritores podem ser interessantes e poderosos.

Veja abaixo como fica o Exemplo 17.13 ao transformar o atributo `altura` em uma propriedade.

Exemplo 17.13 - revisitado**Inclusão de altura como propriedade**

```
class Retangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    @property
    def base(self):
        return self._base

    @base.setter
    def base(self, valor):
        if not isinstance(valor, int | float) or valor <= 0:
            raise ValueError('Número positivo esperado')
        self._base = valor

    @property
    def altura(self):
        return self._altura
```



```
@altura.setter
def altura(self, valor):
    if not isinstance(valor, int | float) or valor <= 0:
        raise ValueError('Número positivo esperado')
    self._altura = valor

def calc_area(self):
    return self.base * self.altura

def exibe(self):
    print(f'retângulo {self.base} x {self.altura}')
    print(f' área = {self.calc_area()}')
```

(exemplo interativo feito com IDE Idle - destaques em negrito feitos pelo autor)

Compare os trechos em azul (propriedade `base`) e verde (propriedade `altura`) e verifique como os dois trechos de código são essencialmente iguais, com a óbvia distinção apenas quanto ao nome da propriedade. É muito comum que isso aconteça quando estamos desenvolvendo classes com certo grau de complexidade, que tem a necessidade de muitas propriedades e existe a grande chance de várias delas terem comportamento e funcionalidades parecidas, diferenciando-se apenas quanto à natureza do valor contido.

Em virtude da frequente e repetitiva ocorrência de trechos de código assim é possível tornar o código mais compacto se forem usados os descritores (em inglês, *descriptors*).

Um descritor é uma classe que segue o chamado protocolo de descritor (em inglês, *descriptor protocol*). Para seguir esse protocolo, basta que na classe seja implementado um dos métodos mágicos listados abaixo:

```
__get__(self, obj, type=None) # deve retornar um objeto (getter)
__set__(self, obj, value)     # deve armazenar o valor passado em value (setter)
__delete__(self, obj)         # executado na exclusão do objeto (deleter)
```

Uma classe que **implemente pelo menos um destes métodos será um descritor**.

Por outro lado, um descritor pode ser embutido em uma classe e desse modo permitir a criação de propriedades de uma maneira mais compacta e sem redundância de código.

Adicionalmente para que um descritor seja efetivamente útil é preciso que a classe que o contém tenha acesso ao nome do objeto (instância) e para isso deve-se, também, implementar o método `__set_name__()`.

```
__set_name__(self, owner, name) # proverá acesso ao nome da instância no código
```

Para saber mais sobre descritores
acesse este link

<https://docs.python.org/pt-br/3/howto/descriptor.html>
e você encontrará um artigo muito completo
abordando todos os aspectos sobre os assuntos

Assim, vejamos como isso é feito em termos práticos, através do exemplo 17.16

Exemplo 17.16



```
# módulo m_exemplo_17_16.py

class NumeroPositivo:
    def __set_name__(self, owner, name):
        self._name = name

    def __get__(self, obj, objtype=None):
        return obj.__dict__[self._name]

    def __set__(self, obj, value):
        if not isinstance(value, int | float) or value <= 0:
            raise ValueError('Número positivo esperado')
        obj.__dict__[self._name] = value

class Retangulo:
    base = NumeroPositivo()
    altura = NumeroPositivo()

    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calc_area(self):
        return self.base * self.altura

    def exhibe(self):
        print(f'retângulo {self.base} x {self.altura}')
        print(f' área = {self.calc_area()}')
```

(módulo - destaques em negrito feitos pelo autor)

No código acima, foi declarada a classe `NumeroPositivo` que é o descritor a ser usado para implementar os getters e setters da classe `Retangulo`. Nesta classe temos os seguintes detalhes:

- O método `NumeroPositivo.__set_name__(self, owner, name)` é usado para que se tenha acesso ao nome do atributo que está sendo manipulado. Este nome será usado nas operações de recuperação e atribuição de conteúdo para o atributo;
- O método `NumeroPositivo.__get__()` seu argumento `obj` se refere à instância para a qual o descritor está sendo chamado e é usado para retornar o objeto, funcionando como um getter;
- O método `NumeroPositivo.__set__()` seu argumento `obj` se refere à instância para a qual o descritor está sendo chamado, e o argumento `value` é usado para armazenar seu valor no atributo, funcionando como um setter.

Depois de pronto o descritor, passamos à codificação da classe `Retangulo`, dentro da qual usaremos o descritor `NumeroPositivo`.

Para implementar uma propriedade com o uso do descritor é preciso criar um atributo de classe desta forma:

```
base = NumeroPositivo()
```

Esse atributo passa a ser uma instância da classe `NumeroPositivo` e, por consequência, passa a ser uma propriedade, ou atributo gerenciado. O mesmo vale para o atributo `altura`.

E assim, com um código mais compacto temos dois atributos gerenciados que se comportam de modo semelhante, porém armazenando dados diferentes. Isso pode ser visto no programa principal mostrado abaixo para testar o funcionamento dessa versão da classe `Retangulo`.

Exemplo 17.16 - continuação

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
>>> from cap17.m_exemplo_17_16 import Retangulo
>>> r1 = Retangulo(12.1, 7.8)
>>> r1.calc_area()
94.38
>>> r1.base = -7.8
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    r1.base = -7.8
  File "D:\Python\ModuloAvancado\cap17\m_exemplo_17_16.py", line 10, in __set__
    raise ValueError('Número positivo esperado')
ValueError: Número positivo esperado
>>> r1.altura = -3.5
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    r1.altura = -3.5
  File "D:\Python\ModuloAvancado\cap17\m_exemplo_17_16.py", line 10, in __set__
    raise ValueError('Número positivo esperado')
ValueError: Número positivo esperado
```

(exemplo interativo feito com IDE Idle)

17.15 HERANÇA E HIERARQUIA DE CLASSES

A herança é um recurso muito importante e poderoso da programação orientada a objetos. Consiste na criação de um relacionamento hierárquico entre duas classes, onde uma será a classe base e outra será a herdeira. O trecho de código a seguir ilustra a forma que essa relação assume:

```
class Base:
    # todo o código da classe base vem aqui

class Derivada(Base):
    # todo o código da classe herdeira vem aqui
```

Nessa relação a classe `Base` é chamada de classe superior e a classe `Derivada` é chamada de herdeira. Esta construção implementa o que é conhecido como uma herança simples, pois a classe herdeira tem apenas uma classe superior.

Em Python, uma classe pode ter várias classes superiores, caracterizando o que se conhece como herança múltipla. Este aspecto não será abordado neste texto.

É comum que uma classe base dê origem a várias classes herdeiras.

A herança é um recurso importante pois permite reduzir a quantidade de código, através da reutilização de código que se aplica a diferentes classes que possuem aspectos em comum. Tais aspectos em comum serão implementados em uma classe superior e com isso não haverá duplicação de código nas classes herdeiras. Além disso, pode tornar seu código mais modular e mais bem organizado.

O exemplo 17.17 mostra um exemplo bastante simples e seu objetivo é ilustrar uma implementação de relação de herança entre uma classe base e duas classes herdeiras. Pela simplicidade do exemplo não é possível extrair todo o potencial da herança, mas ele serve para ilustrar a forma de implementação de herança em Python.

Exemplo 17.17



```
# módulo m_exemplo_17_17.py

class FormaGeometrica:
    def __init__(self):
        self.tipo_forma = 'forma não definida'
    def exibe_tipo_forma(self):
        print(self.tipo_forma)

class Circulo(FormaGeometrica):
    def __init__(self, raio):
        self.tipo_forma = 'Círculo'
        self.raio = raio
    def exibe_dados(self):
        super().exibe_tipo_forma()
        print(f' detalhes: raio = {self.raio}')

class Retangulo(FormaGeometrica):
    def __init__(self, base, altura):
        self.tipo_forma = 'Retângulo'
        self.base = base
        self.altura = altura
    def exibe_dados(self):
        super().exibe_tipo_forma()
        print(f' detalhes: {self.base} x {self.altura}')
```

(módulo)

A classe `FormaGeometrica` possui dois métodos: o método não-público `__init__()` e o método público `exibe_tipo_forma()`. Note que enquanto o primeiro cria o atributo nome e o inicializa com um string vazio, o segundo método exibe o argumento de instância `tipo_forma`. Esta classe estabelece elementos básicos que farão parte da hierarquia de classes nela baseada.

Em seguida são definidas duas classes especializadas: `Circulo` e `Retangulo`. Essas duas classes são herdeiras da classe `FormaGeometrica` e implementam dois métodos `__init__()` e `exibe_dados()`.

Em cada uma dessas classes o método `__init__()` tem argumentos específicos conforme o caso. O método `exibe_dados()` de cada uma delas, por sua vez, utiliza o método `tipo_forma()`. Note que as classes especializadas não possuem tal método definido em seu próprio código. No entanto, essas classes possuem esse método através da herança, pois ele está definido na classe base.

A seguir ilustramos o uso dessas três classes.

No código abaixo criamos e usamos um objeto de cada uma das classes contidas no módulo `m_exemplo_17_17.py`.

Exemplo 17.17 - continuação

