

alpha

<ed/tech>

Python
Tipagem

Aula 07

<Módulo 07/>

Tipagem de Variáveis

Tipagem em Python: Dinâmica vs Estática

Conceitos Fundamentais



1. Fortemente Tipada

Python é uma linguagem **fortemente tipada** porque: - Cada objeto tem um tipo definido - O tipo do objeto determina quais operações podem ser realizadas - Não permite operações entre tipos incompatíveis sem conversão explícita

```
# Exemplo de tipagem forte
```

```
numero = 42  
texto = "10"
```

```
# Isso gerará erro:
```

```
resultado = numero + texto # TypeError: unsupported operand type(s)
```

```
# Precisa de conversão explícita:
```

```
resultado = numero + int(texto) # OK: 52
```

2. Tipagem Dinâmica

Python é **dinamicamente tipada** porque: - O tipo é associado ao valor, não à variável - Uma variável pode receber valores de diferentes tipos - O tipo é determinado em tempo de execução

```
# Exemplo de tipagem dinâmica
```

```
x = 42 # x é int  
x = "texto" # agora x é str  
x = [1, 2, 3] # agora x é list
```

Comparação com Linguagens Estaticamente Tipadas

Java (Tipagem Estática):

```
int numero = 42;  
numero = "texto"; // Erro de compilação
```

Python (Tipagem Dinâmica):

```
numero = 42  
numero = "texto" # Funciona, mas pode não ser uma boa prática
```

Type Hints (Tipagem Estática Opcional)

Python 3.5+ introduziu type hints, permitindo indicar tipos:

```
def calcular_idade(ano_nascimento: int) -> int:  
    return 2024 - ano_nascimento
```

```
# As hints são opcionais e não afetam a execução
```

```
def calcular_idade(ano_nascimento): # Também funciona  
    return 2024 - ano_nascimento
```

Exemplos Práticos

1. Tipagem Forte em Ação

```
# Operações precisam respeitar tipos
numero = 10
texto = "20"

# Erro:
resultado = numero + texto # TypeError

# Correto:
resultado = str(numero) + texto # "1020"
# ou
resultado = numero + int(texto) # 30
```

2. Tipagem Dinâmica em Ação

```
def processar_dado(valor):
    if isinstance(valor, str):
        return valor.upper()
    elif isinstance(valor, int):
        return valor * 2
    else:
        return None

# A mesma função processa diferentes tipos
print(processar_dado("python")) # PYTHON
print(processar_dado(5))       # 10
```

3. Type Hints em Uso

```
from typing import Union, List

def processar_lista(items: List[Union[str, int]]) -> List[str]:
    return [str(item).upper() for item in items]

# O código funciona com ou sem as hints
resultado = processar_lista([1, "dois", 3]) # ["1", "DOIS", "3"]

def processa_lista(lista: List[int]) -> Tuple[int, int]:
    return min(lista), max(lista)
```

Vantagens e Desvantagens

Vantagens da Tipagem Dinâmica:

1. Código mais flexível
2. Desenvolvimento mais rápido
3. Menos verbosidade

Vantagens das Type Hints:

1. Melhor documentação
2. Catch de erros antes da execução
3. Melhor suporte de IDE

Passo a Passo da Tipagem

Anotação de Tipo (Type Annotation)

Introdução

Nesta aula, vamos mergulhar no módulo typing, uma ferramenta poderosa que foi introduzida para suportar a adição de anotações de tipo ao código Python.

Embora Python seja uma linguagem dinamicamente tipada, as anotações de tipo, quando combinadas com ferramentas de análise estática, trazem uma série de benefícios que podem melhorar significativamente a qualidade, a legibilidade e a manutenibilidade do seu código.

Problemática Inicial: Falta de Sugestões de Métodos

Imagine que você está trabalhando em um projeto Python e define uma função que recebe um argumento string.

Dentro dessa função, você deseja usar métodos desse argumento (métodos de string), mas o VSCode (ou outra IDE) não sugere os métodos apropriados.

Isso ocorre porque, por padrão, o Python não sabe o tipo do argumento,

Exemplo

Digamos que você quer definir uma função `cumprimentar(nome)` que retorna o nome em letras maiúsculas.

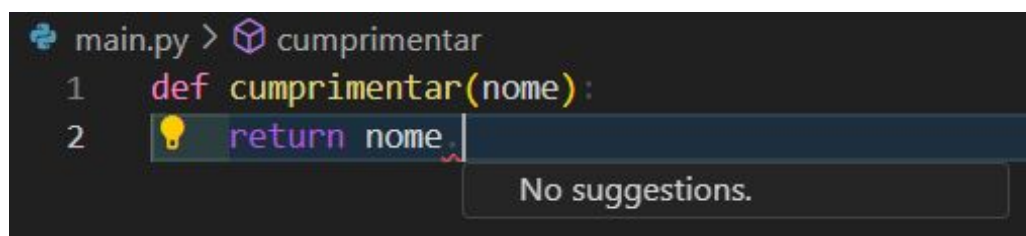
A função completa é assim:

```
def cumprimentar(nome):  
    return nome.upper()
```

Você pode testar a função e ver que ela funciona:

```
cumprimentar("João")  
# Saída:  
# 'JOÃO'
```

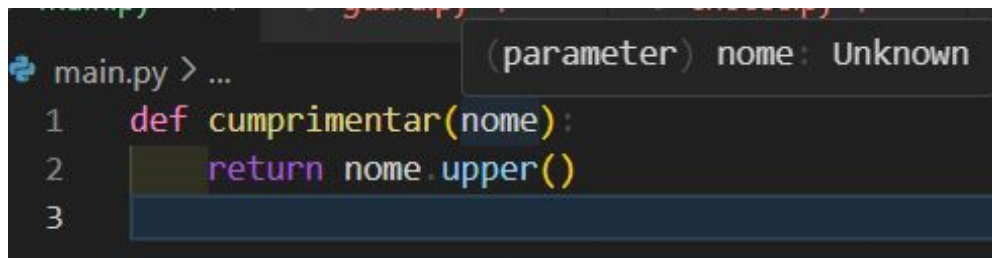
Ela funciona, mas a experiência de escrever esta função não é a melhor possível, porque enquanto você digita o código, ao digitar `nome.`, o VSCode não sugere os métodos de string:



Passo a Passo da Tipagem

Ao tentar acessar métodos de string após nome., o VSCode não oferece sugestões porque não sabe que nome é do tipo string.

