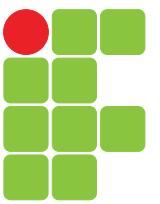
Introdução ao Python

notas de aula

Instituto Federal de Ciência e Tecnologia de Santa Catarina
Departamento Acadêmico de Eletrotécnica
Mestrado Profissional em Sistemas de Energia
Programação Aplicada a Sistemas de Energia — NVP68300

Prof. Jackson Lago, Dr.
jackson.lago@ifsc.edu.br



Sumário

Orientações Iniciais e Convenções

Este tutorial oferece uma introdução concisa ao Python, voltada para os alunos ingressantes no curso de Mestrado Profissional em Sistemas de Energia.

Por ter um caráter introdutório, o texto apresenta exemplos e conceitos sem a intenção de esgotar completamente o tema ou fornecer soluções otimizadas para problemas específicos.

Os exemplo aqui descritos assumem a utilização de Python 3.10 (ou superior) que deve estar devidamente instalado (https://www.python.org/downloads/).

Também recomenda-se a utilização de alguma IDE (*Integrated Development Environment*) para facilitar a escrita e manipulação do código. As duas mais populares para desenvolvimento em Python são:

- `PyCharm`: uma IDE dedicada ao Python, desenvolvida pela JetBrains. Oferece uma configuração padrão robusta e inclui funcionalidades como autocomplete, análise de código, refatoração, debugging, testes integrados e integração com Git. É uma ferramenta profissional ideal para projetos complexos, embora possa ser pesada em máquinas mais simples ou antigas. Disponível em duas versões: Professional (paga) e Community Edition (gratuita). Embora a versão gratuita tenha algumas limitações, ela é mais do que suficiente para este tutorial e até mesmo para projetos pessoais de grande porte. (https://www.jetbrains.com/pycharm/download/)
- `VS Code`: um editor de código aberto, leve e flexível, não restrito a uma única linguagem. O suporte ao Python é obtido por meio de extensões, que também habilitam ferramentas de análise de código. Seu uso exige uma configuração inicial, incluindo a instalação da extensão Python (Microsoft). Outras extensões relacionadas à produtividade podem ser obtidas pelo marketplace integrado ao VS Code. (https://code.visualstudio.com/download/)

No texto, códigos Python serão identificados por uma caixa preta:

exemplo_codigo_python.py

```python
print('Hello, World!')
```

enquanto comandos e mensagens impressas no terminal estarão em uma caixa azul:

```
C:\curso_python\> python exemplo_codigo_python.py
Hello, World!
```

Capítulo 1

Variáveis e Tipos

A computação em seu nível mais básico baseia-se na manipulação de dados armazenados na memória.

As linguagens naturais são ambíguas e, muitas vezes, subjetivas, o que pode gerar interpretações variadas. Diferentemente delas, uma linguagem de programação deve ser precisa e determinística, garantindo previsibilidade no processamento da informação.

Através dessas linguagens, comunicamos ao hardware como manipular os dados armazenados na memória, transformando-os em novos valores. Esses valores, por sua vez, são interpretados e associados a algo significativo no mundo real, seja um número, um texto ou uma representação visual.

Blocos de dados armazenados na memória que representam coletivamente um conceito comum são chamados objetos. Os tipos associados a esses objetos formam uma camada de abstração que define como um conjunto de bits deve ser interpretado. Essa abstração permite representar diversos conceitos, como números inteiros, valores fracionários, caracteres e outros tipos de dados, tornando a manipulação dessas informações mais estruturada e intuitiva.

Diferente de algumas linguagens, em Python não é necessário declarar o tipo da variável explicitamente — o interpretador determina automaticamente o tipo com base no valor atribuído.

Python possui diversos tipos primitivos, que representam os dados mais básicos da linguagem. Estes podem ser utilizados diretamente para expressar algum conceito real ou combinados para formar tipos compostos (ou estruturados), capazes de representar ideias e relações mais complexas.

1.1: Tipos primitivos

Os principais tipos primitivos em Python são:

- `int`: utilizado para armazenar números inteiros, como `10`, `-5`, e `0`
- `float`: representa valores com casas decimais, como `3.14`, `-0.5`, `1.6e-3`
- `complex`: representa números complexos, como `3.2 + 4.7j`

- `str`: string utilizado para armazenar textos, como `'Hello, world!'` ou `"python"`[1].
- `bool`: representa uma grandeza booleana, que pode assumir apenas `True` ou `False`, essencial para estruturas de decisão
- `NoneType`: pode assumir apenas o valor `None`, e pode ser explicitamente atribuído a uma variável para sinalizar a ausência de um valor.

Em código a criação de variáveis com os tipos acima descritos ficaria:

```
idade = 25
peso = 78.6
impedancia = 5 + 3j
ativo = True
nome = 'Fulano de Tal'
sem_valor = None

print(f"'idade' é uma variável do tipo {type(idade)} e valor: {idade}")
print(f"'peso' é uma variável do tipo {type(peso)} e valor: {peso}")
print(f"'impedancia' é uma variável do tipo {type(impedancia)} e valor: {impedancia}")
print(f"'ativo' é uma variável do tipo {type(ativo)} e valor: {ativo}")
print(f"'nome' é uma variável do tipo {type(nome)} e valor: {nome}")
print(f"'sem_valor' é uma variável do tipo {type(sem_valor)} e valor: {sem_valor}")
```

O código acima apenas cria diversas variáveis de diferentes tipos e para cada uma delas imprime o seu tipo e valor no terminal:

```
'idade' é uma variável do tipo <class 'int'> e valor: 25
'peso' é uma variável do tipo <class 'float'> e valor: 78.6
'impedancia' é uma variável do tipo <class 'complex'> e valor: (5+3j)
'ativo' é uma variável do tipo <class 'bool'> e valor: True
'nome' é uma variável do tipo <class 'str'> e valor: Fulano de Tal
'sem_valor' é uma variável do tipo <class 'NoneType'> e valor: None
```

Antes de prosseguirmos e começarmos a utilizar essas variáveis para realizar tarefas úteis, é importante refletir sobre o conceito de tipagem e as diferentes formas como as linguagens de programação a tratam.

Algumas linguagens possuem tipagem dinâmica, onde o tipo das variáveis é determinado durante a execução do programa. Outras adotam tipagem estática, exigindo que os tipos sejam declarados antecipadamente e validados durante a compilação (quando aplicável). Essa diferença impacta diretamente a maneira como escrevemos, mantemos e depuramos código.

Python é uma linguagem interpretada e de tipagem dinâmica, o que significa que seu código não é compilado previamente e que o tipo de uma variável é determinado em tempo de execução, sem exigir uma declaração explícita por parte do programador. Isso não significa que as variáveis não possuam um tipo associado, mas sim que o próprio interpretador do Python gerencia automaticamente os tipos conforme necessário.

Por exemplo, no seguinte código:

```
descricao = 'Geladeira frost free 500l'  # tipo: str
quantidade = 10                          # tipo: int
peso_unitario = 72.7                     # tipo: float
```

as variáveis `descricao`, `quantidade` e `peso_unitario`[2] são automaticamente criadas com os tipos `str`, `int` e `float`, respectivamente. O interpretador infere esses tipos com base nos valores atribuídos a cada variável (o valor à direita de `=`).

[1] Em Python, strings literais podem ser delimitadas por aspas simples (') ou duplas ("). Essa flexibilidade é útil quando queremos incluir o outro tipo de aspas dentro do texto, sem a necessidade de caracteres de escape.

[2] Em Python, o estilo `snake_case` é tradicionalmente usado para nomear variáveis, separando palavras com underscores (_) e usando minúsculas.

Além disso, uma variável pode ser redefinida posteriormente, até mesmo para um tipo diferente. Essa abordagem pode ser vantajosa para tornar o código mais flexível e dinâmico, mas também pode introduzir problemas ao mascarar erros difíceis de identificar.

Por exemplo, em

```
descricao = 'Geladeira frost free 500l'  # tipo: str
quantidade = 10                          # tipo: int
peso_unitario = 72.7                     # tipo: float

print(f"pré: 'descricao' é do tipo {type(descricao)} com valor: {descricao}")
descricao = quantidade   # redefine descricao para int
print(f"pós: 'descricao' é do tipo {type(descricao)} com valor: {descricao}")
```

```
pré: 'descricao' é do tipo <class 'str'> com valor: Geladeira frost free 500l
pós: 'descricao' é do tipo <class 'int'> com valor: 10
```

a variável `descricao` inicialmente criada como `str` e valor `'Geladeira frost free 500l'`, é redefinida para `int`, passando a armazenar o valor `10`. Esse comportamento exemplifica a tipagem dinâmica do Python, onde uma mesma variável pode assumir diferentes tipos ao longo da execução do código.

Em Python, uma variável é apenas um nome que referencia um objeto em memória. Quando uma variável é redefinida, ela passa a apontar para outro objeto, que não necessariamente precisa ter o mesmo tipo.

Esse código é válido e não resultará em erros de execução por si só. No entanto, pode representar um erro lógico caso essa alteração de tipo não esteja alinhada com a intenção do programador. Cabe ao programador garantir a coerência do código, tratando `descricao` como `str` na parte inicial do programa e como `int` posteriormente, conforme necessário.

Embora essa redefinição com mudança de tipo não configure um erro sintático e possa ser útil em alguns casos, sua prática indiscriminada é desencorajada, pois pode dificultar legibilidade e a manutenção do código.

A tipagem dinâmica do Python torna a linguagem mais simples e flexível, facilitando o desenvolvimento, especialmente para iniciantes. No entanto, essa característica exige atenção, pois erros de tipagem podem surgir em tempo de execução se os tipos das variáveis não forem corretamente tratados.

Por exemplo, considere o seguinte código:

```
descricao = 'Geladeira frost free 500l'  # tipo: str
quantidade = 10                          # tipo: int
peso_unitario = 72.7                     # tipo: float

peso_total = peso_unitario * quantidade
print(f'Peso total: {peso_total} kg')
```

```
    Peso total: 727.0 kg
```

As variáveis `quantidade` (`int`) e `peso_unitario` (`float`) são multiplicadas, gerando um novo valor do tipo `float`, que é atribuído a `peso_total`. Essa operação é perfeitamente válida, tanto do ponto de vista sintático da linguagem quanto do ponto de vista lógico.

Agora, observe o que aconteceria se o programador, por descuido, digitasse `descricao` ao invés de `peso_unitario`.

```
descricao = 'Geladeira frost free 500l'  # tipo: str
quantidade = 10                          # tipo: int
peso_unitario = 72.7                     # tipo: float

peso_total = peso_unitario * descricao   # <- erro de digitação
print(f'Peso total: {peso_total} kg')
```

```
Traceback (most recent call last):
  File "c:\curso_python\aula1\variaveis.py", line 5, in <module>
    peso_total = peso_unitario * descricao
                 ~~~~~~~~~~~~~~~^~~~~~~~~~
TypeError: can't multiply sequence by non-int of type 'float'
```

Durante a execução do código, o interpretador encontrou uma operação inválida e gerou um erro de incompatibilidade de tipos, pois a multiplicação entre um `float` com um `str` não está definida.

Perceba que esse erro só ocorre em tempo de execução, no momento em que o interpretador Python tenta realizar a multiplicação entre um `float` e um `str`. Trata-se de um erro de lógica, não de sintaxe, pois, sendo Python uma linguagem de tipagem dinâmica, `peso_unitario` e `descricao` poderiam armazenar qualquer tipo de dado ao longo da execução do código, podendo sofrer mudanças não só de valor, mas também de tipo, até atingir a linha onde ocorre a tentativa e multiplicação e, consequentemente, o erro.

Esse erro seria facilmente detectado e corrigido no momento da escrita do código ou durante a compilação em linguagens de tipagem estática (como C, C++, Java, Rust, etc), impedindo sua ocorrência em tempo de execução. Esse exemplo foi incluído apenas para ilustrar também as desvantágens e os possíveis desafios da tipagem dinâmica.

Embora Python não suporte tipagem estática, ele suporta opcionalmente anotação de tipos. As anotações de tipos são declarações do programador quanto ao tipo intentado para cada variável criada[3].

Apesar das anotações de tipos não influenciarem a execução do código — sendo ignoradas pelo interpretador Python durante a execução — elas são extremamente úteis. Essas anotações são utilizadas por ferramentas de análise estática integradas à maioria das IDEs modernas para identificar possíveis erros de incompatibilidade de tipos no momento da escrita do código[4] além de melhorar a legibilidade e a manutenção, minimizando assim os riscos associados à tipagem dinâmica, mantendo todos os seus benefícios.

Reescrevendo o exemplo anterior com anotação de tipos, a IDE agora pode nos alertar sobre um provável erro antes mesmo da execução:

```
descricao: str = 'Geladeira frost free 500l'
quantidade: int = 10
peso_unitario: float = 72.7
                                  Operator '*' not supported for types 'float' and 'str'
peso_total = peso_unitario * descricao  # <- erro de digitação
print(f'Peso total: {peso_total} kg')
```

Além dos tipo primitivos, Python possui mais duas formas de armazenar informações: através de classes customizadas `class`, que serão discutidas no capítulo 5 e através de coleções de dados (`collections`)

[3] Além da anotação de tipo de variáveis, podemos também anotar tipos para parâmetros e retorno de funções/métodos

[4] A análise estática de tipos (ou *type checking*) está habilitada por padrão na IDE PyCham. Já no VS Code, ela pode ser habilitada incluindo os campos `'python.analysis.autoImportCompletions': true, 'python.analysis.typeCheckingMode': 'standard'` no arquivo de configurações `settings.json`.

1.2: Collections

Em Python, coleções são estruturas que armazenam múltiplos valores em uma única variável. As quatro coleções mais comuns da linguagem são: `list`, `tuple`, `set` e `dict`. Cada uma delas possui características específicas que as tornam mais adequadas para diferentes situações.

1.2.1: `list`

As listas (`list`) são uma das estruturas mais versáteis do Python, representam sequências de elementos ordenados e mutáveis. Ous seja, os elementos inseridos nela são mantidos na mesma ordem e podem ser alterados a qualquer momento, seja adicionando, removendo modificando ou substituindo itens.