```
>>> import sys
>>> sys.path.append('C:\CursoPython')
```

```
>>> from cap17.m_exemplo_17_17 import *

>>> c = Circulo(4.4)
>>> c.exibe_dados()
Circulo
    detalhes: raio = 4.4

>>> r = Retangulo(3.6, 7.2)
>>> r.exibe_dados()
Retângulo
    detalhes: 3.6 x 7.2

>>> r.base = 8.8
>>> r.altura = 4.5
>>> r.exibe_dados()
Retângulo
    detalhes: 8.8 x 4.5

>>> f = FormaGeometrica()
>>> f.exibe_dados()
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    f.exibe_dados()
AttributeError: 'FormaGeometrica' object has no attribute 'exibe_dados'

>>> f.exibe_tipo_forma()
forma não definida
```

(exemplo interativo feito com IDE Idle)

Quando se trata de usar as classes `Circulo` e `Retangulo` os métodos públicos `.exibe_tipo_forma()` e `.exibe_dados()` estão disponíveis para ambas. O primeiro através da herança, pois ele está implementado na classe base `FormaGeometrica`, e o segundo porque é implementado nas próprias classes específicas.

Como mostrado na segunda parte do exemplo 17.17, no programa principal podemos usar tanto a classe base como as classes herdeiras, desde que respeitemos os métodos existentes em cada uma.

Exercício Proposto 17.1

Enunciado: Escreva uma classe Python para conter dados de um Estoque de produtos. Os atributos dessa classe devem ser: código, nome, quantidade, preço de custo; e os métodos devem ser: adicionar_item, atualizar_item, exhibir_item.

Para armazenamento, interno use um dicionário, no qual a chave seja o código e o conteúdo seja uma lista com os demais valores.

O método adicionar_item deve levantar uma exceção caso o código já exista.

O método atualizar_item deve levantar uma exceção caso o código não exista

Exercício Proposto 17.2

Enunciado: Escreva uma classe Python que simule uma TV.

Atributos canal; volume; brilho, contraste e modo (TV ou INTERNET)

Métodos alterar_canal – recebe o número do canal, valida e guarda (os canais válidos são de 100 a 499)

alterar_volume – recebe o número, valida e guarda (valores válidos entre 0 e 100)

alterar_brilho e alterar_contraste – idem ao alterar_volume

alterar_modos – recebe o modo, valida e guarda (apenas os modos TV e INTERNET são válidos)

exibir_detalhe – deve exibir na tela todos os parâmetros configurados no momento

As validações deve levantar exceções em caso de erro

Pgm principal deve testar o uso dessa classe criando um objeto e permitindo manipular seus atributos.

Capítulo 18

Uso DE PYTHON COM SQLITE 3

Anteriormente neste curso vimos como usar arquivos texto para armazenar dados de forma permanente. Neste capítulo vamos trabalhar com o gerenciador de banco de dados SQLite 3, que representa uma outra forma de armazenamento de dados.

Um dos objetivos deste capítulo é mostrar como um bom conjunto de módulos de funções amplia grandemente a capacidade de Python 3 ser usada como uma ferramenta de programação, poderosa e produtiva.

18.1 SISTEMAS GERENCIADORES DE BANCOS DE DADOS

Há décadas os sistemas gerenciadores de banco de dados (SGBD) vem sendo desenvolvidos. Esses gerenciadores são softwares especializados no armazenamento, recuperação e manutenção de volumes de dados.

Como consequência desse desenvolvimento surgiram duas vertentes distintas dessa categoria de software, cada uma com suas especificidades e características. Elas são conhecidas de forma genérica por:

- Bancos de Dados relacionais: neste tipo de gerenciador é usada uma linguagem conhecida como SQL, que é a abreviação de Structured Query Language;
- Bancos de Dados não-relacionais: também conhecidos como NoSQL, pois nesse tipo de gerenciador não é usada a linguagem SQL;

Existem vários SGBD's SQL como: Oracle, Microsoft SQL Server, MySQL, etc. A lista é bem grande e o SQLite é um deles. Alguns deles são softwares proprietários e licenças precisam ser adquiridas para que se possa utilizá-los. Por outro lado, há os que são softwares livres e podem ser usados sob determinadas condições, a depender da licença de software livre sob a qual é disponibilizado.

Neste capítulo vamos ver como utilizar o SQLite em conjunto com a linguagem Python 3. Com isso, você terá uma visão geral de como é possível usar Python para interagir com gerenciadores de bancos de dados. Os conceitos essenciais vistos com SQLite se aplicam também a outros SGBDs.

Mas, antes de começar precisamos apresentar e compreender conceitos básicos essenciais dos gerenciadores de banco de dados relacionais. Então faremos uma pausa em Python e nas próximas seções apresentamos conceitos de banco de dados para depois seguirmos com o Python.

18.2 BANCOS DE DADOS RELACIONAIS

A organização dos dados em um SGBD relacional está fundamentada em tabelas, como a mostrada na figura 18.1.

Figura 18.1 – Elementos de uma tabela de banco de dados relacional

Nome da tabela: **Contratos_Aluguel**

colunas são chamadas de campos

numeros dos campos	NumContrato	CPFInquilino	IdImovel	DtInicio	Duracao	Aluguel	ReajusteEm
linhas são registros	3318	999.999.999-99	3M-005	01/03/2022	36 meses	1250,00	31/03/2024
	3319	999.999.998-98	1B-012	15/10/2023	24 meses	3310,00	14/10/2024
	3320	999.999.997-97	4T-001	01/06/2017	Indeterminado	2188,14	31/05/2024

chave primária

fonte: o Autor

Esta figura contém os elementos essenciais de uma tabela de banco de dados relacional, a saber:

- **Nome da tabela:** toda tabela existente na base de dados deve ter um nome de acordo com as regras de nomes adotadas no SGBD escolhido. Normalmente se utilizam letras, números e o caractere underline "_". Na Figura 18.1 a tabela se chama Contratos_Aluguel.
- **Registro:** cada linha da tabela é denominada registro e representa uma coleção heterogênea de dados interligados. Os registros são subdivisíveis em campos.
- **Campo:** é o elemento no qual um dado é armazenado. Um conjunto de campos forma um registro, ou linha da tabela. Os campos apresentam tipo e, quando pertinente, tamanho definidos. Embora haja certa variação entre os diversos SGBDs, os tipos básicos de campos são: texto, número inteiro, número real, data, hora, ou data e hora juntos, lógico (true/false), entre outros. Campos são identificados por nomes que seguem regras semelhantes aos nomes de tabelas.
- **Chave primária:** é responsável pela identificação de cada registro no banco de dados. Pode ser constituída por um ou mais campos, e é obrigatório que seja única e não nula. No exemplo, a chave primária é o campo NumContrato.

Os bancos de dados podem conter muitas tabelas e recursos que estabelecem relacionamentos entre as tabelas. Cada tabela pode ter grande quantidade de campos e milhões de registros. Além das tabelas, os bancos de dados também têm outros elementos, como índices (*indexes*), visões (*views*), gatilhos (*triggers*), procedimentos armazenados (*stored procedures*), direitos de acesso (*grants*) etc.

Para a criação de uma estrutura completa de banco de dados são necessários conhecimentos de gerais de modelagem de dados e também conhecimentos específicos a respeito do SGBD adotado em uma aplicação. É bastante coisa a ser aprendida e, como você pode perceber pela leitura desta seção, tal volume de conceitos requer um tempo considerável de estudo e prática, justificando a existência das

disciplinas de banco de dados nos cursos de desenvolvimento de sistemas. Não há espaço neste curso para todo esse aprofundamento, mas podemos aprender um conjunto básico de elementos de banco de dados, o suficiente para que possamos aplicá-lo em muitos exercícios envolvendo a linguagem Python e o gerenciador de banco de dados SQLite.

18.3 A LINGUAGEM SQL

SQL – como já apontamos, abreviação de Structured Query Language – é um termo que se refere à linguagem utilizada para criar, armazenar, recuperar e atualizar dados em um banco de dados relacional. Seu desenvolvimento teve início nos anos 1970 pela IBM como parte do projeto que levou à criação do modelo relacional de bancos de dados. Desde então muitos SGBDs surgiram no mundo da computação e incorporaram essa linguagem.

Não se trata de uma linguagem de uso geral, como C, Java ou Python. SQL é específica e exclusivamente utilizada para interagir com bancos de dados relacionais. Seus comandos divididos em categorias, segundo as operações que realizam. A seguir listamos essas categorias:

- **DQL (Data Query Language):** nesta categoria há apenas um comando que é o `select`. O `select` é usado para buscar dados que já estão armazenados em tabelas. É o mais usado de todos os comandos SQL e tem muitas opções e possibilidades. Vamos usá-lo bastante neste capítulo.
- **DML (Data Manipulation Language):** são os comandos usados para realizar alterações nos conteúdos das tabelas. Com os comandos dessa categoria é possível fazer inclusões, alterações e exclusões de dados nas tabelas, a saber: `insert`, `update` e `delete`. Também vamos usá-los neste capítulo.
- **DDL (Data Definition Language):** são os comandos usados para manipular a estrutura das tabelas e seus elementos associados. Eles permitem criar, alterar e excluir tais elementos. Esses comandos são: `create`, `alter` e `drop`. Também vamos usá-los.
- **DCL (Data Control Language):** são os comandos utilizados para controlar o acesso aos dados das tabelas. Com eles é possível definir a visibilidade e os direitos de realizar operações aos usuários do banco de dados. Não veremos esses comandos.
- **DTL (Data Transaction Language):** são os comandos relacionados ao controle de transações no banco de dados, indicando quando e como os dados devem ser fisicamente salvos ou descartados. Existem com a finalidade de garantir a integridade de dados inter-relacionados. São eles: `commit` e `rollback`. Também vamos usá-los.

No que diz respeito aos comandos dessas categorias, há muito a ser aprendido e não temos espaço suficiente neste curso para tudo. Porém, vamos apresentar o suficiente para que você consiga ter um bom conhecimento inicial dos comandos e consiga aprofundar-se em estudos posteriores. Faremos a implementação de diversos programas Python que vão se conectar ao SQLite para realizar tarefas de criação de tabelas; seu preenchimento e recuperação de dados armazenados; alteração da estrutura de tabelas; exclusão de tabelas; importação de dados a partir de arquivos externos; entre outras.

Adiante, quando formos usar um comando SQL explicaremos sua estrutura e como ele é usado.

18.4 BANCO DE DADOS SQLITE

Há muitas opções de gerenciadores de bancos de dados e para este curso precisávamos escolher algum. O escolhido foi o SQLite por vários motivos: ele é muito simples, totalmente gratuito, amplamente disponível em muitas plataformas. Mas principalmente porque não exige qualquer instalação e muito menos configurações, que poderiam fazer o aluno iniciante perder muito tempo e provavelmente frustrar-se no processo, se as coisas dessem errado logo no início.

Se você já instalou a linguagem Python no seu computador, o SQLite está instalado também e o que é melhor: pronto para uso. Como o SQLite acompanha o pacote de Python você não precisará fazer mais nada: não requer download, não requer instalação, não requer configurações difíceis de entender, não requer um administrador de banco de dados (DBA) para preparar o ambiente para você.

Por outro lado, não imagine que você vai utilizar o SQLite em aplicações capazes de conter tabelas com milhões de registros e muitos giga bytes de tamanho ou atender um alto volume de transações e tráfego com milhares de acessos simultâneos em um segundo. Não, definitivamente, o SQLite não foi criado para isso.

O uso do SQLite é recomendado quando a simplicidade da administração, implementação e manutenção são mais importantes do que recursos relacionados a aplicações complexas, de grande volume e alto desempenho. Por mais improvável que pareça, a ocorrência dessa simplicidade é muito comum, mais comum do que muitos imaginam.

É disponibilizado na forma de software livre, com código-fonte aberto e está disponível para diversas plataformas como Windows, Linux, macOS, Android e iOS. Há milhões de aplicações que o utiliza e, como é nativo em instalações Android e iOS, está instalado em bilhões de equipamentos ao redor do mundo. A versão mais recente disponível quando este texto foi escrito é a 3.45.1, de 30 de janeiro de 2024 (fonte: www.sqlite.org, acessado em 10/02/2024).

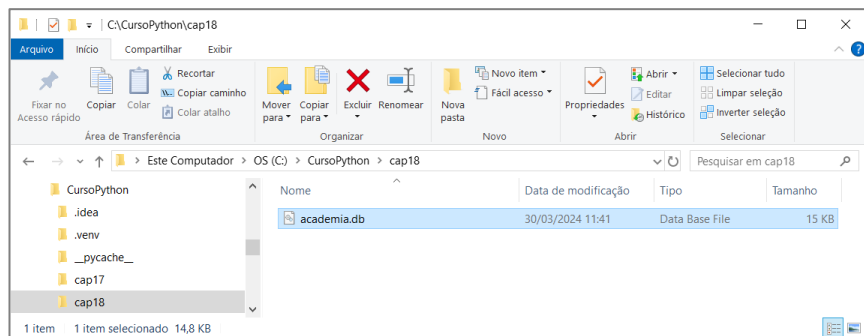
Para mais informações sobre o SQLite,
acesse seu site oficial
<https://www.sqlite.org/>

Além da home page, vale a pena dar uma olhada
na página de grandes usuários do SQLite
<https://www.sqlite.org/famous.html>

18.5 IDE PARA SQLITE

IDE é a sigla para Integrated Development Environment, ou seja, um ambiente de desenvolvimento. Vamos precisar de um IDE para "enxergar" um banco de dados SQLite. Imagine que, depois de criado, um banco de dados SQLite nada mais é do que um arquivo gravado no seu computador. Veja a figura 18.2 onde mostramos o arquivo academia.db.

Figura 18.2 – Arquivo do BD SQLite gravado no dispositivo



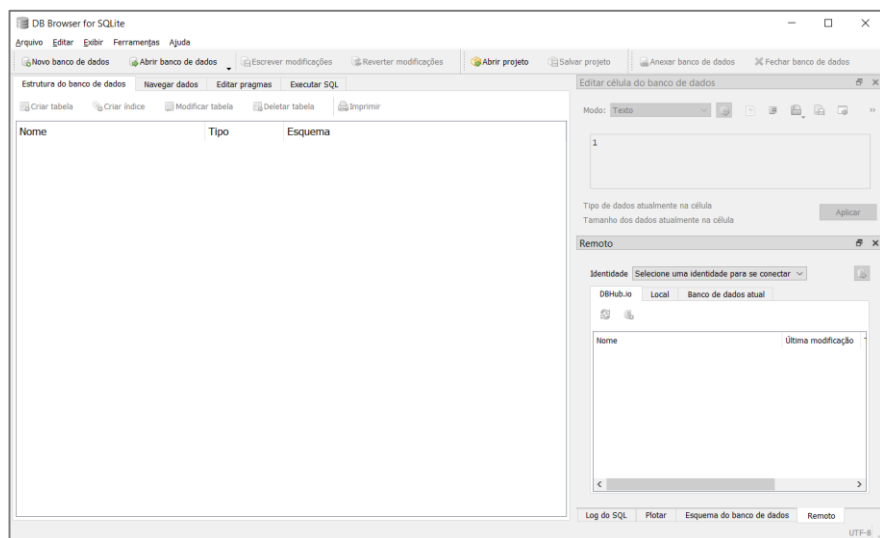
fonte: o Autor

Como enxergar o que há dentro dele?

Resposta simples: precisaremos de um IDE para isso.

Há muitos IDE's adequados para abrir e explorar um banco de dados SQLite. Neste curso usaremos o DB Browser for SQLite, que tem a aparência mostrada na figura 18.3.

Figura 18.3 – DB Browser for SQLite



fonte: o Autor

O uso deste IDE é bastante simples. Faça seu download, instale no seu computador e comece a explorá-lo. Sua interface está disponível em português e rapidamente você absorverá as operações que são possíveis com ele. Em caso de dúvida recorra à sua página oficial (em inglês) e você obterá informações que necessita.

Você poderá baixar o DB Browser for SQLite
a partir do endereço
<https://sqlitebrowser.org/>

18.6 PYTHON COM SQLITE

Agora que você já tem uma noção do que é o SQLite, vamos começar a utilizá-lo para escrever aplicações usando Python.

18.6.1 PRIMEIRO BANCO DE DADOS

O primeiro passo é saber que precisamos importar o módulo `sqlite3` e fazer a conexão com um banco de dados. Isso é mostrado no exemplo 18.1, no qual criaremos, dentro da pasta "C:\CursoPython\cap18", um arquivo de banco de dados com o nome "loja.db". A extensão .db no nome do arquivo não é necessária, mas facilita sua abertura no IDE DB Browser.

Exemplo 18.1



