Trabalhando com if

Live de Python # 293

Roteiro

1. Estruturas de controle

Uma revisão teórica

2. Formas do if

Gramática, bloco e expressão

3. Expressões

O que o if valida

4. Algumas dicas

Que podem ser úteis

O objetivo dessa live não é ser uma introdução ao assunto `if`.

Observações

Mas, uma fundamentação teórica para quem precisa e também um entendimento mais aprofundado de como o python lida com as coisas

Observações

apoia.se/livedepython

pix.dunossauro@gmail.com

patreon.com/dunossauro

Ajude o projeto

Albano Maywitz, Alexandre Costa, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alfredomorais, Alfredo Neto, Alysson Oliveira, Andre, Andre Makoski, André Oliveira, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Augusst0o, Augusto Domingos, Aurelio Costa, Belisa Arnhold, Beltzery, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Bug_elseif, Canibasami, Carlos Gonçalves, Carlos Henrique, Cauã Oliveira, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Brito, Daniel Bruno, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, Darcioalberico_sp, David Couto, Dh44s, Diego Guimarães, Dilan Nery, Dunossauro, Edgar, Elias Soares, Emerson Rafael, Érico Andrei, Esdras, Everton Silva, Ewertonbello, Fábio Belotto, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Ferrabras, Fichele Marias, Fightorfall, Francisco Aclima, Franklin Sousa, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giuliano Silva, Glauber Duma, Gnomo Nimp, Grinaode, Guibeira, Guilherme Felitti, Guilherme Ostrock, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Idlelive, Ivan Santiago, Ivansantiagojr, Janael Pinheiro, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jerry Ubiratan, Jhonata Medeiros, Jhon Gonçalves, Joaocuginotto, João Pena, Joao Rocha, Joarez Wernke, Jonas Araujo, Jonatas Leon, Jonatas_silva11, Jose Barroso, Joseíto Júnior, José Predo), Josir Gomes, Jota_lugs, Jplay, Jrborba, Ju14x, Juan Felipe, Juli4xpy, Juliana Machado, Julio Franco, Julio Gazeta, Julio Silva, Kacoplay, Kaio Peixoto, Kakaroto, Knaka, Krisquee, Lara Nápoli, Leandro Pina, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucasgcode, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Polo, Lucas Schneider, Luciano_ratamero, Luciano Ratamero, Lúcia Silva, Luidduarte, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Marina Passos, Mateusamorim96, Mateus Ribeiro, Matheus Mendez, Matheus Vian, Medalutadorespacialx, Mlevi Lsantos, Mrnoiman, Murilo Carvalho, Nhambu, Omatheusfc, Oopaze, Ostuff_fps, Otávio Carneiro, Patrick Felipe, Pytonyc, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Ramon Lobo, Renan, Renan Sebastião, Renato José, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Ric_fv, Richard Sousa, Rinaldo Magalhaes, Robsonpiere, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogeriocampos7, Rogério Nogueira, Rui Jr, Rwallan, Samael Picoli, Samanta Cicilia, Santhiago Cristiano, Scrimf00x, Scrimfx, Sherlock Holmes, Shinodinho, Shirakawa, Tarcísio Miranda, Tenorio, Téo Calvo, Teomewhy, Tharles Andrade, Thiago, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tomás Tamantini, Trojanxds, Valdir, Varlei Menconi, Viniciusfk9, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vladimir Lemos, Waltennecarvalho, Williamslews, Willian Lopes, Will_sca, Xxxxxxxxx, Zero! Studio

Obrigado você <3

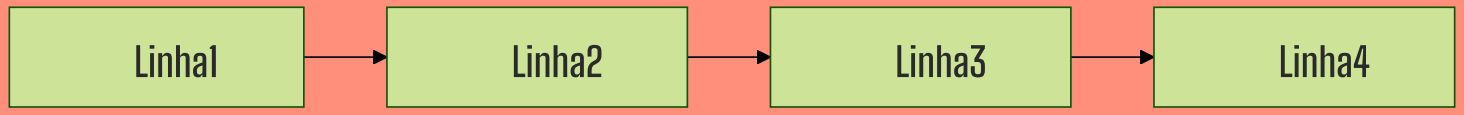
Estruturas de

Con trole

Fluxo de execução

Quando pensamos na execução do código, geralmente imaginamos algo linear, onde as instruções acontecem uma após a outra, como em uma "receita de bolo" que deve ser seguida passo a passo.

Que seguem um **fluxo de execução linear**, linha após linha.



Algo como:

```
a = 10
b = 20
print(a + b)
```

Estruturas de controle

Alguns problemas podem ser difíceis e repetitivos quando vistos de forma linear. Para "deslinearizar" o fluxo, existem **estruturas de controle**.

