
Aprendendo Python: Conceitos Intermediários

Asimov Academy

ASIMOV

Conteúdo

01. Introdução ao curso	6
Motivação do curso	6
Estrutura do curso	6
02. Conjuntos e seus métodos	7
Propriedades de conjuntos	8
Valores únicos	8
Sem ordem	8
Elementos imutáveis	8
Operações de conjuntos	9
União	9
Intersecção	9
Diferença	10
Diferença simétrica	10
03. Compreensão de lista	11
A estrutura de uma compreensão de lista	11
Criando a compreensão de lista	11
Por que usar compreensão de lista?	12
04. Compreensão de dicionários e conjuntos	13
Compreensão de dicionários	13
Compreensão de conjuntos	13
05. Desafio – Conjuntos e compreensão de lista	14
Desafios	14
Desafio 01	14
Desafio 02	14
Desafio 03	14
Soluções	15
Solução do desafio 01	15
Solução do desafio 02	15
Solução do desafio 03	15
06. O bloco with	16
Exemplos de uso do bloco with	16

07. O bloco match case	18
match case em padrões estruturais	18
08. Operadores de divisão	20
O operador piso	20
O operador módulo	20
Usos para os operadores de divisão	21
09. Expressão condicional	22
Operadores ternários	22
Expressões condicionais: operadores ternários em Python	22
Usando expressões condicionais	23
10. Identidade e o operador is	24
O operador is	24
Identidade dos valores True, False e None	25
11. O operador “morsa” (expressão de atribuição)	26
Outros exemplos de uso do operador morsa	26
Vale a pena usar?	27
12. Desafio – Operadores	28
Desafios	28
Desafio 01	28
Desafio 02	28
Soluções	28
Solução do desafio 01	28
Solução do desafio 02	29
13. Funções de iteração	30
A função enumerate	30
A função sorted	30
A função reversed	31
A função zip	31
14. Desempacotando sequências	33
Desempacotamento aninhado	33
Desempacotamento com o operador *	34

15. Iteração avançada com itertools	36
itertools.chain	36
itertools.zip_longest	36
itertools.product	37
itertools.combinations	37
itertools.cycle	37
16. Desafio – Iteração	39
Desafios	39
Desafio 01	39
Desafio 02	39
Soluções	39
Desafio 01	39
Desafio 02	40
17. Argumentos arbitrários com *args e **kwargs	43
Funções com número arbitrário de argumentos	43
*args e **kwargs	43
Desmembrando argumentos na chamada de funções	44
Adaptando o desafio para qualquer número de listas	45
18. Funções built-in relevantes	46
Função isinstance	46
Funções any e all	46
map	47
filter	48
19. Funções lambda (anônimas)	49
O que é uma função lambda?	49
Para quê usar uma função lambda?	49
Usando funções anônimas em ordenamento	50
20. Decoradores	52
Funções são objetos em Python	52
Usando decoradores	53
Anexo: script meus_decoradores.py	54
21. Como decoradores funcionam?	56
Criando um decorador	56
Ajustando o número de argumentos	57

Mudando o nome da variável <code>meu_pacote</code> para o nome da função original	58
“Açúcar sintático”: o uso da sintaxe <code>@decorador</code>	59
22. Utilidades do módulo <code>functools</code>	60
<code>functools.cache</code>	60
<code>functools.partial</code>	60
23. Desafio – Funções	62
Desafios	62
Desafio 01	62
Desafio 02	62
Soluções	62
Desafio 01	62
Desafio 02	63
24. Interação com arquivos e sistema operacional	64
Módulo <code>os</code>	64
Módulo <code>pathlib</code>	64
Módulo <code>shutil</code>	65
Quer mais informações sobre arquivos e caminhos?	65
25. Trabalhando com tempo, datas e horas	66
Criando e manipulando datas com <code>datetime</code>	66
Criando data e hora	66
Lendo e formatando datas de a partir de texto	67
Aritmética com datas	68
Módulo <code>zoneinfo</code> : lidando com fusos horários	68
Usando <code>zoneinfo</code> em Windows	69
26. Manipulação e busca em textos	70
Módulo <code>locale</code>	70
Expressões regulares com o módulo <code>re</code>	70
O que é uma expressão regular?	71
O objeto <code>Match</code>	71
Criando expressões regulares com metacaracteres	72
Buscando pelos CPFs	73
27. Desafio – Biblioteca padrão	75
Desafios	75
Desafio 01	75

Desafio 02	75
Soluções	75
Desafio 01	75
Desafio 02	76
28. Os erros mais comuns em Python	78
1. IndentationError	78
2. SyntaxError	78
3. NameError	79
4. TypeError	79
5. ValueError	80
6. IndexError	80
7. KeyError	80
8. AttributeError	81
9. ImportError	81
10. FileNotFoundError	82
29. Depurando código com debugger	83
Usando um debugger	83
Breakpoints e depuração linha a linha	85
30. Lidando com erros com blocos try/except	87
A estrutura de um try/except	87
Lidando com erros específicos	88
31. Criando nossas próprias exceções com raise	90
Para quê criar Exceções?	90
Gerando Exceções a partir de outra	91
32. Usando type hints - tipagem de dados em Python	93
O que é tipagem de dados?	93
Tipagem de dados compostos	93
Tipagem de funções	93
Indicando o retorno de funções	94
Outros detalhes de tipagem	95

01. Introdução ao curso

Bem-vindo a curso de Python Intermediário da Asimov Academy!

Motivação do curso

Criamos este curso como uma continuação dos conhecimentos dos nossos cursos mais básicos. Aqui, você aprenderá formas mais eficazes de trabalhar com Python, seja aprendendo e utilizando funções novas, ou conhecendo outras estruturas possíveis de serem criadas em Python.

Python é uma linguagem altamente expressiva, com a qual conseguimos criar ideias relativamente complexas com poucas linhas. Mas para conseguir “dominar” esta complexidade, precisamos conhecer a fundo todas as expressões, operadores e possibilidades que a linguagem nos oferece. E isso obviamente não é possível de ser aprofundado nos nossos cursos introdutórios.

É nessa linha que entra o conteúdo deste curso. Não há nada que possa ser considerado “fundamental” dentro deste curso, no sentido de que é impossível programar em Python sem conhecer o assunto X ou Y. Dito isso, vamos abordar diversos conhecimentos muito úteis para quem quiser se tornar cada vez mais proficiente com a linguagem. Nesse sentido, é como aprender uma nova língua (natural): eu não preciso necessariamente saber a língua a fundo para conseguir me comunicar nela em um nível básico, mas certas habilidades requerem um nível mais avançado para que sejam desempenhadas com sucesso.

Estrutura do curso

O conteúdo está dividido por módulos que abordam assuntos similares. Ao final de cada módulo, apresentamos alguns desafios práticos (com resoluções) para praticarmos os conhecimentos. Mesmo assim, vocês verão que o conteúdo é quase que uma “coletânea de ferramentas” de Python. Em alguns casos, vamos aprender formas alternativas ou mais eficazes para realizar tarefas que já conhecemos, como iterações. Em outros, aprenderemos formas de trabalhar com dados específicos, como datas, textos ou funções.

Quando nos aprofundamos tanto em uma área, a melhor forma de aprendermos é sempre aliando a teoria com a prática. Portanto, os melhores “desafios” possíveis para quem está nesse nível é tentar construir algum projeto próprio de Python (como os muitos que temos disponíveis na Asimov Academy). É aí que cada uma das ferramentas que vamos conhecer no curso terão seu momento para brilhar!

02. Conjuntos e seus métodos

Conjuntos em Python são **sequências não ordenadas e de elementos únicos**. A ideia dessa estrutura de dados vem dos conjuntos matemáticos (aqueles dos diagramas de Venn):

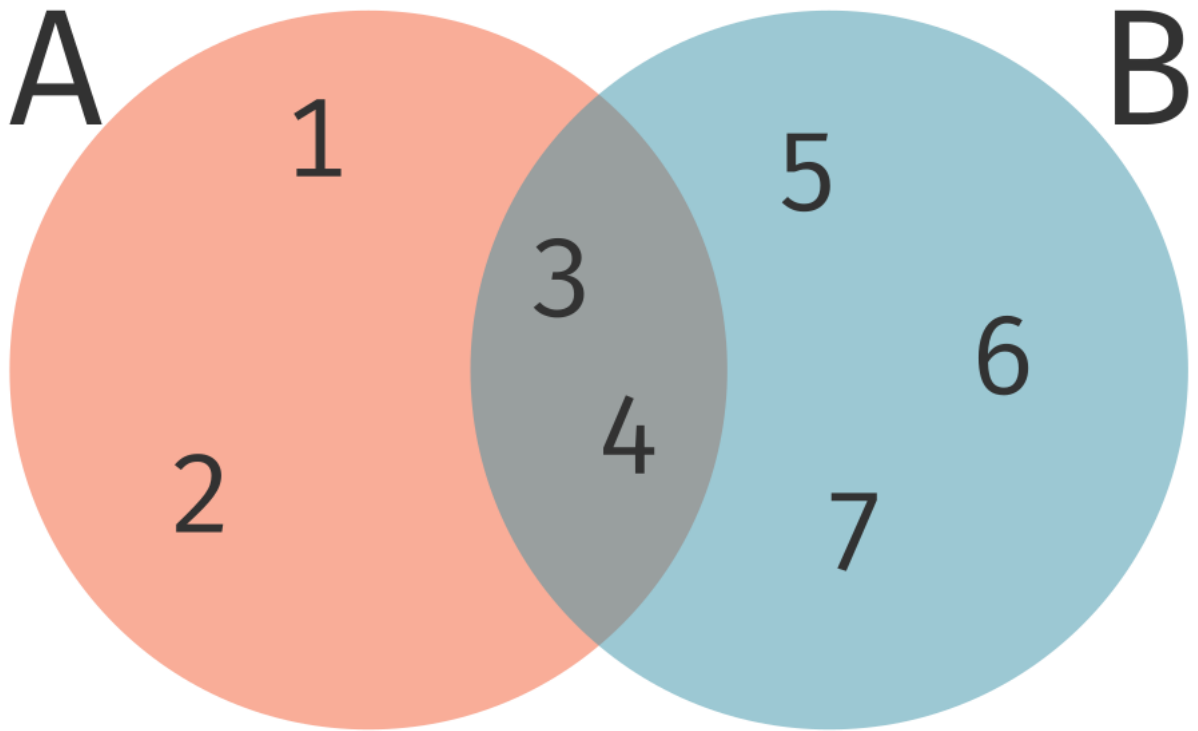


Figura 1: Representação visual de conjuntos como um diagrama de Venn

Para criar um conjunto, usamos colchetes `{ }` e separamos seus elementos com vírgulas (fica parecido com um dicionário, porém sem os `:` que associam chaves a valores):

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6, 7}
```

Também é possível usar a função `set()` para criá-los (neste caso é preciso passar uma sequência de elementos, como uma lista):

```
A = set([1, 2, 3, 4])
B = set([3, 4, 5, 6, 7])
```


Propriedades de conjuntos

Valores únicos

Os valores em um conjunto são **únicos**: não há valores duplicados, e criar um conjunto com valores duplicados ou adicionar valores repetidos não faz nada:

```
A = {1, 2, 1, 1, 2, 1, 2}
print(A)
# output: {1, 2}
```

```
A.add(1)
A.add(2)
A.add(3)
print(A)
# output: {1, 2, 3}
```

Podemos usar essa propriedade para filtrar valores únicos de uma lista:

```
numeros = [1, 2, 3, 1, 2, 1, 4]
numeros_unicos = list(set(numeros))
print(numeros_unicos)
# output: [1, 2, 3, 4]
```

Sem ordem

Os valores em um conjunto estão **desordenados**: não podemos pegar o “primeiro” ou “último” item de um conjunto, ou garantir que os elementos serão iterados em uma ordem qualquer:

```
A = {10, 'Python', 1.0, False}
print(A[0])
# TypeError: 'set' object is not subscriptable

for elemento in A:
    print(elemento)
# output:
# False
# Python
# 10
# 1.0
```

Elementos imutáveis

Por fim, os elementos precisam ser **imutáveis**. Dados mutáveis, como listas e dicionários, não podem ser elementos de conjuntos:

```
A = {10, [1, 2, 3]}
# TypeError: unhashable type: 'list'
```

Por outro lado, tuplas (que são imutáveis) podem estar dentro de um conjunto.

Se pensarmos em como conjuntos não possuem valores repetidos, essa limitação faz sentido. Caso contrário, poderíamos criar conjuntos contendo listas diferentes, mas que se tornassem idênticas em algum momento do código, acabando assim com valores repetidos no conjunto!

Operações de conjuntos

Conjuntos são muito utilizados para fazer checagens de pertencimento com `in`. Compare o tempo de execução com listas (é necessário usar um console de IPython ou Jupyter Notebook para o código abaixo):

```
numeros = list(range(1_000))
%timeit 500 in numeros
# 2.7 µs ± 30 ns per loop

conj_numeros = set(numeros)
%timeit 500 in conj_numeros
# 19.5 ns ± 0.131 ns per loop
```

Além disso, é possível realizar as operações “clássicas” de conjuntos, com origem na matemática e teoria de grupos:

União

O método `set.union()` retorna a união de dois ou mais conjuntos. Também é possível usar o operador `|` para isso:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6, 7}

print(A.union(B))
# output: {1, 2, 3, 4, 5, 6, 7}

print(A | B)
# output: {1, 2, 3, 4, 5, 6, 7}
```

Intersecção

O método `set.intersection()` retorna a intersecção (elementos comuns) de dois ou mais conjuntos. Também é possível usar o operador `&` para isso:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6, 7}
```

```
print(A.intersection(B))  
# output: {3, 4}
```

```
print(A & B)  
# output: {3, 4}
```

Diferença

O método `set.difference()` retorna a diferença de dois ou mais conjuntos. Também é possível usar o operador `-` para isso.

Note que a diferença de A com B não é a mesma coisa que a diferença de B com A:

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6, 7}
```

```
print(A.difference(B))  
# output: {1, 2}
```

```
print(B.difference(A))  
# output: {5, 6, 7}
```

```
print(A - B)  
# output: {1, 2}
```

```
print(B - A)  
# output: {5, 6, 7}
```

Diferença simétrica

O método `set.symmetric_difference()` retorna a diferença simétrica de dois ou mais conjuntos - isto é, elementos que estão em um conjunto ou no outro, mas não em ambos.

Não há um operador específico para esta ação, mas ela é o resultado da *união das diferenças* entre os conjuntos:

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6, 7}
```

```
print(A.symmetric_difference(B))  
# output: {1, 2, 5, 6, 7}
```

```
print((A - B) | (B - A))  
# output: {1, 2, 5, 6, 7}
```

03. Compreensão de lista

Quando queremos filtrar valores em Python, é muito comum desenvolvermos um código como este:

```
valores = list(range(10))

maiores_que_cinco = []
for valor in valores:
    if valor > 5:
        maiores_que_cinco.append(valor)

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Podemos recriar esta lógica em uma única linha, usando uma **compreensão de lista**!