```
>>> import os # módulo de interação com o sistema operacional
>>> os.chdir('C:\\CursoPython\\cap18') # troca da pasta default no Idle

>>> import sqlite3
>>> conector = sqlite3.connect('loja.db') # cria o arquivo
>>> conector.close()

(exemplo interativo feito com IDE Idle)
```

Este código simples é responsável pela criação do arquivo do banco de dados, porém ele estará vazio, como é mostrado no vídeo do exemplo.

O método `.connect()` do módulo `sqlite3` fará a conexão do Python com o arquivo do banco de dados, caso ele exista. Caso não exista, esse arquivo é criado. O nome do arquivo deve ser fornecido e pode incluir um caminho qualificado, indicando a unidade de disco e a pasta desejados. Deste modo poderíamos substituir o uso do método `.os.chdir()` mostrado acima, pela linha abaixo:

```
>>> conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
```

Toda conexão aberta com o banco de dados, deve ser encerrada com o método `.close()` para liberar eventuais recursos alocados.

Agora vamos ampliar um pouco esse exemplo, com a criação de uma tabela e a inserção de alguns dados.

Exemplo 18.1

continuação feita no PyCharm

```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
# Criação da tabela
cursor = conector.cursor()
sql = """
    create table produto
    (codigo integer, descr text, preco numeric, qtdeestq integer)
    """
cursor.execute(sql)
# inserção de alguns dados
sql = """
    insert into produto (codigo, descr, preco, qtdeestq)
    values (1138, 'lápiz preto', 1.90, 376)
    """
cursor.execute(sql)
sql = """
    insert into produto (codigo, descr, preco, qtdeestq)
    values (1251, 'papel sulfite A4 100fls', 7.25, 188)
    """
cursor.execute(sql)

conector.commit() # salva a tabela e os dados no disco
```

```
cursor.close()
conector.close()
print('\nFim do Programa')
Fim do Programa
```

Para executar comandos SQL é preciso criar um cursor. E essa criação é feita a partir do objeto criado no momento da conexão:

```
cursor = conector.cursor()
```

A partir do cursor já criado, usando-se o método `.execute()` é possível executar os comandos da linguagem SQL para interagir com o gerenciador de banco de dados. Para que você possa compreender a execução dos dois comandos SQL envolvidos neste programa destacamos as explicações no quadro 18.1.

Quadro 18.1 – Comandos SQL usados no exemplo 18.1

Comando	Descrição
<pre>create table produto (codigo integer, descr text, preco numeric, qtdeestq integer)</pre>	<p>Este SQL da categoria DDL (<i>Data Definition Language</i>)</p> <p>Ele é usado para criar tabelas. Se a tabela já existir, ocorre um erro. Deve-se fornecer o nome da tabela, no caso: "produto"</p> <p>Devem-se fornecer o nome e o tipo de dados de cada campo que a tabela conterá. Neste exemplo, são quatro campos: "código" e "qtdeestq" são números inteiros (integer), "descr" é do tipo string (text) e preco é número real (numeric).</p> <p>Para esta tabela não foi definida uma chave primária.</p>
<pre>insert into produto (codigo, descr, preco, qtdeestq) values (1138, 'lápiz preto', 1.90, 376)</pre>	<p>Este SQL é da categoria DML (<i>Data Manipulation Language</i>)</p> <p>Ele é utilizado para inserir dados em uma tabela.</p> <p>O nome da tabela deve ser fornecido seguido dos campos que receberão dado. Não sendo obrigatório que todos os campos da tabela estejam presentes.</p> <p>E na sequência da cláusula <i>values</i> colocam-se os dados que vão preencher os campos especificados</p>

Os comandos SQL, em geral, são extensos e é normal escrevermos usando mais de uma linha. Assim, cada um desses comandos é um string que pode ser atribuído um objeto ou posicionado diretamente como argumento do método `.execute()`. No programa usamos o objeto `sql` para isso.

Quando o método `.execute()` é executado as alterações ao banco de dados estarão apenas em memória. Para torná-las permanentes é preciso executar o método `.commit()`, que irá transferir para o disco todas as modificações efetuadas com `.execute()`.

Observação Importante

Experimente executar esse programa duas vezes seguidas. Você obterá um erro na segunda execução, pois a tabela já existirá e haverá erro ao tentar criá-la novamente.

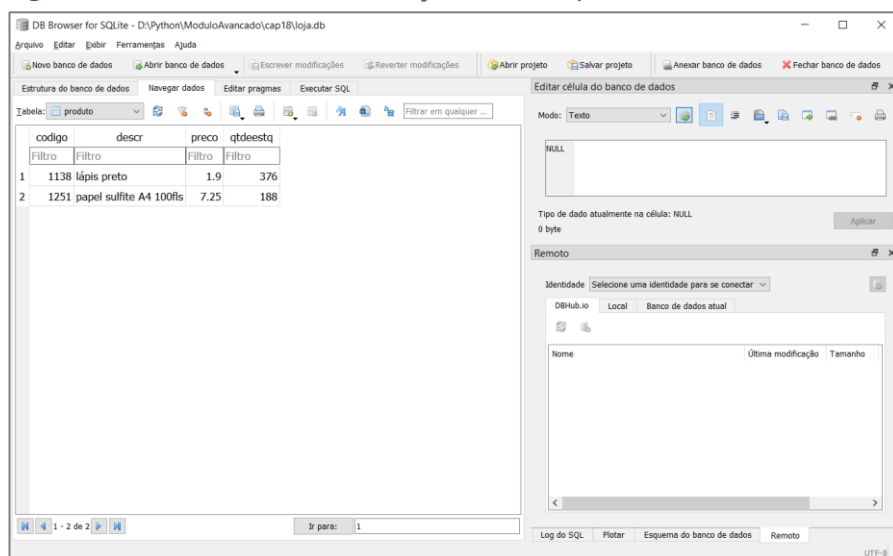
18.6.2 SÍNTESE DO PRIMEIRO EXEMPLO

Este primeiro exemplo, em síntese faz as seguintes tarefas.

1. Carrega a biblioteca `sqlite3`;
2. Conecta-se com um BD existente, ou cria o BD caso não exista;
3. Define um cursor para envio de comandos;
4. Cria a tabela `produto`;
5. Insere dois registros de dados na tabela `produto`;
6. Salva (`commit`) tudo no disco e encerra a transação;
7. Encerra o cursor e a conexão.

E na figura 18.2 podemos verificar, usando o DB Browser, a existência da tabela `produto`, com os dois registros presentes nela.

Figura 18.4 – Resultado da execução do exemplo 18.1



fonte: o Autor

Como limitação deste exemplo, destacamos que a tabela criada não contém uma chave primária. A chave primária é um campo (ou campos combinados) usado para identificação única de cada registro na tabela. A chave primária não pode ter valor nulo, nem valores repetidos. Na próxima seção criaremos uma tabela com chave primária.

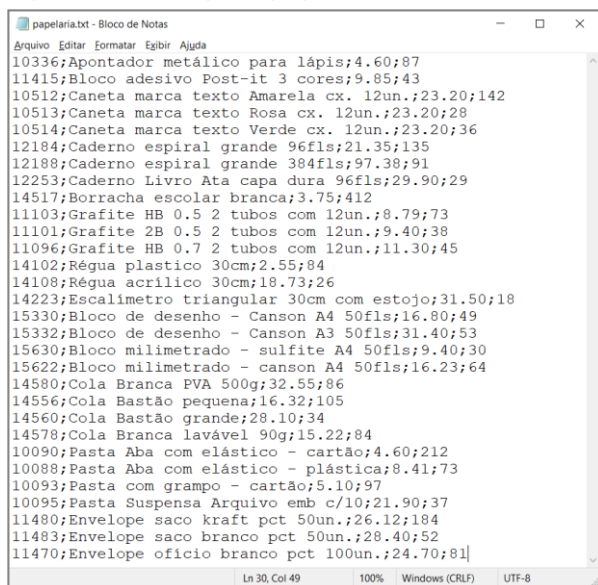
18.7 GERANDO UM BANCO DE DADOS MAIS EXTENSO

Nesta seção vamos criar um banco de dados com a mesma tabela de produtos mostrada no exemplo 18.1, porém desta vez vamos carregá-la a partir de dados lidos de um arquivo do tipo CSV gravado em disco.

O arquivo CSV fornecido e que será usado neste exemplo tem o nome "papelaria.txt", contém 30 linhas com produtos e foi gravado com padrão de codificação "utf-8". A figura 18.3 mostra todos os dados contidos nele. Note que embora o nome do arquivo tenha extensão `.txt`, a forma como está gravado é característica de arquivos CSV, com os dados em cada linha separados pelo caractere `';`.

O exemplo 18.2 tem um aspecto diferente do anterior. Neste exemplo a tabela `produto` será criada contendo uma chave primária, que será o campo `codigo`.

Figura 18.5 – Arquivo papelaria.txt



fonte: o Autor

O programa do exemplo 18.2 faz a leitura desse arquivo e a importação para o banco de dados.

Exemplo 18.2



```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
# exclui a tabela, caso exista
try:
    sql = "drop table produto"
    cursor.execute(sql)
    conector.commit()
except sqlite3.OperationalError:
    pass
# cria a tabela
sql = """
create table produto
(codigo integer not null, descr text, preco numeric, qtdeestq integer,
 primary key (codigo))
"""
# note a cláusula primary key que define a chave primária
cursor.execute(sql)
sql = """
insert into produto (codigo, descr, preco, qtdeestq)
values(?, ?, ?, ?)
"""
nome_arq = 'C:\\CursoPython\\cap18\\papelaria.txt'
for linha_arq in open(nome_arq, encoding='utf-8'):
    dados = linha_arq.rstrip().split(';')
    print(dados)
    cursor.execute(sql, dados)
conector.commit()
cursor.close()
conector.close()
print('\nFim do Programa')

['10336', 'Apontador metálico para lápis', '4.60', '87']
['11415', 'Bloco adesivo Post-it 3 cores', '9.85', '43']
['10512', 'Caneta marca texto Amarela cx. 12un.', '23.20', '142']
['10513', 'Caneta marca texto Rosa cx. 12un.', '23.20', '28']
['10514', 'Caneta marca texto Verde cx. 12un.', '23.20', '36']
['12184', 'Caderno espiral grande 96fls', '21.35', '135']
```