Além disso, ao passar o mouse sobre nome, você verá nome: Unknown:



```
main.py > ...  
1 def cumprimentar(nome):  
2     return nome.upper()  
3
```

(parameter) nome: Unknown

Isso significa que para o VSCode, o tipo do parâmetro nome é desconhecido (unknown).

Solução: Anotação de Tipo

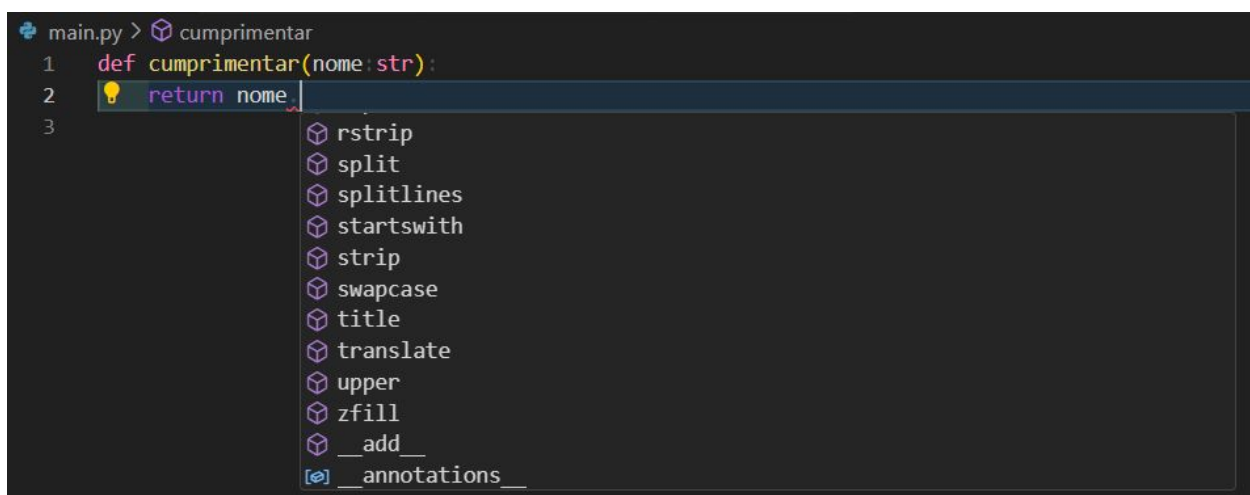
Para resolver esse problema, podemos declarar explicitamente o tipo do parâmetro na definição da função.

Fazemos isso colocando : str logo após o parâmetro:

```
def cumprimentar(nome: str):  
    return nome.upper() # Agora o VSCode sugere métodos de string
```

Leia "nome: str" como "nome é uma string".

Agora sim o VSCode sugere os métodos de string, entre eles o upper:

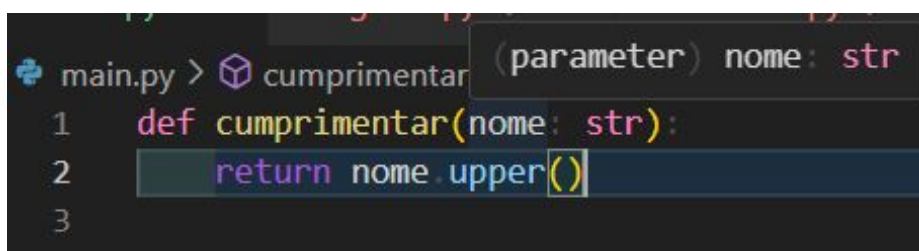



```
main.py > cumprimentar  
1 def cumprimentar(nome: str):  
2     return nome.  
3
```

- rstrip
- split
- splitlines
- startswith
- strip
- swapcase
- title
- translate
- upper
- zfill
- __add__
- __annotations__

Passo a Passo da Tipagem

Além disso, ao passar o mouse sobre nome, você verá "nome: str":



```
main.py >  cumprimentar (parameter) nome: str
1 def cumprimentar(nome: str):
2     return nome.upper()
3
```

Vantagens da Anotação de Tipo

A anotação de tipo em Python oferece várias vantagens significativas para o desenvolvimento de software. Aqui estão algumas das principais vantagens:

1. Melhoria na Experiência de Desenvolvimento

- **Suporte a Autocompletar:** Com a anotação de tipo, IDEs como o VSCode podem oferecer sugestões mais precisas e relevantes enquanto você digita. Isso torna o processo de escrita de código mais eficiente e reduz a necessidade de consultar documentação para lembrar de métodos e atributos.
- **Informações de Tipo em Tempo de Desenvolvimento:** Ao passar o mouse sobre uma variável ou parâmetro, você pode ver imediatamente seu tipo, o que ajuda a entender melhor o fluxo do programa e a evitar erros de tipo.

2. Redução de Erros

- **Deteção de Erros em Tempo de Desenvolvimento:** Muitas IDEs e ferramentas de análise de código podem detectar erros de tipo antes mesmo de você executar o código. Isso inclui tentativas de chamar métodos que não existem para um tipo específico ou passar argumentos de tipos incorretos para funções.
- **Mensagens de Erro Mais Claras:** Quando erros ocorrem, a anotação de tipo pode ajudar a fornecer mensagens de erro mais informativas, indicando claramente o que está errado, em vez de apenas reportar um erro genérico.