A estrutura de uma compreensão de lista

A estrutura de uma compreensão de lista tem o seguinte formato:

NOVA_LISTA = [RESULTADO para cada ELEMENTO em SEQUÊNCIA se CONDIÇÃO]

Pode parecer confuso. Mas pensarmos no código anterior, já tínhamos esta estrutura básica distribuída no bloco central:

```
NOVA_LISTA = []
para cada ELEMENTO em SEQUÊNCIA:
    se CONDIÇÃO:
        RESULTADO entra em NOVA_LISTA
```

Criando a compreensão de lista

Se reestruturarmos o código acima para uma compreensão de lista, temos:

```
valores = list(range(10))

maiores_que_cinco = [valor for valor in valores if valor > 5]

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Para deixar ainda mais claro, podemos até reordenar cada bloco dentro da lista:

```
valores = list(range(10))

maiores_que_cinco = [
    valor # RESULTADO
    for valor in valores # para cada ELEMENTO em SEQUÊNCIA
    if valor > 5 # se CONDIÇÃO
]

print(maiores_que_cinco)
# output: [6, 7, 8, 9]
```

Também é possível modificar os valores na lista original de acordo com uma lógica qualquer:

```
def somar_dois_ao_quadrado(valor):
    return (valor ** 2) + 2

valores = list(range(10))

resultado = [
    somar_dois_ao_quadrado(valor)
    for valor in valores
    if valor not in [0, 5]
]

print(resultado)
# output: [3, 6, 11, 18, 38, 51, 66, 83]
```

Por que usar compreensão de lista?

Pode parecer apenas um detalhe, mas usar compreensão de lista facilita bastante a escrita de código:

- Forma padronizada para criar novas listas/dicionários/conjuntos a partir de sequências
- Sintaxe mais simples de ser lida e compreendida do que recriar toda a estrutura de um for loop tradicional
- Otimizado para Python: executa mais rapidamente que outras formas de criar/filtrar sequências

04. Compreensão de dicionários e conjuntos

Compreensão de dicionários

A compreensão de dicionários segue a mesma lógica da compreensão de lista, porém criando associações de chaves e valores:

NOVO_DIC = { CHAVE: VALOR para cada ELEMENTO em SEQUÊNCIA se CONDIÇÃO }

Vamos usar esta estrutura para converter o dicionário `valores_em_dolares` para `valores_em_reais` abaixo:

```
valores_em_dolares = {
    'leite': 0.78,
    'carne': 4.60,
    'maçã': 0.35,
}

fator_conversao = 4.93
valores_em_reais = {
    produto: round(preco * fator_conversao, 2)
    for (produto, preco) in valores_em_dolares.items()
}

print(valores_em_reais)
# output: {'leite': 3.85, 'carne': 22.68, 'maçã': 1.73}
```

Note que aqui já “desempacotamos” cada elemento recebido do `valores_em_dolares.items()` nas variáveis `produto` e `preco`. Conseguimos fazer isso porque sabemos que o método `.items()` itera sempre por tuplas de 2 elementos (a chave do dicionário e o seu valor correspondente).

Compreensão de conjuntos

Seguem a mesma lógica de compreensão de listas e dicionários. Para construí-los, basta trocar os colchetes (`[]`) de uma compreensão de lista por chaves (`{ }`):

NOVO_CONJ = {RESULTADO para cada ELEMENTO em SEQUÊNCIA se CONDIÇÃO}

Exemplo de código:

```
valores = set(range(10))

maiores_que_cinco = {valor for valor in valores if valor > 5}
print(maiores_que_cinco)
# output: {6, 7, 8, 9}
```

05. Desafio – Conjuntos e compreensão de lista

Desafios

Desafio 01

Converta o loop abaixo para uma compreensão de lista:

```
valores = [30, 50, 100, 120]

triplos = []
for valor in valores:
    triplo = valor * 3
    triplos.append(triplo)

print(triplos)
```

Desafio 02

Crie uma compreensão de dicionários a partir de uma lista de palavras. No dicionário resultante, cada chave é a palavra em letras minúsculas, e cada valor associado é o número de caracteres da palavra, sem contar espaços em branco.

Exemplo de lista de palavras e o dicionário resultante:

```
palavras = ['Olá', 'Python', 'Juliano', 'Asimov Academy']

dict_caracteres = {'olá': 3, 'python': 6, 'juliano': 7, 'asimov academy': 13}
```

Desafio 03

Meus amigos possuem os seguintes interesses:

- **Gostam de programação:** Ricardo, Roberto, Pedro, Vinicius
- **Gostam de jogar futebol:** Mateus, Roberto, Paulo, Vinicius
- **Estudam na Asimov Academy:** Ricardo, Mateus, Paulo, Pedro

Usando operações de conjunto, encontre o conjunto de amigos que gostam de programação e estudam na Asimov Academy, mas que *não* gostam de jogar futebol.

Soluções

Solução do desafio 01

```
valores = [30, 50, 100, 120]
triplos = [valor * 3 for valor in valores]

print(triplos)
```

Solução do desafio 02

```
palavras = ['Olá', 'Python', 'Juliano', 'Asimov Academy']

dict_caracteres = {
    palavra.lower(): len(palavra.replace(' ', ''))
    for palavra in palavras
}

print(dict_caracteres)
# output: {'olá': 3, 'python': 6, 'juliano': 7, 'asimov academy': 13}
```

Solução do desafio 03

```
gostam_programacao = {'Ricardo', 'Roberto', 'Pedro', 'Vinicius'}
gostam_futebol = {'Mateus', 'Roberto', 'Paulo', 'Vinicius'}
estudam_asimov = {'Ricardo', 'Mateus', 'Paulo', 'Pedro'}

# Usando métodos de conjuntos
resultado = gostam_programacao.intersection(estudam_asimov).difference(gostam_futebol)

# Usando operadores de conjuntos
resultado = (gostam_programacao | estudam_asimov) - gostam_futebol

print(resultado)
# output: {'Pedro', 'Ricardo'}
```


06. O bloco `with`

Algumas ações em Python precisam ser feitas de forma sequencial. Por exemplo, para escrevermos um arquivo de texto, é preciso:

- Abrir o arquivo com a função `open()`.
- Usar o método `.write()` do arquivo para escrever o texto nele.
- Usar o método `.close()` para fechá-lo.

O exemplo abaixo mostra estes passos, usando a função `open()` para abrir um arquivo chamado `notas.txt` em modo de escrita (`w`):

```
arquivo = open('notas.txt', 'w')
arquivo.write('Esta é uma anotação especial!')
arquivo.close()
```

Se não fecharmos os arquivos, pode ser que outros programas entrem em conflito com nosso arquivo (por exemplo, em Windows você pode receber um “acesso negado” caso tente acessar um arquivo aberto em outro processo).

Para não correremos o risco de esquecer o arquivo aberto, podemos usar o **bloco `with`**:

```
with open('notas.txt', 'w') as arquivo:
    arquivo.write('Esta é uma anotação especial!')
```

O **bloco `with`** representa um **gerenciador de contexto** (do inglês *context manager*). Estes blocos executam uma lógica automática tanto antes quanto depois do bloco indentado para a direita.

No caso da função `open()`, a lógica executada é fechar o arquivo automaticamente após o bloco. Isso acontece mesmo se Python gerar algum erro no código dentro do bloco `with`!

Exemplos de uso do bloco `with`

Criar pasta temporária com módulo `tempfile`:

```
import tempfile

with tempfile.TemporaryDirectory() as temp_dir:
    print(f'Diretório temporário criado: {temp_dir}')
    input()

# Diretório não existe mais aqui!
```

Abrir, criar e fechar arquivo `.zip` com módulo `zipfile`:

```
import zipfile
```

```
nome_arquivo = 'notas.txt'

with open(nome_arquivo, 'w') as arquivo_aberto:
    arquivo_aberto.write('Esta é uma anotação!')

with zipfile.ZipFile('compactado.zip', 'w') as arquivo_zip:
    arquivo_zip.write(nome_arquivo)

# Arquivo .zip foi fechado aqui!

arquivo_zip.write(nome_arquivo)
# ValueError: Attempt to write to ZIP archive that was already closed
```

07. O bloco `match case`

O bloco `match case` é uma introdução relativamente recente de Python (versão 3.10). Ele funciona como uma forma mais concisa de um bloco `if/elif/else`, especialmente quando queremos testar valores específicos:

```
op = 1

# Versão com if/elif/else
if op == 1:
    print('Opção 1')
elif op == 2:
    print('Opção 2')
else:
    print('Opção inválida!')

# Versão com match case
match op:
    case 1:
        print('Opção 1')
    case 2:
        print('Opção 2')
    case _:
        print('Opção inválida!')
```

`match case` em padrões estruturais

Até aqui, o `match case` apenas parece uma forma diferente de fazer um `if/elif/else`. Mas a principal diferença está em procurar **padrões estruturais** através do `match case`.

Por exemplo: o `match case` do código abaixo consegue diferenciar entre diferentes estruturas do dicionário `notas`. Há 2 pontos para destacar:

- A variável `_` é uma forma de simbolizar “qualquer valor”.
- O operador `|` (barra horizontal ou *pipe*) pode ser usado como forma de testar valores alternativos em um mesmo `case`.

```
notas = {
    'João': 10,
    'Maria': 9,
    'Mateus': 9.2,
}

match notas:
    case {'Lucas': _} | {'Marcos': _}:
        print('Lucas ou Marcos estão no dicionário!')
    case {'João': 10, 'Maria': 10}:
        print('João e Maria estão no dicionário, e ambos tiraram 10!')
```

```
case {'João': 10, 'Maria': _}:  
    print('João e Maria estão no dicionário, e só João tirou 10!')  
case _:  
    print('Nenhuma informação encontrada!')
```

08. Operadores de divisão

Já conhecemos o operador de divisão “comum” (/). Contudo, Python possui outros 2 operadores de divisão,

- O operador //, que representa uma divisão inteira, sem fração (operador “ piso”)
- O operador %, que representa o “resto” da divisão (operador “módulo”)

Estes operadores são utilizados com menos frequência, mas cada um possui utilidade em alguns cenários:

O operador piso

Usado para obter a parte inteira de uma divisão:

```
print(9 // 3) # Número 3 cabe 3 vezes no 9, sem resto
# output: 3

print(10 // 3) # Número 3 cabe 3 vezes no 10, resta 1
# output: 3

print(14 // 3) # Número 3 cabe 4 vezes no 14, resta 2
# output: 4

print(15 // 3) # Número 3 cabe 5 vezes no 15, sem resto
# output: 5
```

O operador módulo

Usado para obter a parte restante de uma divisão (de certa forma, é o “complemento” do operador piso):

```
print(9 % 3) # Número 3 cabe 3 vezes no 9, sem resto
# output: 0

print(10 % 3) # Número 3 cabe 3 vezes no 10, resta 1
# output: 1

print(14 % 3) # Número 3 cabe 4 vezes no 14, resta 2
# output: 2

print(15 % 3) # Número 3 cabe 5 vezes no 15, sem resto
# output: 0
```

Usos para os operadores de divisão

Extrair valor de horas, minutos e segundos:

```
segundos_totais = 65

minutos = segundos_totais // 60
segundos = segundos_totais % 60

print(f'{segundos_totais}s = {minutos}m{segundos}s')
```

Descobrir se um número é par:

```
n = 5
par = (n % 2) == 0

print(f'{n} é par? {par}')
```

Descobrir se um número é divisor de outro:

```
n = 5
div = 3
divisor = (n % div) == 0

print(f'{div} é divisor de {n}? {divisor}')
```

09. Expressão condicional

Em Python, muitas vezes temos que definir uma variável de acordo com o resultado de uma condição `if/else`:

```
x = 4
y = 5

if x > y:
    maior_valor = x
else:
    maior_valor = y
```

Podemos “condensar” este bloco com um **operador ternário**, também chamadas de **expressões condicionais** em Python.

Operadores ternários

Em linguagens de programação, operadores ternários tem a seguinte estrutura:

VAR = CONDIÇÃO ? VALOR SE VERDADEIRO ; VALOR SE FALSO

Essa estrutura é altamente utilizada, desde a linguagem C até a função SE= do Excel!

Expressões condicionais: operadores ternários em Python

Em Python não é diferente, porém a ordem dos operadores é ligeiramente diferente para melhorar a legibilidade do código:

VAR = VALOR_VERDADEIRO SE CONDIÇÃO SENÃO VALOR_FALSO

Voltando para o primeiro exemplo, podemos reduzir a o bloco de código com uma expressão condicional da seguinte forma:

```
x = 4
y = 5

maior_valor = x if x > y else y
```

É importante deixar claro que estas expressões condicionais são apenas **formas reduzidas de criar blocos `if/else`**. Não há nenhuma funcionalidade nova, apenas um ganho na redução do código e legibilidade!

Usando expressões condicionais

É bastante comum encontrarmos expressões condicionais em funções curtas. Por exemplo, uma função que retorna a cor a partir de um valor (para plotar a cor em um gráfico):

```
def pegar_cor(valor):  
    return 'vermelho' if valor < 0 else 'azul'  
  
for valor in [10, -2, 3]:  
    print(pegar_cor(valor))
```

Também podemos usá-las para usar um valor-padrão, caso o valor não passe por algum teste:

```
entrada = input("Digite um número (padrão: 0) ->")  
n = int(entrada) if entrada.isdigit() else 0  
print(f"Seu número é: {n}")
```

É até possível utilizar expressões condicionais dentro de compreensão de listas!

```
numeros = [1, 2, 3, 4]  
  
pares_quadrados = [  
    n**2  
    if n % 2 == 0  
    else n  
    for n in numeros  
]  
print(pares_quadrados)  
# output: [1, 4, 3, 16]
```

O código acima é equivalente a:

```
numeros = [1, 2, 3, 4]  
  
pares_quadrados = []  
for n in numeros:  
    if n % 2 == 0:  
        resultado = n ** 2  
    else:  
        resultado = n  
    pares_quadrados.append(resultado)  
  
print(pares_quadrados)  
# output: [1, 4, 3, 16]
```


10. Identidade e o operador `is`

Pergunta rápida: estas listas são iguais?

```
lista_a = [1, 2, 3, 4, 5]
lista_b = [1, 2, 3, 4, 5]
```

Do ponto de vista de Python, elas são sim iguais...

```
lista_a = [1, 2, 3, 4, 5]
lista_b = [1, 2, 3, 4, 5]
```

```
print(lista_a == lista_b)
# output: True
```

Porém elas não são a mesma lista: no momento em que altero a `lista_a`, elas deixam de ser iguais:

```
lista_a = [1, 2, 3, 4, 5]
lista_b = [1, 2, 3, 4, 5]
```

```
lista_a.append(6)
```

```
print(lista_a == lista_b)
# output: False
```

O operador `is`

No exemplo anterior, cada lista é considerada um objeto próprio. Ou seja, a identidade da `lista_a` não é a mesma que a identidade da `lista_b`, ainda que ambas possam possuir exatamente os mesmos valores.

Podemos testar a identidade em Python com o operador `is`. Note que as duas listas nunca terão a mesma identidade:

```
lista_a = [1, 2, 3, 4, 5]
lista_b = [1, 2, 3, 4, 5]
```

```
print(lista_a is lista_b)
# output: False
```

Uma forma de entendermos identidade em Python é pensando em bolos. Dois bolos podem ser exatamente idênticos, porém não são o mesmo objeto: quando eu corto um dos bolos, o outro não é automaticamente cortado também!