```
[ '12188', 'Caderno espiral grande 384fls', '97.38', '91' ]
[ '12253', 'Caderno Livro Ata capa dura 96fls', '29.90', '29' ]
[ '14517', 'Borracha escolar branca', '3.75', '412' ]
[ '11103', 'Grafite HB 0.5 2 tubos com 12un.', '8.79', '73' ]
[ '11101', 'Grafite 2B 0.5 2 tubos com 12un.', '9.40', '38' ]
[ '11096', 'Grafite HB 0.7 2 tubos com 12un.', '11.30', '45' ]
[ '14102', 'Régua plástico 30cm', '2.55', '84' ]
[ '14108', 'Régua acrílico 30cm', '18.73', '26' ]
[ '14223', 'Escalímetro triangular 30cm com estojo', '31.50', '18' ]
[ '15330', 'Bloco de desenho - Canson A4 50fls', '16.80', '49' ]
[ '15332', 'Bloco de desenho - Canson A3 50fls', '31.40', '53' ]
[ '15630', 'Bloco milimetrado - sulfite A4 50fls', '9.40', '30' ]
[ '15622', 'Bloco milimetrado - canson A4 50fls', '16.23', '64' ]
[ '14580', 'Cola Branca PVA 500g', '32.55', '86' ]
[ '14556', 'Cola Bastão pequena', '16.32', '105' ]
[ '14560', 'Cola Bastão grande', '28.10', '34' ]
[ '14578', 'Cola Branca lavável 90g', '15.22', '84' ]
[ '10090', 'Pasta Aba com elástico - cartão', '4.60', '212' ]
[ '10088', 'Pasta Aba com elástico - plástica', '8.41', '73' ]
[ '10093', 'Pasta com grampo - cartão', '5.10', '97' ]
[ '10095', 'Pasta Suspensa Arquivo emb c/10', '21.90', '37' ]
[ '11480', 'Envelope saco kraft pct 50un.', '26.12', '184' ]
[ '11483', 'Envelope saco branco pct 50un.', '28.40', '52' ]
[ '11470', 'Envelope ofício branco pct 100un.', '24.70', '81' ]
```

Fim do Programa

Quadro 18.2 – Comandos SQL usados no exemplo 18.2

Comando	Descrição
drop table produto	Este SQL é da categoria DDL (<i>Data Definition Language</i>) Ele é usado para eliminar tabelas. Se a tabela não existir, ocorre um erro. Deve-se fornecer o nome da tabela, no caso: "produto"
insert into produto (codigo, descr, preco, qtdeestq) values (?, ?, ?, ?)	Este SQL é da categoria DML (<i>Data Manipulation Language</i>) Neste formato trata-se de um comando parametrizado no qual as interrogações serão substituídas por valores fornecidos no momento da execução do comando.

18.7.1 COMANDOS SQL PARAMETRIZADOS

Neste exemplo um elemento chave é o comando SQL escrito da seguinte forma:

```
sql = """
insert into produto (codigo, descr, preco, qtdeestq)
values(?, ?, ?, ?)
"""
```

Perceba que em sua cláusula `values` foram posicionadas as interrogações `(?, ?, ?, ?)` representando parâmetros que devem ser atribuídos posteriormente. Esse tipo de construção é chamada de parametrização, ou comando SQL parametrizado.

E para executar esse comandos SQL assim precisamos utilizar um segundo parâmetros no método `.execute()`, da seguinte forma:

```
cursor.execute(sql, lista_de_parametros)
```


onde o objeto `lista_de_parametros` deve ser uma sequência – lista ou tupla – contendo a exata quantidade de parâmetros que irão substituir as interrogações posicionadas no string `sql`.

A substituição é feita por posição, ou seja, a primeira interrogação receberá o primeiro elemento da lista, a segunda interrogação receberá o segundo elemento e assim por diante. Por isso, a quantidade de elementos na lista deve coincidir com a quantidade de interrogações e se isso não ocorrer haverá erro.

18.8 EXCLUSÃO DE TABELA

Outro ponto de destaque neste exemplo 18.2 está no início do programa, quando fazemos a exclusão da tabela `produtos`, com este código.

```
try:
    sql = "drop table produto" # SQL para exclusão de tabela
    cursor.execute(sql)
    conector.commit()
except sqlite3.OperationalError:
    pass
```

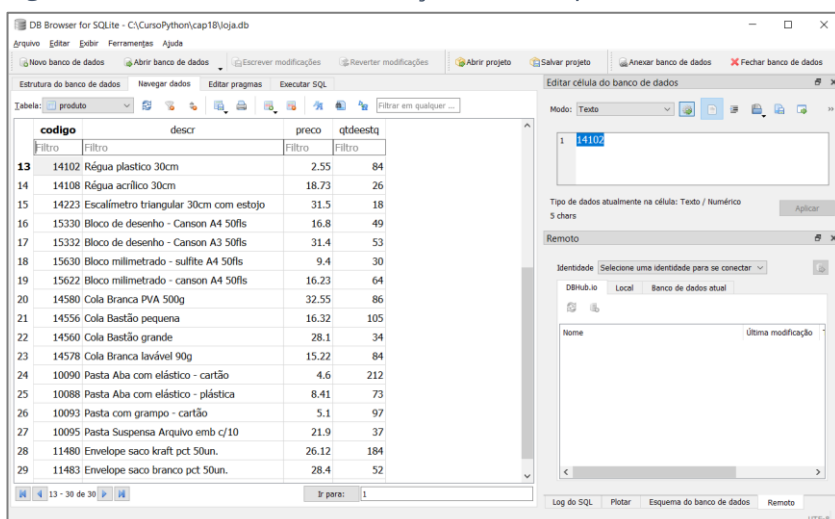
Este código elimina a tabela já existente com o comando DDL `drop table`. Porém, existe o risco de ocorrer um erro caso a tabela a ser eliminada não exista. Para que o programa não seja cancelado fizemos a proteção deste trecho de código com `try-except`. No entanto, não temos nada a fazer caso o erro ocorra, pois estamos em uma situação em que queremos eliminar a tabelas, mas a tabela não existe (quando o BD é novo). Em casos assim, como a cláusula `except` é obrigatória, basta usar o comando `pass` que não fará nada, mas permitirá a consistência sintática no nosso código.

Neste programa fizemos a opção de criar a tabela antes de iniciar a importação dos dados. Porém, ao tentar criar a tabela obteríamos um erro, caso ela já existisse. Para facilitar a execução do programa várias vezes, durante os testes, teríamos que escolher uma de duas estratégias:

- Eliminar a tabela existente e, caso ela não exista porque o banco de dados é novo, fazer o tratamento da exceção na ocorrência de um erro; (esta foi a opção que escolhemos)
- Tentar criar a tabela e tratar a exceção caso a tabela já exista. Nesta opção, e caso a tabela já exista, teríamos que limpá-la, excluindo todos seus registros antes da importação;

Por fim, após a execução deste programa teremos a tabela de produtos com 30 registros.

Figura 18.6 – Resultado da execução do exemplo 18.2



codigo	descr	preco	qtdeestq
13	14102 Régua plástico 30cm	2.55	84
14	14108 Régua acrílico 30cm	18.73	26
15	14223 Escalímetro triangular 30cm com estojo	31.5	18
16	15330 Bloco de desenho - Canson A4 50fls	16.8	49
17	15332 Bloco de desenho - Canson A3 50fls	31.4	53
18	15630 Bloco milimetrado - sulfite A4 50fls	9.4	30
19	15622 Bloco milimetrado - canson A4 50fls	16.23	64
20	14580 Cola Branca PVA 500g	32.55	86
21	14556 Cola Bastão pequena	16.32	105
22	14560 Cola Bastão grande	28.1	34
23	14578 Cola Branca lavável 90g	15.22	84
24	10090 Pasta Aba com elástico - cartão	4.6	212
25	10088 Pasta Aba com elástico - plástica	8.41	73
26	10093 Pasta com grampo - cartão	5.1	97
27	10095 Pasta Suspensa Arquivo emb c/10	21.9	37
28	11480 Envelope saco kraft pct 50un.	26.12	184
29	11483 Envelope saco branco pct 50un.	28.4	52

fonte: o Autor

18.9 COMO ACESSAR OS DADOS INSERIDOS USANDO PYTHON

Nesta seção vamos fazer a mais comum de todas as operações com banco de dados. Vamos consultar a tabela existente e já preenchida e exibir na tela os dados retornados. Veja o código a seguir:

Exemplo 18.3



```
import sqlite3
conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
sql = "select * from produto"
cursor.execute(sql)
dados = cursor.fetchall()
cursor.close()
conector.close()