Além disso, possuem tamanho arbitrário, crescendo conforme novos elementos são inseridos.

Uma lista pode ser criada utilizando colchetes `[]` e pode armazenar qualquer tipo de dado, inclusive, diferentes tipos para cada elemento. Por exemplo:

```
numeros = [8, 2, 0, -4, 15]               # lista de inteiros
nomes = ['Fulano', 'Beltrano', 'Sicrano']  # lista de strings
misto = [42, 'texto', 3.14, True]          # lista com tipos variados
```

A lista é uma coleção dinâmica, e pode crescer mesmo apos sua criação para acomodar novos elementos:

```
cores = ['azul', 'amarelo', 'cinza', 'vermelho']
print(f'pré: A lista 'cores' possui {len(cores)} elementos: {cores}')

cores.append('verde')  # adiciona um novo elemento de valor 'verde' ao final da lista
print(f'pós: A lista 'cores' possui {len(cores)} elementos: {cores}}')
```

```
pré: A lista 'cores' possui 4 elementos: ['azul', 'amarelo', 'cinza', 'vermelho']
pós: A lista 'cores' possui 5 elementos: ['azul', 'amarelo', 'cinza', 'vermelho', 'verde']
```

Esse código introduz um elemento novo: a chamada `cores.append()`, onde `append()` é um método da classe `list`. Métodos são semelhantes a funções, mas estão associados a um tipo de dado específico (a uma `class`) e normalmente modificam o próprio objeto ao qual pertencem.

A chamada de um método geralmente impacta diretamente o objeto a partir do qual ele é invocado. No caso de `append()`, um novo elemento é adicionado ao final da lista, alterando sua estrutura inicial.

Esse conceito será revisitado e formalizado com mais detalhes no Capítulo 5, quando exploraremos a construção e uso de classes em Python.

Note que a lista cresce automaticamente para acomodar novos elementos, sem que o programador precise gerenciar ou alocar memória adicional manualmente. Essa característica torna as listas particularmente flexíveis para armazenar conjuntos dinâmicos de dados.

Além disso, no último exemplo, introduzimos a função `len()`, que retorna o número de elementos presentes na coleção. Essa é uma função essencial para validar o tamanho de uma lista antes de realizar operações que dependam da quantidade de itens.

Também podemos acessar elementos individuais dessa lista, tanto para leitura como para escrita. Essa indexação é realizada com a seguinte sintaxe `nome_da_lista[indice]`, veja o código exemplo:

```python
cores = ['azul', 'amarelo', 'cinza', 'vermelho', 'verde']

# a indexação dos elemntos da listas por um inteiro inicia em 0
selecionada1 = cores[1]  # indice 1 equivale ao 2° elemento
print(f'selecioanda1 é: {selecionada1}')

# indice negativos podem ser usados para indexar elemntos do final para o início da listas
selecionada2 = cores[-2]  # -2 indica o penúltimo elemento
print(f'selecioanda2 é: {selecionada2}')

# é possível a indexação por slice, ou faixa de índices m:n (n é não inclusivo)
selecionadas = cores[0:3]  # 0:3 corresponde aos indices 0,1,2
print(f'selecionadas são: {selecionadas}')

# elementos da lista também podem ser individualemente modificados
cores[2] = 'marrom'  # altera o terceiro elemento de 'cinza' para 'marrom'
print(f'pós modificação: {cores}')

# podemo excluir elementos, o que desloca dos indices dos elementos a sua direita
del cores[3]
print(f'pós excusão por indice: {cores}')

# ou ainda podemo excluir não por indice mas por valor
cores.remove('azul')
print(f'pós excusão por valor: {cores}')

# tentar acessar índices que não existe resultará em um erro
cores[99]
```

```
selecioanda1 é: amarelo
selecioanda2 é: vermelho
selecionadas são: ['azul', 'amarelo', 'cinza']
pós modificação: ['azul', 'amarelo', 'marrom', 'vermelho', 'verde']
pós excusão por índice: ['azul', 'amarelo', 'marrom', 'verde']
pós excusão por valor: ['amarelo', 'marrom', 'verde']
Traceback (most recent call last):
  File "c:\curso_python\aula1\variaveis.py", line 28, in <module>
    cores[99]
    ~~~~~^^^^
IndexError: list index out of range
```

Vários outros métodos e funções para manipular listas estão disponíveis por padrão. Segue uma breve enumeração dos mais úteis:

- `len(lista)` – Retorna o número de elementos na lista.
- `lista.append(valor)` – Adiciona um novo elemento ao final da lista.
- `lista.insert(indice, valor)` – Insere um elemento em uma posição específica.
- `lista.remove(valor)` – Remove a primeira ocorrência do valor especificado.
- `lista.pop(indice)` – Remove e retorna o elemento na posição indicada (ou o último, se omitido).
- `lista.index(valor)` – Retorna o índice da primeira ocorrência do valor.
- `lista.count(valor)` – Retorna o número de vezes que o valor aparece na lista.
- `lista.sort()` – Ordena a lista em ordem crescente (por padrão).
- `lista.reverse()` – Inverte a ordem dos elementos da lista.
- `sorted(lista)` – Retorna uma nova lista ordenada, sem modificar a original.
- `lista.copy()` – Retorna uma cópia superficial da lista.
- `lista.clear()` – Remove todos os elementos da lista.

- `lista[i:j:k]` – fatiamento (*slicing*) de `i` até `j` com passo `k`.

Esses métodos permitem desde operações simples, como acessar elementos, até manipulações mais avançadas, como ordenação e remoção de elementos. No decorrer desse tutorial, abordaremos mais detalhes sobre a aplicação desses métodos em exemplos práticos.

1.2.2: `tuple`

Assim como `list`, a `tuple` (ou tupla) é uma estrutura de dados que permite armazenar uma sequência de valores. No entanto, enquanto listas são mutáveis (ou seja, seus elementos podem ser alterados, adicionados ou removidos), tuplas são imutáveis: uma vez criadas, seus elementos não podem ser modificados.

Essa característica confere à tupla maior segurança e previsibilidade, tornando-a ideal para representar coleções fixas de dados que não devem ser alteradas acidentalmente, ou até valores de retorno de uma função.

Tuplas são definidas por uma sequência de valores separados por vírgula (`,`), não necessariamente, mas comumente inseridos entre parênteses (`()`).

```
coordenada = (46.8, 22.3)
print(coordenada)
```

```
(46.8, 22.3)
```

Para definir uma tupla de um único elemento, é obrigatório adicionar uma vírgula, caso contrário, o Python interpretará o valor como um tipo isolado:

```
numero = (5)   # não é tupla, é apenas um int
tupla = (5,)  # tupla com um único elemento
print(f'numero é do tipo {type(numero)} e valor: {numero}')
print(f'tupla é do tipo {type(tupla)} e valor: {tupla}')
```

```
numero é do tipo <class 'int'> e valor: 5
tupla é do tipo <class 'tuple'> e valor: (5,)
```

A indexação de tuplas segue o mesmo princípio das listas, permitindo acessar elementos por índices (`0` para o primeiro item, índices negativos para contar do final)

```
cores = 'azul', 'verde', 'vermelho', 'amarelo', 'roxo'
print(cores[0])     # azul
print(cores[-1])    # roxo
print(cores[1:4])   # ('verde', 'vermelho', 'amarelo')
```

```
azul
roxo
('verde', 'vermelho', 'amarelo')
```

Embora tupla possa parecer uma versão limitada de lista, elas são estruturas complementares, com propósitos distintos. Essas limitações justamente tornam operações com tuplas mais previsíveis, rápidas e seguras.

1.2.3: `set`

Um `set` é uma coleção semelhante à lista (`list`), porém não permite valores duplicados e não mantém uma ordem fixa dos elementos. Isso faz com que seja uma estrutura útil para armazenar itens únicos e realizar operações como união, interseção e diferença.

Os conjuntos podem ser criados utilizando chaves `{}` ou a função `set()`:

```
cores = {'azul', 'vermelho', 'verde', 'azul'}
print(cores)
```

```
{'azul', 'vermelho', 'verde'}
```

Observe que, mesmo que `'azul'` tenha sido declarado duas vezes, ele aparece apenas uma vez no conjunto.

Outra forma de criar um conjunto é utilizando `set()` passando uma `list` como argumento:

```
valores = set([1, 2, 3, 4, 4, 5])
print(valores)
```

```
{1, 2, 3, 4, 5}
```

Os conjuntos são ideais quando precisamos garantir que os elementos sejam únicos, eliminando valores repetidos automaticamente.

Por não manter uma ordem fixa dos elementos, `set` não permite acesso por índice. Isso significa que não podemos utilizar a notação `set[indice]` para recuperar ou modificar elementos específicos, como fazemos com listas. Em vez disso, a manipulação de conjuntos é realizada por métodos próprios, como adição, remoção, união, interseção e diferença, que operam sobre o conjunto como um todo:

```
a = {1, 2, 3, 4}
b = {3, 4, 5}
b.add(6)

print(f'união: {a | b}')
print(f'interseção: {a & b}')
print(f'diferença: {a - b}')
```

```
união: {1, 2, 3, 4, 5, 6}
interseção: {3, 4}
diferença: {1, 2}
```

Embora `set` não suporte indexação direta, ainda é possível iterar sobre seus elementos utilizando laços como `for`. Dessa forma, podemos percorrer todos os itens do conjunto sem precisar acessar um elemento específico por índice. A iteração sobre coleções de dados será abordada no capítulo 3.

Aqui está uma lista resumida dos principais métodos e funções para manipulação de `set` em Python:

- `len(set)` – Retorna o número de elementos no conjunto.
- `set.add(valor)` – Adiciona um novo elemento ao conjunto.
- `set.remove(valor)` – Remove um elemento existente (gera erro se o valor não existir).
- `set.discard(valor)` – Remove um elemento sem gerar erro caso ele não exista.
- `set.pop()` – Remove e retorna um elemento aleatório do conjunto.
- `set.clear()` – Remove todos os elementos do conjunto.
- `set.copy()` – Retorna uma cópia do conjunto.

1.2.4: `dict`

Em Python, um dicionário (`dict`) é uma estrutura de dados que implementa uma *hash table*, permitindo o armazenamento de pares `key: value` (chave e valor). Essa abordagem garante acesso eficiente aos valores a partir de suas chaves, tornando a busca em grandes coleções extremamente rápida.

Diferente das listas (`list`) e conjuntos (`set`), que organizam elementos sequencialmente ou de forma desordenada, o dicionário associa cada valor a uma chave única, permitindo acessos diretos sem necessidade de percorrer toda a estrutura. Ná prática, essas chaves funcionam como os índices em listas.

Cada chave (`key`) pode ser qualquer tipo de dado que suporte a função `hash()`, como números, strings ou tuplas imutáveis. Já o valor (`value`) pode ser qualquer objeto do Python, incluindo listas, outras coleções, classes definidas pelo usuário e até mesmo funções.

Podemos criar um dicionário utilizando chaves (`{}`) e inserindo os pares `key: value` dentro delas. Por exemplo:

```
dados = {'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}
print(dados)
```

```
{'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}
```

Para recuperar um valor armazenado em um dicionário, utilizamos sua chave entre colchetes `[]`:

```
dados = {'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}
nome = dados['nome']
print(f'nome: {nome}')
```

```
nome: Fulano
```

Tentar acessar uma chave inexistente gera um erro `KeyError`:

```
dados = {'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}
telefone = dados['telefone']
```

```
Traceback (most recent call last):
  File "c:\curso_python\aula1\variaveis.py", line 2, in <module>
    telefone = dados['telefone']
    ~~~~~^^^^^^^^^^^^
KeyError: 'telefone'
```

Já escrever em uma chave previamente inexistes, expande o `dict` incluindo esse novo par:

```
dados = {'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}

dados['telefone'] = '(48)99999-9999'
print(dados)
```

```
{'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis', 'telefone': '(48)99999-9999'}
```

Dicionários são mutáveis, permitindo que seus elementos sejam alterados, adicionados ou removidos. Podemos modificar ou remover um valor acessando sua chave diretamente:

```
dados = {'nome': 'Fulano', 'idade': 25, 'cidade': 'Florianópolis'}

dados['idade'] = 26   # modifica o valor associado à chave 'idade'
del dados['cidade']   # remove o valor associado à chave 'cidade'
print(dados)
```

```
{'nome': 'Fulano', 'idade': 26}
```

Aqui estão alguns métodos úteis para trabalhar com dicionários:

- `len(dicionario)` – Retorna a quantidade de pares chave-valor no dicionário.
- `dicionario.keys()` – Retorna todas as chaves do dicionário.
- `dicionario.values()` – Retorna todos os valores do dicionário.
- `dicionario.items()` – Retorna todos os pares chave-valor como tuplas.
- `dicionario.pop(chave)` – Remove um item e retorna seu valor.
- `dicionario.update(outro_dicionario)` – Atualiza o dicionário com novos pares chave-valor.
- `dicionario.clear()` – Remove todos os elementos do dicionário.

Dicionários são ferramentas poderosas para armazenar e acessar dados de forma eficiente. Eles são amplamente utilizados em Python para estruturar informações e otimizar operações de busca.

1.3: Referência e mutabilidade na atribuição de variáveis

Em Python, as variáveis não armazenam diretamente os valores, mas referências para objetos na memória. Dessa forma, o efeito da atribuição (`=`) a uma variável depende da mutabilidade do objeto que ela referencia, definindo se a variável continuará apontando para o mesmo objeto ou passará a referenciar um novo.

Os objetos em Python são divididos em dois grupos: imutáveis e mutáveis, conforme sua capacidade de sofrer alterações após a criação.

Objetos imutáveis, como `int`, `float`, `str`, `bool` e `tuple`, não podem ser modificados após sua criação. Qualquer operação que tente alterar uma variável que referencia um imutável resulta na criação de um novo objeto, com um novo endereço de memória, e a variável passa a apontar para ele, sem afetar o objeto original.

Por outro lado, objetos mutáveis, como `list`, `dict`, `set` e instâncias de classes, permitem alterações diretas em seu conteúdo sem a necessidade de criar um novo objeto. Assim, modificações feitas em uma referência são refletidas em todas as variáveis que apontam para o mesmo objeto.