```
bolo1 = {
    'sabor': 'chocolate',
    'tamanho': 'grande',
    'preço': 50,
```

```
}
bolo2 = {
    'sabor': 'chocolate',
    'tamanho': 'grande',
    'preço': 50,
}

print(bolo1 == bolo2)
# output: True
print(bolo1 is bolo2)
# output: False

bolo1['preço'] = 80

print(bolo1 == bolo2)
# output: False
print(bolo1 is bolo2)
# output: False
```

Identidade dos valores True, False e None

Os valores True, False e None são especiais em Python. Como efetivamente representam conceitos abstratos e imutáveis (verdadeiro, falso e nulo, respectivamente), estes 3 valores possuem uma única identidade.

Logo, para estes valores nós podemos usar o operador `is` em comparações tradicionais:

```
print(True is True)
# output: True
print(False is True)
# output: False
print(None is None)
# output: True
```

Pode parecer um detalhe, mas isto nos ajuda a tornar Python ainda mais legível, pois permite construir expressões com legibilidade quase como em inglês:

```
x = 5
y = 2
resultado = x > y
if resultado is True: # if resultado == True:
    print(f'{x} é maior que {y}')
else:
    print(f'{x} não é maior que {y}')

valor = None
if valor is not None: # if resultado != None:
    print(f'valor {valor} não é nulo')
else:
    print(f'O valor {valor} é nulo')
```

11. O operador “morsa” (expressão de atribuição)

Em muitos códigos de Python, é necessário pegar um valor e logo em seguida testá-lo para decidir o que fazer com ele.

Por exemplo, se nosso código se conectar a um banco de dados, é muito comum testarmos se o valor retornado é nulo (o que significaria que a busca não retornou nada):

```
valor_de_busca = 'xxx'

resultado = buscar_no_banco_de_dados(valor_de_busca)
if resultado is None:
    print(f'Nada foi encontrado para o valor de busca "{valor_de_busca}".')
else:
    print(f'Resultados encontrados para valor de busca "{valor_de_busca}": {resultado}')
```

O **operador morsa** (nome técnico: *expressão de atribuição*) nos permite pegar o resultado e testá-lo em uma única linha:

```
valor_de_busca = 'xxx'

if (resultado := buscar_no_banco_de_dados(valor_de_busca)) is None:
    print(f'Nada foi encontrado para o valor de busca "{valor_de_busca}".')
else:
    print(f'Resultados encontrados para valor de busca "{valor_de_busca}": {resultado}')
```

O nome **expressão de atribuição** deixa claro o que está acontecendo: em uma única linha, fazemos uma **atribuição** do tipo `var = func()`, e a colocamos em uma **expressão**, como `if var is None`:

Já o nome **operador morsa** é apenas uma referência ao formato do operador :)

Outros exemplos de uso do operador morsa

- Passar condição de parada de um loop *while* para sua definição:

```
# Sem operador morsa
n = 5
while n > 0:
    n -= 1
    print(n)

# Com operador morsa
n = 5
while (n := n-1) >= 0:
    print(n)
```

- Pegar valor e testá-lo em uma única linha:

```
# Sem operador morsa
valor = calcular_valor()
if valor > 0:
    processar_valor(valor)

# Com operador morsa
if (valor := calcular_valor()) > 0:
    processar_valor(valor)
```

- Evitar chamar funções mais de uma vez (em compreensões de lista, por exemplo):

```
numeros = [1, 2, 3, 4, 5]

# Sem compreensão de lista
quadrados_maiores_que_8 = []
for n in numeros:
    quadrado = n ** 2
    if quadrado > 8:
        quadrados_maiores_que_8.append(quadrado)

# Com compreensão de lista, sem operador morsa
quadrados_maiores_que_8 = [
    n ** 2
    for n in numeros
    if n ** 2 > 8
]

# Com compreensão de lista, com operador morsa
quadrados_maiores_que_8 = [
    quadrado
    for n in numeros
    if (quadrado := n ** 2) > 8
]
```

Vale a pena usar?

O operador morsa é um detalhe, mas que em alguns casos ajuda a tornar o código mais conciso e legível. E em casos bem específicos, evita recalcular alguma função.

Cabe a você decidir se faz sentido usar em cada caso. Dependendo da expressão ou do que seu código tenta fazer, usar o operador morsa pode torná-lo menos legível!

12. Desafio – Operadores

Desafios

Desafio 01

Crie uma função que retorna se um número inteiro n (maior que zero) é primo.

Dica: um número primo é um número que só é divisível por 1 ou por ele mesmo.

Desafio 02

Temos duas funções abaixo que simulam operações em bancos de dados:

- `busca_dados`, que retorna informação de um banco de dados, mas que falha 50% das vezes em que é executada.
- `processa_dados`, que processa a informação obtida a partir do banco de dados.

Você consegue usar os operadores e expressões que aprendemos para simplificar o bloco de código ao final do script? (Não modifique o corpo das funções)

```
import random

def busca_dados():
    if random.random() > 0.5:
        return None
    return 'xxxxx'

def processa_dados(dados):
    return f'Dados "{dados}" foram processados'

dados_banco = busca_dados()
if dados_banco is not None:
    dados_processados = processa_dados(dados_banco)
else:
    dados_processados = 'N/A'

print(f'Resultado: {dados_processados}')
```

Soluções

Solução do desafio 01

Lógica utilizada:

- Para os números 1 e 2, retornamos sempre True (por definição, são números primos).
- Para os demais números n, criamos um range de divisores d, indo de 2 até n (sem incluir n).
- Se o resto da divisão (operador módulo ou %) de n por algum divisor d for zero, então n não é primo. Caso contrário, n é primo.

```
def primo(n):
    if n <= 2: # 1 e 2 são primos
        return True
    for div in range(2, n):
        if n % div == 0:
            return False
    return True

# Testando a função
for n in [1, 5, 10, 13, 15, 17]:
    print(n, 'é número primo?', primo(n))
```

Solução do desafio 02

Combinando uma expressão condicional com o operador morsa, é possível transformar o bloco de código em uma única linha:

```
import random

def busca_dados():
    if random.random() > 0.5:
        return None
    return 'xxxxx'

def processa_dados(dados):
    return f'Dados "{dados}" foram processados'

dados_processados = processa_dados(dados_banco) if (dados_banco := busca_dados()) is not None
↪ else 'N/A'

print(f'Resultado: {dados_processados}')
```

Nota: nem sempre menos linhas = código melhor. É fundamental que o código seja legível e compreensível!

13. Funções de iteração

Python possui diversas funções especializadas em iteração. Dessa forma, não precisamos gastar muito tempo pensando em implementar formas específicas de iterar sobre elementos, pois podemos usar as funções built-in da própria linguagem para isso.

A função `enumerate`

Esta função é usada para **iterar sobre elementos e índices** de uma sequência ao mesmo tempo.

Ao passarmos qualquer sequência para `enumerate` e iterarmos sobre ela, vamos acabar iterando sobre pares de índice-valor, conforme exemplo abaixo:

```
nomes = ['Juliano', 'José', 'Lucas', 'Luiza']
```

```
for i, nome in enumerate(nomes):  
    print(f'Índice {i} -> Nome: {nome}')
```

Podemos até passar um segundo argumento para `enumerate` que descreve o índice a partir do qual começar a contar (muitas vezes, vamos querer usar 1 como valor):

```
nomes = ['Juliano', 'José', 'Lucas', 'Luiza']
```

```
for i, nome in enumerate(nomes, 1):  
    print(f'Posição {i} da lista -> Nome: {nome}')
```

A função `sorted`

Esta função **ordena elementos** de qualquer sequência (ainda que a sequência em si não seja ordenável, como um conjunto).

Para fazer isso, ela *sempre retorna uma lista*:

```
conj = {1, 10, -1, 4}  
ordenados = sorted(conj)  
  
print(ordenados)  
# output: [-1, 1, 4, 10]
```

Para listas, usar `sorted` é diferente de usar o método `list.sort()`, pois uma nova lista é criada (a original não é tocada):

```
lista1 = [1, 10, -1, 4]  
lista2 = sorted(lista1)  
  
print(lista1)  
# output: [1, 10, -1, 4]
```

```
print(lista2)
# output: [-1, 1, 4, 10]
```

```
lista1.sort()
```

```
print(lista1)
# output: [-1, 1, 4, 10]
```

A função `sorted` também aceita o argumento `reverse=True` para ordenar de forma invertida:

```
conj = {1, 10, -1, 4}
ordenados = sorted(conj, reverse=True)
```

```
print(ordenados)
# output: [10, 4, 1, -1]
```

E por fim, note que os elementos devem ser comparáveis entre si (Python não “sabe” comparar ints e strings, por exemplo):

```
conj = {1, 10, 'Python', 4}
ordenados = sorted(conj, reverse=True)
# TypeError: '<' not supported between instances of 'int' and 'str'
```

A função `reversed`

Esta função cria um iterável que passa sobre elementos de uma sequência qualquer, **indo do último para o primeiro**:

```
for i in reversed(range(10)): # Equivalente a "for i in range(9, -1, -1):"
    print(i)
```

```
nomes = ['Juliano', 'José', 'Amanda', 'Tiago']
for nome in reversed(nomes):
    print(nome)
```

A sequência precisa ser “reversível” (por exemplo, conjuntos não funcionam pois não possuem ordem):

```
conj = {1, 4, -10}
for n in reversed(conj):
    print(n)
# TypeError: 'set' object is not reversible
```

A função `zip`

Esta função é usada para **passar por elementos de dois ou mais iteráveis “lado a lado”**. Veja o exemplo:


```
nomes = ['Juliano', 'Laura', 'Roberto', 'Guilherme']
idades = [30, 24, 19, 47]
```

```
for elemento in zip(nomes, idades):
    print(elemento)
```

```
# output:
# ('Juliano', 30)
# ('Laura', 24)
# ('Roberto', 19)
# ('Guilherme', 47)
```

Note que sempre são recebidas tuplas contendo os primeiros, segundos, terceiros, ... elementos de cada lista.

Como explicado anteriormente, é possível usar a função `zip` com mais de duas sequências. O único detalhe é que, se as sequências não tiverem o mesmo tamanho, a iteração acabará quando a menor das sequências for finalizada:

```
nomes = ['Juliano', 'Laura', 'Roberto', 'Guilherme']
idades = [30, 24, 19, 47]
cpfs = ['xxx', 'yyy', 'zzz']
emails = ['juliano@empresa.com', 'laura@empresa.com']
```

```
for elemento in zip(nomes, idades, cpfs, emails):
    print(elemento)
```

```
# output:
# ('Juliano', 30, 'xxx', 'juliano@empresa.com')
# ('Laura', 24, 'yyy', 'laura@empresa.com')
```

No caso acima, como a lista `emails` possui apenas 2 elementos, depois da segunda iteração o loop já se encerrou.

14. Desempacotando sequências

Qualquer sequência em Python pode ser “desempacotada” em variáveis individuais:

```
seq = (10, 20, 30)
```

```
a, b, c = seq
```

```
print(a)
# output: 10
```

```
print(b)
# output: 20
```

```
print(c)
# output: 30
```

Também costumamos fazer isso ao iterarmos sobre sequências:

```
dic = {
    'chave1': 'valor1',
    'chave2': 'valor2',
    'chave3': 'valor3',
}

for k, v in dic.items():
    print(f'Chave: {k}', f'Valor: {v}')
```

Desempacotamento aninhado

E se tivermos uma iteração mais complexa, como essa:

```
nomes = ['Juliano', 'Laura', 'Roberto', 'Guilherme']
idades = [30, 24, 19, 47]
```

```
for elemento in enumerate(zip(nomes, idades)):
    print(elemento)
```

```
# output:
# (0, ('Juliano', 30))
# (1, ('Laura', 24))
# (2, ('Roberto', 19))
# (3, ('Guilherme', 47))
```

Como usamos o `enumerate`, então sabemos que a variável `elemento` é composta pelo índice e pelos elementos sobre os quais estamos iterando.

Contudo, como dentro do `enumerate` estamos usando `zip`. Logo, cada elemento sendo iterado é, na realidade, uma tupla de dois elementos!

Neste caso, podemos usar parênteses para desempacotar cada valor individual - tanto o índice vindo do `enumerate`, quanto os valores sendo iterados lado a lado pelo `zip`:

```
nomes = ['Juliano', 'Laura', 'Roberto', 'Guilherme']
idades = [30, 24, 19, 47]

for i, (nome, idade) in enumerate(zip(nomes, idades)):
    print(f'Índice {i} -> {nome}, {idade} anos')

# output:
# Índice 0 -> Juliano, 30 anos
# Índice 1 -> Laura, 24 anos
# Índice 2 -> Roberto, 19 anos
# Índice 3 -> Guilherme, 47 anos
```

Desempacotamento com o operador *

Um detalhe importante no desempacotamento de sequências é que o número de variáveis **deve** bater com o número de elementos na iteração:

```
a, b = (1, 2, 3)
# ValueError: too many values to unpack (expected 2)

dic = {
    'chave1': 'valor1',
    'chave2': 'valor2',
    'chave3': 'valor3',
}

for a, b, c in dic.items():
    print(f'Chave: {a}', f'Valor: {b}')
# ValueError: not enough values to unpack (expected 3, got 2)
```

Nestes exemplos, é meramente uma questão de ajustarmos a sintaxe, porque sabemos o que são os valores que estamos tentando desempacotar.

Contudo, podemos não ter conhecimento de antemão sobre quantos valores serão desempacotados, ou até mesmo trabalhar com sequências de diferentes tamanhos. Nestes casos, podemos usar o operador * (asterisco) para “agrupar” valores que podem sobrar ou faltar.

Exemplo:

```
minha_lista = [1, 2, 3, 4, 5]

primeiro, *meio, ultimo = minha_lista

print(primeiro)
# output: 1
print(meio)
# output: [2, 3, 4]
```

```
print(ultimo)
# output: 5
```

Note que a variável `meio` “absorveu” todos os valores que precisava, de modo a deixar as variáveis `primeiro` e `ultimo` com 1 elemento cada.

Outros exemplos:

```
primeiro, *resto = (1, 2, 3, 4)
print(primeiro)
# output: 1
print(resto)
# output: [2, 3, 4]
```

```
*resto, penultimo, ultimo = (1, 2, 3, 4)
print(resto)
# output: [1, 2]
print(penultimo)
# output: 3
print(ultimo)
# output: 4
```

```
primeiro, *meio, ultimo = (1, 2, 3)
print(primeiro)
# output: 1
print(meio)
# output: [2]
print(ultimo)
# output: 3
```

```
primeiro, *meio, ultimo = (1, 2)
print(primeiro)
# output: 1
print(meio)
# output: []
print(ultimo)
# output: 2
```

Note aqui que a variável com `*` **sempre retorna uma lista de valores**, ainda que a sequência original seja outro tipo de dado (como tuplas), ou que a lista acabe com um ou nenhum elemento nela.

Por fim, é muito comum usarmos `*` para simbolizar valores que queremos ignorar. Se só quisermos o segundo valor de uma lista, é possível escrever:

```
_, segundo, *_ = (1, 2, 3, 4, 5)

print(segundo)
# output: 2
```

15. Iteração avançada com `itertools`

O módulo `itertools` da biblioteca padrão de Python contém muitas funções úteis para iteração. Vamos explorar algumas delas aqui:

`itertools.chain`

Usado para “encadear” diversas sequências em uma única sequência. Assim é possível iterar sobre todas as sequências, uma atrás da outra:

```
import itertools

seq1 = (1, 2, 3)
seq2 = ['a', 'b', 'c']

for elemento in itertools.chain(seq1, seq2):
    print(elemento)

# output:
# 1
# 2
# 3
# a
# b
# c
```

`itertools.zip_longest`

Funciona da mesma forma que a função `zip`, mas não para na sequência mais curta. Quando acabarem, as sequências curtas retornam o valor definido em `fillvalue`:

```
import itertools

nomes = ['Juliano', 'Laura', 'Roberto', 'Guilherme']
idades = [30, 24, 19, 47]
cpfs = ['xxx', 'yyy', 'zzz']
emails = ['juliano@empresa.com', 'laura@empresa.com']

for elemento in itertools.zip_longest(nomes, idades, cpfs, emails, fillvalue='???'):
    print(elemento)

# output:
# ('Juliano', 30, 'xxx', 'juliano@empresa.com')
# ('Laura', 24, 'yyy', 'laura@empresa.com')
# ('Roberto', 19, 'zzz', '???')
# ('Guilherme', 47, '???' , '???)
```

itertools.product

Itera sobre o **produto** de diversas sequências, isto é, cada uma das combinações possíveis de serem feitas pegando 1 elemento de cada sequência passada para `itertools.product`:

```
import itertools

comidas = ['Churrasco', 'Pizza', 'Sushi']
bebidas = ('Refrigerante', 'Água')

for prato in itertools.product(comidas, bebidas):
    print(prato)

# output:
# ('Churrasco', 'Refrigerante')
# ('Churrasco', 'Água')
# ('Pizza', 'Refrigerante')
# ('Pizza', 'Água')
# ('Sushi', 'Refrigerante')
# ('Sushi', 'Água')
```

itertools.combinations

Itera sobre as combinações possíveis de serem feitas com uma sequência. O segundo argumento é o tamanho de cada combinação.

Exemplo: todas as combinações possíveis de 4 nomes, agrupando a cada 3:

```
import itertools

nomes = ['Marcos', 'Lucas', 'Rodrigo', 'Carlos']
for comb in itertools.combinations(nomes, 3):
    print(comb)