3. Melhoria na Documentação e Legibilidade

- **Código Auto-Documentado:** A anotação de tipo serve como uma forma de documentação inline, tornando mais fácil para outros desenvolvedores (e para você mesmo, após um tempo) entender o que o código espera e o que ele retorna, sem a necessidade de ler extensivamente a documentação ou o próprio código.
- **Comunicação de Intenção:** Ao especificar explicitamente os tipos, você comunica melhor sua intenção e os requisitos do código, o que pode evitar mal-entendidos e erros.

Passo a Passo da Tipagem

4. Integração com Ferramentas de Análise de Código

- **Análise de Código Estático:** Ferramentas de análise de código estático podem oferecer insights mais profundos e sugestões de melhoria com base nas anotações de tipo, ajudando a identificar problemas potenciais antes que eles causem erros em produção.
- **Type Checkers:** O Python tem type checkers como o mypy e o pyright, que podem verificar a consistência dos tipos em todo o seu código-base, oferecendo uma camada adicional de segurança contra erros de tipo.

5. Facilitação da Refatoração e Manutenção

- **Segurança ao Refatorar:** Com anotações de tipo, você pode refatorar o código com mais confiança, sabendo que as ferramentas podem alertá-lo sobre possíveis erros de tipo introduzidos durante o processo.
- **Manutenção de Código Legado:** Ao adicionar anotações de tipo a código legado, você pode melhorar significativamente a sua manutenibilidade, tornando-o mais fácil de entender e modificar sem introduzir novos erros.

Em resumo, a anotação de tipo em Python, embora não seja obrigatória, oferece uma série de vantagens que podem melhorar significativamente a qualidade, a manutenibilidade e a legibilidade do código, além de facilitar o desenvolvimento e a depuração.

Passo a Passo da Tipagem

Configurando o VSCode para Relatar Erros de Tipo

Por padrão, o VSCode pode não mostrar erros de tipo.

Por exemplo, imagine que você tenha um código como este:

```
def cumprimentar(nome: str):  
    return nome.upper()  
  
# Ops ! Passamos um inteiro ao invés de uma string  
# por engano  
print(cumprimentar(100))
```

Nesse código, cometemos um erro de tipo, ou seja, passamos um argumento com o tipo errado para uma função.

O erro foi passar um inteiro ao invés de uma string como argumento para a função cumprimentar.

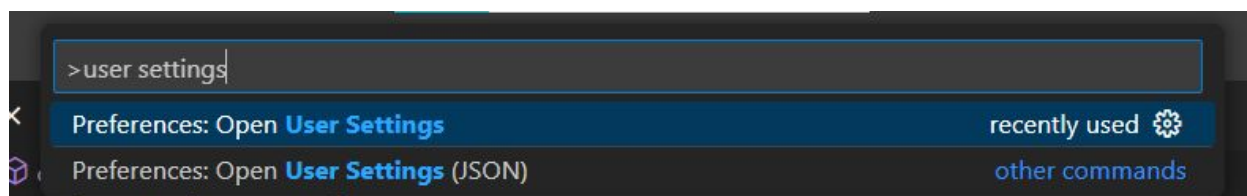
É um erro porque o parâmetro nome tem uma Anotação de Tipo dizendo que ele deve ser uma string.

Claramente nesse exemplo ilustrativo o erro é “bobo”, mas num código maior e realista esse tipo de erro é comum. E você só perceberia ao executar o código e vê-lo falhar.

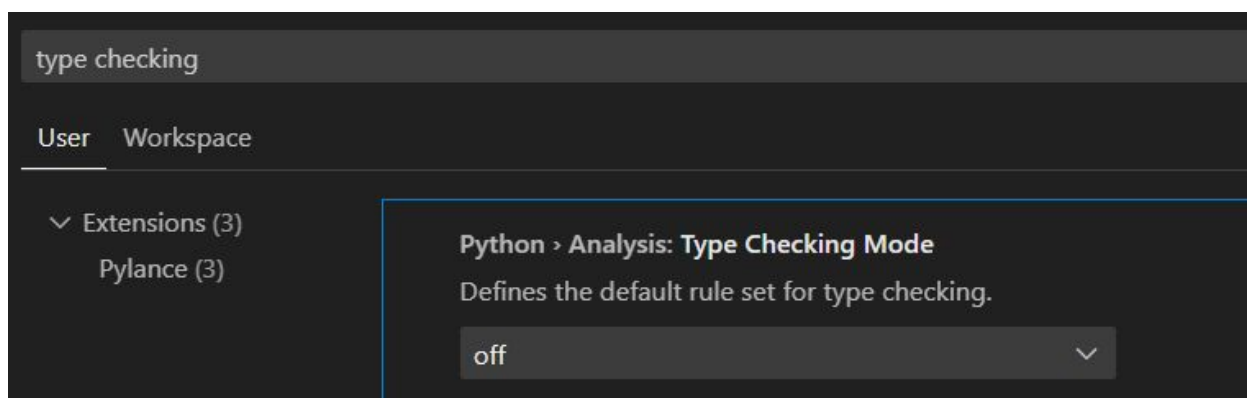
Mas agora que temos anotação de tipo, é possível configurar o VSCode para mostrar erros de tipo dentro do editor, sem precisar executar o código.