Veja um exemplo com variáveis imutáveis, como `int`:

```
x = 10    # x referencia um inteiro imutável
print(f"{type(x)=}, {id(x)=}, {x}")

y = x    # y agora referencia o mesmo objeto que x
print(f"{type(y)=}, {id(y)=}, {y}")

y = y + 5    # um novo objeto é criado e y passa a referenciá-lo
print(f"{type(y)=}, {id(y)=}, {y}")

z = 10    # uma nova variável z referencia o inteiro 10
print(f"{type(z)=}, {id(z)=}, {z}")
```

```
type(x)=<class 'int'>, id(x)=140706534401224, 10
type(y)=<class 'int'>, id(y)=140706534401224, 10
type(y)=<class 'int'>, id(y)=140706534401384, 15
type(z)=<class 'int'>, id(z)=140706534401224, 10
```

A função `id(x)`, utilizada em `print`, retorna o identificador único do objeto referenciado por `x`, que equivale ao seu endereço de memória na implementação CPython.

A saída do código mostra que `x` e `y` inicialmente referenciam o mesmo objeto `10`, compartilhando o mesmo identificador único (`id`). No entanto, ao modificar `y`, Python cria um novo objeto com valor `15`, e `y` passa a apontar para ele, sem alterar `x`.

Já `z`, criado posteriormente, também referencia o objeto `10`. Como esse valor é um inteiro imutável, Python pode reutilizar a mesma referência na memória[5], garantindo que o objeto permaneça inalterável. Dessa forma, qualquer modificação em `x` faz com que ele passe a apontar para outro objeto, sem impactar `z`.

Agora, um exemplo similar com variáveis mutáveis (lista de um elemento inteiro):

```
x = [10]    # x referencia um lista mutável com apenas um elemento
print(f"{type(x)=}, {id(x)=}, {x}")

y = x    # y agora referencia o mesmo objeto que x
print(f"{type(y)=}, {id(y)=}, {y}")

y[0] = y[0] + 5    # nenhum novo objeto é criado, o valor em y[0] é alterado
print(f"{type(y)=}, {id(y)=}, {y}")

z = [10]    # uma nova variável z, idêntica, mas independente de x
print(f"{type(z)=}, {id(z)=}, {z}")
```

```
type(x)=<class 'list'>, id(x)=1945482649600, [10]
type(y)=<class 'list'>, id(y)=1945482649600, [10]
type(y)=<class 'list'>, id(y)=1945482649600, [15]
type(z)=<class 'list'>, id(z)=1945484459648, [10]
```

Neste caso, `x` e `y` são referências para a mesma lista, que é mutável. Ao modificar `y[0]`, o conteúdo da lista é alterado diretamente, sem criar um novo objeto, o que é evidenciado pelo fato de que o identificador único (`id`) permanece o mesmo.

Já `z` contém outra lista `[10]`, alocada em um novo espaço de memória, o que explica seu identificador diferente.

Caso se deseje copiar os valores de uma lista para outra, em vez de apenas referenciar o mesmo objeto, é necessário usar `copy()` ou `deepcopy()`.

A função `copy()` cria uma nova lista contendo os mesmos elementos da lista original, mas se algum elemento for um objeto mutável, como outra lista ou um dicionário, a cópia conterá apenas referências para esses objetos, e modificações nos elementos internos afetarão ambas as listas.

Já `deepcopy()` copia recursivamente todos os objetos mutáveis dentro da estrutura, garantindo que cada nível da cópia seja independente da original.

Veja o exemplo:

[5] Essa reutilização de objetos é uma otimização da implementação CPython conhecida como *internin*, geramente aplicada para pequenos objetos.

```python
from copy import deepcopy

x = [10]    # lista mutável com apenas um elemento
print(f"{type(x)=}, {id(x)=}, {x}")

y = x    # y agora referencia o mesmo objeto que x
print(f"{type(y)=}, {id(y)=}, {y}")

z = deepcopy(x)    # nova lista com copias dos valores de x
print(f"{type(z)=}, {id(z)=}, {z}")
```

```
type(x)=<class 'list'>, id(x)=2511057429440, [10]
type(y)=<class 'list'>, id(y)=2511057429440, [10]
type(z)=<class 'list'>, id(z)=2511059612160, [10]
```

A compreensão da mutabilidade será crucial quando analisarmos a passagem de argumentos para funções, tema abordado no próximo capítulo.

Capítulo 2

Expressões e Funções

Uma expressão é qualquer fragmento de código que, ao ser executado, retorna um valor. Ela pode ser composta por valores, variáveis, operadores e chamadas de funções, que são avaliadas para produzir um resultado. Esse resultado pode ser armazenado em uma variável ou utilizado como entrada para outra expressão ou função.

Um exemplo comum de expressão são as operações aritméticas:

```
a = 3
b = 7

y = a * 5 + b
```

Aqui, as variáveis `a` e `b` são inicializadas com valores fixos (ou *literals*, como são chamados em programação), enquanto `y` recebe o resultado do processamento da expressão `a * 5 + b`, que retorna `22`.

Os operadores aritméticos em Python são: adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`), divisão inteira (`//`), resto da divisão (`%`) e potenciação (`**`).

Outro pilar essencial da programação são as funções, que permitem a organização e reutilização de código. Elas ajudam a evitar repetições desnecessárias, encapsular blocos lógicos e contribuir para uma estrutura mais eficiente e modular.

Uma função é um bloco de código reutilizável, projetado para executar uma ação específica. Sua declaração possui três elementos fundamentais:

1. Parâmetros de entrada, que atuam como espaços reservados para valores que serão fornecidos posteriormente
2. quando a função for chamada.
3. O corpo da função, onde a lógica é definida e executada.
4. O valor de retorno, que é o resultado da função após seu processamento.

Quando chamamos a função, fornecemos argumentos, que são os valores reais atribuídos aos parâmetros de entrada. Esses argumentos carregam informações essenciais vindas do contexto externo, ou seja, da parte do programa que chamou a função. A função, então, processa esses dados e executa a lógica necessária para produzir um resultado.

Outro aspecto crucial das funções é o escopo de suas variáveis locais (internas). Variáveis criadas dentro de uma função existem apenas durante sua execução e são automaticamente descartadas ao seu término. Isso evita interferências indesejadas em outras partes do código e melhora a eficiência do uso de memória.

Para ilustrar esse conceito, podemos criar a função `eq_reduzida_reta()`, que encapsula o processamento da expressão matemática utilizada anteriormente. Nesse caso, ela receberá três parâmetros de entrada: `a`, `b` e `x`. No corpo da função, a expressão matemática será avaliada para calcular `y`, que será então retornado como resultado.

A chamada a essa função constitui uma expressão, de forma que esse valor de retorno pode ser capturado e armazenado em uma variável ou usada como valor para outra expressão, ou ainda como argumento para outra função.

Veja como essa função pode ser definida em Python:

```
# definição da função
# não executa nenhum código, apenas define o que será executado quando a função for chamada
def eq_reduzida_reta(a, x, b):
    y = a * x + b
    return y

# chamada da função com diferentes argumentos
resultado1 = eq_reduzida_reta(3, 5, 7)  # argumentos posicionais 3, 5, 7
resultado2 = 5 + eq_reduzida_reta(4, 2, 6)  # resultado como entrada para uma expressao
resultado3 = eq_reduzida_reta(x=-2, b=10, a=2)  # argumentos nomeados

print(f'resultado1: {resultado1}')
print(f'resultado2: {resultado2}')
print(f'resultado3: {resultado3}')
```

```
resultado1: 22
resultado2: 19
resultado3: 6
```

A declaração da função é indicada pela *keyword* `def`, seguida do nome da função e pela lista de parâmetros de entrada, declarados entre parênteses. Após os dois pontos (`:`), inicia-se o corpo da função.

Diferente de algumas outras linguagens onde a indentação é apenas estética, em Python ela é essencial, pois define o bloco de código pertencente à função. Isso garante a organização da estrutura e evita ambiguidades na execução.

Por fim, o comando `return` especifica o valor que será retornado ao invocador da função. Se omitida, a função não retorna um valor explícito, e seu retorno será implicitamente `None`.

Uma chamada de função é feita pelo seu nome, seguido por um par de parênteses. Dentro dos parênteses, passamos os argumentos, que devem corresponder aos parâmetros definidos pela função.

Em Python, essa correspondência pode ocorrer de duas formas:

- Argumentos posicionais, em que os valores são passados na ordem definida pela função.
- Argumentos nomeados, onde especificamos explicitamente qual valor será atribuído a cada parâmetro, permitindo uma ordem arbitrária.

Também podemos combinar ambos os tipos de argumentos em uma chamada de função. No entanto, os argumentos posicionais devem obrigatoriamente vir primeiro, seguidos pelos nomeados.

Python também permite a definição de parâmetros opcionais, o que proporciona maior flexibilidade na chamada de funções. Isso significa que ao invocar uma função, podemos fornecer todos os argumentos necessários, ou omitir aqueles marcados como opcionais.

Um parâmetro é considerado opcional quando, na definição da função, atribuímos a ele um valor padrão. Esse valor será usado automaticamente caso o chamador da função não forneça um argumento correspondente.

Por exemplo, podemos reescrever a função `eq_reduzida_reta()` tornando `b` um parâmetro opcional, atribuindo a ele um valor padrão de `0`. O código correspondente seria:

```
# definição da função com parâmetro opcional
def eq_reduzida_reta(a, x, b=0):
    y = a * x + b
    return y

# chamada da função com diferentes quantidades de argumentos
resultado1 = eq_reduzida_reta(3, 5, 7)  # arg b = 7 (fornecido explicitamente)
resultado2 = eq_reduzida_reta(3, 5)     # arg b omitido, valor padrão é usado (b=0)

print(f'resultado1: {resultado1}')
print(f'resultado2: {resultado2}')
```

```
resultado1: 22
resultado2: 15
```

Esse conceito permite maior flexibilidade na chamada de funções, reduzindo a necessidade de fornecer todos os argumentos manualmente quando faz sentido na lógica da função possuir comportamento padrão. Se não indicarmos um valor para `b`, a função usará automaticamente `b=0`. Os demais parâmetros (`a` e `b`) não possuem valor padrão, portanto, são obrigatórios. Chamar a função sem fornecê-los resultaria em um erro.

Assim como na criação de variáveis, também podemos anotar os tipos dos parâmetros e do valor de retorno de uma função. Essas anotações não afetam a execução do programa, pois são ignoradas pelo interpretador durante a execução. No entanto, elas podem ser analisadas por um *type checker* da IDE, que alerta sobre possíveis incompatibilidades antes da execução do código.

Isso melhora a legibilidade do código e ajuda na detecção de erros potenciais, tornando a programação mais segura e organizada, além de documentar as intenções do programador quando escreveu tal função.

```
# definição da função com parâmetro opcional e anotação te tipos esperados
def eq_reduzida_reta(a: int, x: int, b: int = 0) -> int:
    y = a * x + b
    return y
                                        'float' is not assignable to 'int'
                                          'Literal['texto']' is not assignable to 'int'
# chamada da função
resultado1 = eq_reduzida_reta(3, 5.6, 7)    # 'float' incompatível com declaração
resultado2 = eq_reduzida_reta(3, 'texto')   # 'str' incompatível com declaração
resultado3 = eq_reduzida_reta(3, 5, 7)      # ok
```

Aqui, todos os parâmetros de entrada e o valor de retorno foram definidos com o tipo `int`, o que garante que apenas números inteiros sejam aceitos. Se um valor `str` ou `float` for passado incorretamente, um *type checker* pode alertar sobre a incompatibilidade.

Contudo, para essa função em específico, números fracionários (`float`) seriam completamente válidos. Para permitir esse tipo de entrada, podemos utilizar anotações de múltiplos tipos com o operador `|`, como no exemplo abaixo:

```python
# Definição da função com suporte a múltiplos tipo
def eq_reduzida_reta(a: int | float, x: int | float, b: int | float = 0) -> int | float:
    y = a * x + b
    return y
```

Além disso, nada impede que parâmetros e valores de retorno tenham tipos diferentes—tudo depende da lógica que a função deve executar. A anotação de tipos é opcional, mas é altamente encorajada para manter a clareza do código e detectar possíveis erros antes da execução.

Inda sobre funções, os parâmetros de entrada e o valor de retorno são elementos opcionais em uma função e sua presença depende da ação que se deseja realizar. Algumas funções podem simplesmente executar uma tarefa sem receber valores externos ou sem produzir e retornar um resultado.

Um exemplo disso é a função padrão `print`, que já utilizamos ao longo deste material sem muitas explicações. Essa função recebe um argumento, geralmente uma `str` contendo o texto a ser exibido, e imprime essa informação no terminal. No entanto, ela não retorna um valor[1] para quem a chamou no código.

Internamente ela apenas faz uma chamada para o sistema operacional para imprimir um texto no terminal.

Em Python, funções também podem retornar múltiplos valores simultaneamente, o que torna o código mais flexível e evita a necessidade de usar estruturas, listas ou dicionários para armazenar múltiplos resultados.

Essa funcionalidade é especialmente útil quando uma função precisa retornar vários cálculos ou informações relacionadas ao mesmo tempo, além de possibilitar o retorno de dados sobre erros encontrados durante sua execução

Isso é possível porque Python permite que uma função retorne uma `tupla`, que pode ser desempacotada na chamada da função. Veja um exemplo:

```python
import math

def bhaskara(a: float, b: float, c: float) -> tuple[float, float]:
    delta = b**2 - 4*a*c
    if delta < 0:
        raise ValueError("A equação não possui raízes reais. O delta é negativo.")

    raiz1 = (-b + math.sqrt(delta)) / (2 * a)
    raiz2 = (-b - math.sqrt(delta)) / (2 * a)
    return raiz1, raiz2    # retorna ambas as raízes

r1, r2 = bhaskara(a=0.5, b=14.3, c=45.0)
print(f'{r1 = }')
print(f'{r2 = }')
```

```
r1 = -3.5999999999999996
r2 = -25.0
```

Observe que `return` retorna uma `tuple` contendo dois valores do tipo `float`. Essa tupla é então desempacotada pelo código que chama a função, e cada valor retornado é atribuído a uma variável distinta (`r1` e `r2`).