# output:
# ('Marcos', 'Lucas', 'Rodrigo')
# ('Marcos', 'Lucas', 'Carlos')
# ('Marcos', 'Rodrigo', 'Carlos')
# ('Lucas', 'Rodrigo', 'Carlos')
```

Há também `itertools.permutations`, que faz a mesma coisa mas com permutações (onde a ordem dos elementos importa).

itertools.cycle

É um iterador infinito. Itera sobre todos os elementos da sequência indefinidamente, voltando sempre para o começo:

```
import itertools

for cor in itertools.cycle(['azul', 'vermelho']):
    print(cor)
    input() # Pausar código para conseguirmos ver funcionando
```

Para que ele não bloqueie o código indefinidamente, é preciso ter uma condição de parada na iteração. Uma ideia é usá-lo junto do `zip`:

```
import itertools

nomes = ['Marcos', 'Lucas', 'Rodrigo', 'Carlos']
cores = itertools.cycle(['azul', 'vermelho'])

for nome, cor in zip(nomes, cores):
    print(f'{nome} vai para equipe {cor}')
```

16. Desafio – Iteração

Desafios

Desafio 01

Crie uma função que retorna os valores de duas listas de forma intercalada, mesmo quando as listas têm tamanho diferente. Por exemplo, dadas as listas:

```
L1 = [1, 2, 3]
L2 = ['a', 'b', 'c', 'd', 'e']
```

A função deve retornar:

```
[1, 'a', 2, 'b', 3, 'c', 'd', 'e']
```

Desafio 02

Imagine que você tem um restaurante com os seguintes itens no seu menu:

```
comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Strogonoff': 29.90,
    'Feijoada': 32.90,
}

bebidas = {
    'Água': 3.90,
    'Refrigerante': 5.90,
    'Suco': 7.90,
}
```

Crie um novo dicionário chamado combos onde cada chave é uma tupla contendo (comida, bebida), e o seu respectivo valor é o preço daquela combinação de comida e bebida.

Soluções

Desafio 01

Aqui usamos `itertools.zip_longest` para iterar sobre todas as listas recebidas. Passamos o valor `fillvalue=None` e usamos uma lógica adicional para evitar passar estes valores para dentro da lista resultado. Assim, apenas os valores originais das listas de entrada aparecem na lista resultante (ainda que uma lista seja menor que a outra):


```
import itertools

def retorna_intercalado(lista1, lista2):
    resultado = []
    for valor1, valor2 in itertools.zip_longest(lista1, lista2, fillvalue=None):
        if valor1 is not None:
            resultado.append(valor1)
        if valor2 is not None:
            resultado.append(valor2)
    return resultado

L1 = [1, 2, 3]
L2 = ['a', 'b', 'c', 'd', 'e']

resultado = retorna_intercalado(L1, L2)
print(resultado)
```

Desafio 02

Solução usando dois for loops aninhados:

```
comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Stroganoff': 29.90,
    'Feijoadá': 32.90,
}

bebidas = {
    'Água': 3.90,
    'Refrigerante': 5.90,
    'Suco': 7.90,
}

combo = {}

for chave_comida, preco_comida in comidas.items():
    for chave_bebida, preco_bebida in bebidas.items():
        chave_combo = (chave_comida, chave_bebida)
        preco_combo = preco_comida + preco_bebida
        combo[chave_combo] = round(preco_combo, 2)
print(combo)
```

Solução usando `itertools.product`:

```
import itertools

comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Stroganoff': 29.90,
```

```
        'Feijoadada': 32.90,
    }

    bebidas = {
        'Água': 3.90,
        'Refrigerante': 5.90,
        'Suco': 7.90,
    }

    combo = {}

    for chave_combo in itertools.product(comidas, bebidas):
        chave_comida, chave_bebida = chave_combo
        preco_combo = comidas[chave_comida] + bebidas[chave_bebida]
        combo[chave_combo] = round(preco_combo, 2)
    print(combo)
```

Dica: se passarmos o método `.items()` para `itertools.product()`, veja que recebemos uma sequência de tuplas, contendo cada combinação de pares chave-valor:

```
import itertools

comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Strogonoff': 29.90,
    'Feijoadada': 32.90,
}

bebidas = {
    'Água': 3.90,
    'Refrigerante': 5.90,
    'Suco': 7.90,
}

for tuplas in itertools.product(comidas.items(), bebidas.items()):
    print(tuplas)
```

Podemos então adaptar a lógica do loop para pegar a chave e preço do dicionário combo a partir dessa sequência de tuplas.

Aqui, usamos a mesma sintaxe de compreensão de lista para extrair o primeiro elemento de cada tupla para a chave_combo, e somar os segundos elementos para a variável preco_combo:

```
comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Strogonoff': 29.90,
    'Feijoadada': 32.90,
}

bebidas = {
    'Água': 3.90,
```

```
'Refrigerante': 5.90,
'Suco': 7.90,
}

combo = {}

for tuplas in itertools.product(comidas.items(), bebidas.items()):
    chave_combo = tuple(tup[0] for tup in tuplas) # Sintaxe de comp. de lista
    preco_combo = sum(tup[1] for tup in tuplas) # Sintaxe de comp. de lista
    combo[chave_combo] = round(preco_combo, 2)
print(combo)
```

Note que, como a sintaxe de compreensão de lista funciona para qualquer número de dicionários, não precisamos mais adaptar a lógica do código quando usamos mais dicionários:

```
import itertools

comidas = {
    'Prato Feito': 24.90,
    'Salada': 21.90,
    'Strogonoff': 29.90,
    'Feijoadada': 32.90,
}

bebidas = {
    'Água': 3.90,
    'Refrigerante': 5.90,
    'Suco': 7.90,
}

entradas = {
    'Batata Frita': 12,
    'Picadinho': 15,
}

combo = {}

for tuplas in itertools.product(comidas.items(), bebidas.items(), entradas.items()):
    chave_combo = tuple(tup[0] for tup in tuplas)
    preco_combo = sum(tup[1] for tup in tuplas)
    combo[chave_combo] = round(preco_combo, 2)
print(combo)
```

17. Argumentos arbitrários com `*args` e `**kwargs`

Funções com número arbitrário de argumentos

Até aqui, sempre que definimos uma função, tivemos que dizer exatamente quantos argumentos ela deve receber:

```
def somar(a, b):  
    return a + b
```

```
resultado = somar(1, 2)  
print(resultado)
```

```
resultado = somar(1, 2, 3)  
# TypeError: somar() takes 2 positional arguments but 3 were given
```

Porém, existem funções dentro do próprio Python que aceitam um **número arbitrário de argumentos**:

```
print('Olá')  
  
print('Olá', 'Python', '!!!')
```

Para reproduzirmos este comportamento, podemos utilizar o operador asterisco (*) para representar um número qualquer de argumentos:

```
def somar(*valores):  
    return sum(valores)
```

```
resultado = somar(1, 2)  
print(resultado)
```

```
resultado = somar(1, 2, 3)  
print(resultado)
```

`*args` e `**kwargs`

Há duas sintaxes usadas para criar funções que recebam um número variável de argumentos:

- `*var`: acumula argumentos em uma **tupla**, exceto aqueles que forem passados por palavra-chave.
- `**var`: acumula argumentos passados por palavra-chave em um **dicionário**.

Para esclarecer, argumentos por palavra-chave são aqueles que passarmos no formato `param="valor"`. No exemplo abaixo, `a` e `b` não foram passados por palavra-chave, enquanto `c` e `d` foram:

```
resultado = func(a, b, c=10, d=20)
```

A convenção é usar `*args` para representar os argumentos sem palavra-chave, e `**kwargs` (derivado de *keyword arguments*) para os argumentos com palavra-chave.

Abaixo, temos um exemplo de função que aceita *qualquer argumento*. Ela apenas exibe os argumentos passados a ela (distinguindo entre argumentos com e sem palavra-chave), mas poderíamos adaptá-la para fazer qualquer ação:

```
def exibe_argumentos(*args, **kwargs):
    print(f'Argumentos passados sem palavra-chave: {args}')
    print(f'Argumentos passados com palavra-chave: {kwargs}')

exibe_argumentos(1)
exibe_argumentos(param='valor')
exibe_argumentos(1, 2, x=3, y=[1, 2, 3])
```

Desmembrando argumentos na chamada de funções

Podemos também usar os operadores `*` e `**` para desmembrar sequências ou dicionários na hora de chamarmos uma função:

```
def exibe_argumentos(*args, **kwargs):
    print(f'Argumentos passados sem palavra-chave: {args}')
    print(f'Argumentos passados com palavra-chave: {kwargs}')

valores = [1, 2, 3]
dic = {
    'nome': 'Juliano',
    'idade': 30,
}

exibe_argumentos(valores, dic=dic) # Passa lista e dicionário sem desmembrá-los
exibe_argumentos(*valores, **dic) # Passa lista e dicionário com desmembramento
```

Pode parecer confuso, mas a primeira chamada da função (sem usar os operadores asterisco) equivale a:

```
exibe_argumentos([1, 2, 3], dic={'nome': 'Juliano', 'idade': 30})
```

Já a segunda chamada desmembra os valores antes de chamar a função, sendo equivalente a:

```
exibe_argumentos(1, 2, 3, nome=Juliano, idade=30)
```

O desmembramento de dicionários é particularmente útil para organizar argumentos de funções com diversos parâmetros, para chamá-las em outro momento:

```
params = {
    'nome': 'Juliano',
    'idade': 30,
    'senha': pegar_senha(),
    'filtro': None,
}
params['outro_argumento'] = pegar_argumento()

resultado = minha_funcao_complexa(**params)
```

Adaptando o desafio para qualquer número de listas

Este código abaixo é o desafio 1 da aula 16, adaptado para aceitar **qualquer número** de listas utilizando o `*args`.

```
def retorna_intercalado(*listas):
    resultado = []
    for valores in itertools.zip_longest(*listas, fillvalue=None):
        for valor in valores:
            if valor is not None:
                resultado.append(valor)
    return resultado

L1 = [1, 2, 3]
L2 = ['a', 'b', 'c', 'd', 'e']
L3 = ['xxx', 'yyy']
L4 = [-10, 55, 40, 20]

resultado = retorna_intercalado(L1, L2, L3, L4)
print(resultado)
```

18. Funções built-in relevantes

Vamos ver aqui sobre outras funções úteis de Python que fazem parte das funções embutidas (*built-in*). Isto significa que não é preciso importar nenhuma biblioteca para utilizá-las.

Função `isinstance`

Usado para **chechar o tipo de dado** de um valor qualquer:

```
valor = 2
if isinstance(valor, int):
    print(f'{valor} é do tipo int')
else:
    print(f'{valor} não é do tipo int')
```

Sempre prefira usar `isinstance` no lugar de checagens como `if type(valor) == int`, tanto por questão de clareza (essa função foi criada justamente para isso), quanto por praticidade (é possível checar “subtipos”, e mais de um tipo de uma vez):

```
valor = True
if isinstance(valor, int):
    print(f'{valor} é do tipo int')
else:
    print(f'{valor} não é do tipo int')

valor = 10.5
if isinstance(valor, (int, float)):
    print(f'{valor} é numérico')
else:
    print(f'{valor} não é numérico')
```

Funções `any` e `all`

São usadas para checar se algum valor (`any`) ou todos os valores (`all`) de uma sequência são verdadeiros:

```
booleanos = [True, False, True]

print(all(booleanos)) # Todos são True?
# output: False

print(any(booleanos)) # Algum é True?
# output: True
```

É usado com mais frequência junto de alguma expressão. Por exemplo, para checar se todos/algum dos valores é de certo tipo de dado:

```
valores = [1, 2, 2.5, 3]

if all(isinstance(valor, int) for valor in valores):
    print(f'Todos os valores de {valores} são do tipo int')
else:
    print(f'Nem todos os valores de {valores} são do tipo int')

if any(isinstance(valor, int) for valor in valores):
    print(f'Ao menos um valor de {valores} é do tipo int')
else:
    print(f'Nenhum valor de {valores} é do tipo int')
```

map

Usado para “mapear” cada valor de uma sequência, passando-os por uma função e retornando cada resultado como uma nova sequência:

```
def somar_dois(n):
    return n + 2

numeros = [1, 3, 6, 10]

mapa = map(somar_dois, numeros)
numeros_mais_dois = list(mapa)

print(numeros_mais_dois)
# output: [3, 5, 8, 12]
```

A função map não é tão utilizada atualmente, pois pode ser substituída por uma compreensão de lista:

```
```python
def somar_dois(n):
 return n + 2

numeros = [1, 3, 6, 10]

numeros_mais_dois = [somar_dois(n) for n in numeros]

print(numeros_mais_dois)
output: [3, 5, 8, 12]
```

Mesmo assim, para quem está acostumado a pensar em funções, ou com outras linguagens de programação, pode ser mais fácil de entender.



## **filter**

Mesma utilidade que `map`, mas ao invés de transformar cada valor, os filtra a partir de alguma condição (geralmente uma função que retorne `True` ou `False`):

```
def meu_filtro(n):
 return n > 5

numeros = [1, 3, 6, 10]

filtro = filter(meu_filtro, numeros)
maiores_que_cinco = list(filtro)

print(maiores_que_cinco)
output: [6, 10]
```

E assim como `map`, a função `filter` é frequentemente substituída por uma compreensão de lista:

```
```python  
def meu_filtro(n):  
    return n > 5  
  
numeros = [1, 3, 6, 10]  
  
maiores_que_cinco = [n for n in numeros if meu_filtro(n)]  
  
print(maiores_que_cinco)  
# output: [6, 10]
```

19. Funções Lambda (anônimas)

Como vimos anteriormente, podemos passar funções para definir o comportamento de `filter`:

```
def meu_filtro(x):  
    return x > 2  
  
filtro = filter(meu_filtro, [1, 2, 3, 4])  
print(list(filtro))  
# output: [3, 4]
```

Note que a função `meu_filtro` é bastante básica. Além disso, o único motivo que temos para defini-la é usar no filtro.

Seria útil ter a lógica de filtragem junto do código que gera o filtro. Podemos fazer isso transformando a função `meu_filtro` em uma função `lambda` (também conhecida por “função anônima”):

```
filtro = filter(lambda x: x > 2, [1, 2, 3, 4])  
print(list(filtro))  
# output: [3, 4]
```

O que é uma função Lambda?

As funções `lambda` são uma forma “resumida” de definir uma função em uma única linha. Note a sintaxe:

`lambda <ARGUMENTOS DA FUNÇÃO>: <LÓGICA DA FUNÇÃO>`

Traduzindo: definimos uma função anônima com a palavra-chave `lambda`, seguido dos argumentos que a função aceita, dois pontos (:), e o valor retornado pela função.

Por exemplo, a função a seguir:

```
def func(a, b, c):  
    return (a + b) > c
```

Pode ser reescrita como uma função `lambda` da forma:

```
lambda a, b, c: (a + b) > c
```

Note que a função não possui um “nome”, como é o caso quando criamos uma função com `def`. Justamente por isso, são rotineiramente chamadas de “funções anônimas”.

Para quê usar uma função Lambda?

Até podemos usar funções `lambda` como sendo funções “normais”, se as passarmos para dentro de uma variável:

```
def func(a, b, c):  
    return (a + b) > c  
  
resultado = func(1, 2, 5)  
print(resultado)  
# output: False  
  
func = lambda a, b, c: (a + b) > c  
resultado = func(1, 2, 5)  
print(resultado)  
# output: False
```

Porém, isso não é aconselhado, porque acaba poluindo o código: se algo se comporta como uma função a ser chamada, espero encontrá-la junto de um bloco com `def`, que simbolize isso.

A verdadeira utilidade de funções `lambda` é definir uma lógica para funções que utilizam outras funções como argumento, tais quais `map` e `filter`:

```
mapa = map(lambda x: str(x + 2), [1, 2, 3])  
print(list(mapa))  
# output = ['3', '4', '5']  
  
filtro = filter(lambda x: len(x) <= 3, ['xxx', 'y', 'abcde', '.....'])  
print(list(filtro))  
# output: ['xxx', 'y']
```

Usando funções anônimas em ordenamento

Outra função onde `lambdas` são muito utilizados é a função `sorted`, que também vimos anteriormente. Podemos passar uma função para o argumento `key`, que ditará a lógica de ordenação dos elementos.

O código abaixo ordena os elementos de uma lista com strings e ints:

```
lista = ['abc', 1, 4, 6.5, '.', '22']  
  
lista_ordenada = sorted(lista, key=lambda x: len(x) if isinstance(x, str) else x)  
print(lista_ordenada)  
# output: [1, '.', '22', 'abc', 4, 6.5]
```

Normalmente, não é possível comparar strings e ints, porém criamos uma função `lambda` capaz de fazer isso. Sua lógica é:

- Se o elemento da lista for um string, use o comprimento do string para comparação
- Caso contrário, use o próprio elemento

Ou seja, na hora de ordenar os valores, o valor `'abc'` terá valor 3, e o valor `'22'` terá valor 2. Isso devido à lógica que determinamos com a função `lambda`!

Apenas para deixar claro: é perfeitamente possível reproduzir o exemplo acima usando uma função tradicional, com `def`:

```
def ordenamento(x):  
    return len(x) if isinstance(x, str) else x
```

```
lista = ['abc', 1, 4, 6.5, '.', '22']
```

```
lista_ordenada = sorted(lista, key=ordenamento)  
print(lista_ordenada)  
# output: [1, '.', '22', 'abc', 4, 6.5]
```

Contudo, como estamos interessados apenas em criar uma lógica para este ordenamento específico (e não em ter uma função própria de ordenamento que será usada múltiplas vezes), em muitos casos é mais conciso usar `lambdas` nessas situações!

20. Decoradores

Em poucas palavras, decoradores **modificam a forma com que uma função opera**, porém sem mudar a lógica da função em si.

Podemos pensar em decoradores como algo que “adiciona utilidade extra” a alguma função já existente. Antes de entendermos como usar decoradores, precisamos entender algumas coisas sobre funções.

Funções são objetos em Python

Isto significa que elas podem ser **assinaladas para alguma variável**:

```
def func():  
    return 2  
  
minha_funcao = func  
retorno = minha_funcao()  
  
print(retorno)  
# output: 2
```

Passadas como argumento de outra função:

```
def func():  
    return 2  
  
def exibe_func(f):  
    print(f'Objeto de função recebido: {f}')  
    print(f'Nome da função: "{f.__name__}"')  
  
exibe_func(func)
```

Definidas dentro de outra função - inclusive aceitando argumentos que vêm da “função externa”:

```
def func_externa(x):  
    def func_interna():  
        return x + 2  
  
    valor = func_interna()  
    return valor  
  
resultado = func_externa(3)  
print(resultado)  
# output: 5
```

Retornadas a partir de outra função:

```
def pega_func_interna(x):  
    def func_interna(y):  
        return x + y + 2  
    return func_interna
```

```
f = pega_func_interna(1)  
resultado = f(2)  
print(resultado)  
# output: 5
```

Nestes exemplos acima, não aconteceu nada de muito “emocionante” - apenas passamos funções simples entre algumas variáveis e outras funções. Contudo, são estas propriedades que torna possível o uso e criação de decoradores em Python.

Usando decoradores

A próxima aula discute como usar as propriedades apresentadas para criar um decorador. Porém para simplesmente **usar** um decorador, não precisamos necessariamente conhecê-las.

Podemos adicionar um decorador a uma função simplesmente adicionando um `@nome_do_decorador` diretamente acima da definição da função.

O exemplo abaixo utiliza dois decoradores importados a partir do arquivo `meus_decoradores.py` (disponível nos materiais de aula, e também no bloco de código ao final desta aula). Não se preocupe em como os decoradores funcionam aqui: **foque em como são usados!**

O decorador `fazer_duas_vezes` roda uma mesma função uma segunda vez, com apenas uma única execução:

```
from meus_decoradores import fazer_duas_vezes
```

```
@fazer_duas_vezes  
def printar_exclamações():  
    print('!!!!')
```

```
printar_exclamações()  
# output:  
# !!!!  
# !!!!
```

Já o decorador `medir_tempo` exibe quanto tempo uma função leva para executar (aqui usamos `time.sleep` apenas para fazer a função durar algum tempo, já que ela é curta):

```
import time

from meus_decoradores import medir_tempo

@medir_tempo
def dizer_oi(nome):
    time.sleep(3)
    return f'Olá, {nome}!'

resultado = dizer_oi("Juliano")
print(resultado)
# # output:
# A função dizer_oi levou 3.00023 segundos para executar.
# Olá, Juliano!
```

Note que, depois de decorarmos as funções com `@nome_do_decorador`, adicionamos funcionalidade a cada uma delas - sem que isso modificasse a função original, nem o código que chama a função!

Na “vida real”, encontramos muitas bibliotecas que utilizam decoradores. Por exemplo, a biblioteca Dash, que usamos em diversos projetos da Asimov Academy, utiliza decoradores para indicar que uma função é um `callback`, isto é, que ela será executada em reação a algum clique no dashboard.

Se tudo o que quisermos é apenas usar decoradores, basta seguir as instruções das bibliotecas em que foram definidas (qualquer biblioteca relativamente boa deve indicar como seus decoradores devem ser usados). Para quem quiser saber como, de fato, um decorador funciona, vamos para a próxima aula!

Anexo: script `meus_decoradores.py`

Para testar o código dessa aula, coloque o código abaixo no arquivo `meus_decoradores.py`, dentro da mesma pasta do seu script principal:

```
import time

def fazer_duas_vezes(f):
    def meu_pacote(*args, **kwargs):
        _ = f(*args, **kwargs)
        retorno = f(*args, **kwargs)
        return retorno
    return meu_pacote

def medir_tempo(f):
    def meu_pacote(*args, **kwargs):
        inicio = time.time()
```

```
    retorno = f(*args, **kwargs)
    final = time.time()
    print(f'A função {f.__name__} levou {round(final - inicio, 5)} segundos para executar.')
    return retorno
return meu_pacote
```


21. Como decoradores funcionam?

Criando um decorador

Vamos partir de uma função simples:

```
def dizer_oi(nome):  
    return f'Olá, {nome}!'
```

```
resultado = dizer_oi('Juliano')  
print(resultado)  
# output: Olá, Juliano!
```

Se quisermos fazer com que essa função retorne o string em letras maiúsculas, poderíamos fazer algo como:

```
def dizer_oi(nome):  
    return f'OLÁ, {nome}!'
```

```
resultado = dizer_oi('Juliano').upper()  
print(resultado)  
# output: OLÁ, JULIANO!
```

O problema aqui é que, se quisermos inserir esta mudança, precisaríamos alterar toda vez em que chamamos a função `dizer_oi`. Se ela for chamada 100 vezes pelo nosso script, precisamos encontrar e alterar todas as 100 chamadas da função!

Também poderíamos alterar o corpo da função, o que é bastante simples nesse caso. Mas imagine que a funcionalidade que você quer adicionar na função não seja algo tão simples, ou até mesmo que você não tenha acesso ou não queira encostar no código da função (por exemplo: ele veio de alguma biblioteca).

Neste caso, conseguimos fazer um pequeno truque para “empacotar” a função `dizer_oi` em outra função. Vamos chamá-la de `meu_decorador`:

```
def dizer_oi(nome):  
    return f'Olá, {nome}!'
```

```
def meu_decorador(func):  
    def meu_pacote(nome):  
        retorno = func(nome=nome)  
        return retorno.upper()  
    return meu_pacote
```

```
meu_pacote = meu_decorador(dizer_oi)
```

Preste atenção no que fizemos:

- Criamos uma função chamada `meu_decorador`, que aceita como argumento uma função qualquer, chamada `func`.
- Dentro da função `meu_decorador`, é criada uma nova função, que chamamos de `meu_pacote`. Essa função recebe um argumento `nome`.
- No corpo da função `meu_pacote`, escrevemos que ela é responsável por chamar a função `func`, passando o argumento `nome`. O valor retornado por `func` é, por sua vez, transformado com o método `.upper()`.

Ao chamarmos nossa nova função `meu_pacote`, a função original `dizer_oi` é devidamente executada com os argumentos entregues à `meu_pacote`, além do ajuste no valor retornado com `.upper()`:

```
print(meu_pacote('João'))  
# output: OLÁ, JOÃO!  
  
print(meu_pacote('Ana'))  
# output: OLÁ, ANA!  
  
print(meu_pacote('Mateus'))  
# output: OLÁ, MATEUS!
```

Esta é, basicamente, o “truque” por trás de um decorador. Há apenas alguns poucos ajustes a serem feitos, para que nossa função `meu_decorador` se comporte de fato como um decorador “padrão” em Python.

Ajustando o número de argumentos

A nossa função `meu_pacote` aceita apenas um argumento `nome`. Fizemos isso para que os argumentos batessem com aqueles esperados pela função `dizer_oi`.

Contudo, a maioria dos decoradores não tem como saber qual os argumentos das funções que irão decorar (isso porque, em geral, eles devem conseguir decorar qualquer função).

Conseguimos contornar isso usando os operadores `*args` e `**kwargs` que vimos anteriormente. Dessa forma, quaisquer argumentos passados para `meu_pacote` serão passados adiante para a função sendo decorada:

```
def dizer_oi(nome):  
    return f'Olá, {nome}!'  
  
def meu_decorador(func):  
    def meu_pacote(*args, **kwargs):  
        retorno = func(*args, **kwargs)
```

```
        return retorno.upper()
    return meu_pacote
```

```
meu_pacote = meu_decorador(dizer_oi)
```

Com este pequeno ajuste, garantimos que os argumentos que a função original aceita serão os mesmos após ela ter sido decorada!

Mudando o nome da variável `meu_pacote` para o nome da função original

Este “truque” fará com que a função original seja “sobrescrita” pela função `meu_pacote`. Ou seja, do ponto de vista prático, é como se tivéssemos “atualizado” a funcionalidade de `dizer_oi`, mas continuando a chamá-la da mesma forma:

```
def dizer_oi(nome):
    return f'Olá, {nome}!'

def meu_decorador(func):
    def meu_pacote(*args, **kwargs):
        retorno = func(*args, **kwargs)
        return retorno.upper()
    return meu_pacote
```

```
dizer_oi = meu_decorador(dizer_oi)
```

Após a última linha acima, chamadas subsequentes à função usarão a função decorada:

```
def dizer_oi(nome):
    return f'Olá, {nome}!'

def meu_decorador(func):
    def meu_pacote(*args, **kwargs):
        retorno = func(*args, **kwargs)
        return retorno.upper()
    return meu_pacote

print(dizer_oi('Juliano'))

dizer_oi = meu_decorador(dizer_oi)

print(dizer_oi('Juliano'))
```

“Açúcar sintático”: o uso da sintaxe @decorador

Funcionalmente, o decorador está pronto. Contudo, ainda não usamos a sintaxe @decorador para defini-lo. Na realidade, esta sintaxe nada mais é do que uma outra forma de escrever a linha que já escrevemos anteriormente.

Os dois blocos abaixo são equivalentes:

```
# Definindo o decorador
def meu_decorador(func):
    def meu_pacote(*args, **kwargs):
        retorno = func(*args, **kwargs)
        return retorno.upper()
    return meu_pacote

# Decorando a função sem usar @meu_decorador
def dizer_oi(nome):
    return f'Olá, {nome}!'

dizer_oi = meu_decorador(dizer_oi)
```

```
# Decorando a função usando @meu_decorador
@meu_decorador
def dizer_oi(nome):
    return f'Olá, {nome}!'
```

O ponto-chave aqui é que colocar @meu_decorador acima de uma função qualquer func é **exatamente o mesmo** que escrever func = meu_decorador(func).

A sintaxe @meu_decorador é apenas uma forma mais óbvia e legível de atingir o mesmo objetivo, pois fica evidente ao leitor que a função está decorada sem precisar procurar embaixo da definição da função.

Por ser apenas um recurso sintático de Python, o uso do @meu_decorador é também chamado de “açúcar sintático”.

22. Utilidades do módulo `functools`

O módulo `functools` possui algumas utilidades e decoradores que podemos usar.

`functools.cache`

Decorador usado para criar um cache de resultados de uma função. Cada vez que a função é chamada com o mesmo argumento, ela não é executada novamente; no lugar disso, o resultado já computado no cache é retornado.

No exemplo abaixo, a função `fatorial(n)` usa recursão para chamar a si mesmo, até chegar no valor “base” de `fatorial(1)`. Ao longo do caminho, os valores intermediários são mantidos em cache. Podemos ver isso acompanhando as mensagens sendo printadas (elas são omitidas quando um valor está no cache):

```
import functools

@functools.cache
def fatorial(n):
    print(f'Valor de n: {n}')
    if n == 1:
        return 1
    return n * fatorial(n-1)

print(fatorial(5))
print(fatorial(3))
print(fatorial(2))
print(fatorial(8))
print(fatorial(7))
```

`functools.partial`

Função usada para criar uma função “parcial”, isto é, uma função com argumentos previamente preenchidos.

O primeiro argumento é a função a ser preenchida, e os demais são os argumentos e seus valores (que podem ser passados por palavra-chave ou não):

```
import functools

def somar(a, b):
    return a + b

somar_dois = functools.partial(somar, 2)
```

```
print(somar_dois)
print(somar_dois(3))
print(somar_dois(-2))

somar_cinco = functools.partial(somar, b=5)
print(somar_cinco)
print(somar_cinco(5))
print(somar_cinco(500))
```

Ideia de uso: criar função “pré-pronta” de ordenação:

```
import functools

ordenar_por_tamanho = functools.partial(sorted, key=lambda x: len(x) if isinstance(x, str)
    ↪ else x)

lista = ['abc', 1, 4, 6.5, '.', '22']

lista_ordenada = ordenar_por_tamanho(lista)
print(lista_ordenada)
# output: [1, '.', '22', 'abc', 4, 6.5]
```

23. Desafio – Funções

Desafios

Desafio 01

Com base no `for` loop abaixo:

```
valores = [1, 2, 3, 5, 10]

quadrados_maiores_que_tres = []
for valor in valores:
    if valor > 3:
        quadrado = valor ** 2
        quadrados_maiores_que_tres.append(quadrado)

print(quadrados_maiores_que_tres)
# output: [25, 100]
```

Crie uma compreensão de lista que gera a mesma lista `quadrados_maiores_que_tres`. Em seguida, use as funções `map` e `filter` para fazer a mesma coisa.

Desafio 02

Crie uma função chamada `multiplicar_por`. O que esta função faz é retornar uma nova função `f` capaz de multiplicar um número qualquer pelo fator `n` passado à função `multiplicar_por`.

Exemplo de uso:

```
dobrar = multiplicar_por(2)
print(dobrar(3))
# output: 6
print(dobrar(10))
#output: 20

vezes_cinco = multiplicar_por(5)
print(vezes_cinco(3))
# output: 15
print(vezes_cinco(10))
# output: 50
```

Soluções

Desafio 01

```
valores = [1, 2, 3, 5, 10]
```

```
# Por compreensão de lista
quadrados_maiores_que_tres = [valor ** 2 for valor in valores if valor > 3]
print(quadrados_maiores_que_tres)
# output: [25, 100]

# Por map e filter
quadrados_maiores_que_tres = list(map(lambda x: x**2, filter(lambda x: x > 3, valores)))
print(quadrados_maiores_que_tres)
# output: [25, 100]
```

Desafio 02

Os exemplos abaixo representam diferentes formas de implementar a solução:

```
import functools

# Usando functools.partial para preencher um dos
# argumentos de uma função de multiplicação
def mult(a, b):
    return a * b

def multiplicar_por(n):
    return functools.partial(mult, b=n)

# Mesma lógica anterior, porém criando mult como uma
# função lambda dentro de multiplicar_por
def multiplicar_por(n):
    return functools.partial(lambda a, b: a * b, b=n)

# Usando funções aninhadas
# (lembre-se: funções são objetos!)
def multiplicar_por(n):
    def f(x):
        return x * n
    return f
```


24. Interação com arquivos e sistema operacional

Há muitos módulos da biblioteca padrão que podemos usar para interagir com o nosso sistema operacional, e com arquivos de forma geral. Vamos entender como usá-los aqui:

Módulo `os`

Usado para interação direta com o sistema operacional e arquivos:

```
import os

# Lê pasta atual
dir_atual = os.getcwd()
print(f'Pasta atual: {dir_atual}')

# Lê arquivos na pasta atual
arquivos = os.listdir()
print(f'Arquivos na pasta atual: {arquivos}')

# Tenta mudar de pasta
pasta = 'Desktop'
print(f'Tentando ir para pasta {pasta}...')

if not os.path.isdir(pasta): # Pasta não existe, impossível entrar nela
    caminho_completo = os.path.join(dir_atual, pasta)
    print(f'Pasta {caminho_completo} não existe!')
else: # Pasta existe, é possível entrar nela
    os.chdir(pasta)
    dir_atual = os.getcwd()
    print(f'Pasta atual: {dir_atual}')
```

O módulo `os` não é resumido a apenas arquivos e pastas. Podemos acessar variáveis de ambiente que estão rodando no nosso sistema também:

```
import os

print(os.environ)
# output: dicionário com todas as variáveis de ambiente

print(os.environ['PATH'])
# output: valor da variável de ambiente PATH
```

Módulo `pathlib`

Alternativa mais “moderna” ao `OS`. Usamos objetos `Path` que são um pouco mais fáceis de trabalhar do que as funções do `os`:

```
from pathlib import Path

# Criando um Path na pasta atual (relativo e absoluto)
p = Path('.')
print(f'''
--- Pasta atual ---
Caminho relativo: {p}
Caminho absoluto: {p.absolute()}
''')

# Pegando arquivos do caminho
arquivos = list(p.iterdir())
print(f'Arquivos na pasta atual:\n{arquivos}')

# Criando caminhos e checando se existe
caminho_desktop = p / 'Desktop'
print(f'Caminho para pasta {caminho_desktop} existe? {caminho_desktop.exists()}')
```

Módulo `shutil`

Usado para fazer algumas operações com arquivos, como recortar, copiar e colar.