Para habilitar a verificação de erros de tipo:

1. Abra as configurações de usuário do VSCode (pressione F1, digite “User Settings” e selecione a opção):



2. Na barra de pesquisa, digite “type checking”.

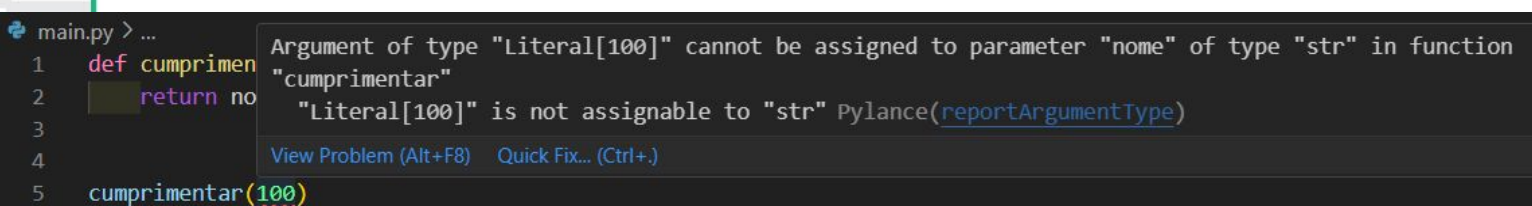


3. No campo “Python > Analysis: Type Checking Mode”, escolha “standard”.

Passo a Passo da Tipagem

Depois de alguns segundos para a configuração se aplicar, o VSCode deve começar a mostrar erros de tipo.

Você verá um sublinhado vermelho no 100 e, ao passar o mouse, verá a mensagem de erro:



```
main.py > ...  
1 def cumprimen  
2     return no  
3  
4  
5 cumprimentar(100)
```

Argument of type "Literal[100]" cannot be assigned to parameter "nome" of type "str" in function "cumprimentar"
"Literal[100]" is not assignable to "str" Pylance([reportArgumentType](#))
[View Problem \(Alt+F8\)](#) [Quick Fix... \(Ctrl+.\)](#)

Traduzindo, está dizendo que "o número 100 não pode ser atribuído ao parâmetro 'nome' do tipo 'string' na função 'cumprimentar'"

Passo a Passo da Tipagem

Análise Estática e Type Checkers

Quem ou o quê analisa as anotações de tipo que você escreve no código ? A resposta é: ferramentas de análise estática, também conhecidas como *type checkers* (verificadores de tipo).

Explicação do Conceito

Análise estática é o processo de analisar o código-fonte sem executá-lo.

Ferramentas de análise estática, como *type checkers*, examinam o código para detectar erros, inconsistências e possíveis problemas antes mesmo de o código ser executado.

Quando se trata de anotações de tipo em Python, os *type checkers* são as ferramentas que verificam se o seu código está usando os tipos corretamente, de acordo com as anotações que você forneceu.

Quem Faz a Análise de Tipos?

No contexto do VSCode, quando você tem a extensão oficial "Python" da Microsoft instalada, o *type checker* padrão que realiza a análise de tipos é o **Pylance**, que é baseado no **Pyright**, também desenvolvido pela Microsoft.

Outra alternativa popular é o **MyPy**, desenvolvido pela comunidade e amplamente adotado.

Estas ferramentas funcionam em segundo plano, analisando seu código à medida que você o escreve.

Quando encontram um erro de tipo - por exemplo, você chama uma função com um argumento do tipo errado, de acordo com as anotações de tipo - elas exibem um aviso ou erro diretamente no editor.

Por Que o Nome "Análise Estática"?

O termo "análise estática" vem do fato de que essas ferramentas analisam o código "estático", ou seja, o código como ele está escrito, sem executá-lo.

Elas não rodam seu programa; em vez disso, examinam a estrutura e o fluxo do código com base apenas no texto do código-fonte.

Isso contrasta com a "análise dinâmica", que envolve a execução do código e a observação de seu comportamento em tempo de execução.

Passo a Passo da Tipagem

Anotações de Tipo Não São Checadas em Tempo de Execução

É importante ressaltar que o interpretador Python não verifica as anotações de tipo durante a execução do programa.

Isso significa que, mesmo que seu código tenha erros de tipo detectados por um *type checker*, ele ainda pode ser executado pelo interpretador Python (desde que não haja outros erros de sintaxe ou de tempo de execução).

Por exemplo:

```
def dobrar(numero: int) -> int:
    return numero * 2

resultado = dobrar("olá") # Erro de tipo, mas o código ainda executa
print(resultado) # Imprime "oláolá"
```

Neste exemplo, a função `dobrar` está anotada para aceitar um inteiro, mas estamos chamando-a com uma string.

Um *type checker* detectaria esse erro e o mostraria no VSCode.

No entanto, se você executar este código, o interpretador Python não reclamará. Ele simplesmente executará a função, resultando na string "olá" sendo multiplicada por 2, o que produz "oláolá".

Uma boa maneira de pensar é a seguinte:

- Enquanto você escreve o código, o Pylance analisa as anotações de tipo e avisa os erros de tipos que você cometer.
- Ao executar o código, é como se ele não tivesse anotações de tipo. O interpretador Python não verifica as anotações de tipo em tempo de execução.

Atenção

Não dissemos que é **impossível** analisar as anotações de tipo em tempo de execução (i.e. quando o código é executado).

Dissemos que **o interpretador Python** não faz isso.

Mas é possível de ser feito, e há bibliotecas que fazem isso (por motivos diversos), como Pydantic, FastAPI, Marshmallow, attrs, Enforce, Beartype, entre outras.

Resumo

- *Type checkers* (verificadores de tipo) são ferramentas que realizam análise estática de código Python, verificando o uso correto dos tipos com base nas anotações.
- No VSCode com a extensão Python da Microsoft, o *type checker* padrão é o Pylance (baseado no Pyright). Outra opção popular é o MyPy.
- A análise é dita "estática" porque examina o código sem executá-lo.
- O interpretador Python não verifica as anotações de tipo em tempo de execução; portanto, um código com erros de tipo ainda pode ser executado.
- Existem bibliotecas que fazem uso das anotações de tipo em tempo de execução (i.e. no momento que o código é executado).