No código acima, a linha `import math` é uma declaração ao interpretador para importar o módulo `math`, já que `sqrt`, usada para calcular a raiz quadrada, não é uma função embutida

[1] Tecnicamente, toda função python tem um retorno. Se `retorn` não for usado dentro da função então por padrão o interpretador Python retornará `None`, que justamente é o objeto que indica a ausência de valor.

(não está disponível por padrão). No entanto, ela faz parte da biblioteca padrão do Python, então não é necessário instalar nenhum pacote externo de terceiros.

Ainda nesse exemplo, introduzimos o comando `raise`, que lança uma exceção (geralmente associada a um erro) caso a função encontre raízes complexas, uma condição para a qual ela não foi projetada para lidar. Se essa exceção não for tratada[2] pelo código que chama a função, a execução do programa será interrompida.

2.1: Mutabilidade dos argumentos de funções

Outro aspecto importante sobre funções em Python é que os argumentos são sempre passados por referência (compartilhamento de objeto). Ou seja, os nomes das variáveis dentro da função apontam para os mesmos objetos que foram passados como argumento.

No entanto, há uma diferença de comportamento entre objetos imutáveis (como `int`, `float`, `str`, `bool` e `tuple`) e objetos mutáveis (como `list`, `dict`, `set` e classes definidas pelo usuário):

- Objetos imutáveis não podem ser alterados diretamente. Por isso, mesmo que a referência seja compartilhada, qualquer tentativa de modificação resulta na criação de um novo objeto. Na prática, eles se comportam como se fossem passados por cópia.
- Já objetos mutáveis podem ser alterados diretamente. Isso significa que modificações feitas dentro da função afetam o objeto original fora dela.

Abaixo vemos um exemplo de uma função que recebe uma variável imutável:

```
def quadrado(x):
    print(f"dentro da função antes da modificação: {id(x)=}, {x=}")
    x = x**2
    print(f"dentro da função após a modificação: {id(x)=}, {x=}")

a = 5
quadrado(a)
print(f"fora da função: {id(a)=}, {a=}")
```

```
dentro da função antes da modificação: id(x)=140706763580456, x=5
dentro da função após a modificação: id(x)=140706763581096, x=25
fora da função: id(a)=140706763580456, a=5
```

Fica evidente que a instrução `x = x**2` não modifica o objeto originalmente apontado por `x`, mas sim cria um novo objeto em uma nova posição da memória. A variável `x` passa, então, a referenciar esse novo objeto, ocultando (ou *shadowing*) a referência anterior dentro do escopo da função. A variável `a`, fora da função, permanece inalterada, continuando a referenciar o objeto original.

Agora, observe o comportamento distinto ao lidarmos com argumentos mutáveis:

```
def quadrado(x):
    print(f"dentro da função antes da modificação: {id(x)=}, {x=}")
    x[0] = x[0]**2
    x[1] = x[1]**2
    print(f"dentro da função após a modificação: {id(x)=}, {x=}")

a = [5, 3]
quadrado(a)
print(f"fora da função: {id(a)=}), {a=}")
```

[2] O tratamento adequado de exceções em Python é realizado por meio da estrutura `try-except`, permitindo capturar e gerenciar erros de maneira controlada. No entanto, essa abordagem foge ao escopo desse texto introdutório.

```
dentro da função antes da modificação: id(x)=2075912921088, x=[5, 3]
dentro da função após a modificação: id(x)=2075912921088, x=[25, 9]
fora da função: id(a)=2075912921088), a=[25, 9]
```

Como a lista é um objeto mutável, a função altera diretamente o conteúdo do objeto referenciado, mantendo o mesmo `id`. Isso mostra como mudanças feitas dentro da função refletem fora dela — o parâmetro ainda aponta para o mesmo objeto, agora modificado.

Compreender essa distinção é fundamental, pois muitas funções em Python podem não retornar um valor explícito, mas ainda assim modificar o estado de objetos mutáveis passados como argumento. Esse comportamento pode gerar efeitos colaterais significativos na execução do programa — especialmente quando não é antecipado pelo desenvolvedor.

Por outro lado, esse mecanismo também é uma ferramenta poderosa para evitar cópias desnecessárias de objetos grandes, permitindo que funções operem diretamente sobre estruturas complexas sem comprometer o desempenho ou o uso de memória.

Funções desse tipo — que alteram diretamente os dados aos quais têm acesso — são conhecidas como funções com efeitos colaterais (*side-effect functions*) e são bastante comuns em operações sobre estruturas de dados, como listas, dicionários ou objetos de classes definidas pelo usuário.

O oposto de uma *side-effect function* é uma função pura (*pure function*). Funções puras garantem que seus parâmetros de entrada sejam apenas lidos, sem qualquer modificação. Isso é possível mesmo quando se trabalha com objetos mutáveis, desde que se evite alterar seu estado. Para isso, é comum criar cópias dos objetos antes de aplicar transformações — o que preserva a integridade dos dados originais, mas implica em custos de duplicação do objeto, com maior uso de memória e tempo de processamento.

Em Python, a cópia de objetos pode ser realizada com as funções `copy()` ou `deepcopy()` do módulo `copy`, dependendo da profundidade desejada.

Capítulo 3

Estruturas de controle de fluxo

Até agora, os pequenos trechos de código que escrevemos seguiram uma sequência linear e bem definida, executando instruções em uma ordem preestabelecida, onde cada linha era processada incondicionalmente — como seguir uma receita de bolo à risca.

No entanto, essa abordagem não é suficiente para representar comportamentos mais complexos, onde o próprio algoritmo precisa tomar decisões com base no estado atual. Por exemplo, em determinado ponto da execução, se uma condição específica for verdadeira, o programa pode seguir pelo caminho A; caso contrário, deve seguir pelo caminho B. Cada um desses caminhos corresponde a blocos de código distintos, que realizam ações diferentes para atingir um certo objetivo.

Além das decisões condicionais, os programas frequentemente precisam executar uma mesma ação várias vezes, sem que o programador precise escrever o mesmo código repetidamente. Além de ser impraticável, muitas vezes a lógica desejada exige que a decisão sobre quantas vezes executar uma tarefa seja tomada durante a própria execução do programa, baseada no estado atual.

Para isso, utilizamos as estruturas de repetição (ou loops), que permitem a execução contínua de um bloco de código enquanto certas condições forem atendidas, ou garantem que um bloco de código seja executado para todos os valores de uma lista. Dessa forma, o programa pode automatizar tarefas, tornando o código mais eficiente e adaptável.

Em qualquer programa de computador, controlar a ordem de execução das instruções é fundamental para garantir que a aplicação possa lidar com diferentes cenários, tomar decisões dinamicamente e repetir ações conforme necessário. Esse controle é realizado por meio das estruturas de controle de fluxo, que possibilitam modificar o comportamento de um programa, tornando-o mais flexível e inteligente

Como aludido anteriormente, as estruturas de controle de fluxo podem ser classificadas em duas categorias: estruturas condicionais e estruturas de repetição. Sua implementação pode variar entre diferentes linguagens, e até mesmo dentro de uma mesma linguagem, podem existir múltiplas variantes para tornar seu uso mais conveniente em diferentes situações.

Por exemplo, em Python, decisões condicionais podem ser feitas com `if-else`, uma abordagem mais direta, ou com `match-case`, que facilita comparações mais organizadas entre múltiplas possibilidades. Da mesma forma, repetições podem ser controladas com `while`,

quando se depende de uma condição, ou com `for`, que é ideal para percorrer sequências.

Essa flexibilidade permite que programadores escolham a estrutura mais adequada para cada contexto, tornando o código mais simples, expressivo e eficiente.

Nas próximas seções, exploraremos cada categoria detalhadamente, com exemplos para ilustrar seu funcionamento.

3.1: Estruturas condicionais

As estruturas condicionais permitem que um programa tome decisões durante sua execução com base em condições estabelecidas. Em Python, existem diferentes variantes dessa estrutura, tornando-a mais versátil para diversas situações. A abordagem mais comum é o uso de `if-else`, onde um bloco de código é executado apenas se uma condição for verdadeira. Para comparações múltiplas podes usar `if-elif-else` ou `match-case`, introduzido no Python 3.10, oferece uma alternativa estruturada, útil quando há diversas possibilidades de decisão.

3.1.1: `if-else`

Exemplo do uso de `if-else`:

```
idade = 20

if idade >= 18:
    print("Você é maior de idade.")  # é executado se idade for maior ou igual a 18
else:
    print("Você é menor de idade.")  # é executado se idade menor que 18
```

O comando `if` sempre espera uma expressão que resulte em um valor booleano, ou seja, `True` (verdadeiro) ou `False` (falso).

Quando a condição for avaliada como `True`, o bloco de código associado ao `if` será executado. Caso contrário, se houver um `else` (que é opcional), seu respectivo bloco será executado.

Cada bloco pode ter um tamanho arbitrário e conter múltiplas instruções, incluindo estruturas `if-else` aninhadas, permitindo expressar deci soes mais complexas. Assim como nas funções, a delimitação de um bloco `if` é feita pela indentação.

No exemplo, dependendo do valor da variável idade a comparação `idade >= 18` retorna `True` ou `False`, e baseado nesse valor o `if` escolhe qual bloco de código executar, exibindo a mensagem apropriada.

Essas expressões booleanas geralmente envolvem o uso de operadores relacionais (`==`, `!=`, `>`, `<`, `>=`, `<=`), que podem ser combinados com operadores lógicos (`and`, `or`, `not`) para formar condições lógicas mais complexas. Além disso, o operador `is` verifica se duas variáveis referenciam o mesmo objeto na memória, diferindo do `==`, que compara os valores desses objetos. O operador `in` é útil para verificar se um determinado valor está presente em uma coleção, como uma lista. Outra função embutida `isinstance(obj, class)` é especialmente útil para verificar o tipo de um objeto dentro dessas expressões.

3.1.2: `if-elif-else`

Segue um exemplo de uma função que verifica o cadastro de um usuário para identificar um meio de contato válido. A prioridade é o email; caso não esteja disponível, o número de telefone será retornado como alternativa.

```python
def obter_contato(cadastro: dict) -> str:
    if 'email' in cadastro and cadastro['email'] != '':
        return cadastro['email']
    elif 'telefone' in cadastro and cadastro['telefone'] != '':
        return cadastro['telefone']
    else:
        return 'nenhum contato cadastrado'

usuario1 = {'nome': 'Fulano', 'email': 'fulano@ifsc.edu.br', 'telefone': '(48)99999-9999'}
usuario2 = {'nome': 'Beltrano', 'telefone': '(48)99999-8888'}
usuario3 = {'nome': 'Sicrano', 'telefone': ''}

print('contato usuario1:', obter_contato(usuario1))
print('contato usuario2:', obter_contato(usuario2))
print('contato usuario3:', obter_contato(usuario3))
```

```
contato usuario1: fulano@ifsc.edu.br
contato usuario2: (48)99999-8888
contato usuario3: nenhum contato cadastrado
```

3.1.3: `match-case`

A seguir temos um exemplo usando `match-case` para verificar se um determinado dia da semana é um final de semana ou um dia útil:

```python
dia_da_semana = "sábado"

match dia_da_semana.lower():
    case "sábado" | "domingo":
        print("É fim de semana! Aproveite para descansar.")
    case "segunda" | "terça" | "quarta" | "quinta" | "sexta":
        print("É dia útil. Hora de trabalhar ou estudar!")
    case _: # executado caso não haja matching
        print("Dia inválido. Certifique-se de inserir um nome correto.")
```

Esse código verifica o valor da variável `dia_da_semana` e determina se o dia pertence ao final de semana ou se é um dia útil, comparando-o com as opções especificadas nos diferentes casos (`case`) dentro da estrutura `match-case`[1].

Cada `case` dentro de um `match` representa um possível caso de correspondência, permitindo que o programa escolha dinamicamente a execução adequada de apenas um deles.

O operador `|` possibilita agrupar múltiplos casos numa mesma cláusula, tornando o código mais legível e eficiente.

Além disso, a conversão para minúsculas (`lower()`) garante que a entrada do usuário funcione corretamente, independentemente de letras maiúsculas ou minúsculas.

3.2: Estruturas de repetição

As estruturas de repetição permitem que um programa execute um bloco de código múltiplas vezes de forma controlada, eliminando repetições manuais e tornando o código mais legível e organizado. Além de aprimorar a clareza, elas aumentam a eficiência e versatilidade do desenvolvimento ao possibilitar que o número de execuções seja definido dinamicamente durante a execução, adaptando-se às exigências da lógica que está sendo implementada.

Em Python, assim como na maioria das linguagens, há duas principais formas de implementar repetições: o comando `while`, ideal para execuções dependentes de uma condição dinâmica,

[1] em Python `match-case` é uma estrutura mais completa que não realiza apenas comparações de valores, mas sim *pattern matching*, que envolve conceitos de desestruturação e correspondência de padrões. Conceitos um pouco mais avançados que ficarão de fora desse tutorial introdutório.

e o comando `for`, mais adequado para percorrer elementos de uma sequência de maneira previsível. Cada uma dessas estruturas apresenta vantagens específicas e pode ser aplicada conforme a necessidade do programa, abrangendo desde a iteração sobre sequências ou coleções de dados até algoritmos que requerem monitoramento contínuo e definição dinâmica do critério de parada, como cálculos iterativos em modelos computacionais.

3.2.1: `while`

O comando `while` permite a repetição de um bloco de código enquanto uma condição for verdadeira, sendo especialmente útil em situações onde o número de iterações não pode ser determinado previamente. Essa abordagem possibilita a execução de tarefas que dependem de eventos externos, verificações contínuas ou condições de parada dinâmicas.