```
from pathlib import Path
import shutil

nome_arquivo = 'notas.txt'

with open(nome_arquivo, 'w') as arquivo_texto:
    arquivo_texto.write('Estou aprendendo Python!')

pasta = Path('.').absolute() / 'minhas_notas'
pasta.mkdir(exist_ok=True)

print(f'Movendo arquivo {nome_arquivo} para pasta {pasta}...')
shutil.move(nome_arquivo, pasta / nome_arquivo)
print('Finalizado!')
```

Cuidado ao usar `shutil`: ele não pede nenhuma mensagem de confirmação para fazer qualquer operação, *inclusive deletar arquivos*!

Confira se você está fazendo exatamente o que pensa que está fazendo (use mensagens de `print` por exemplo), antes de executar “pra valer”.

Quer mais informações sobre arquivos e caminhos?

O curso `Lendo e Escrevendo Arquivos` da Asimov Academy aprofunda bastante no uso do `Path` e `shutil` para construir caminhos e interagir com arquivos!

25. Trabalhando com tempo, datas e horas

Python possui módulos na biblioteca padrão para interagirmos com datas e tempo.

Criando e manipulando datas com `datetime`

Esta é a principal biblioteca para manipular datas e horas! Quando você tiver que fazer qualquer tarefa relacionada a estes elementos (contar intervalos de tempo, transformar string para data, determinar que dia é final de semana, ...), comece por aqui.

Criando data e hora

```
import datetime

# Cria objetos de dia e hora
data = datetime.date(2022, 1, 5)
hora = datetime.time(18, 6, 36)

print(f'Data: {data}, hora: {hora}')

# Cria um único objeto representando um dia e hora
instante = datetime.datetime(2022, 1, 5, 18, 6, 36)
print(f'Instante: {instante}')
```

Não conseguimos criar datas “impossíveis”:

```
import datetime

datetime.datetime(2022, 13, 10)
# ValueError: month must be in 1..12

datetime.datetime(2022, 12, 100)
# ValueError: day is out of range for month
```

O `datetime` considera até os dias de cada mês, e casos mais complicados como anos bissextos:

```
import datetime

datetime.datetime(2022, 2, 30)
# ValueError: day is out of range for month

datetime.datetime(2020, 2, 29)
# Funciona porque é ano bissexto!

datetime.datetime(2021, 2, 29)
# ValueError: day is out of range for month
```

Lendo e formatando datas de a partir de texto

Podemos usar `datetime.strftime` para formatar a representação de um `datetime`. Isso nos permite criar um string customizado para exibir a data:

```
import datetime

instante = datetime.datetime(2022, 1, 5, 18, 6, 36)

fmt01 = instante.strftime('Dia %d/%m/%Y, às %H horas e %M minutos')
print(fmt01)

fmt02 = instante.strftime('[ %Y-%m-%d --- %H:%M:%S ]')
print(fmt02)
```

Note que os caracteres logo em seguida do % possuem significado especial: eles representam o valor a ser “extraído” do `datetime`.

Algumas das correspondências estão representadas abaixo (usando o dia 01/12/2023 e hora 12:05:16 como referência):

Código	Significado	Exemplo
%Y	Ano, 4 caracteres	2023
%y	Ano, 2 caracteres	23
%m	Mês	12
%d	Dia	1
%H	Hora	12
%M	Minuto	05
%S	Segundos	16

De forma análoga ao método `datetime.strftime`, podemos usar o método `datetime.strptime` para criar um `datetime` a partir de um string qualquer. Em geral, temos que indicar cada um dos campos a serem “coletados” pelo `datetime` resultante:

```
import datetime

formato_iso = "2023-12-31T08:45:30"

data = datetime.datetime.strptime(formato_iso, "%Y-%m-%dT%H:%M:%S")
print(data)

linha_txt = "Aconteceu no dia 22/10/23, às 16h30min"
data = datetime.datetime.strptime(linha_txt, "Aconteceu no dia %d/%m/%y, às %Hh%Mmin")
print(data)
```

Aritmética com datas

Se subtrairmos um `datetime` do outro, obtemos um objeto `timedelta`, que representa a diferença de tempo entre os dois instantes:

```
import datetime

agora = datetime.datetime.today()  # Retorna o dia atual
ano_2020 = datetime.datetime(2020, 1, 1)

delta = agora - ano_2020
print(delta)

# Resultado em horas
print(delta.total_seconds() / 3600)
```

Módulo `zoneinfo`: lidando com fusos horários

Em todos esses casos, os nossos objetos `datetime` não possuíam a informação de fuso horário.

Podemos adicionar fuso horário a um `datetime` usando objetos `ZoneInfo` (do módulo `zoneinfo`), que representam um fuso específico. Então passamos este objeto ao criar qualquer `datetime` pelo argumento `tz`:

```
import datetime
from zoneinfo import ZoneInfo

fuso_sp = ZoneInfo('America/Sao_Paulo')
fuso_paris = ZoneInfo('Europe/Paris')

agora_sp = datetime.datetime.now(tz=fuso_sp)
agora_paris = datetime.datetime.now(tz=fuso_paris)

print(f'Agora são:\n{agora_sp} em SP\n{agora_paris} em Paris')
print(f'Delta de tempo: {agora_paris - agora_sp}')
```

Note que a “diferença” de tempo entre as duas regiões deu praticamente zero, embora estejam com horas diferentes no relógio!

Os strings usados no `ZoneInfo` são específicos (apenas algumas cidades e regiões são válidas). Para conferir todos os strings possíveis, use o código abaixo:

```
import zoneinfo

print(zoneinfo.available_timezones())
```

Usando zoneinfo em Windows

Atenção! Em alguns sistemas operacionais (em especial em Windows), não há informação “nativa” dentro do sistema sobre fusos horários. Neste caso, é necessário instalar a biblioteca tzdata via pip, com o comando:

```
pip install tzdata
```

O restante do código permanece o mesmo, incluindo a importação de zoneinfo (e não de tzdata).

26. Manipulação e busca em textos

Módulo `locale`

Usado para definir regras de **localização**, isto é, de exibição de nomes e valores de acordo com uma região. Para nós, isso é particularmente útil para configurar a exibição de números e valores monetários sem perder tempo formatando cada caractere:

```
import locale

# Configuração inicial
local_pc = locale.setlocale(locale.LC_ALL, '')
print(local_pc)
# output: 'pt_BR.UTF-8'

x = 12.5
print(locale.str(x))
# output: 12,5
print(locale.currency(x))
# output: R$ 12,20

y = 1279.95402
print(locale.currency(y, grouping=True))
# output: R$ 1.279,95
```

A primeira chamada a `locale.setlocale` é usada para o computador “descobrir” qual a sua linguagem a partir do seu sistema operacional.

Mesmo se o seu sistema não tiver Português como sua linguagem padrão, é possível pedir para utilizá-la no `locale` desde que o suporte para a língua esteja instalada. Neste caso, é preciso usar um código semelhante a:

```
import locale

locale.setlocale(locale.LC_ALL, 'pt_BR.UTF-8')
```

O string `"pt_BR.UTF-8"` pode variar conforme o sistema operacional e sua versão, e nem sempre é fácil descobrir qual o nome correto. Para evitar dores de cabeça, o ideal é já ter o computador configurado com a linguagem correta!

Expressões regulares com o módulo `re`

Imagine que você possui o seguinte texto:

```
texto = """
Nome: Marcos | Idade: 30 | CPF: 012.345.678-90 | País de origem: Brasil
Nome: Ana | Idade: 28 | CPF: 098.765.432-10 | País de origem: Brasil
```

```
Nome: Isadora | Idade: Não informado | CPF:090.080.070-65 | País de origem: Brasil
Nome: Guilherme | Idade: 21 | CPF: 111.222.333-45 | País de origem: Brasil
"""
```

É preciso extrair o CPF de cada pessoa de maneira consistente.

Note que há problemas de formatação: às vezes, há espaços em branco a mais ou a menos, o que impede usarmos métodos como `split()` nos espaços ou buscar por caracteres específicos. Qual lógica podemos aplicar?

Para situações como essa (criar uma lógica para extrair texto), a melhor alternativa é usar uma expressão regular (módulo `re` de Python).

O que é uma expressão regular?

Uma expressão regular, ou **regex**, é uma sintaxe própria, que não existe apenas em Python. Ela serve para **criar um padrão a ser buscado** em algum texto, como por exemplo números ou palavras.

Os exemplos abaixo demonstram alguns padrões possíveis de ser construídos com regex. Usamos sempre a função `re.search` para encontrar o primeiro “match” do padrão com o texto:

```
import re

padrao = '[0-9]+'
texto = "Escrevi este texto há 8 anos atrás."
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(22, 23), match='8'>
```

O objeto Match

A função `re.search` retorna um objeto `Match`. Este objeto guarda a informação do valor encontrado, e de sua posição no string.

Podemos pegar o valor encontrado com os métodos `Match.group()`, ou usando os índices fornecidos por `Match.start()` e `Match.end()`:

```
import re

padrao = '[0-9]+'
texto = "Escrevi este texto há 8 anos atrás."
match = re.search(padrao, texto)

print(match.group())
# output: 8

print(match.start(), match.end())
```



```
# output: 22 23

fatia = texto[match.start(): match.end()]
print(fatia)
# output: 8
```

Note que, se o padrão não for encontrado, a função `re.search` retorna o valor `None`:

```
import re

padrao = '[0-9]'
texto = "Escrevi este texto há oito anos atrás."
match = re.search(padrao, texto)

print(match)
# output: None
```

Criando expressões regulares com metacaracteres

Alguns caracteres tem funcionalidade especial dentro de um padrão de regex, sendo chamados de **metacaracteres**.

A tabela abaixo mostra os alguns caracteres que podem ser usados em um regex:

Caractere	Significado no regex
.	Representa qualquer caractere
^	Representa o começo do string
\$	Representa o final do string
*	Zero ou mais repetições do caractere/sequência anterior
+	Uma ou mais repetições do caractere/sequência anterior
?	Zero ou uma repetição do caractere/sequência anterior
{ }	Número específico de repetições do caractere/sequência anterior
[]	Representa um grupo específico de caracteres (ex: <code>[0-9]</code> , <code>[a-z]</code> , <code>[A-Z]</code>)

Veja alguns exemplos de uso abaixo:

```
import re

texto = "O ônibus saiu com 45min de atraso, Estava previsto para sair 16h30"

padrao = '[0-9]{2}' # Exatamente 2 dígitos seguidos
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(18, 20), match='45'>

padrao = '[0-9]{2}h' # Match anterior seguido de caractere 'h'
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(61, 64), match='16h'>

padrao = '[0-9]{2}$' # Exatamente 2 dígitos no final da frase
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(64, 66), match='30'>

padrao = 'E.' # Caractere "E" seguido de qualquer caractere
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(35, 37), match='Es'>

padrao = 'E.*' # Caractere "E" seguido de quaisquer caracteres (plural!)
match = re.search(padrao, texto)
print(match)
# output: <re.Match object; span=(35, 66), match='Estava previsto para sair 16h30'>
```

Um aprofundamento maior em regex vai além do escopo deste curso, mas é importante saber que estes conceitos existem para poder buscá-los quando for necessário. Há muitos sites e ferramentas que ajudam você a construir um regex, como o [RegExr](#).

Buscando pelos CPFs

Testando no site acima, conseguimos ver que o regex para pegar o CPF é

```
[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}
```

onde:

- `[0-9]{3}` representa 3 dígitos de 0 a 9
- `\.` representa um caractere `.` (precisamos da contra barra para indicar que é o caractere em si, e não um metacaractere)
- `-[0-9]{2}` representa um caractere de hífen seguido de dois dígitos

O código completo em Python para extrair os CPFs do texto é:

```
import re
```

```
texto = """
Nome: Marcos | Idade: 30 | CPF: 012.345.678-90 | País de origem: Brasil
Nome: Ana | Idade: 28 | CPF: 098.765.432-10 | País de origem: Brasil
Nome: Isadora | Idade: Não informado | CPF: 090.080.070-65 | País de origem: Brasil
Nome: Guilherme | Idade: 21 | CPF: 111.222.333-45 | País de origem: Brasil
"""

padrao = '[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}'
match = re.search(padrao, texto)

print(match)
# output: <re.Match object; span=(34, 48), match='012.345.678-90'>
```

Note que o `re.search` retorna apenas um único objeto `match`, representando o primeiro resultado encontrado. No lugar disso, podemos usar `re.findall` para retornar todos os resultados em uma lista:

```
import re

texto = """
Nome: Marcos | Idade: 30 | CPF: 012.345.678-90 | País de origem: Brasil
Nome: Ana | Idade: 28 | CPF: 098.765.432-10 | País de origem: Brasil
Nome: Isadora | Idade: Não informado | CPF: 090.080.070-65 | País de origem: Brasil
Nome: Guilherme | Idade: 21 | CPF: 111.222.333-45 | País de origem: Brasil
"""

padrao = '[0-9]{3}\.[0-9]{3}\.[0-9]{3}-[0-9]{2}'
cpfs = re.findall(padrao, texto)

print(cpfs)
# output: ['012.345.678-90', '098.765.432-10', '090.080.070-65', '111.222.333-45']
```

Sucesso!

27. Desafio – Biblioteca padrão

Desafios

Desafio 01

Crie uma função chamada `ler_datas`, que recebe um texto qualquer e extrai todas as datas que estejam escritas no formato DD/MM/AAAA (como objetos `datetime`). Use o texto abaixo como exemplo:

```
texto = """
A reunião está marcada para o dia 15/03/2023.
Lembre-se de entregar o relatório até 28/02/2023.
O evento acontecerá em 10/04/2023 no auditório principal.
"""
```

Desafio 02

Crie uma função chamada `diff_tempo`, que aceita dois strings no formato HH:MM:SS e retorna a diferença de tempo entre eles em um string de mesmo formato.

Exemplo de uso:

```
inicio = '08:34:21'
fim = '13:55:09'

diff = diff_tempo(inicio, fim)
print(diff)
# output:
# 5:20:48
```

Soluções

Desafio 01