Passo a Passo da Tipagem

Inferência de Tipo

Motivação

Mesmo sem anotações de tipo explícitas, o VSCode, em muitos casos, consegue inferir o tipo de uma variável com base no valor que ela recebe.

Isso é chamado de **inferência de tipo** e é uma funcionalidade poderosa que ajuda a manter o código limpo e legível.

A inferência de tipo é a capacidade do Python de deduzir automaticamente o tipo de uma variável com base no seu valor inicial.

Isso funciona tanto para variáveis dentro de funções quanto para variáveis globais.

Exemplo

```
def exemplo():  
    idade = 30 # Python infere que 'idade' é um int  
    nome = "Alice" # Python infere que 'nome' é uma str  
  
exemplo()  
  
saldo = 1000.50 # Python infere que 'saldo' é um float
```

Neste exemplo, o Python infere corretamente os tipos das variáveis idade, nome e saldo com base nos valores atribuídos.

Resumo:

A inferência de tipo é uma funcionalidade poderosa do Python que deduz automaticamente os tipos das variáveis com base em seus valores iniciais, ajudando a manter o código limpo e legível.

Passo a Passo da Tipagem

Anotação para Tipos Primitivos e Classes Definidas pelo Usuário

Motivação

A anotação de tipo funciona não apenas para tipos primitivos, como `int`, `str`, `float` e `bool`, mas também para classes definidas pelo usuário.

Isso permite uma grande flexibilidade e poder na especificação dos tipos em seu código.

Exemplo para Tipos Primitivos:

```
def exemplo_primitivos(idade: int, nome: str, altura: float, ativo: bool):  
    print(f"Nome: {nome}, Idade: {idade}, Altura: {altura}, Ativo: {ativo}")  
  
exemplo_primitivos(30, "Alice", 1.75, True)
```

Exemplo para Classes Definidas pelo Usuário:

```
class Pessoa:  
    # Anotações de tipo como já visto.  
    # Observe que não colocamos anotação  
    # de tipo no 'self'  
    def __init__(self, nome: str, idade: int):  
        self.nome = nome  
        self.idade = idade  
  
    # Anotação de tipo para o parâmetro 'pessoa'.  
    # O python vai mostrar as sugestões dos  
    # métodos pessoa.nome e pessoa.idade ao digitar  
    def apresentar(pessoa: Pessoa):  
        print(f"Olá, meu nome é {pessoa.nome} e tenho {pessoa.idade} anos.")  
  
alice = Pessoa("Alice", 30)  
apresentar(alice)  
  
# Se passarmos um argumento que não é  
# uma instância de Pessoa, o VSCode vai avisar  
# uma mensagem de erro  
apresentar(100)
```

Passo a Passo da Tipagem

Subtipos e Atribuibilidade

Motivação

Em Python, alguns tipos são considerados "subtipos" de outros.

Por exemplo, `int` é considerado um subtipo de `float`.

Isso tem implicações importantes para a anotação de tipo, pois um valor de um subtipo pode ser atribuído a uma variável ou parâmetro anotado com o tipo pai.

Atribuibilidade refere-se à capacidade de "inserir" um valor de um tipo em uma variável ou parâmetro que tem um tipo anotado.

Em geral, um valor de um subtipo pode ser atribuído a uma variável ou parâmetro do tipo pai.

Exemplo

```
def dobrar(numero: float) -> float:
    return numero * 2

# Correto, pois int é um subtipo de float
resultado = dobrar(5)

print(resultado)
```

O tipo `int` é considerado um subtipo de `float`.

Portanto, você pode passar um `int` para a função `dobrar`, mesmo que ela esteja anotada para aceitar um `float`.

Isso se chama **Atribuibilidade**, dizemos que **o tipo `int` é atribuível ao tipo `float`**. Isso quer dizer que podemos passar um `int` a um lugar que espera `float`.

Passo a Passo da Tipagem

Tipo do Retorno de Função

Embora o Python geralmente infira corretamente o tipo de retorno de uma função, anotar explicitamente o tipo de retorno pode trazer benefícios significativos para a clareza e a manutenibilidade do código.

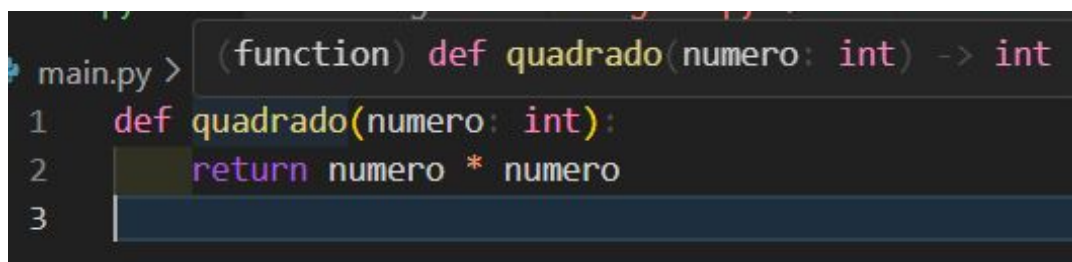
Exemplo de Inferência Automática

Na maioria dos casos, o Python consegue inferir o tipo de retorno de uma função com base nas instruções `return` dentro dela.

```
def quadrado(numero: int):  
    return numero * numero  
  
resultado = quadrado(5) # Python infere que 'resultado' é int  
                        # porque numero é int então  
                        # int * int é int
```

Neste exemplo, o Python infere corretamente que a função `quadrado` retorna um `int` porque a expressão `numero * numero` é uma operação entre inteiros.