print('\nConsulta ao Banco de Dados "loja.db" \n')
print('Dados da tabela "produto"')
print("-" * 61)
print("{:7}{:40}{:>8}{:>6}".format('Código', 'Descrição', 'Preço', 'Estq'))
print("- " * 31)
for d in dados:
    print("{:<7}{:40}{:8.2f}{:6}".format(d[0], d[1], d[2], d[3]))
print("-" * 61)
print("Encontrados {} registros".format(len(dados)))
print("\n\nFim do programa")
```

Consulta ao Banco de Dados "loja.db"

Dados da tabela "produto"

Código	Descrição	Preço	Estq
10336	Apontador metálico para lápis	4.60	87
11415	Bloco adesivo Post-it 3 cores	9.85	43
10512	Caneta marca texto Amarela cx. 12un.	23.20	142
10513	Caneta marca texto Rosa cx. 12un.	23.20	28
10514	Caneta marca texto Verde cx. 12un.	23.20	36
12184	Caderno espiral grande 96fls	21.35	135
12188	Caderno espiral grande 384fls	97.38	91
12253	Caderno Livro Ata capa dura 96fls	29.90	29
14517	Borracha escolar branca	3.75	412
11103	Grafite HB 0.5 2 tubos com 12un.	8.79	73
11101	Grafite 2B 0.5 2 tubos com 12un.	9.40	38
11096	Grafite HB 0.7 2 tubos com 12un.	11.30	45
14102	Régua plástico 30cm	2.55	84
14108	Régua acrílico 30cm	18.73	26
14223	Escalímetro triangular 30cm com estojo	31.50	18
15330	Bloco de desenho - canson A4 50fls	16.80	49
15332	Bloco de desenho - canson A3 50fls	31.40	53
15630	Bloco milimetrado - sulfite A4 50fls	9.40	30
15622	Bloco milimetrado - canson A4 50fls	16.23	64
14580	Cola Branca PVA 500g	32.55	86
14556	Cola Bastão pequena	16.32	105
14560	Cola Bastão grande	28.10	34
14578	Cola Branca lavável 90g	15.22	84
10090	Pasta Aba com elástico - cartão	4.60	212
10088	Pasta Aba com elástico - plástica	8.41	73
10093	Pasta com grampo - cartão	5.10	97
10095	Pasta Suspensa Arquivo emb c/10	21.90	37
11480	Envelope saco kraft pct 50un.	26.12	184
11483	Envelope saco branco pct 50un.	28.40	52
11470	Envelope ofício branco pct 100un.	24.70	81

Encontrados 30 registros

Fim do programa

Este exemplo consiste em fazer um acesso à tabela produto, recuperar todos os seus registros e depois fazer a exibição em tela. Observe que claramente o programa é dividido em duas partes distintas. Na primeira parte é feita a conexão com o banco de dados e executada a consulta. Na segunda parte fazemos a exibição dos dados em tela.

Quadro 18.3 – Comandos SQL usados no exemplo 18.3

Comando	Descrição
<code>select * from produto</code> ou <code>select código, descr from produto</code>	Este SQL é um DQL (<i>Data Query Language</i>) Ele é usado para recuperar registros já armazenados em alguma tabela. O caractere * significa que todos os campos da tabela serão retornados. É possível substituí-lo por uma lista de campos.

O comando SQL `select` é executado normalmente com o método `.execute()`. Porém, ele tem um efeito diferente dos outros comandos SQL.

Enquanto os comandos SQL de outras categorias realizam uma ação sem produzir retorno de dados, o `select` produz um retorno de dados. Por isso, após o `.execute()`, precisamos usar um dos métodos de recuperação de dados (`fetch`). Existem três opções:

- **`.fetchall()`** – retorna de uma única vez uma lista com todos os registros produzidos pelo `select`, sendo que cada registro é uma sublista dentro da lista. Deve-se tomar cuidado com esta opção quando a quantidade de registros retornados é muito grande;
- **`.fetchmany(size)`** – retorna uma lista com uma certa quantidade de registros produzidos pelo `select`, sendo que cada registro é uma sublista dentro da lista. A quantidade retornada é especificada pelo argumento `size`. Com esta opção é possível recuperar sucessivamente um bloco de `size` registros por vez;
- **`.fetchone()`** – retorna um único registro produzido pelo `select`, os dados estarão uma lista. Com esta opção é possível recuperar sucessivamente um registro por vez;

18.10 INSERÇÃO DE NOVOS REGISTROS A PARTIR DA LEITURA DE DADOS VIA TECLADO

No próximo exemplo, o programa permanecerá em laço lendo os dados a serem inseridos na tabela produto. Todos os dados devem ser digitados em uma única linha separados por vírgulas. Os dados são: código, descrição do produto, preço e quantidade em estoque. O programa deve permanecer em laço lendo os dados e inserindo-os no banco. Quando uma linha vazia for digitada o laço termina.

Este programa não cria a tabela. Ele supõe que a tabela já exista.

Quanto ao comando SQL necessário à inserção dos dados não há qualquer novidade, pois é o mesmo `insert into` usado no exemplo 18.2.

Exemplo 18.4

```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
sql = """
    insert into produto (codigo, descr, preco, estq)
```

```

values (?, ?, ?, ?)
"""
print('Digite os dados separados por vírgulas')
print('Codigo, Descrição, Preço, Estoque')
Ler = input()
while Ler != '':
    D = Ler.split(',')
    try:
        cursor.execute(sql, D)
        conector.commit()
    except:
        print('{} Dados inválidos'.format(D))
    else:
        print(' '*30, '...dados inseridos com sucesso')
    finally:
        print('\nCodigo, Descrição, Preço, Estoque')
    Ler = input()

```

```

Digite os dados separados por vírgulas
Codigo, Descrição, Preço, Estoque
19010, Produto 19010, 19.01, 100
['19010', 'Produto 19010', '19.01', '100']
...dados inseridos com sucesso

```

```

Codigo, Descrição, Preço, Estoque
19020, Produto 19020, 19.02, 50
['19020', 'Produto 19020', '19.02', '50']
...dados inseridos com sucesso

```

```

Codigo, Descrição, Preço, Estoque

```

O resultado da execução do código acima pode ser visto na imagem 18.7 a seguir, na qual evidenciamos que novos registros estão presentes na tabela `produto`, contendo os dados digitados para os dois produtos, 19010 e 19020.

Figura 18.7 – Resultado da inserção de dados na tabela "produto"

The screenshot shows the DB Browser for SQLite interface. The 'produto' table is selected, and its data is displayed in a table view. The table has four columns: 'codigo', 'descr', 'preco', and 'qtdeestq'. Two new records, '19010 Produto 19010' and '19020 Produto 19020', are highlighted with a red box. The interface also shows a 'Remoto' (Remote) connection panel on the right.

codigo	descr	preco	qtdeestq
16	12184 Caderno espiral grande 96fls	21.35	135
17	12188 Caderno espiral grande 384fls	97.38	91
18	12253 Caderno Livro Ata capa dura 96fls	29.9	29
19	14102 Régua plástico 30cm	2.55	84
20	14108 Régua acrílico 30cm	18.73	26
21	14223 Escalímetro triangular 30cm com ...	31.5	18
22	14517 Borracha escolar branca	3.75	412
23	14556 Cola Bastão pequena	16.32	105
24	14560 Cola Bastão grande	28.1	34
25	14578 Cola Branca lavável 90g	15.22	84
26	14580 Cola Branca PVA 500g	32.55	86
27	15330 Bloco de desenho - canson A4 50fls	16.8	49
28	15332 Bloco de desenho - canson A3 50fls	31.4	53
29	15622 Bloco milimetrado - canson A4 50fls	16.23	64
30	15630 Bloco milimetrado - sulfite A4 50fls	9.4	30
31	19010 Produto 19010	19.01	100
32	19020 Produto 19020	19.02	50

fonte: o Autor

18.11 FILTROS E ORDENAÇÃO DE REGISTROS

Nesta seção vamos retomar o exemplo 18.3 estendendo o uso do comando SQL `select` para incluir as cláusulas: de filtro `where` e de ordenação `order by`, conforme destacado no quadro a seguir.

Quadro 18.4 – Forma mais geral do comando `select` com `where` e `order by`

Comando	Descrição
<pre>select * from produto where preco < 10 order by descr</pre>	<p>Este SQL é um DQL (<i>Data Query Language</i>)</p> <p>Ele é usado para recuperar registros já armazenados em alguma tabela.</p> <p>A cláusula <code>where</code> é usada para filtrar registros segundo o critério especificado através de uma ou mais condições que acompanham a palavra-chave <code>where</code>.</p> <p>A cláusula <code>order by</code> é usada para ordenar o conjunto de registros retornados. A ordem pode ser crescente ou decrescente.</p>

No vídeo do exemplo 18.3 acrescentamos essas duas cláusulas e mostramos possibilidades variadas de filtragem e ordenação. Como no exemplo abaixo onde filtramos quantidades menores que 30 e ordenamos pela descrição do produto.

Exemplo 18.3 - continuação