```
import datetime
import re

texto = """
A reunião está marcada para o dia 15/03/2023.
Lembre-se de entregar o relatório até 28/02/2023.
O evento acontecerá em 10/04/2023 no auditório principal.
"""

padrao = '[0-9]{2}/[0-9]{2}/[0-9]{4}'
```

```
datas = []
for data_str in re.findall(padrao, texto):
    data = datetime.datetime.strptime(data_str, '%d/%m/%Y')
    datas.append(data)

print(datas)
# output:
# [
#     datetime.datetime(2023, 3, 15, 0, 0),
#     datetime.datetime(2023, 2, 28, 0, 0),
#     datetime.datetime(2023, 4, 10, 0, 0),
# ]
```

Reduzindo a solução acima para uma compreensão de lista:

```
import datetime
import re

texto = """
A reunião está marcada para o dia 15/03/2023.
Lembre-se de entregar o relatório até 28/02/2023.
O evento acontecerá em 10/04/2023 no auditório principal.
"""

padrao = '[0-9]{2}/[0-9]{2}/[0-9]{4}'

datas = [
    datetime.datetime.strptime(data_str, '%d/%m/%Y')
    for data_str in re.findall(padrao, texto)
]

print(datas)
# output:
# [
#     datetime.datetime(2023, 3, 15, 0, 0),
#     datetime.datetime(2023, 2, 28, 0, 0),
#     datetime.datetime(2023, 4, 10, 0, 0),
# ]
```

Desafio 02

```
import datetime

def diff_tempo(inicio, fim):
    formato = '%H:%M:%S'
    dt_inicio = datetime.datetime.strptime(inicio, formato)
    dt_fim = datetime.datetime.strptime(fim, formato)
    delta = dt_fim - dt_inicio
    delta_seconds = delta.total_seconds()

    # Calculando "manualmente" cada elemento
    h = delta_seconds // 3600
```

```
m = (delta_seconds % 3600) // 60
s = delta_seconds % 60
return f'{int(h)}:{int(m)}:{int(s)}'

# Usando formatação automática ao transformar timedelta em str
# (comente a seção anterior para usar esta versão)
return str(delta)

inicio = '08:34:21'
fim = '13:55:09'

diff = diff_tempo(inicio, fim)
print(diff)
# output:
# 5:20:48
```

28. Os erros mais comuns em Python

Nesta aula, vamos entender alguns dos erros mais comuns em Python, abordando sua causa e como os evitar.

1. IndentationError

Acontece quando a **indentação** (o espaço em branco à esquerda) não está correta, seja por ela não estar consistente, ou não ter sido aplicada corretamente conforme as regras de Python.

- Indentação inconsistente em múltiplas linhas:

```
x = 1
  y = 2 # IndentationError: unexpected indent
```

- Indentação faltando em alguma linha que a requer (por exemplo, em for loops):

```
for valor in [1, 2, 3]:
print(valor) # IndentationError: expected an indented block after 'for' statement on line 1
```

2. SyntaxError

Problemas com a **sintaxe** (regras de escrita) de Python. Algumas das ocorrências mais comuns são:

- Parênteses / aspas / chaves ... não fechadas corretamente:

```
print('olá' # SyntaxError: '(' was never closed

print('olá) # SyntaxError: unterminated string literal (detected at line 1)
```

- Falta de dois pontos ou parênteses em blocos que necessitam:

```
for valor in (1, 2, 3) # SyntaxError: expected ':'
    print(valor)

def minha_funcao # SyntaxError: expected '('
    print('Esta é minha função!')
```

- Uso de caracteres inválidos ou palavras-chave reservadas como variáveis

```
meu!valor = 2 # SyntaxError: invalid syntax

for = 2 # SyntaxError: invalid syntax
```

3. NameError

Este erro acontece sempre quando você tenta **acessar uma variável que não existe**:

```
x = 1
y = 2

print(x)
print(y)
print(z)  # NameError: name 'z' is not defined
```

Os motivos clássicos para este tipo de erro são:

- Usar uma variável antes de criá-la
- Erro de digitação no nome da variável
- Variável que não foi importada

Também pode acontecer quando tentamos acessar uma variável fora do seu escopo de criação. No exemplo abaixo, a variável `resultado` existe apenas dentro da função. Não temos acesso a ela mesmo se chamarmos a função:

```
def minha_funcao():
    resultado = 42

minha_funcao()
print(resultado)  # NameError: name 'resultado' is not defined
```

Se você se deparar com esta situação, o que você precisa fazer é retornar o valor de dentro da função para acessá-lo no escopo global (fora de qualquer função):

```
def minha_funcao():
    resultado = 42
    return resultado

resultado = minha_funcao()
print(resultado)
# output: 42
```

4. TypeError

Este erro ocorre quando o **tipo de dado** para uma determinada ação está incorreto, como:

- Operações com tipos incompatíveis:

```
resultado = 10 + '10'  # TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- Argumentos de tipo incorreto para funções:


```
resultado = len(10) # TypeError: object of type 'int' has no len()
```

- Modificar objetos imutáveis:

```
t = (1, 2, 3)
t[0] = 100 # TypeError: 'tuple' object does not support item assignment
```

- Tentar fazer indexação de tipos de dado que não suportam esta operação:

```
n = 100
valor = n[0] # TypeError: 'int' object is not subscriptable
```

- Tentar “chamar” algo que não é uma função:

```
n = 100
valor = n() # TypeError: 'int' object is not callable
```

5. ValueError

Este erro ocorre quando o tipo de dado está correto, porém seu **valor é inválido** para uma determinada ação:

```
int('10.0.1') # ValueError: invalid literal for int() with base 10: '10.0.1'

seq = [10, 20, 30]
seq.remove(40) # ValueError: list.remove(x): x not in list
```

6. IndexError

Este erro ocorre quando o **índice** passado para alguma sequência vai além dos seus limites:

```
seq = [10, 20, 30]

print(seq[0])
# output: 10

print(seq[10]) # IndexError: list index out of range
```

7. KeyError

Este erro ocorre quando a **chave** passada para algum dicionário não existe:

```
dic = {'chave1': 'valor1', 'chave2': 'valor2'}

print(dic['chave1'])
# output: valor1

print(dic['chave10']) # KeyError: 'chave10'
```

Além de dicionários, este erro também aparece em outros contextos onde tentamos “pegar” algum valor com base em um nome (a “chave”). Um dos casos mais comuns é tentando pegar um nome de coluna que não existe em DataFrames do pandas.

8. AttributeError

Este erro ocorre quando tentamos acessar um **atributo inexistente** de algum objeto:

```
s = 'Python'
print(s.upper())
# output: PYTHON

x = 100
print(x.upper()) # AttributeError: 'int' object has no attribute 'upper'
```

Muito comum de acontecer quando um valor pode ser None, e não lidamos com esta possibilidade:

```
dic = {'chave1': 'valor1', 'chave2': 'valor2'}

for chave in ['chave1', 'chave2', 'chave3']:
    valor = dic.get(chave) # Pode retornar None se a chave não existir!
    print(valor.upper())

# na última iteração...
# AttributeError: 'NoneType' object has no attribute 'upper'
```

9. ImportError

Este erro ocorre quando tentamos importar um módulo que não é encontrado pelo Python:

```
import meu_modulo_inexistente
# ModuleNotFoundError: No module named 'meu_modulo_inexistente'
```

Em geral, este erro tem duas causas possíveis:

- Se o módulo que tentamos importar foi baixado externamente (com pip), possivelmente o módulo foi instalado em um interpretador de Python, e estamos tentando executar nosso código com outro interpretador. Isso pode acontecer quando temos duas ou mais instalações de Python em um mesmo computador, ou quando foram criados ambientes virtuais para alguma das instalações.
- Se o módulo foi criado por nós mesmos, então o Python está com dificuldades de encontrar o arquivo. Neste caso, é preciso conferir se o nome do arquivo está correto na importação, e em que pasta o Python está executando (lembre-se do `os.getcwd()`).

10. FileNotFoundError

Este erro ocorre quando Python tenta abrir algum arquivo que não é encontrado:

```
with open('arquivo_inexistente.txt') as arquivo:
    print(arquivo.read())
# FileNotFoundError: [Errno 2] No such file or directory: 'arquivo_inexistente.txt'
```

Se você encontrar este erro, mas vê que o arquivo existe de fato, então confira os seguintes pontos:

- Veja se o nome está correto: cuide com letras maiúsculas e minúsculas e confira a extensão (dependendo da sua configuração, o explorador de arquivos do Windows pode omitir a extensão).
- Confira se o caminho até o arquivo está correto: cheque a pasta de onde Python está executando, printando o valor de `os.getcwd()`, e veja se está fazendo sentido.

29. Depurando código com debugger

Considere o código abaixo:

```
nomes = ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
idades = [20, 16, 18, 40, 12]

for nome, idade in zip(nomes, idades):
    if idade > 18:
        maior_de_idade = True
    elif idade < 18:
        maior_de_idade = False
    print(f'{nome} tem {idade} anos. É maior de idade? {maior_de_idade}')
```

Ele parece executar sem problemas. Mas agora vamos testar com outros dados:

```
nomes = ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
idades = [18, 16, 18, 40, 12]

for nome, idade in zip(nomes, idades):
    if idade > 18:
        maior_de_idade = True
    elif idade < 18:
        maior_de_idade = False
    print(f'{nome} tem {idade} anos. É maior de idade? {maior_de_idade}')
# NameError: name 'maior_de_idade' is not defined
```

Encontramos um erro! Vamos ver como solucioná-lo com um debugger.

Usando um debugger

Se executarmos o mesmo código acima no modo de **depuração/debug**, o programa deve parar na linha que causa o erro:

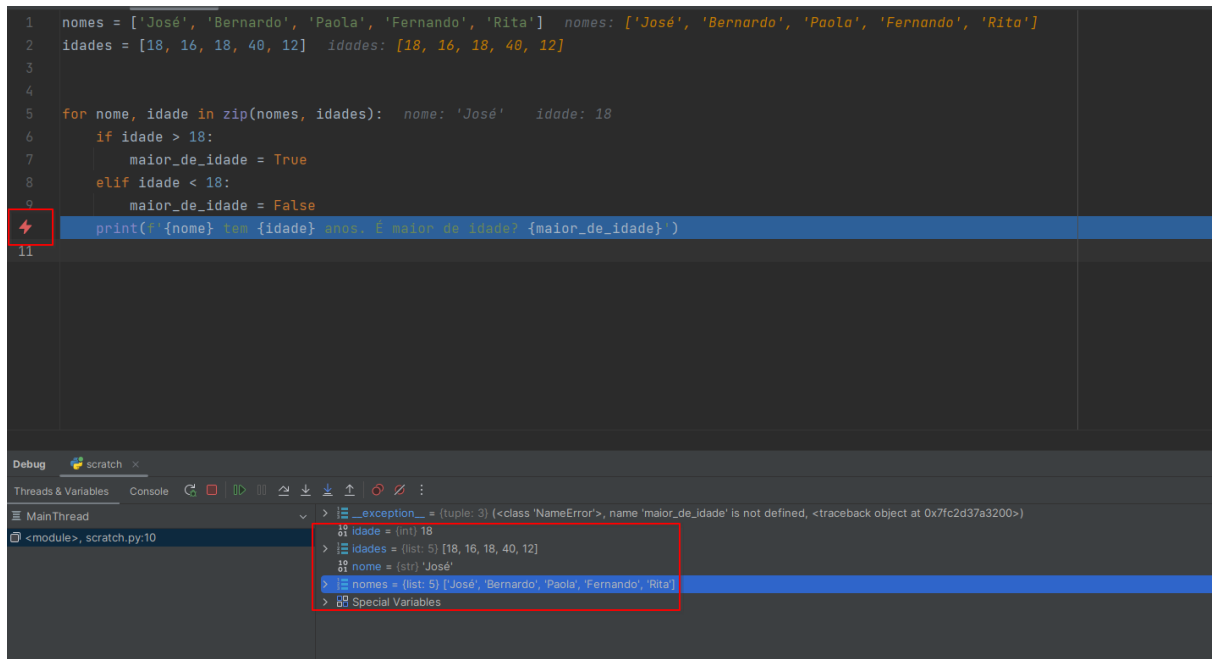


Figura 2: Visão geral do debugger.

Note que as variáveis ficam disponíveis para serem visualizadas junto do painel de debugger. Além disso, podemos usar o console de debugger para checar seus valores.

Com esta inspeção do código, conseguimos entender o problema:

- Não há lógica para lidar com a situação onde `idade == 18`. Isso significa que, se isso ocorrer na primeira iteração do loop, a variável `maior_de_idade` não será definida, e causará o erro.
- Por outro lado, se isso ocorrer em outra iteração, então o valor da variável `maior_de_idade` virá do loop anterior, o que causa um bug silencioso!

Modificando o código para a lógica abaixo corrige o problema:

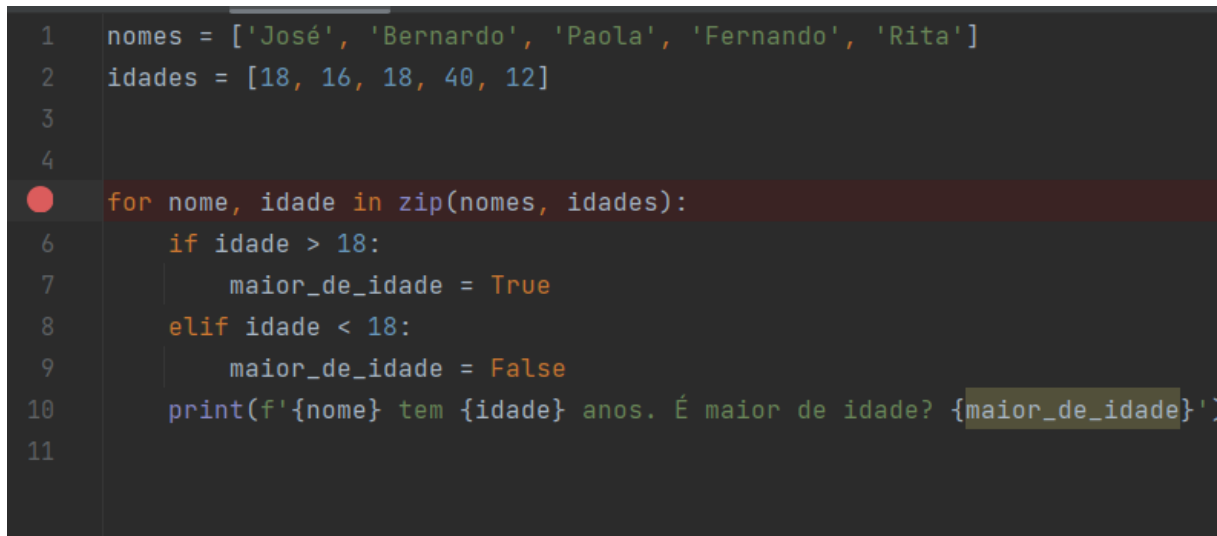
```
nomes = ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
idades = [18, 16, 18, 40, 12]

for nome, idade in zip(nomes, idades):
    if idade >= 18:
        maior_de_idade = True
    elif idade < 18:
        maior_de_idade = False
    print(f'{nome} tem {idade} anos. É maior de idade? {maior_de_idade}')
    # NameError: name 'maior_de_idade' is not defined
```

Breakpoints e depuração linha a linha

É possível pausar a execução do código em outros momentos, além de quando ocorre um erro. Para isso, usamos **breakpoints**.