Além disso podemos ver o tipo de retorno da função passando o mouse sobre o nome dela:



The screenshot shows a code editor with a tooltip displayed over the function name 'quadrado'. The tooltip text is '(function) def quadrado(numero: int) -> int'. The code in the background is as follows:

```
main.py >  
1 def quadrado(numero: int):  
2     return numero * numero  
3
```

Aquele `-> int` no final significa "retorna um `int`".

Passo a Passo da Tipagem

Exemplo de Anotação Manual

Uma das principais vantagens de anotar manualmente o tipo de retorno da função é que o VSCode (com a verificação de tipo habilitada) acusará um erro se você tentar retornar um tipo diferente do anotado.

Isso ajuda a evitar erros causados por falta de atenção ao programar o corpo da função.

Para anotar o tipo de retorno de uma função, colocamos uma seta "->" seguida do tipo que a função deve retornar:

```
def saudacao(nome: str) -> str:
    if not nome:
        return 0 # VSCode mostrará um erro aqui
    return f"Olá, {nome}!"
```

Neste exemplo, a função `saudacao` está anotada para retornar uma string.

No entanto, há um caminho (if) onde ela retorna um inteiro.

Com a anotação de tipo de retorno, o VSCode mostrará um erro, indicando que a função nem sempre retorna o tipo anotado.

Isso permite que você perceba e corrija o erro antes mesmo de executar o código.

Exemplo de Função Sem Retorno

Se uma função não tem uma instrução `return` ou tem uma instrução `return` sem um valor, o tipo de retorno deve ser anotado como `None` (ou deixado sem anotação de tipo):

```
# Função que não retorna nada
def imprimir_mensagem(mensagem: str) -> None:
    print(mensagem)

imprimir_mensagem("Olá!")
```

Resumo

- O Python geralmente infere o tipo de retorno automaticamente.
- Anotar manualmente o tipo de retorno melhora a clareza e a documentação.
- Se a função não retorna nada, anote o tipo de retorno como `None`.
- A principal vantagem da anotação manual é a detecção de erros pelo VSCode, ajudando a escrever código correto e a evitar erros por desatenção.

Passo a Passo da Tipagem

União de tipos (operador |)

Motivação

Em Python, é comum encontrar situações em que uma variável ou um parâmetro de função pode precisar aceitar valores de mais de um tipo.

O operador de união (|) permite que você especifique que um argumento, variável ou valor de retorno pode aceitar valores de diferentes tipos. Ele é uma forma concisa e legível de expressar que um elemento do seu código pode ser de um tipo ou de outro(s).

Exemplo com Valor Padrão None

Um padrão comum em Python é ter um argumento de função com um valor padrão None, mas que também pode aceitar valores de um tipo específico. Nesses casos, você pode usar Tipo | None.

Por exemplo sem anotações de tipo:

```
def saudacao(nome = None):  
    """Retorna uma saudação personalizada ou uma saudação genérica."""  
    if nome is None:  
        return "Olá, visitante!"  
    else:  
        return f"Olá, {nome}!"  
  
print(saudacao("Alice"))  
print(saudacao())
```

A intenção dessa função é: ela recebe como parâmetro ou uma string ou nada (None), e retorna uma string.

Veja como fazemos a anotação de tipo:

```
def saudacao(nome: str | None = None) -> str:  
    """Retorna uma saudação personalizada ou uma saudação genérica."""  
    if nome is None:  
        return "Olá, visitante!"  
    else:  
        return f"Olá, {nome}!"  
  
print(saudacao("Alice"))  
print(saudacao())
```

Neste exemplo, o parâmetro nome pode ser ou uma string (str) ou None. A função saudacao verifica se nome é None e retorna uma saudação apropriada.

Passo a Passo da Tipagem

União de Múltiplos Tipos

Você não está limitado a apenas dois tipos na união. Pode usar `|` para unir quantos tipos forem necessários:

```
def processar_dado(dado: int | str | None) -> None:
    """Processa um dado que pode ser um inteiro, uma string ou None."""
    if isinstance(dado, int):
        print(f"Processando inteiro: {dado}")
    elif isinstance(dado, str):
        print(f"Processando string: {dado}")
    else:
        print("Não há dados para processar")

processar_dado(10)
processar_dado("Olá")
processar_dado(None)
```

Neste exemplo, a função `processar_dado` pode aceitar um inteiro, uma string ou `None`.

Sintaxe Antiga: Union e Optional

Antes do Python 3.10, a união de tipos era expressa usando `Union` do módulo `typing`.

Para unir `None` a um tipo, usava-se `Optional`.

Ou seja, em vez de `int | str` usava-se `Union[int, str]`.

E se um dos tipos fosse `None`, por exemplo `int | None`, usava-se `Optional[int]`.

```
from typing import Union, Optional

# "Optional[str]" é o mesmo que "str | None"
def saudacao_antiga(nome: Optional[str] = None) -> str:
    """Retorna uma saudação personalizada ou uma saudação genérica (versão antiga)."""
    if nome is None:
        return "Olá, visitante!"
    else:
        return f"Olá, {nome}!"

# "Union[int, str, None]" é o mesmo que "int | str | None"
def processar_dado_antigo(dado: Union[int, str, None]) -> None:
    """Processa um dado (versão antiga)."""
    # ... (mesma lógica da versão nova) ...
```

Embora essa sintaxe ainda funcione em versões mais recentes do Python, a nova sintaxe com `|` é mais concisa e legível.

Resumo

- O operador de união (`|`) permite que você especifique múltiplos tipos possíveis para um elemento do seu código.
- É comum usar `Tipo | None` para argumentos que podem ser `None` ou de um tipo específico.
- Você pode unir quantos tipos forem necessários, não apenas dois.
- Antes do Python 3.10, usava-se `Union` e `Optional` do módulo `typing`, mas a nova sintaxe com `|` é mais concisa.