O exemplo a seguir apresenta um programa interativo no qual o usuário deve tentar adivinhar um número secreto (57). Caso acerte, vence o jogo e o programa é encerrado. Se errar, poderá continuar tentando até acertar.

```
numero_secreto = 57
print("Tente adivinhar o número entre 0 e 99.")
palpite = int(input("Digite seu palpite: "))

while palpite != numero_secreto:
    if palpite < numero_secreto:
        print("O número secreto é maior. Tente novamente!")
    else:
        print("O número secreto é menor. Tente novamente!")

    palpite = int(input("Digite seu palpite: "))

print("Parabéns! Você ganhou!")
```

Já no exemplo a seguir, o programa imprime no terminal os números naturais menores que `10`. (valor definido pelo usuário):

```
contador = 0
while contador < 10:
    print(contador, end=' ')
    contador += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

Nesse contexto, a variável `contador` desempenha o papel de variável de controle, sendo responsável por acompanhar e atualizar o estado da repetição a cada iteração.

Como o loop `while` apenas avalia uma expressão booleana, é necessário que essa variável seja inicializada, monitorada e modificada manualmente. Sua inicialização fora do laço e seu incremento em `1` a cada iteração é essencial para garantir o avanço do programa e evitar loops infinitos.

Esse processo de percorrer uma sequência pre-definida (ou até coleção de dados) é tão fundamental em programação que a maioria das linguagens, incluindo Python, oferece uma estrutura específica para essa finalidade. Com o comando `for`, o estado da iteração é gerenciado automaticamente, sem necessidade de controle manual pelo programador, tornando o código mais conciso e legível.

3.2.2: `for`

O comando `for` é utilizado principalmente para percorrer elementos de uma sequência, garantindo que cada item seja processado de maneira previsível.

O exemplo anterior pode ser reescrito substituíndo o `while` por um `for` da seguinte forma:

```
for i in range(10):
    print(i, end=' ')
```

```
0 1 2 3 4 5 6 7 8 9
```

Aqui, a variável `i` não precisa ser declarada ou incrementada manualmente, pois o próprio comando `for` atribui a ela cada valor dentro da sequência gerada por `range(10)`. Na primeira iteração, `i` recebe o valor `0`, na segunda `1`, e assim por diante, até atingir `9`. O valor `10` é não inclusivo, ou seja, o loop termina antes de alcançá-lo, e já que `10` gera uma sequência de `0` a `9`, temos exatamente `10` repetições do loop.

A estrutura `for` é particularmente útil ao trabalhar com listas, tuplas, dicionários e outras coleções de dados, permitindo percorrer diretamente seus elementos sem necessidade de manipular índices manualmente. Além disso, o `for` em Python é conceitualmente equivalente ao `for-each` presente em linguagens como Java e C#, pois permite acessar diretamente cada elemento de um iterável, tornando o processo de iteração mais intuitivo.

O exemplo a seguir demostra isso:

```
nomes = ['Fulano', 'Beltrano', 'Sicrano']

for nome in nomes:
    print(f"Saudações, {nome}!")
```

```
Saudações, Fulano!
Saudações, Beltrano!
Saudações, Sicrano!
```

Aqui, o loop percorre diretamente os elementos da lista `nomes`, atribuindo cada valor individualmente à variável `nome` em cada iteração, sem exigir um índice explícito. Dessa forma, o processamento ocorre um a um, garantindo que cada elemento seja tratado separadamente. Essa abordagem torna o código mais claro, eliminando a necessidade de manipulação manual de índices e facilitando a interação direta com os dados da coleção.

Em situações onde é necessário acessar tanto o índice quanto o valor de cada elemento durante a repetição, o método `enumerate()` pode ser utilizado. Ele retorna pares de valores, onde cada elemento da coleção recebe seu respectivo índice de forma automática.

```
nomes = ['Fulano', 'Beltrano', 'Sicrano']

for indice, nome in enumerate(nomes):
    print(f"{indice}: Saudações, {nome}!")
```

```
0: Saudações, Fulano!
1: Saudações, Beltrano!
2: Saudações, Sicrano!
```

Python também oferece diversas funções embutidas para manipular iteradores, como `reversed()`, permitem percorrer uma sequência de trás pra frente:

```
nomes = ['Fulano', 'Beltrano', 'Sicrano']

for nome in reversed(nomes):
    print(f"Saudações, {nome}!")
```

```
Saudações, Sicrano!
Saudações, Beltrano!
Saudações, Fulano!
```

Além disso, `zip()` possibilita a iteração simultânea sobre múltiplas coleções, associando pares de valores um a um a cada repetição:

```
nomes = ['Fulano', 'Beltrano', 'Sicrano']
idades = [22, 35, 65]

for nome, idade in zip(nomes, idades):
    print(f"{nome} tem {idade} anos.")
```

```
Fulano tem 22 anos.
Beltrano tem 35 anos.
Sicrano tem 65 anos.
```

Tanto o laço `while` quanto o `for` podem ser interrompidos a qualquer momento com o comando `break`. Esse mecanismo permite finalizar a execução do loop imediatamente, ignorando qualquer condição restante e continuando a execução do programa a partir da próxima instrução após o laço.

Capítulo 4

Exercícios

O desenvolvimento do raciocínio lógico e sua aplicação na programação são habilidades que se aprimoram, sobretudo, pela prática. Por isso, antes de explorarmos novos recursos, é essencial consolidarmos o que já estudamos.

Até agora, apresentamos pequenos trechos de código para ilustrar os principais mecanismos da linguagem. Python oferece uma ampla gama de funcionalidades, mas os conceitos abordados até aqui — variáveis, tipos, expressões, funções e controle de fluxo — são fundamentais em qualquer linguagem de programação e já possibilitam a construção de diversas soluções.

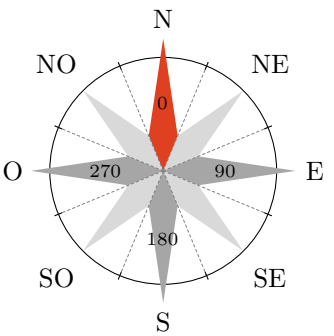
Os problemas propostos neste capítulo, embora em alguns casos se inspirem em desafios práticos reais, não têm como objetivo desenvolver soluções otimizadas para produção. Em vez disso, buscam fortalecer o raciocínio lógico, a capacidade de resolver problemas e a habilidade de transcrever soluções para o computador. Além disso, buscam desenvolver uma maior familiaridade com a linguagem, a IDE e o ecossistema que a envolve.

Questão 4.1: `bussola`

Implemente uma função uma função `bussola(deg: float) -> str` que recebe um ângulo em graus e retorna o ponto cardeal ou colateral mais próximo como uma string (`'Sul'`, `'Sudeste'`, `'Leste'`, etc).

A função deve tratar corretamente valores negativos ou maiores que `360`, assegurando a normalização do ângulo por meio da aritmética modular.

Adote a convenção em que `0` graus corresponde ao Norte e `90` graus ao Leste.



Implemente o código:

```
def bussola(deg: float) -> str:
    # seu código aqui

#teste
print(bussola(20.7))
print(bussola(169.2))
print(bussola(-60.3))
print(bussola(868.5))
```

Saída esperada:

```
Norte
Sul
Noroeste
Sudoeste
```

Questão 4.2: `desvio_padrao`

Escreva uma função que recebe como argumento uma lista numérica **x** de tamanho arbitrário e retorna o valor calculado do desvio padrão considerando probabilidades iguais para todos os elementos. Ou seja:

$$\sigma(\mathbf{x}) = \sqrt{\frac{1}{N-1}\sum_{n=1}^{N}(x_n - \bar{x})^2} \tag{4.1}$$

sendo $\bar{x}$ a média aritmética dos elementos de **x** e N o número de elementos de **x**.

Compare o resultado de sua função com a função padrão `stdev()` do módulo `statistics`.

Questão 4.3: `login`

Escreva uma função para validar usuário e senha. Ao ser executado, o programa solicita ao usuário (via terminal) seu login e senha. Os valores são então comparados a um cadastro pré-definido (armazenado em um dicionário global). A função retorna `True` ou `False`, além de exibir uma mensagem no terminal informando se o acesso foi concedido ou negado.

```
def validacao(usuario: str, senha: str) -> bool:
    global CADASTROS
    # seu código aqui


# logins e senhas previamente cadastrados
CADASTROS = {
    'fulano'  : '8g7&4A5',
    'beltrano': 'Hgr787@',
    'sicrano' : 'po78BA-'
    }

# interrompe a execução e solicita usuário e senha
usuario = input("Digite seu login:")
senha = input("Digite sua senha:")

# chama a função de validação
acesso = validacao(usuario, senha)
```

Desafio:

Modifique o programa para permitir ao usuário três tentativas de acesso. Caso a senha seja digitada incorretamente três vezes consecutivas, essa senha deve ser permanentemente invalidada. Crie uma nova mensagem informando essa situação. A invalidação da senha de um usuário não deve afetar os demais.

Importante:

Tenha em mente que este exercício tem como objetivo explorar conceitos introdutórios de comparação e controle de fluxo. No entanto, por razões óbvias de segurança, senhas nunca devem ser armazenadas ou comparadas diretamente. Em aplicações reais, a validação de credenciais sempre envolve criptografia robusta e bancos de dados especializados, projetados para garantir a proteção das informações.

Questão 4.4: `separa_pares_impares`

Escreva uma função que recebe uma lista de números inteiros de tamanho arbitrário e retorne duas listas, a primeira contendo os elementos pares da lista original, e a segunda os ímpares.

```python
def separa_pares_impares(lista):
    # seu código aqui

numeros = [5, 7, -9, 6, 4, 7, 9, 3, 8, 6]

pares, impares = separa_pares_impares(numeros)
print(f"pares: {pares}")
print(f"ímpares: {impares}")
```

Resultado esperado:

```
pares: [6, 4, 8, 6]
ímpares: [5, 7, -9, 7, 9, 3]
```

Desafio:

Reescreva a função adicionando anotação de tipos tanto para os parâmetros como para o retorno.

Questão 4.5: `seno`

Escreva uma função `seno(alpha: float) -> float` que receba um ângulo em radianos e retorne seu valor aproximado de seno, utilizando a seguinte série de Taylor:

$$\text{sen}(\alpha) = \alpha - \frac{\alpha^3}{3!} + \frac{\alpha^5}{5!} - \frac{\alpha^7}{7!} + \cdots \tag{4.2}$$

A expansão deve ser truncada no primeiro termo em que:

$$\left| \frac{\alpha^k}{k!} \right| \leq 0.0001 \tag{4.3}$$

A série acima é convergente para ângulos pertencentes ao intervalo [-π, +π]. Portanto, antes de calcular a série, caso o ângulo informado esteja fora desse intervalo, ele deve ser mapeado para o intervalo de convergência.

Além da função de cálculo do seno, implemente um script de teste que compare os valores obtidos com os retornados pela função `math.sin()` do Python.

Questão 4.6: `produto_mais_vendido`

Crie um programa que receba um dicionário de produtos vendidos, contendo o nome do produto e a quantidade vendida. A função deve calcular e retornar uma tupla com o nome e a quantidade do produto mais vendido.

```python
def produto_mais_vendido(dados: dict) -> tuple[str, int]:
    # seu código aqui

vendas = {
    'Notebook': 15,
    'Smartphone': 32,
    'Tablet': 12,
    'Fones de ouvido': 45
}
nome, quantidade = produto_mais_vendido(vendas)
print(f"O produto mais vendido foi '{nome}' com {quantidade} vendas.")
```

Resultado esperado:

```
O produto mais vendido foi 'Fones de ouvido' com 45 vendas.
```

Questão 4.7: `alunos_aprovados`

Crie uma função que receba um dicionário de alunos e suas notas e retorne uma lista contendo apenas os nomes dos alunos aprovado (nota mínima 6.0).

```python
def alunos_aprovados(dados):
    # seu código aqui

alunos = {
    'Alice': 8.5,
    'Bruno': 5.2,
    'Carlos': 9.0,
    'Maria': 7.1,
    'Pedro': 5.8
}
aprovados = alunos_aprovados(alunos)
print(f"Aprovados: {aprovados}")
```

Resultado esperado:

```
Aprovados: ['Alice', 'Carlos', 'Maria']
```

Desafio:

Substitua a linha contendo o `print` para uma chamada a uma nova função (a ser desenvolvida por você) que recebe a lista de aprovados e não retorna nada, mas que imprima os nomes dos aprovados de forma mais legível, com apenas um nome um por linha. Resultado esperado:

```
Aprovados:
  Alice
  Carlos
  Maria
```

Questão 4.8: `media_movel`

Escreva uma função `media_movel(serie: list[float], n: int) -> list[float]` que recebe como argumentos uma lista numérica `serie` de tamanho arbitrário e um inteiro `n`, indicando o número de elementos da janela de cálculo da média móvel a ser computada na lista `serie`. A função deve retornar uma nova série, na forma de uma lista de `float` com o mesmo

tamanho de `serie`, contendo os valores calculados da média móvel. Ou seja, para elementos cujo índice é maior ou igual a `n`:

$$y_k = \frac{1}{n} \sum_{i=k-n+1}^{k} x_i \quad , \quad k \geq n \tag{4.4}$$

onde x é a série original de entrada e y a que contem as médias moveis.

Preencha os elementos da serie de retorno cujos índices são menores que `n` com `float('nan')`.

Por exemplo, para a lista `[ 1.0, 3.0, 2.0, 4.0, 6.0, 2.0, 1.0 ]` e `n=3`:

$$x = [\ \underbrace{1\quad 3\quad 2}_{y_3=\frac{1+3+2}{3}=2}\quad 4\quad 6\quad 2\quad 1\ ]$$

$$x = [\ 1\quad \underbrace{3\quad 2\quad 4}_{y_4=\frac{3+2+4}{3}=3}\quad 6\quad 2\quad 1\ ]$$

$$x = [\ 1\quad 3\quad \underbrace{2\quad 4\quad 6}_{y_5=\frac{2+4+6}{3}=4}\quad 2\quad 1\ ]$$

$$x = [\ 1\quad 3\quad 2\quad \underbrace{4\quad 6\quad 2}_{y_6=\frac{4+6+2}{3}=4}\quad 1\ ]$$

$$x = [\ 1\quad 3\quad 2\quad 4\quad \underbrace{6\quad 2\quad 1}_{y_7=\frac{6+2+1}{3}=3}\ ]$$

$$y = [\ \cdot\quad \cdot\quad 2\quad 3\quad 4\quad 4\quad 3\ ]$$

```python
def media_movel(serie: list[float], n: int) -> list[float]:
    # seu código aqui

print(media_movel([ 1.0, 3.0, 2.0, 4.0, 6.0, 2.0, 1.0 ], 3))
```

Resultado esperado:

```
[nan, nan, 2.0, 3.0, 4.0, 4.0, 3.0]
```

Questão 4.9: `ordena`

Escreva uma função `ordena(valores)`, que recebe uma lista de números inteiros de tamanho arbitrário e ordena seus elementos em ordem crescente. A função deve retornar duas listas: a primeira contendo os valores ordenados da lista original; a segunda contendo os índices dos elementos da lista original que correspondem à posição dos valores ordenados.