```
sql = "select * from produto where qtdeestq < 30 order by descr"
```

Consulta ao Banco de Dados "loja.db"

Dados da tabela "produto"

Código	Descrição	Preço	Estq
12253	Caderno Livro Ata capa dura 96fls	29.90	29
10513	Caneta marca texto Rosa cx. 12un.	23.20	28
14223	Escalímetro triangular 30cm com estojo	31.50	18
14108	Régua acrílico 30cm	18.73	26

Encontrados 4 registros

Fim do programa

18.12 ALTERAÇÃO DE TABELAS E ATUALIZAÇÃO DE DADOS

Há situações em que é necessário alterar a estrutura de tabelas existentes, com inclusão e/ou exclusão de campos, acréscimo de índices e, eventualmente, alteração de chave primária. Também é possível renomear os campos e a própria tabela. Em um projeto de um sistema novo e bem projetado é raro que isso aconteça. Porém, ao longo da vida útil de um sistema, mudanças assim podem acontecer provocadas por alterações nas regras de negócio, implantação de novas funcionalidades, adoção de novas tecnologias, entre outras motivações.

Nos próximos dois exemplos realizaremos esse tipo de operação.

No exemplo 18.5 são incluídos três novos campos na tabela "produto", a saber:

- **custo** (numeric) – para conter o preço de custo do produto;
- **aliqicms** (numeric) – para conter a alíquota do imposto ICMS aplicado à venda do produto;
- **qtdemin** (integer) – para conter a quantidade mínima de estoque para o produto;

É de se supor que tabelas já existentes contenham registros. Em casos assim, quando um novo campo é inserido, os registros já armazenados passarão a conter esse campo mas não haverá nenhum dado para ele, de modo que seu estado será *NULL* (nulo).

Após a inclusão do novo campo é possível substituir esses valores nulos por algum valor adequado. No exemplo 18.5, como os campos são numéricos, vamos atribuir o valor zero para os campos inseridos.

O quadro 18.5 mostra os comandos SQL que usaremos nesse exemplo. É importante destacar que o comando SQL `alter table` só atua em um campo por vez. Como queremos incluir três campos, teremos que usá-lo três vezes. O comando `update` deve ser usado para alterar o conteúdo dos campos e ele pode atuar sobre vários campos simultaneamente.

Quadro 18.5 – Inclusão de campos em tabela existente

Comando	Descrição
<pre>alter table produto add custo numeric</pre>	<p>Este SQL é um DDL (Data Definition Language)</p> <p>Ele é usado para acrescentar campos a uma tabela já existente. Os novos campos inseridos não terão valor e estarão com estado nulo (<i>NULL</i>).</p> <p>Deve-se usar um comando deste para cada campo a ser incluído e se o campo já existir ocorrerá o erro "duplicate column".</p>
<pre>update produto set custo = 0, aliqicms = 0, qtdemin = 0</pre>	<p>Este SQL é um DML (Data Manipulation Language)</p> <p>Ele permite atualizar os registros da tabela colocando em cada campo o valor que lhe é atribuído por meio da cláusula <code>set</code>.</p> <p>É possível usar a cláusula <code>where</code> para selecionar quais registros serão alterados.</p>

Assim, o código do exemplo 18.5 tem duas partes: na primeira os campos são incluídos, um por vez.; na segunda parte seus valores são atualizados para 0.

Exemplo 18.5



```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
# insere os novos campos
sql = "alter table produto add custo numeric"
cursor.execute(sql)
sql = "alter table produto add aliqicms numeric"
cursor.execute(sql)
sql = "alter table produto add qtdemin integer"
cursor.execute(sql)
# atribui 0 aos novos campos
sql = "update produto set custo = 0, aliqicms = 0, qtdemin = 0"
cursor.execute(sql)
conector.commit()
print('Os três campos foram inseridos e inicializados com zero')
cursor.close()
conector.close()
```

Os três campos foram inseridos e inicializados com zero

Como resultado da execução do programa acima temos a tabela produto alterada.

Figura 18.8 – Tabela produto após a execução do exemplo 18.5

codigo	descr	preco	qtdeestq	custo	aliqicms	qtdeemin
10088	Pasta Aba com elástico - plástica	8.41	73	0	0	0
10090	Pasta Aba com elástico - cartão	4.6	212	0	0	0
10093	Pasta com grampo - cartão	5.1	97	0	0	0
10095	Pasta Suspensa Arquivo emb c/10	21.9	37	0	0	0
10336	Apontador metálico para lápis	4.6	87	0	0	0
10512	Caneta marca texto Amarela cx. 12un.	23.2	142	0	0	0
10513	Caneta marca texto Rosa cx. 12un.	23.2	28	0	0	0
10514	Caneta marca texto Verde cx. 12un.	23.2	36	0	0	0
11096	Grafite HB 0.7 2 tubos com 12un.	11.3	45	0	0	0
11101	Grafite 2B 0.5 2 tubos com 12un.	9.4	38	0	0	0
11103	Grafite HB 0.5 2 tubos com 12un.	8.79	73	0	0	0
11415	Bloco adesivo Post-it 3 cores	9.85	43	0	0	0
11470	Envelope ofício branco pct 100un.	24.7	81	0	0	0
11480	Envelope saco kraft pct 50un.	26.12	184	0	0	0
11483	Envelope saco branco pct 50un.	28.4	52	0	0	0
12184	Caderno espiral grande 96fls	21.35	135	0	0	0
12188	Caderno espiral grande 384fls	97.38	91	0	0	0

fonte: o Autor

Nesse programa não há nenhum aspecto novo no que diz respeito ao Python. São os mesmos comandos utilizados nos exemplos anteriores para fazer as tarefas típicas de banco de dados, a saber: conexão com o banco; criação do cursor; execução dos comandos DDL e DML, commit e encerramento da conexão. Esse programa não interage com o usuário, e ao seu término uma mensagem é mostrada na tela indicando que o BD foi atualizado.

18.13 ATUALIZAÇÃO DE DADOS A PARTIR DE UM ARQUIVO

Em uma situação real não será muito útil incluir campos zerados no cadastro de produtos da loja. É de se esperar que os campos preço de custo, alíquota de ICMS e quantidade mínima devam ter valores.

No exemplo 18.6 faremos a atualização desses campos atribuindo valores provenientes de um arquivo CSV que será lido e importado para tabela produto. Cada linha deste arquivo contém o código do produto (chave primária e que será usada na identificação dos registros) seguido do preço de custo, alíquota e quantidade mínima de estoque. A figura 18.9 mostra o aspecto deste arquivo.

Figura 18.9 – Arquivo papelaria_atualiz.txt

```

10336;3.60;7;50
11415;7.90;18;20
10512;18.70;12;30
10513;18.70;12;30
10514;18.70;12;30
12184;17.00;7;100
12188;79.00;12;100
12253;23.90;18;50
14517;3.00;7;50
11103;7.00;18;30
11101;7.50;18;30
11096;9.20;18;30
14102;1.95;7;50
14108;15.10;18;20
14223;25.20;18;10
15330;13.90;7;30
15332;25.95;12;30
15630;7.60;12;30
15622;13.22;12;30
14580;26.50;12;20
14556;13.50;7;50
14560;22.50;7;30
14578;12.45;12;100
10090;3.70;18;100
10088;6.35;18;100
10093;4.00;18;100
10095;17.65;18;50
11480;21.75;18;100
11483;22.75;18;100
11470;20.12;18;100

```

fonte: o Autor

O comando SQL necessário para realizar a atualização é mostrado no quadro a seguir.

Quadro 18.6 – SQL para atualização da tabela produto

Comando	Descrição
<pre>update produto set custo = ?, aliqicms = ?, qtdemin = ? where codigo = ?</pre>	<p>Este SQL é um DML (Data Manipulation Language)</p> <p>Ele permite atualizar um registro da tabela colocando em cada campo o valor que lhe é atribuído por meio da cláusula <code>set</code>.</p> <p>Na cláusula <code>where</code> foi usado o campo <code>codigo</code> que é a chave primária. Portanto, apenas um registro será atualizado quando este comando for executado.</p>

Note que nesse este comando `update` é parametrizado com quatro interrogações. Isso implica que no momento da sua execução precisaremos passar uma lista com quatro valores que serão usados como parâmetro e a ordem importa, ou seja, a ordem em que cada interrogação aparece no `update` deve ser usada na lista. E essa ordem não corresponde à ordem que obteremos ao ler uma linha do arquivo. A ordem a ser usada é esta:

```
# indicação da ordem dos argumentos para o update
dados = [custo, aliqicms, qtdemin, codigo]
```

Com isso após obter a linha do arquivo e fazer sua separação com o método `.split()` usamos as duas linhas abaixo para posicionar o `codigo` no final da linha, removendo-o da primeira posição:

```
dados = linha_arq.rstrip().split(';')
dados.append(dados[0])          # coloca o código no final da lista dados
del(dados[0])                  # elimina o código do início
```

O programa completo é mostrado no código 18.6 a seguir.

Exemplo 18.6



```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
sql = """
    update produto
    set custo = ?, aliqicms = ?, qtdemin = ?
    where codigo = ?
"""

nome_arq = 'C:\\CursoPython\\cap18\\papelaria_atualiz.txt'
for linha_arq in open(nome_arq, encoding='utf-8'):
    dados = linha_arq.rstrip().split(';')
    dados.append(dados[0]) # coloca o código no final da lista
    del(dados[0]) # elimina o código do início da lista
    print(dados)
    cursor.execute(sql, dados)
conector.commit()
print('A tabela produto foi atualizada')
cursor.close()
conector.close()
```

A tabela produto foi atualizada

O resultado obtido pode ser visto com o DB Browser

Figura 18.10 – Tabela produto após atualização

	codigo	descr	preco	qtdeestq	custo	aliqicms	qtdeemin
1	10088	Pasta Aba com elástico - plástica	8.41	73	6.35	18	100
2	10090	Pasta Aba com elástico - cartão	4.6	212	3.7	18	100
3	10093	Pasta com grampo - cartão	5.1	97	4	18	100
4	10095	Pasta Suspensa Arquivo emb c/10	21.9	37	17.65	18	50
5	10336	Apontador metálico para lápis	4.6	87	3.6	7	50
6	10512	Caneta marca texto Amarela cx. 12un.	23.2	142	18.7	12	30
7	10513	Caneta marca texto Rosa cx. 12un.	23.2	28	18.7	12	30
8	10514	Caneta marca texto Verde cx. 12un.	23.2	36	18.7	12	30
9	11096	Grafite HB 0.7 2 tubos com 12un.	11.3	45	9.2	18	30
10	11101	Grafite 2B 0.5 2 tubos com 12un.	9.4	38	7.5	18	30
11	11103	Grafite HB 0.5 2 tubos com 12un.	8.79	73	7	18	30
12	11415	Bloco adesivo Post-it 3 cores	9.85	43	7.9	18	20
13	11470	Envelope ofício branco pct 100un.	24.7	81	20.12	18	100
14	11480	Envelope saco kraft pct 50un.	26.12	184	21.75	18	100
15	11483	Envelope saco branco pct 50un.	28.4	52	22.75	18	100
16	12184	Caderno espiral grande 96fls	21.35	135	17	7	100
17	12188	Caderno espiral grande 384fls	97.38	91	79	12	100

fonte: o Autor

18.14 EXCLUSÃO DE REGISTROS DE UMA TABELA

A exclusão de registros é outra operação frequente executada em tabelas de banco de dados. Para isso usamos o comando SQL `delete`. O quadro 18.7 mostra o formato desse comando e o exemplo 18.7 contém um programa que exemplifica seu uso.

Quadro 18.7 – SQL para atualização da tabela produto

Comando	Descrição
<pre>delete from produto where codigo = ?</pre>	<p>Este SQL é um DML (Data Manipulation Language). Ele permite excluir registros da tabela especificada. A cláusula <code>where</code> é opcional, mas tome cuidado, pois se ela não for especificada o comando excluirá todos os registros. Na cláusula <code>where</code> deste exemplo foi usado o campo <code>codigo</code> que é a chave primária. Portanto, apenas um registro será excluído quando o comando for executado. Se não existe um registro com o código digitado não ocorre erro e o comando não terá qualquer efeito.</p>

O exemplo 18.7 permanece em laço lendo o código do produto e excluindo o registro correspondente.

Exemplo 18.7