Passo a Passo da Tipagem

Tipos de Listas, Tuplas e Dicionários

Motivação

Ao trabalhar com coleções como listas, tuplas e dicionários em Python, você pode usar a simples anotação de tipo como `list`, `tuple` ou `dict`.

Mas essa anotação de tipo pode não ser suficiente para fornecer informações precisas ao sistema de tipos estáticos.

Isso ocorre porque essas anotações, por si só, não especificam os tipos dos elementos contidos nas coleções.

Sem essa informação adicional, as ferramentas de análise estática não podem realizar verificações de tipo completas e a IDE pode não ser capaz de fornecer sugestões precisas de código.

Listas: Problema com `list`

Quando você anota uma variável apenas como `list`, o sistema de tipos não sabe qual o tipo dos elementos dentro da lista.

Isso limita a capacidade das ferramentas de análise estática de verificar se seu código está usando a lista corretamente:

```
def processar_nomes(nomes: list):  
    for nome in nomes:  
        print(nome.upper()) # Problema: o VSCode não sabe o tipo de 'nome'
```

Neste exemplo, como `nomes` está anotado apenas como `list`, o VSCode não pode sugerir métodos de string como `upper()` porque não sabe que `nome` é uma string.

Além disso, se você tentar acessar um método que não existe em strings (por exemplo, `nome.append()`), não haverá um aviso de erro.

Solução com `list[Tipo]`

Para resolver esse problema, usamos `list[Tipo]` para especificar o tipo dos elementos contidos na lista.

```
def processar_nomes(nomes: list[str]) -> None:  
    for nome in nomes:  
        print(nome.upper()) # Agora o VSCode sugere métodos de string
```

Agora, ao anotar `nomes` como `list[str]`, o VSCode sabe que cada elemento em `nomes` é uma string e pode oferecer sugestões apropriadas, como métodos de string.

Além disso, se você tentar usar um método que não é de string em `nome`, o VSCode mostrará um aviso.

Leia "`list[str]`" como "lista de strings".

Tuplas

Tuplas, sendo sequências imutáveis que podem conter uma mistura de tipos, são anotadas usando `tuple[Tipo1, Tipo2, ...]`, onde você especifica os tipos dos elementos em cada posição da tupla.

```
def descrever_ponto(ponto: tuple[int, int, str]) -> None:  
    # vscode infere que x e y são int, e descricao é str  
    x, y, descricao = ponto  
    print(f"Ponto ({x}, {y}): {descricao}")  
  
ponto = (10, 20, "Origem")  
descrever_ponto(ponto)
```

Passo a Passo da Tipagem

Dicionários

Para dicionários, que são coleções de pares chave-valor, usamos `dict[TipoChave, TipoValor]` para especificar os tipos das chaves e dos valores.

```
def imprimir_notas(notas: dict[str, float]) -> None:
    # vscode infere que nome é str e nota é float
    for nome, nota in notas.items():
        print(f"{nome}: {nota}")

notas = {"Alice": 9.5, "Bob": 8.0}
imprimir_notas(notas)
```

Claro que isso só funciona quando os valores do dicionário são do mesmo tipo (nesse exemplo todos são float).

Se você quiser especificar as chaves de um dicionário separadamente e qual o tipo de cada uma, pode usar `TypedDict` do módulo `typing`, veja a documentação:

- [TypedDict](#)

Sintaxe Antiga

Antes do Python 3.9, as anotações de tipo para listas, tuplas e dicionários eram feitas usando `List`, `Tuple` e `Dict` do módulo `typing`.

```
from typing import List, Tuple, Dict

def processar_nomes_antigo(nomes: List[str]) -> None: # ...

def descrever_ponto_antigo(ponto: Tuple[int, int, str]) -> None:
```

Embora essa sintaxe ainda funcione em versões mais recentes do Python, a nova sintaxe com `list`, `tuple` e `dict` é mais concisa.

Resumo

- Anotar apenas como `list` não especifica o tipo dos elementos, limitando a análise estática.
- Usar `list[Tipo]` resolve esse problema, permitindo que as ferramentas de análise estática entendam o tipo dos elementos da lista.
- Tuplas são anotadas com `tuple[Tipo1, Tipo2, ...]`, especificando os tipos em cada posição.
- Dicionários são anotados com `dict[TipoChave, TipoValor]`.
- Antes do Python 3.9, usava-se `List`, `Tuple` e `Dict` do módulo `typing`, mas a nova sintaxe é mais concisa e consistente.

Passo a Passo da Tipagem

Tipo Literal

Motivação

Em alguns casos, você pode querer restringir um argumento a um conjunto finito de valores literais. Por exemplo, uma função que lida com os dias da semana pode aceitar apenas as strings "segunda", "terça", etc.

O tipo `Literal` permite que você especifique que um argumento ou variável pode aceitar apenas um conjunto específico de valores literais.

Exemplo

```
from typing import Literal

# Literal["fácil", "médio", "difícil"]
# significa que o parâmetro nível só
# aceita uma dessas três possíveis strings
def configurar_dificuldade(nível: Literal["fácil", "médio", "difícil"]) -> None:
    if nível == "fácil":
        print("Configurando nível fácil")
    elif nível == "médio":
        print("Configurando nível médio")
    else:
        print("Configurando nível difícil")

configurar_dificuldade("fácil")
configurar_dificuldade("impossível") # VSCode mostra um erro aqui
```

Neste exemplo, a função `configurar_dificuldade` só aceita as strings "fácil", "médio" ou "difícil". Se você tentar passar qualquer outro valor, mesmo que seja string, o VSCode mostrará um erro.

Passo a Passo da Tipagem

Anotações de Tipo em Classes

Assim como nas funções, as anotações de tipo também são extremamente úteis em classes.

Elas ajudam a documentar e validar os tipos dos atributos, parâmetros de métodos e valores de retorno.