```python
def ordena(valores: list[int]) -> tuple[list[int], list[int]]:
    # seu código aqui

vord, ids = ordena([1, 4, 3, 5, 8, 2])
print(f"vord: {vord}\n ids: {ids}")
```

Resultado esperado:

```
vord: [1, 4, 3, 5, 8, 2]
 ids: [4, 3, 1, 2, 5, 0]
```

Não utilize a função `sort()` ou `sorted()` do Python. Implemente sua própria lógica de ordenação.

Depois de implementar seu própria algoritmo, utilize as soluções embutidas da linguagem.

Questão 4.10: `procura`

Escreva uma função `procura(lista, valor)`, que recebe uma lista de números de tamanho arbitrário e um valor a ser buscado dentro dessa lista. A função deve retornar uma nova lista contendo os índices das ocorrências do valor na lista original.

Se o valor não for encontrado, a função deve retornar uma lista vazia.

```
def procura(lista, valor):
    # seu código aqui

ids = procura([2, 4, 8, 3, 0, 3, 5, 7], 3)
print(ids)
```

Resultado esperado:

```
[3, 5]
```

Desafio:

Além de criar sua própria lógica para a busca, tente também resolver o problema utilizando o função embutida `filter()`. Para isso, você precisará aplicar uma função `lambda`, que não foi abordada neste tutorial. Pesquise sobre essa técnica e descubra como utilizá-la para filtrar os índices corretamente!

Questão 4.11: `tictactoe`

Escreva uma função `eval_move(board_state, next_move)` que processe a lógica do jogo tic-tac-toe (jogo da velha) e informa o resultado da partida.

O primeiro argumento recebido pela função (`board_state`) é uma matriz 3×3 (implementada na forma de `list[list[int]]`) que contém o estado atual do tabuleiro. Elementos da matriz contendo `0` indicam que aquela posição ainda não foi jogada. Posições da matriz contendo `1` indicam que ela foi tomada pelo jogador ✗, enquanto posições contendo `-1` foram tomadas pelo jogador *O*.

Valores diferentes de `-1`, `0` e `1` constituem um estado inválido do tabuleiro. Outras formas de estados inválidos também devem ser considerados. Por exemplo, muitas posições tomadas por um mesmo jogador ou o jogador *O*com mais posições tomadas que ✗(considerando que ✗sempre inicia jogando). Abaixo são apresentados alguns valores de `board_state` e o que cada um deles representa.

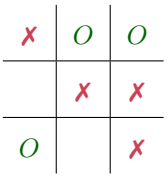
```
# valid state
board_state = [
    [ 1, -1,  0],
    [ 0,  1,  1],
    [-1,  0, -1]
]
```

✗	*O*	
	✗	✗
O		*O*

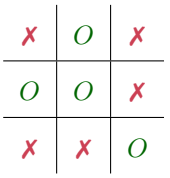
```
# invalid state
board_state = [
    [ 1, -1,  0],
    [ 0, -1,  1],
    [-1,  0, -1]
]
```

✗	*O*	
	O	✗
O		*O*

```
# player x wins!
board_state = [
    [ 1, -1, -1],
    [ 0,  1,  1],
    [-1,  0,  1]
]
```



```
# draw!
board_state = [
    [ 1, -1,  1],
    [-1, -1,  1],
    [ 1,  1, -1]
]
```



O segundo argumento recebido (`next_move`) é uma tupla de dois inteiros `(l,c)`, indicando respectivamente a linha e a coluna jogada pelo próximo jogador. Valores de `l` e `c` diferentes de `0`, `1` e `2` constituem uma tentativa de joga inválida. Também é uma jogada inválida qualquer posição `(l,c)` previamente ocupada.

A função deve inferir qual dos jogadores (✗ ou O) está realizando a jogada `next_move` a partir do estado do tabuleiro (`board_state`), garantindo a alternância entre os jogadores. O jogador ✗joga primeiro.

Uma vez validado o estado atual, o jogador da vez e a posição a ser jogada por este, a função `eval_move()` deve modificar o estado do tabuleiro executando a jogada solicitada (*side-effect function*). Se a jogada solicitada ou a posição prévia do tabuleiro for inválida, a função mantém o `board_state` inalterado.

Alem de modificar `board_state`, a função também retorna um valor (`outcome`), indicando se a jogada ou posição são inválidas, se houve um vencedor, se houve empate ou se o jogo ainda não terminou. O retorno `outcome` deve ser um inteiro que codifica essas situações da seguinte forma:

- `-4` se a posição do tabuleiro passado como argumento for ilegal;
- `-2` se a próxima jogada for ilegal;
- `0` se a jogada foi executada com sucesso, mas ainda não houve um ganhador;
- `1` se a jogada foi executada com sucesso e o jogador ✗venceu;
- `2` se a jogada foi executada com sucesso e o jogador Ovenceu;
- `3` se a jogada foi executada com sucesso, mas o jogou terminou em empate.

Complete:

```
def eval_move(board_state: list[list[int]], next_move: tuple[int, int]) -> int:
    #seu código aqui

board_state = [
    [ 1, -1,  0],
    [ 0,  0,  1],
    [-1,  0,  0]
]
outcome = eval_move(board_state, (1, 1))
print(f"{board_state = }, {outcome = }")

outcome = eval_move(board_state, (2, 2))
print(f"{board_state = }, {outcome = }")

outcome = eval_move(board_state, (0, 1))
print(f"{board_state = }, {outcome = }")

outcome = eval_move(board_state, (1, 0))
print(f"{board_state = }, {outcome = }")
```

Resultado esperado:

```
board_state = [[1, -1, 0], [0, 1, 1], [-1, 0, 0]], outcome = 0
board_state = [[1, -1, 0], [0, 1, 1], [-1, 0, -1]], outcome = 0
board_state = [[1, -1, 0], [0, 1, 1], [-1, 0, -1]], outcome = -2
board_state = [[1, -1, 0], [1, 1, 1], [-1, 0, -1]], outcome = 1
```

Note que a função deve realizar várias ações:

- Validação do estado atual do tabuleiro;
- Determinação do próximo jogador;
- Verificação da próxima jogada;
- Aplicação da jogada no tabuleiro;
- Avaliação do resultado (vitória, empate ou jogo em andamento).

Convém quebra a implementação da função `eval_move` em funções menores, dedicadas a cada uma dessa sub-tarefas (nomeie essas funções com nomes significativos).

Desafio:

Pesquise sobre `Enums` e incorpore-os na representação de `outcome` para deixar o código mais legível e organizado.

Questão 4.12: `tictactoe` (continuação)

Em `https://github.com/j-Lago/pyTicTacToe` você encontrá o repositório de um código python para uma interface gráfica do jogo da questão anterior. Ao executar `main.py`, o programa abrirá uma janela gráfica com um tabuleiro tic-tac-toe vazio. Ao clicar com o botão esquerdo do mouse em uma das posições do tabuleiro, o programa chamará a função `eval_move()` do arquivo `eval_move.py` (o arquivo existe, mas a implementação da função está faltando).

Edite o arquivo `eval_move.py`, incluindo a sua implementação para a função desenvolvida na questão anterior, execute `main.py` e verifique se sua função realmente implementa a lógica do jogo.

Capítulo 5

Classes

Além dos tipos primitivos e coleções introduzidos no Capítulo 1, podemos ainda criar nossos próprios tipos personalizados.

Uma classe descreve um tipo composto que reúne, em um único objeto, um conjunto coerente de variáveis (atributos) e as funções que operam sobre elas (métodos), para a manipulação desses dados.

Embora as classes sejam um dos pilares da Programação Orientada a Objetos (OOP), viabilizando conceitos como encapsulamento, abstração, herança e polimorfismo, este texto introdutório não se propõe a discutir, muito menos advogar por OOP como abordagem universal.

Ainda assim, o uso de classes pode melhorar significativamente a organização do código, mesmo sem aderir estritamente aos princípios da OOP.

Além disso, compreender a estrutura básica de classes em Python é essencial, especialmente para entender e utilizar pacotes especializados, que serão explorados em capítulos futuros.

Nada melhor do que demonstrar na prática as vantagens do uso de classes. Para isso, vamos revisitar um dos exemplos apresentados no Capítulo 3, onde criamos uma função para buscar o contato de um usuário a partir de seu cadastro.

O código original é replicado abaixo:

```
def obter_contato(cadastro: dict) -> str:
    if 'email' in cadastro and cadastro['email'] != '':
        return cadastro['email']
    elif 'telefone' in cadastro and cadastro['telefone'] != '':
        return cadastro['telefone']
    else:
        return 'nenhum contato cadastrado'

usuario1 = {'nome': 'Fulano', 'email': 'fulano@ifsc.edu.br', 'telefone': '(48)99999-9999'}
usuario2 = {'nome': 'Beltrano', 'telefone': '(48)99999-8888'}
usuario3 = {'nome': 'Sicrano', 'telefone': ''}

print('contato usuario1:', obter_contato(usuario1))
print('contato usuario2:', obter_contato(usuario2))
print('contato usuario3:', obter_contato(usuario3))
```

```
contato usuario1: fulano@ifsc.edu.br
contato usuario2: (48)99999-8888
contato usuario3: nenhum contato cadastrado
```

Note que a função `obter_contato()` possui um único parâmetro `cadastro`, que é um dicionário com algumas chaves esperadas: `'nome'`, `'telefone'` e `'email'`.

O uso de um dicionário, nesse contexto, foi a forma encontrada para agrupar as informações relacionadas a um mesmo indivíduo, evitando que tais dados fiquem soltos, espalhados em variáveis independentes. Como os valores associados às chaves `'nome'`, `'telefone'` e `'email'` estão logicamente vinculados, faz sentido organizá-los sob uma mesma estrutura.

Essa abordagem apresenta fragilidades e limitações, uma vez que a função `obter_contato()` depende da existência de chaves específicas no dicionário fornecido para funcionar corretamente. O código não prevê mecanismos estruturados para lidar com dados inconsistentes ou ausentes, delegando ao chamador a responsabilidade de assegurar a organização e a completude do dicionário fornecido. O chamador, de certa forma, precisa conhecer aspectos internos da função.

Uma forma mais estruturada de lidar com essas questões é utilizar classes, que encapsulam dados (`nome`, `telefone`, `email`) e funcionalidade (`obter_contato()`) em um único objeto.

Poderíamos refatorar o programa criando um novo tipo personalizado chamado `Usuario`:

```
1  # definição da classe
2  class Usuario:
3      # o método __init__ é executado ao criar um novo objeto
4      def __init__(self, nome: str, telefone: str = '', email: str = ''):
5          self.nome = nome
6          self.telefone = telefone
7          self.email = email
8
9      def contato(self) -> str:
10         if self.email:
11             return self.email
12         elif self.telefone:
13             return self.telefone
14         else:
15             return 'nenhum contato cadastrado'
16
17 # criando instâncias da classe Usuario
18 usuario1 = Usuario(nome='Fulano', email='fulano@ifsc.edu.br', telefone='(48)99999-9999')
19 usuario2 = Usuario(nome='Beltrano', telefone='(48)99999-8888')
20 usuario3 = Usuario(nome='Sicrano')
21
22 # utilizando o método contato() e apresentando os resultados
23 print('contato usuario1:', usuario1.contato())
24 print('contato usuario2:', usuario2.contato())
25 print('contato usuario3:', usuario3.contato())
```