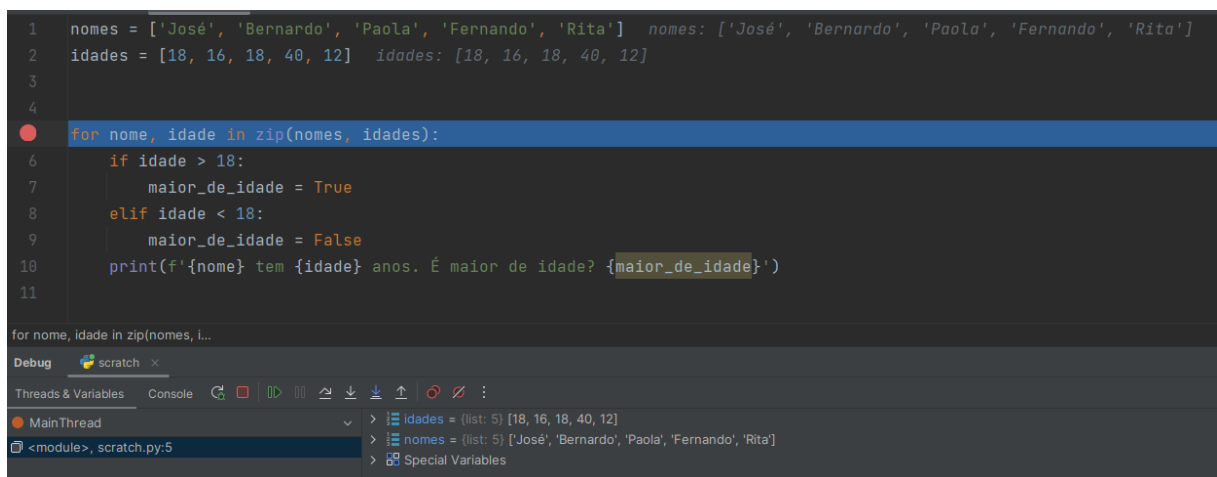
Na maioria das IDEs como PyCharm e VS Code, basta clicar no espaço à esquerda de uma linha para adicionar um breakpoint nela:



```
1  nomes = ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
2  idades = [18, 16, 18, 40, 12]
3
4
5  for nome, idade in zip(nomes, idades):
6      if idade > 18:
7          maior_de_idade = True
8      elif idade < 18:
9          maior_de_idade = False
10     print(f'{nome} tem {idade} anos. É maior de idade? {maior_de_idade}')
11
```

Figura 3: Breakpoint na linha 5.

Agora, se executarmos o código no modo debug, ele vai parar nesta linha:



```
1  nomes = ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']  nomes: ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
2  idades = [18, 16, 18, 40, 12]  idades: [18, 16, 18, 40, 12]
3
4
5  for nome, idade in zip(nomes, idades):
6      if idade > 18:
7          maior_de_idade = True
8      elif idade < 18:
9          maior_de_idade = False
10     print(f'{nome} tem {idade} anos. É maior de idade? {maior_de_idade}')
11
```

for nome, idade in zip(nomes, i...

Debug scratch x

Threads & Variables Console

MainThread > idades = (list: 5) [18, 16, 18, 40, 12]
> nomes = (list: 5) ['José', 'Bernardo', 'Paola', 'Fernando', 'Rita']
> Special Variables

<module>, scratch.py:5

Figura 4: Código executado até o breakpoint.

Veja que a linha atual fica destacada, e as variáveis até este ponto fica carregadas no painel de debugger.

Junto ao painel de debugger, há alguns botões para controlar seu comportamento:

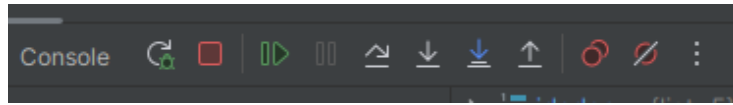


Figura 5: Botões de controle do debugger

Como você poderia esperar, há botões para recomençar/pausar/continuar com a execução de código. Além disso, há botões específicos para acompanhar o debug linha a linha, conforme a tabela abaixo:

Botão	Significado
Step over	Executa a linha atual e pausa na próxima linha. Se a linha atual executar alguma função, essa função é “pulada”
Step into	Executa a linha atual e pausa na próxima linha. Se a linha atual executar alguma função, o debugger entra pra função
Step into my code	Igual a Step into, porém pula funções que não foram escritas por você (biblioteca padrão de Python ou módulos instalados com <code>pip</code>)
Step out	Sai da função atual, parando na primeira linha após sua execução

Neste exemplo, não usamos nenhuma função, então os botões acabam tendo a mesma funcionalidade. Mas teste com algum código seu mais complexo para ver a diferença entre as opções!

30. Lidando com erros com blocos try/except

Veja o código abaixo:

```
x = float(input('Digite um número: '))
y = float(input('Digite um divisor: '))

resultado = x / y
print(resultado)
```

Ele funciona sem problemas **apenas se**:

- Os inputs x e y forem possíveis de ser convertidos em floats.
- O valor de y não for zero.

Já sabemos de antemão que estes são os pontos críticos do código acima.

Podemos ir além: usar um bloco try/except para lidar com qualquer erro que pode ser gerado, fazendo com que o código continue (ainda que não seja possível gerar o resultado esperado):

```
try:
    x = float(input('Digite um número: '))
    y = float(input('Digite um divisor: '))
    resultado = x / y
    print(f'O resultado é: {resultado}')
except:
    print('Ocorreu um erro!')
```

Note que, se qualquer erro acontecer dentro do bloco try, então este erro não fará com que o código pare imediatamente. No lugar disso, o código segue para o bloco except diretamente, e depois segue executando qualquer lógica que possa haver em seguida!

A estrutura de um try/except

O bloco try/except possui múltiplas opções de blocos indentados, incluindo alguns opcionais:

- try: código que está sendo “protegido” de erros.
- except: código que executa apenas se o código no bloco try causar algum erro.
- else: bloco opcional, executa apenas se o código no bloco try não causar nenhum erro.
- finally: bloco opcional, executa independentemente do que acontecer o try.

```
try:
    # Código que pode causar algum erro
except:
    # Executa quando código no try causa erro
else:
    # Executa quando código no try NÃO causa erro
finally:
    # Sempre executa
```


Lidando com erros específicos

Colocar um bloco `try/except` em todo o código, como fizemos anteriormente, não é a melhor solução. Queremos escrever blocos `try/except` que:

- Lidem com um **erro específico**.
- Envolvam apenas o **código específico** capaz de gerar o erro.

Fazemos isso para garantir que sabemos quais erros estamos “pegando” com blocos `try/except`. Caso contrário, é muito fácil criar um bloco que nos impede de ver os erros que nosso programa gera, o que significa que algum bug provavelmente está sendo levado adiante no código!

Reescrevendo o bloco anterior dessa forma, ficamos com:

```
try:
    x = float(input('Digite um número: '))
except ValueError:
    x = 10
    print(f'Valor inválido para x, usando valor padrão {x}')
```

```
try:
    y = float(input('Digite um divisor: '))
except ValueError:
    y = 2
    print(f'Valor inválido para y, usando valor padrão {y}')
```

```
try:
    resultado = x / y
    print(f'O resultado é: {resultado}')
except ZeroDivisionError:
    print(f'Impossível dividir {x} por {y}!')
```

Veja que a estrutura do `try/except` de `x` e `y` ficou muito similar. Vamos transformá-la em uma função:

```
def pegar_numero():
    while True:
        entrada = input('Digite um número: ')
        try:
            valor = float(entrada)
        except ValueError:
            print(f'Impossível transformar {entrada} em um float!')
        else:
            return valor

x = pegar_numero()
y = pegar_numero()
try:
    resultado = x / y
except ZeroDivisionError:
```

```
print(f'Impossível dividir {x} por {y}!')
else:
    print(f'O resultado de {x} dividido por {y} é: {resultado}')
```

31. Criando nossas próprias exceções com raise

Além de lidar com erros usando try/except, podemos usar raise para fazer com que nosso próprio código “lance” uma exceção (erro) em algum ponto:

```
import math

def calcular_raiz_quadrada(numero):
    if numero < 0:
        raise ValueError(f'Impossível calcular a raiz quadrada de {numero}! Apenas valores
        ↪ positivos são válidos.')
    return math.sqrt(numero)

for n in [4, 2, 1, 0, -1]:
    n_raiz = calcular_raiz_quadrada(n)
    print(f'Raiz quadrada de {n}: {n_raiz}')
    # Erro na última iteração:
    # ValueError: Impossível calcular a raiz quadrada de -1! Apenas valores positivos são
    ↪ válidos.
```

Veja que o tipo da exceção e sua mensagem foram criadas por nós!

Para quê criar Exceções?

Neste caso, se não houvesse nosso raise, o código acabaria caindo em uma exceção de qualquer forma:

```
import math

def calcular_raiz_quadrada(numero):
    return math.sqrt(numero)

for n in [4, 2, 1, 0, -1]:
    n_raiz = calcular_raiz_quadrada(n)
    print(f'Raiz quadrada de {n}: {n_raiz}')
    # Erro na última iteração:
    # ValueError: math domain error
```

Dito isso, veja que a nossa mensagem customizada é mais informativa, porque **sabemos especificamente o motivo pelo qual o erro é gerado**.

Além disso, é muito comum usarmos exceções para **garantir um estado específico** do nosso código. Em outras palavras, se o código passar por todas as “checagens” que potencialmente lançam exceções, consigo garantir que as variáveis e dados estão em um estado válido.

Abaixo, temos um exemplo de uma função que valida um conjunto de dados qualquer. Neste exemplo, vamos imaginar que os dados são um DataFrame do pandas, que basicamente representam uma

tabela (não é necessário conhecer pandas para acompanhar o exemplo):

```
def validar_dados(dados):
    # Checa se dados são o valor nulo
    if dados is None:
        raise TypeError("Dados não podem ser nulos!")
    # Checa se as colunas esperadas existem
    for nome_coluna_esperado in ['Data', 'Valor', 'Total']:
        if nome_coluna_esperado not in dados.columns:
            raise ValueError(f"Coluna {nome_coluna_esperado} é necessária, mas não foi
                               ↳ encontrada!")
    # Checa se os dados estão vazios (tabela com colunas mas sem linhas)
    if dados.empty:
        raise ValueError("Não há dados inseridos na tabela!")

dados = ler_dados()
validar_dados(dados=dados)
# A partir daqui, garanto que os dados são exatamente como esperamos!
```

Após a validação, garantimos que os dados possuem os atributos que esperamos (a partir das checagens realizadas na função `validar_dados`). Dessa forma, qualquer processamento posterior dos dados pode se valer do fato de que eles foram validados.

Imagine chegar no final de um processamento de dados demorado e descobrir que deverá ser refeito, porque havia um problema na tabela original? Com o método acima, evitamos essa situação: ou o código finaliza logo de cara e informa o problema, ou segue adiante com dados obrigatoriamente validados!

Gerando Exceções a partir de outra

Em alguns casos, pode ser interessante pegar uma exceção com `try/except` e apenas “relançá-la” com um erro customizado:

```
import math

def dividir(a, b):
    try:
        resultado = a / b
    except ZeroDivisionError as e:
        mensagem = f'Impossível dividir {a} por {b}. Divisão por zero!'
        raise ValueError(mensagem) from e
    return resultado

n = 10
for divisor in [4, 2, 1, 0, -1]:
    resultado = dividir(n, divisor)
    print(f'Resultado de divisão de {n} por {divisor}: {resultado}')
```

```
# Erro na última iteração:  
# ValueError: Impossível dividir 10 por 0. Divisão por zero!
```

O erro fica mais informativo que apenas usar a mensagem padrão do Python.

32. Usando type hints – tipagem de dados em Python

O que é tipagem de dados?

Recentemente, foi introduzido o conceito de **tipagem de dados** em Python.

Isto significa que podemos deixar claro, junto do código, o tipo de dado de determinada variável. Fazemos isso usando a sintaxe:

```
variavel: tipo = valor
```

onde `tipo` representa algum tipo de dado de Python. Esta “indicação do tipo” é chamada em inglês de *type hint*.

Alguns exemplos simples:

```
x: int = 10
y: float = 5.5
nome: str = 'Juliano'
```

Importante: diferente de outras linguagens de programação, Python jamais obriga você a usar type hints. Também não há nenhum tipo de melhoria de performance do código.

Os type hints servem exclusivamente para **ajudar a guiar os desenvolvedores**: é mais fácil entender o que um código faz quando sabemos exatamente o tipo de dado com que estamos trabalhando!

Tipagem de dados compostos

Para algumas estruturas de dados em Python, como listas e tuplas, podemos criar uma tipagem que indique também o **tipo de dado dentro da estrutura**:

```
nomes: list[str] = ['Luiza', 'Henrique', 'Ricardo']

valores: tuple[int] = (0, 1, 2, 3)

produtos: dict[str, float] = {
    'leite': 4.50,
    'pão': 1.20,
    'carne': 15.90,
}
```

Tipagem de funções

Usar tipagem para variáveis “soltas” é útil em alguns casos. Porém, a principal utilidade de type hints é usá-las para **indicar o tipo de dado de funções**.

Podemos usar type hints para indicar tanto o tipo de dado dos **parâmetros da função**, quanto do seu **valor de retorno** (usando uma seta `->` para indicar o retorno):

```
def somar(a: int, b: int) -> int:
    return a + b
```

```
print(somar(1, 2))
# output: 3
```

Note, mais uma vez, que os *type hints* são apenas indicações para ajudar quem lê o código a entendê-lo. O Python nunca reclamará se passarmos dados do tipo “errado” (sob o risco de causarmos algum erro na execução da função):

```
def somar(a: int, b: int) -> int:
    return a + b
```

```
print(somar(1.5, 2.2))
# output: 3.7
```

```
print(somar('Olá', 'Mundo'))
# output: OláMundo
```

Dica: quando as funções possuírem muitos argumentos, use quebras de linha para separar seus argumentos:

```
def somar(
    a: int,
    b: str,
    c: list[int],
    d: tuple,
) -> str:
    # Corpo da função aqui
```

Indicando o retorno de funções

Quando a função não retorna nada, podemos anotar seu retorno como `-> None`:

```
def dar_oi(nome: str) -> None:
    print(f'Olá {nome}!')
```

Em alguns casos, dependendo de sua IDE, ela pode até mesmo entender se você está usando o valor corretamente a partir dos type hints. Por exemplo:

- Quando a função não retorna nada (isto é, retorna `None`), não vamos querer pegar seu retorno em uma variável. As IDEs conseguem perceber quando fazemos isso a partir dos *type hints*:

```
def dar_oi(nome: str) → None:
    print(f'Olá {nome}!')

valor = dar_oi('Juliano')
```

Figura 6: Aviso do PyCharm indicando que a função `dar_oi` não retorna nada, portanto a variável `valor` é apenas um valor `None`

- Se uma função retorna um string (conforme indicado nos *type hints*), então um aviso será gerado se tentarmos somar seu retorno a um número:

```
def dar_oi(nome: str) → str:
    return f'Olá {nome}!'

valor = dar_oi('Juliano')
soma = valor + 2
```

Figura 7: Aviso do PyCharm indicando que a função `dar_oi` retorna um string, e portanto não é possível somar seu retorno a um int

Veja que a tipagem (aliada a uma IDE com inspeções) nos ajuda muito a evitar erros comuns!

Outros detalhes de tipagem

Em estruturas de dados, posso omitir o tipo de dado se não quiser ser explícito, ou não souber qual ele é. O lado negativo é que IDEs vão ficar com menos capacidade de inspeção do código:


```
nomes: list = ['Luiza', 'Henrique', 'Ricardo']

valores: tuple = (0, 1, 2, 3)

produtos: dict = {
    'leite': 4.50,
    'pão': 1.20,
    'carne': 15.90,
}
```

Utilizo barra vertical (|) para indicar alternativas entre dois (ou mais) tipos de dados:

```
coisas = list[int | str] = ['Python', 0, 35, 'Olá!']

produtos: dict[str, int | float] = {
    'leite': 4,
    'pão': 1.20,
    'carne': 15.90,
}

import random

def func() -> float | None:
    if (valor_aleatorio := random.random()) > 0.5:
        return None
    return valor_aleatorio
```

Posso usar `typing.Any` para indicar explicitamente que uma variável representa *qualquer* tipo de dado:

```
from typing import Any

def printar_valores(
    valor1: Any,
    valor2: Any,
) -> None:
    print(valor1)
    print(valor2)
```

Por fim, posso criar um *alias* (apelido) para um tipo de dado específico. Dessa forma, simplifico diversas expressões de tipagem, criando “variáveis” que as representem:

```
Num = int | float
NumList = list[Num]

def somar_elementos(lista: NumList) -> Num:
    return sum(lista)
```