Em classes, você pode anotar os tipos dos atributos na declaração da classe ou no método `__init__`. Os métodos também podem ter anotações de tipo para seus parâmetros e valores de retorno, da mesma forma que as funções.

Exemplo

```
class CarrinhoDeCompras:
    def __init__(self):
        # Anotação de tipo no atributo
        self.itens: list[str] = []

        # Anotação de tipo no método.
        # Note que não se coloca anotação
        # de tipo no self
        def adicionar_item(self, item: str) -> None:
            self.itens.append(item)

        def total(self) -> int:
            return len(self.itens)

carrinho = CarrinhoDeCompras()
carrinho.adicionar_item("maçã")
carrinho.adicionar_item("banana")
print(f"Total de itens no carrinho: {carrinho.total()}")
```

Neste exemplo, o atributo `itens` é anotado como `List[str]`, indicando que é uma lista de strings.

O método `adicionar_item` anota o parâmetro `item` como `str`, e o método `total` anota seu retorno como `int`.

Resumo

As anotações de tipo em classes funcionam de maneira semelhante às funções, permitindo a documentação e validação de tipos de atributos, parâmetros de métodos e valores de retorno.

Aprofundando no Assunto

A capacidade da linguagem Python e da biblioteca `typing` para declarar tipos nos programas vai muito além do que foi mostrado aqui.

Seguem algumas sugestões para ler mais:

- <https://realpython.com/python-type-checking/>
- https://mypy.readthedocs.io/en/stable/getting_started.html
- <https://docs.python.org/3/library/typing.html>

Discussões adicionais

Para quem quer aprofundar no assunto.

Tipos Genéricos

Por que usar Tipos Genéricos?

Problema: Imagine que você precisa criar funções que trabalham com listas de diferentes tipos: > - Uma lista de números > - Uma lista de strings > - Uma lista de objetos customizados

Solução 1: Sem usar Tipos Genéricos

Teríamos que criar várias funções quase idênticas:

```
def primeiro_numero(array: List[int]) -> int:  
    return array[0]
```

```
def primeira_string(array: List[str]) -> str:  
    return array[0]
```

```
def primeiro_usuario(array: List[Usuario]) -> Usuario:  
    return array[0]
```

Solução 2: Usando Tipos Genéricos

```
from typing import TypeVar, List

T = TypeVar('T')

def primeiro(array: List[T]) -> T:
    return array[0]

# Agora podemos usar:
numeros = [1, 2, 3]
textos = ["a", "b", "c"]
usuarios = [Usuario("João"), Usuario("Maria")]

primeiro(numeros)      # Retorna int
primeiro(textos)       # Retorna str
primeiro(usuarios)     # Retorna Usuario
```

Exemplos Práticos**1. Cache Genérico**

```
from typing import TypeVar, Dict, Optional

K = TypeVar('K') # Tipo da chave
V = TypeVar('V') # Tipo do valor

class Cache:
    def __init__(self):
        self._items: Dict[K, V] = {}

    def set(self, key: K, value: V) -> None:
        self._items[key] = value

    def get(self, key: K) -> Optional[V]:
        return self._items.get(key)

# Uso:
cache_strings = Cache[str, str]() # Cache de string para string
cache_numeros = Cache[str, int]() # Cache de string para número
```

2. Pilha Genérica

```
from typing import TypeVar, List

T = TypeVar('T')

class Pilha:
    def __init__(self):
        self._items: List[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> T:
        return self._items.pop()

# Uso:
pilha_int = Pilha[int]() # Pilha só de inteiros
pilha_str = Pilha[str]() # Pilha só de strings
```


3. Restrições em Tipos Genéricos

```
from typing import TypeVar, Comparable

# T deve ser um tipo que suporta comparação
T = TypeVar('T', bound=Comparable)

def encontrar_maior(a: T, b: T) -> T:
    return a if a > b else b

# Uso:
maior_num = encontrar_maior(10, 20)      # Funciona
maior_str = encontrar_maior("a", "b")    # Funciona
```

Benefícios

1. Reutilização de Código:

- Escreva uma função/classe uma vez
- Use com diferentes tipos
- Mantenha a segurança de tipos

2. Prevenção de Erros:

```
numeros = [1, 2, 3]
resultado = primeiro(numeros) # IDE sabe que resultado é int
resultado.upper() # Erro detectado pela IDE - int não tem método upper()
```

3. Melhor Documentação:

- O código se torna autoexplicativo
- IDEs podem fornecer melhor autocompletar

4. Flexibilidade com Segurança:

```
def trocar_posicoes(lista: List[T], i: int, j: int) -> None:
    lista[i], lista[j] = lista[j], lista[i]

# Funciona com qualquer tipo, mas mantém a consistência
```

Type Aliases

```
from typing import Tuple, TypeAlias

Coordenadas: TypeAlias = Tuple[float, float]

def mover(pos: Coordenadas) -> None:
    x, y = pos
    print(f"Movendo para ({x}, {y})")
```

Protocolos (Interfaces)

```
from typing import Protocol

class Animal(Protocol):
    def fazer_som(self) -> None:
        ...

class Cachorro:
    def fazer_som(self) -> None:
        print("Au au!")
```

Casos Práticos Comuns

Problema 1: Validação de Dados

```
from typing import Dict, Any

def validar_usuario(dados: Dict[str, Any]) -> bool:
    if not isinstance(dados.get('idade'), int):
        return False
    if not isinstance(dados.get('nome'), str):
        return False
    return True
```

Problema 2: API Responses

```
from typing import TypedDict

class ResponseUsuario(TypedDict):
    id: int
    nome: str
    ativo: bool

def get_usuario(id: int) -> ResponseUsuario:
    # Simulação de resposta da API
    return {"id": id, "nome": "João", "ativo": True}
```