```
contato usuario1: fulano@ifsc.edu.br
contato usuario2: (48)99999-8888
contato usuario3: nenhum contato cadastrado
```

Vamos explorar as partes fundamentais desse código linha por linha.

Na **linha 2** temos a declaração que estamos definindo uma nova `class` chamada `Usuario`[1].

Essa definição consiste em dois métodos, ou duas funções associadas à classe: `__init__()` (**linha 4**) e `contato()` (**linha 9**).

O método `__init__()` é o construtor da classe, que é executado automaticamente ao criarmos um novo objeto do tipo `Usuario` (o que é feito na **linha 18**). Esse processo é conhecido como instanciação da classe, e resulta na criação de um objeto em memória. Enquanto a classe é

[1] Diferente da convenção utilizada para funções e variáveis, que geralmente seguem o estilo *snake_case* (`exemplo_variavel`), nomes de classes costumam ser escritos em *CamelCase* (`NomeDaClasse`).

apenas uma definição da estrutura e do comportamento dos objetos (um *blueprint*), a instância representa a consolidação de um objeto desse tipo em memória, com valores próprios.

Assim como funções, os métodos podem ter parâmetros que podem ser obrigatórios ou opcionais (recebendo valores padrão). No caso de `__init__()`, os parâmetros são `self`, `nome`, `email` e `telefone`.

A principal diferença desse método quando comparado a uma função padrão, é a presença do parâmetro `self`, que é uma referência à própria instância do objeto[2]. Não precisamos (nem devemos) fornecer `self` manualmente a um método, já que o Python faz isso de forma automática e disponibiliza a referência `self` aos métodos da classe justamente para que estes possam manipular os dados relativos à instância.

Na **linha 5**, criamos o atributo `self.nome`, atribuindo a ele o valor do parâmetro `nome`.

Um atributo é uma variável interna da instância da classe, responsável por armazenar informações específicas de cada objeto criado. São esses atributos que compõem o objeto.

Embora o atributo e o parâmetro tenham o mesmo nome (o que é uma prática comum, mas não necessária), eles são variáveis distintas. O parâmetro `nome` é passado ao método `__init__()` no momento da criação de um novo objeto da classe `Usuario`, e seu valor é armazenado internamente no atributo `self.nome`, garantindo que essa informação fique acessível ao longo da vida útil do objeto.

Além de `nome`, são criados mais dois atributos para armazenar `email` e `telefone`. Esses três atributos formam a estrutura de dados armazenados em memória para um objeto da classe `Usuario`.

Pulando para a **linha 18**, que na realidade é a primeira a ser executada (já que as anteriores pertencem apenas à definição da classe `Usuario`), temos a criação da variável `usuario1`.

À direita do operador de atribuição (`=`) há uma chamada à classe `Usuario()`, o que significa que um novo objeto do tipo `Usuario` está sendo instanciado.

Ao chamar `Usuario(nome='Fulano', email=...)`, os argumentos fornecidos (`nome`, `email` e `telefone`) são passados para o método `__init__()`, que então atribui esses valores aos atributos internos do objeto. Isso resulta na criação de uma instância única, armazenada na variável `usuario1`, contendo todas as informações fornecidas.

Nas **linhas 19** e **20**, criamos mais duas instâncias da classe `Usuario`, com valores distintos. Cada instância da classe representa um objeto único na memoria, contendo suas próprias informações de `nome`, `telefone` e `email`, mas possuem a mesma estrutura definida pela classe.

Na **linha 23**, dentro do comando `print()`, temos a chamada `usuario1.contato()`, que executa o método `contato()` definido na **linha 9**.

Um detalhe importante é que, na definição de `contato(self)`, declaramos um parâmetro `self`, mas não o fornecemos explicitamente na chamada do método.

Isso ocorre porque o Python passa `self` automaticamente, garantindo que o método seja

[2] O nome `self` é apenas uma convenção, e sua função de autoreferência à instância do objeto está diretamente relacionada à posição do parâmetro, que deve sempre ser o primeiro em qualquer método. Independentemente do nome utilizado, esse primeiro parâmetro representa o próprio objeto, permitindo o acesso e a manipulação de seus atributos e métodos. Contudo, usar `self` é altamente recomendado, pois é uma prática amplamente reconhecida e melhora a legibilidade do código.

chamado sobre a instância correta. Ou seja, a chamada `usuario1.contato()` equivale internamente a `contato(usuario1)`, onde `self` recebe `usuario1` como argumento.

Note que o encapsulamento dos dados e funcionalidades contribui para a organização e legibilidade do código. As instâncias de classes geralmente são objetos mutáveis, como os abordados no Capítulo 1. Na realidade, em Python, todos os tipos de dados são implementados como classes, até mesmo os tipos primitivos como inteiros.

Diferente de outras linguagens como Java, C++, C# e Rust, onde existe um sistema explícito de controle de acesso (como `private`, `protected` e `public`), em Python todos os membros de uma classe são públicos por padrão.

Isso significa que qualquer parte do código que tenha acesso a uma instância da classe pode ler e modificar diretamente seus atributos, além de chamar qualquer método definido na classe.

Embora Python não imponha um mecanismo formal de privacidade, existe uma convenção para indicar atributos internos[3].

O exemplo a seguir demostra esse acesso a atributos:

```
class Usuario:
    def __init__(self, nome: str, telefone: str = '', email: str = ''):
        self.nome = nome
        self.telefone = telefone
        self.email = email

usuario = Usuario(nome='Fulano', email='fulano@ifsc.edu.br', telefone='(48)99999-9999')
print(f"{usuario.nome}, {usuario.email}, {usuario.telefone}")   # lendo atributos

usuario.email = 'fulano_de_tal@ifsc.edu.br'   # modificando atributo email
usuario.telefone = '(11)88888-9999'           # modificando atributo telefone
print(f"{usuario.nome}, {usuario.email}, {usuario.telefone}")    # lendo atributos
```

```
Fulano, fulano@ifsc.edu.br, (48)99999-9999
Fulano, fulano_de_tal@ifsc.edu.br, (11)88888-9999
```

As classes em Python oferecem diversas funcionalidades avançadas, como herança, sobrecarga de operadores, métodos mágicos, polimorfismo, abstração, propriedades e metaclasses, que ampliam o suporte à programação orientada a objetos.

Embora sejam conceitos importantes, este tutorial introdutório foca nos princípios básicos. O aprofundamento nesses temas fica a critério do leitor, que poderá explorá-los de forma gradativa, em um momento oportuno, conforme sua necessidade e evolução no aprendizado.

Para praticar os conceitos de classes, refatore o código da questão `tictactoe`, transformando o parâmetro `board_state` em um atributo e a função `eval_move()` método da classe `TicTacTtoe`.

[3] Convencionou-se utilizar um underscore (`_`) antes do nome do atributo para indicar que ele não deve ser acessado diretamente (ex: `self._informacao_privada`). Apesar de ainda ser possível modificar esses atributos externamente, essa prática sinaliza que o atributo é de uso interno, recomendando que sua manipulação ocorra apenas dentro da própria classe por seus métodos. IDEs e ferramentas de desenvolvimento costumam alertar sobre tentativas de acesso direto a esses atributos, reforçando essa boa prática.

Capítulo 6

NumPy e Matplotlib

A lista (`list`) em Python é uma estrutura de dados altamente versátil: cada elemento pode conter qualquer tipo de objeto, independentemente do tamanho ou tipo, e a estrutura pode ser modificada e redimensionada dinamicamente. No entanto, toda essa generalidade e flexibilidade tem um custo, especialmente em termos de desempenho.

Diferente de outras linguagens, o Python não possui um tipo embutido para representar `arrays` no sentido tradicional, ou seja, coleções de dados homogêneos com tamanho fixo. Esse tipo de estrutura, embora bem mais restrita que uma lista, permite otimizações tanto no uso de memória quanto na performance.

Isso torna-se particularmente crítico quando lidamos com grandes volumes de dados numéricos em aplicações científicas, como regressão, otimização, álgebra linear os demais métodos utilizados em computação científica. Nessas situações, a eficiência no processamento é essencial, exigindo estruturas de dados mais especializadas e performáticas que as listas tradicionais do Python.

A solução mais amplamente adotada para esse desafio em Python é o uso da biblioteca `NumPy`, que, embora não venha incluída por padrão na instalação da linguagem, consolidou-se como o padrão de fato no meio científico para computação numérica. Complementarmente, a biblioteca `Matplotlib` é amplamente utilizada para visualização de dados através da geração de gráficos (`plots`).

Como `NumPy` e `Matplotlib` não fazem parte da biblioteca padrão, é necessário instalá-los manualmente. Isso pode ser feito utilizando o `pip`, o gerenciador de pacotes oficial do Python. Para realizar a instalação, basta executar o seguinte comando no terminal:

```
C:\curso_python\> pip install numpy matplotlib
```

A biblioteca `NumPy` fornece a estrutura de dados `ndarray`, que permite representar `arrays` multidimensionais de forma compacta e eficiente. Ela também disponibiliza um vasto conjunto de funções vetorizadas otimizadas, capazes de aplicar operações matemáticas diretamente sobre todo o `array`, dispensando o uso de laços explícitos que podem ser pouco performáticos em Python. Essa abordagem vetorizada resulta em cálculos muito mais rápidos, especialmente ao lidar com grandes volumes de dados.

Uma das aplicações fundamentais da `NumPy` é na representação e manipulação de matrizes. O exemplo a seguir ilustra esse uso:

```
1  import numpy as np
2
3  A = np.array([[2, 3],    # define matriz 2 x 2
4                [-4, 1]])
5  x = np.array([[3],       # define matrix 2 x 1
6                [1]])
7  b = np.array([[-3],      # define matrix 2 x 2
8                [8]])
9
10 y = A @ x + b    # operação de multiplicação e soma matricial
11 print(y)
```

```
[[ 6]
 [-3]]
```

Na **linha 1**, declaramos que utilizaremos o pacote `numpy`, e o faremos por meio do *alias* `np`. Ou seja, para não repetirmos `numpy` diversas vezes ao longo do código, atribuímos a ele o nome abreviado `np`.

Na **linha 3**, criamos a matriz **A** de dimensão 2×2 utilizando `array`. Já nas **linhas 5** e **7**, definimos os vetores coluna **x** e **b**, ambos com dimensão 2×1.

E na **linha 10** realizamos a operação matricial $\mathbf{y} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$. Observe que o operador `@` realiza o produto matricial, conforme definido na álgebra linear. Já o operador `*` também pode ser utilizado com arrays `NumPy`, mas nesse caso representa a multiplicação elemento a elemento, e não a multiplicação de matrizes.

Por fim, na **linha 10**, realizamos a operação matricial $\mathbf{y} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b}$. Note que o operador `@` realiza o produto matricial, conforme definido na álgebra linear. Já o operador `*` também pode ser usado com `arrays` do `NumPy`, mas nesse caso ele representa a multiplicação elemento a elemento, e não o produto de matrizes.

Além das operações matriciais, o `NumPy` também se destaca ao lidar com sequências numéricas unidimensionais, graças à sua poderosa implementação de vetorização — que simplifica tanto a escrita do código quanto sua performance.

No exemplo a seguir, usamos um `array` para representar uma sequência numérica e, com isso, podemos usar vetorização para aplicar uma expressão matemática a cada elemento dessa sequência:

```
1  import numpy as np
2
3  x = np.linspace(0, 3, 7)
4  y = x**2 - 2*x
5
6  print(f"{x = }")
7  print(f"{y = }")
```

```
x = array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. ])
y = array([ 0.  , -0.75, -1.  , -0.75,  0.  ,  1.25,  3.  ])
```

Na **linha 3**, utilizamos o método `linspace` da biblioteca `numpy` para criar um `array` unidimensional (um vetor linha) contendo `7` valores do tipo `float`, igualmente espaçados no intervalo fechado de `0` a `3` — ou seja, incluindo os dois extremos do intervalo.

Na **linha 4**, exploramos as funcionalidades de vetorização da biblioteca `NumPy` para aplicar a expressão matemática diretamente a todos os elementos de `x`. O resultado é um novo `array y`, de mesmo tamanho, cujos valores correspondem à aplicação da expressão $y_n = x_n^2 - 2x_n$ em cada elemento x_n do `array x`, sem a necessidade de laços explícitos.

Nos exemplos anteriores, usamos `np.array([...])` para criar um `array` a partir de uma lista e `np.linspace()` para gerar valores igualmente espaçados dentro de um intervalo. Além dessas abordagens, a biblioteca `NumPy` oferece outras funções bastante úteis para a criação de `arrays`, como:

`np.zeros(shape)` – cria um `array` preenchido com zeros.

`np.ones(shape)` – cria um `array` preenchido com uns.

`np.full(shape, valor)` – cria um `array` preenchido com um valor específico.

`np.eye(n)` – gera a `matriz` identidade de ordem `n`.

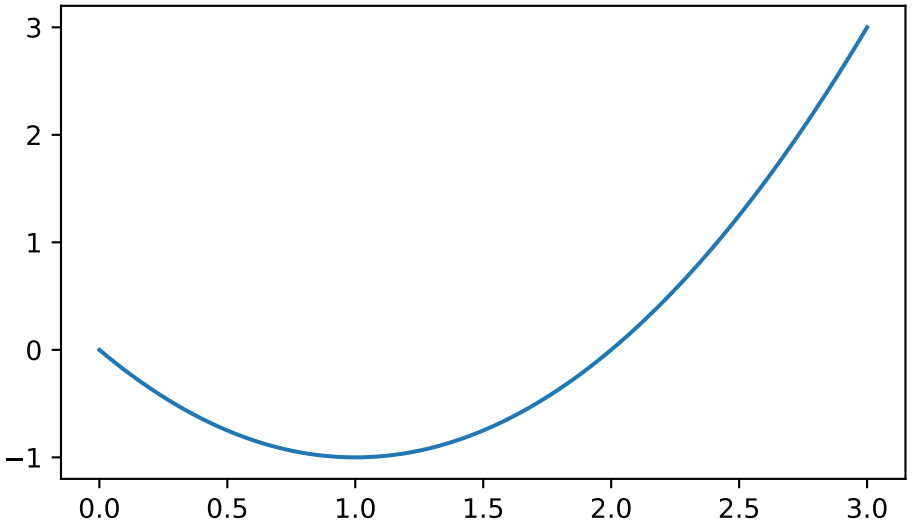
`np.arange(início, fim, passo)` – semelhante à função `range`, mas para `array`.

`np.random.rand(shape)` – cria um `array` com valores aleatórios no intervalo `[0, 1)`.

Outra biblioteca particularmente útil em aplicações de computação científica é a `Matplotlib`. Com uma pequena modificação no exemplo anterior, podemos utilizá-la para gerar um gráfico que representa os vetores `x` e `y` no plano cartesiano.