```
import sqlite3

conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
sql = "delete from produto where codigo = ?"
excluidos = []
codigo = input('Digite o código a ser excluído: ')
while codigo.upper() != 'FIM':
    cursor.execute(sql, [int(codigo)])
    excluidos.append(int(codigo))
    codigo = input('Digite o código a ser excluído: ')
conector.commit()
```

```
print('Foram excluídos os códigos:')
print(excluidos)
cursor.close()
conector.close()

Digite o código a ser excluído: 10088
Digite o código a ser excluído: 10090
Digite o código a ser excluído: 10093
Digite o código a ser excluído: 10095
Digite o código a ser excluído: 10099
Digite o código a ser excluído: FIM

Foram excluídos os códigos:
[10088, 10090, 10093, 10095, 10099]
```

Note que dos códigos acima, o 10099 não existe na tabela e nenhum erro ocorre na sua tentativa de exclusão. Além disso, esse código acaba fazendo parte da lista de códigos que foram excluídos da tabela, o que não é necessariamente verdadeiro, afinal ele não existe no cadastro. Dá para melhorar um pouco isso.

Exercício Resolvido 18.1

Teste este código no PyCharm

Enunciado: Altere o programa do exemplo 18.3 para exibir na tela os três campos inseridos na tabela produto: preço de custo, alíquota ICMS e quantidade mínima de estoque.

```
import sqlite3
conector = sqlite3.connect('C:\\CursoPython\\cap18\\loja.db')
cursor = conector.cursor()
sql = "select * from produto"
cursor.execute(sql)
dados = cursor.fetchall()
cursor.close()
conector.close()

print('\nConsulta ao Banco de Dados "loja.db" \n')
print('Dados da tabela "produto"')
print("-" * 79)
print("{:7}{:40}{:>7}{:>6}{:>7}{:>6}{:>6}".format(
    'Código', 'Descrição', 'Preço', 'Estq', 'Custo', 'Aliq.', 'qMin'))
print("-" * 40)
for d in dados:
    print("{:<7}{:40}{:7.2f}{:6}{:7.2f}{:6}{:6}".format(
        d[0], d[1], d[2], d[3], d[4], d[5], d[6]))
print("-" * 79)
print("Encontrados {} registros".format(len(dados)))
print("\nFim do programa")
```

Consulta ao Banco de Dados "loja.db"

Dados da tabela "produto"

Código	Descrição	Preço	Estq	Custo	Aliq.	qMin
10088	Pasta Aba com elástico - plástica	8.41	73	6.35	18	100
10090	Pasta Aba com elástico - cartão	4.60	212	3.70	18	100
10093	Pasta com grampo - cartão	5.10	97	4.00	18	100
10095	Pasta Suspensa Arquivo emb c/10	21.90	37	17.65	18	50
10336	Apontador metálico para lápis	4.60	87	3.60	7	50
10512	Caneta marca texto Amarela cx. 12un.	23.20	142	18.70	12	30
10513	Caneta marca texto Rosa cx. 12un.	23.20	28	18.70	12	30
10514	Caneta marca texto Verde cx. 12un.	23.20	36	18.70	12	30
11096	Grafite HB 0.7 2 tubos com 12un.	11.30	45	9.20	18	30
11101	Grafite 2B 0.5 2 tubos com 12un.	9.40	38	7.50	18	30
11103	Grafite HB 0.5 2 tubos com 12un.	8.79	73	7.00	18	30

```

11415 Bloco adesivo Post-it 3 cores          9.85    43    7.90    18    20
11470 Envelope ofício branco pct 100un.     24.70    81   20.12    18   100
11480 Envelope saco kraft pct 50un.         26.12   184   21.75    18   100
11483 Envelope saco branco pct 50un.        28.40    52   22.75    18   100
12184 Caderno espiral grande 96fls          21.35   135   17.00     7   100
12188 Caderno espiral grande 384fls         97.38    91   79.00    12   100
12253 Caderno Livro Ata capa dura 96fls     29.90    29   23.90    18    50
14102 Régua plástico 30cm                   2.55     84    1.95     7    50
14108 Régua acrílico 30cm                   18.73    26   15.10    18    20
14223 Escalímetro triangular 30cm com estojo 31.50    18   25.20    18    10
14517 Borracha escolar branca                3.75   412    3.00     7    50
14556 Cola Bastão pequena                   16.32   105   13.50     7    50
14560 Cola Bastão grande                     28.10    34   22.50     7    30
14578 Cola Branca lavável 90g                15.22    84   12.45    12   100
14580 Cola Branca PVA 500g                   32.55    86   26.50    12    20
15330 Bloco de desenho - canson A4 50fls    16.80    49   13.90     7    30
15332 Bloco de desenho - canson A3 50fls    31.40    53   25.95    12    30
15622 Bloco milimetrado - canson A4 50fls   16.23    64   13.22    12    30
15630 Bloco milimetrado - sulfite A4 50fls   9.40     30    7.60    12    30
-----

```

Encontrados 30 registros

Fim do programa

Exercício Proposto 18.1

Enunciado: *Altere o programa do exemplo 18.2 implementando a opção b descrita na seção 18.8*

Exercício Proposto 18.2

Enunciado: *Elabore uma forma de melhorar o exemplo 18.7 de modo que para cada código fornecido se ele existir na tabela, seja excluído mediante uma confirmação do usuário e se não estiver presente na tabela essa situação seja informada.*

Para cada produto excluído seus dados deve ser exibidos na tela ao final do programa.

Dica 1: *Crie uma função que faça a verificação de existência do registro usando um select.*

Dica 2: *Para cada código a ser excluído guarde seus dados em um dicionário e exiba-o no final.*

Exercício Proposto 18.3

Enunciado: *Escreva um programa que crie um banco de dados chamado agenda.db. Nesse banco de dados deve existir a tabela contatos com os seguintes campos:*

Campo	Tipo	Descrição
Id	integer (usar autonumeração)	Chave primária Número inteiro gerado automaticamente na inclusão
Nome	text	Nome da pessoa
Cel	text	Número do celular
Tel	text	Número do tel. fixo
Email	text	Endereço de e-mail
Aniver	text	Data de aniversário da pessoa (armazenar como texto)

Além de exibir a agenda o programa deve permitir efetuar as seguintes ações: cadastrar novas pessoas; alterar os telefones e o email; excluir pessoas da agenda.

Exercício Proposto 18.4

Enunciado: Escreva um programa que crie um banco de dados chamado `musicas.db`. Esse banco de dados deve conter duas tabelas, mostradas a seguir.

Tabela `musicas`

Campo	Tipo	Descrição
id	integer (autonumeração)	Chave primária Número inteiro gerado automaticamente na inclusão
nome	text	Nome da música
artista	text	Artista ou banda
álbum	text	Nome do álbum
ano	text	Ano de lançamento
arquivo	text	Nome do arquivo, incluindo pasta

Tabela `nomeplaylist`

Campo	Tipo	Descrição
nomepl	text	Chave primária Nome da playlist
data	text	Data de criação da playlist

Tabela `playlists`

Campo	Tipo	Descrição	
nomepl	text	Nome da playlist	Estes dois campos compõem a chave primária
id	integer	Id da música	

O programa deve ter um menu com opções para:

- *criar a estrutura do banco de dados*
 - *incluir músicas*
 - *criar uma playlist*
 - *incluir e remover músicas de uma playlist*
 - *excluir músicas (se a música a ser excluída estiver em alguma playlist, deve ser removida)*
 - *excluir uma playlist (apenas a playlist deve ser excluída, as músicas incluídas na playlist devem ser mantidas)*
 - *exportar os dados das músicas e das playlists para arquivos texto no formato CSV*
-