```
import numpy as np
import matplotlib.pyplot as plt

# gera valores para os eixos x e y do gráfico
x = np.linspace(0, 3, 100)
y = x**2 - 2*x

# plota o gráfico
plt.plot(x,y)
plt.show()
```

Esse código gera e exibe a figura:

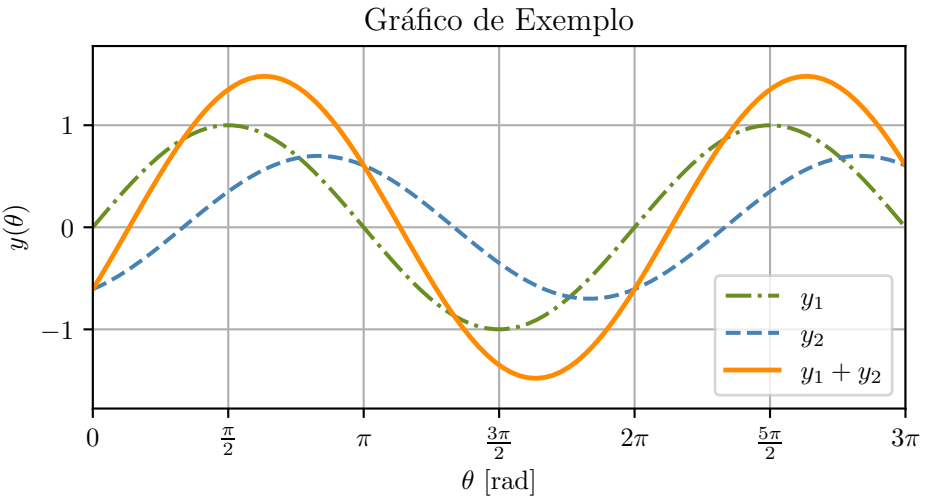


A figura gerada exibe o gráfico construído a partir dos pares ordenados (x_n, y_n). Embora seja uma visualização simples, a biblioteca `matplotlib` oferece uma ampla variedade de funções auxiliares para personalizar o aspecto visual do gráfico.

O exemplo a seguir demostra algumas dessas funcionalidades:

```python
import numpy as np
import matplotlib.pyplot as plt

# gera pontos para os eixo x e y
theta = np.linspace(0, 3*np.pi, 100)
y1 = np.sin(theta)
y2 = 0.7 * np.sin(theta - np.pi/3)
y3 = y1 + y2

# descomete caso queira fidelidade de fontes com Latex (necessário instalar Latex)
#plt.rcParams.update({'text.usetex': True, 'font.family': 'serif'})

plt.figure(figsize=(5, 2.8))    # define o latgura e altura da figura em polegadas
plt.plot(theta, y1, color='olivedrab' , linestyle='-.', label=r'$y_1$')     # plota y1
plt.plot(theta, y2, color='steelblue' , linestyle='--', label=r'$y_2$')     # plota y2
plt.plot(theta, y3, color='darkorange', linewidth=1.8 , label=r'$y_1+y_2$') # plota y3
plt.title('Gráfico de Exemplo')       # adiciona título ao gráfico
plt.xlabel(r'$\theta$ [rad]')         # adiciona identificação do eixo x
plt.ylabel(r'$y(\theta)$')            # adiciona identificação do eixo y
plt.xlim(theta[0], theta[-1])         # define os limites do eixo x
plt.ylim(1.2*y3.min(), 1.2*y3.max())  # define os limites do eixo y
plt.grid(True)       # exibe grade
plt.legend()         # exibe legenda (valores atribuidos a plabel no plot)
plt.tight_layout()   # ajusta o grafico gerado aos limites da figura

# define os ticks do eixo x manualmente como múltiplos de π
xticks = [0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi, 5*np.pi/2, 3*np.pi]
xtick_labels = [r'0', r'$\frac{\pi}{2}$', r'$\pi$', r'$\frac{3\pi}{2}$',
                r'$2\pi$', r'$\frac{5\pi}{2}$', r'$3\pi$']
plt.xticks(xticks, xtick_labels)

# exporta a figura gerada para um arquivo externo do tipo .pdf
plt.savefig(r"seno.pdf", format="pdf")

# exibe a figura na tela
plt.show()
```

Saída do programa:



Mais informações sobre os recursos de Matplotlib podem ser encontradas na documentação oficial da biblioteca, disponível em: https://matplotlib.org/stable/contents.html.

Assim como a na biblioteca Matplotlib, os exemplos apresentados acima têm como objetivo apenas introduzir a NumPy. Ela oferece uma ampla gama de funcionalidades não abordadas, incluindo muitas outras operações vetoriais, manipulação de arrays multidimensionais, transformações estruturais, mapeamento e interpolação de valores, entre muitos outros recursos voltados à computação numérica. Recomenda-se que o leitor explore essas funciona-

lidades conforme sua necessidade. A documentação oficial da biblioteca está disponível em: https://numpy.org/doc.

Além disso, recomenda-se ao leitor explorar a biblioteca SciPy, que complementa e expande as funcionalidades do NumPy ao disponibilizar ferramentas especializadas para áreas como álgebra linear, equações diferenciais, integração numérica, otimização, interpolação, transformadas e processamento de sinais. A documentação oficial da biblioteca está disponível em: https://docs.scipy.org/doc/scipy/.

Capítulo 7

Exercícios

Diferente da lista de exercícios anterior, que tinha como objetivo desenvolver o raciocínio lógico e familiarizar o leitor com os mecanismos básicos da linguagem Python, esta nova lista propõe desafios mais complexos, voltados para situações comuns no cotidiano profissional de engenheiros.

O foco agora é estimular a capacidade de resolver problemas e de traduzi-los para o ambiente computacional, utilizando o Python como ferramenta de apoio.

Questão 7.1: `pll`

Em sistemas de geração fotovoltaica, a energia solar captada pelos painéis é convertida em energia elétrica em corrente contínua. Como a maioria desses sistemas não possui meios de armazenamento, e como energia se conserva, toda a energia gerada pelos painéis deve ser imediatamente injetada na rede elétrica.

Para escoar essa produção, os inversores devem não apenas executar a conversão CC–CA, mas fazê–la de forma a controlar a injeção de potência na rede, garantindo o equilíbrio energético e consequentemente a estabilidade do barramento CC.

Para isso, a tensão imposta pelo inversor deve estar em sincronismo com a da rede. O controle da potência ativa depende diretamente da defasagem entre a tensão da rede e a tensão imposta pelo inversor. Além disso, como a rede elétrica opera com frequência variável em torno dos 60 Hz, o inversor deve estimar continuamente a fase da tensão da rede a fim de realizar os ajustes necessários para manter o sincronismo e o ângulo de carga desejado.

Essa estimação não é trivial já que, além das variações da tensão e frequência em regime permanente, a rede ainda apresenta efeitos transitórios impulsivos e oscilatórios, distorções harmônicas, flutuações, ruídos, afundamentos, etc. Isso é ainda mais desafiador em sistemas monofásicos, onde amplitude, frequência e fase devem ser obtidas a partir de medições de um único sinal de tensão, sujeito aos fenômenos já citados.

Na prática, existem algumas estratégias para se obter uma estimativa instantânea e precisa da fase da tensão da rede, sendo que a maioria delas é baseada na estrutura SOGI–PLL, cujo diagrama de blocos básico é apresentado na Figura 7.1.

O estimador da Figura 7.1 pode ser discretizado e reescrito na forma de equações a diferen-

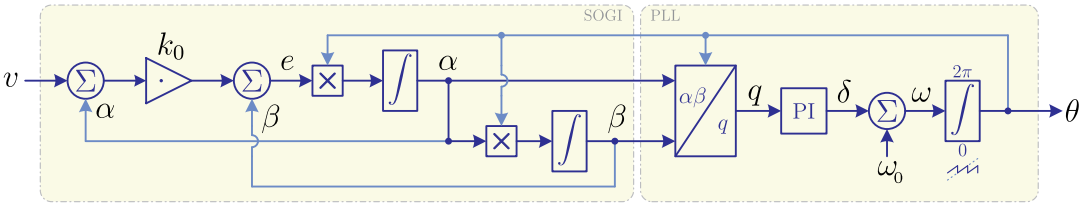


Figura 7.1: SOGI–PLL.

ças, apresentadas nas equações (7.1–7.8), possibilitando sua implementação em controladores digitais embarcados.

O algoritmo discretizado recebe um sinal amostrado da tensão de entrada — tipicamente com frequência de amostragem na ordem de algumas dezenas de kHz, dependendo da frequência de comutação utilizada no inversor.

Com base nessas amostras, estima-se a fase da tensão quase instantânea (com atraso de uma amostra). O sinal de fase estimado é utilizado nas malhas de controle das correntes e potências ativa e reativa injetadas na rede (não abordados nessa questão).

$$e_n = k_0\,(v_n - \alpha_{n-1}) - \beta_{n-1} \tag{7.1}$$

$$\alpha_n = w_{n-1}\,k_1\,(e_n + e_{n-1}) + \alpha_{n-1} \tag{7.2}$$

$$\beta_n = w_{n-1}\,k_1\,(\alpha_n + \alpha_{n-1}) + \beta_{n-1} \tag{7.3}$$

$$d_n = \alpha_n \sin(\theta_{n-1}) - \beta_n \cos(\theta_{n-1}) \tag{7.4}$$

$$q_n = \alpha_n \cos(\theta_{n-1}) + \beta_n \sin(\theta_{n-1}) \tag{7.5}$$

$$\delta_n = k_2 q_n + k_3 q_{n-1} + \delta_{n-1} \tag{7.6}$$

$$\omega_n = \omega_0 + \delta_n \tag{7.7}$$

$$\theta_n = k_1\,(\omega_n + \omega_{n-1}) + \theta_{n-1} \pmod{2\pi} \tag{7.8}$$

A estimativa de fase da rede (*theta*) obtida ainda deve ser validada. O `pll` pode demorar alguns ciclos para se sincronizar com a rede (ou atracar).

Tarefa 1

Em `https://github.com/j-Lago/NVP68300` você encontrará um arquivo `rede.csv` com a tensão (**coluna 2**) amostrada de alguns ciclos da rede elétrica e um registro temporal (**coluna 1**) de cada amostra.

Escreva um script que importe esse sinal para um `array`, e para cada amostra (cada elemento do `array`) execute os cálculos (7.1–7.8). Guarde os resultados de cada amostra de v (tensão de entrada), α (tensão filtrada), d (componente de eixo direto da tensão em base síncrona), q (componente em quadratura) e θ (fase estimada pelo PLL).

Assuma ainda, de forma simplificada, que o sincronismo foi obtido no instante do primeiro cruzamento por zero do sinal θ (saída do PLL) em que $\arctan(q/d) < 0.02$ rad ($\approx 2°$) e $d > 250$ V. Identifique esse instante bem como os demais cruzamentos por zero subsequentes.

Utilize as seguintes constantes para a implementação do controlador e das integrações:

$$\begin{aligned} k_0 &= 0.7 \\ k_1 &= 2.84 \cdot 10^{-6} \\ k_2 &= 0.385715277950311 \\ k_3 &= -0.385713293478261 \\ \omega_0 &= 376.9911184307752 \end{aligned} \tag{7.9}$$

Gere e exporte uma figura apresentado a evolução temporal de cada uma desses sinais, bem como alguma indicação do instante que o PLL conseguiu se sincronizar (atracar) com a rede elétrica. A Figura 7.2 mostra um exemplo de como os resultados do script podem ser apresentados.

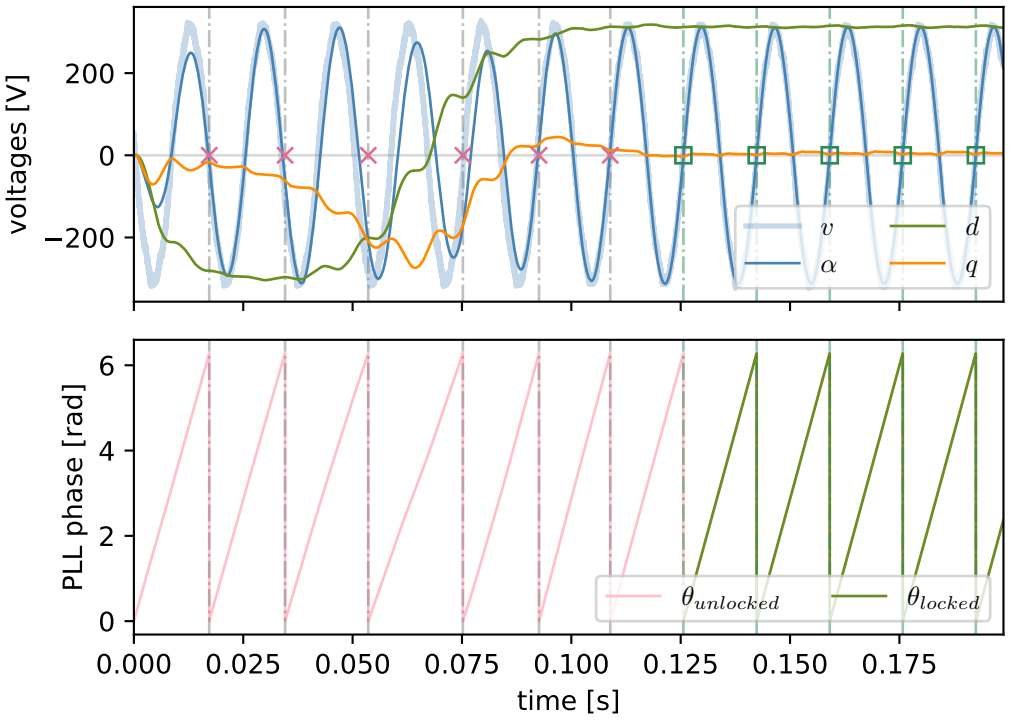


Figura 7.2: Resultado esperado do script.

Tarefa 2

Exporte para um arquivo `.csv` a porção do sinal de tensão de entrada compreendida entre a detecção inicial da sincronização até a amostra que antecede o último cruzamento por zero do ângulo θ, ou seja, o período que compreende todos os ciclos inteiros em que o PLL estava sincronizado. Exporte além da tensão, o período correspondente do vetor de tempo e do ângulo de saída do PLL.

amostras_sincronizadas.csv

```
time,    phase,       voltage
0.12562, 0.001584790, 4.3073
0.12562, 0.003702030, 14.323
0.12563, 0.005819309, -0.32799
0.12564, 0.007936621, 14.62
0.12564, 0.010053976, 14.767
    :         :          :
0.19247, 6.280264753, 10.071
```

Tarefa 3

Os dados exportados na tarefa anterior consistem num intervalo da tensão da rede com ciclos inteiros. Isso nos permite calcular o valor rms da tensão amostrada da rede através de (7.10):

$$V_{\text{rms}} = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} v_n^2} \tag{7.10}$$

Escreva uma função `rms(path: str) -> tuple[float, float]` que recebe como argumento um texto indicando o caminho do arquivo exportado na questão anterior e que calcule e retorne o valor rms da tensão e sua frequência fundamental média.

```python
def rms(path: str) -> tuple[float, float]:
    # seu código aqui

vrms, f1 = rms('amostras_sincronizadas.csv')
print(f"A amostra de tensão possui valor rms de {vrms:.2f} V e frequência {f1:.2f} Hz")
```

Saída esperada:

```
A amostra de tensão possui valor rms de 220.93 V e frequência 59.84 Hz
```

† *Note que (7.10) só é válida para um vetor* $\mathbf{x}$ *contendo ciclos inteiros da componente fundamental. Ainda, para melhorar a precisão do método, recomenda-se realizar o cálculo considerando ao menos dez ciclos completos da fundamental. (o que não é possível nesse exemplo pelo tamanho da amostra inicial)*