Como estruturas de:

- **Seleção**: Bifurcam o fluxo em detrimento de uma **condição**
- **Repetição**: Repetem parte do fluxo baseados em uma **condição**
  - **Interrupção**: Interrompem ou modificam fluxos de repetição
- **Tratamento de Exceções***: Desvia o fluxo em detrimento de um **erro**

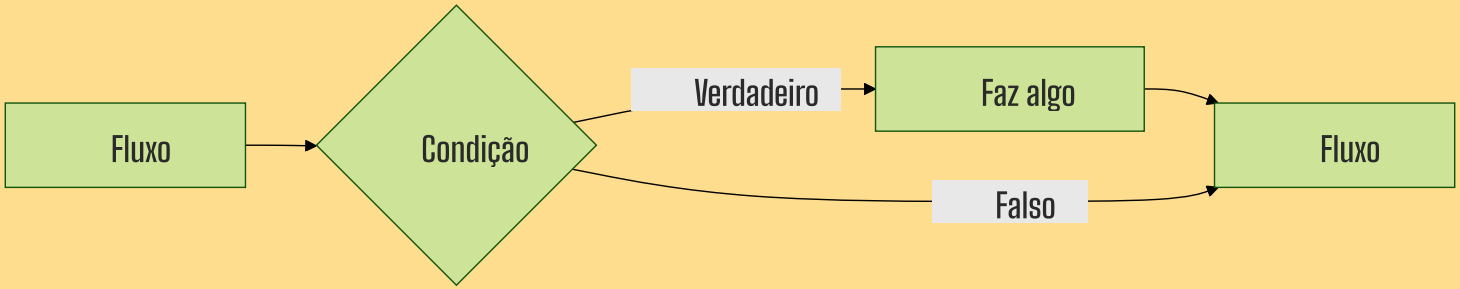
*: Há discussões teóricas sobre a classificação

Alguns exemplos

- **Seleção**: Bifurcam o fluxo em detrimento de uma **condição**
  - exemplos: `if/elif/else` e `match/case`
  - O `match/case` já foi tratado na Live de python #171
- **Repetição**: Repetem parte do fluxo baseados em uma **condição**
  - exemplos: `for/else` e `while/else`
  - **Interrupção**: Interrompem ou modificam fluxos de repetição
    - exemplos: `break`, `continue`, `yield`* e `return`
- **Tratamento de Exceções***: Desvia o fluxo em detrimento de um **erro**
  - exemplo: `try/except/else/finally`
  - Já foi tratado na Live de python #60

Estrutura de seleção

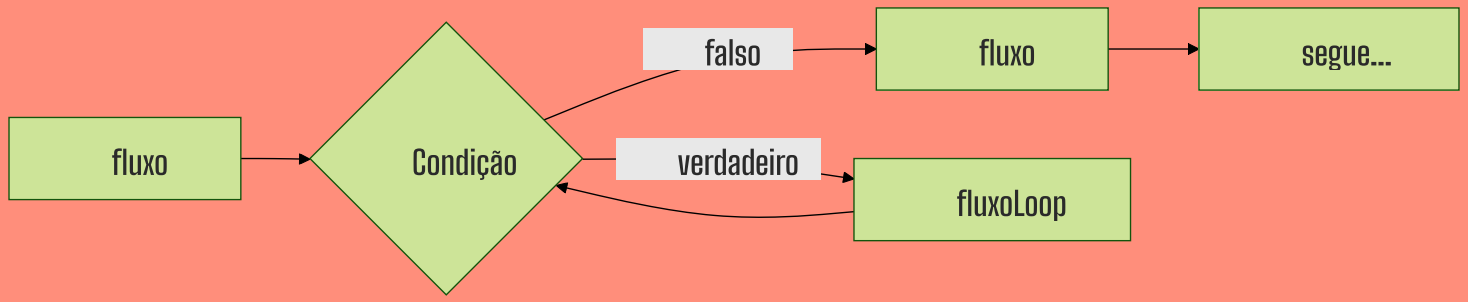
Uma bifurcação pode ser gerada por uma estrutura de seleção como `if`, onde o fluxo do código depende de uma **condição**.




```
# fluxo
if cond:
   print("Faz esse caminho")
# fluxo
```

Estruturas de repetição

Permitem que um sub-fluxo seja executado em relação a uma **condição**.



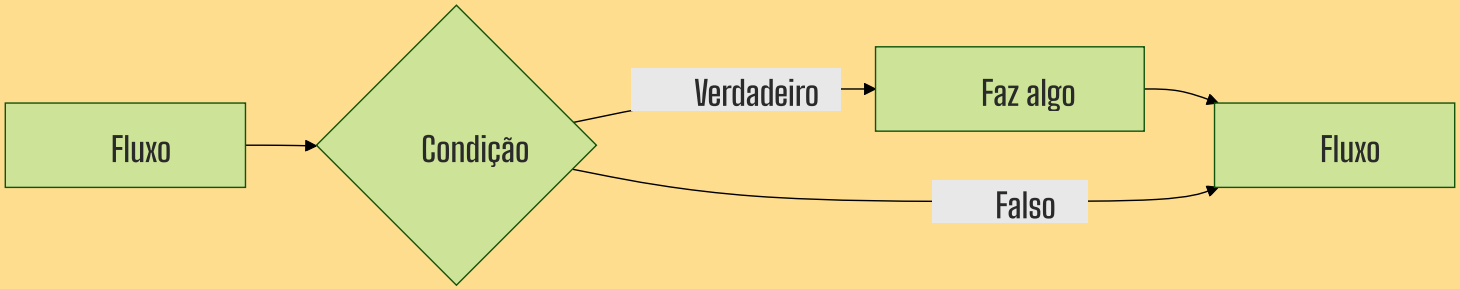
Exemplo:

```
# fluxo
while cond:
   print('Condição verdadeira!')
# fluxo
```

Condição

O **fluxo** de execução é alterado dependendo de uma **condição**, o que é chamado de **desvio condicional**.

A **condição** é um **nó** que define os caminhos possíveis de execução. Ela manipula a direção do fluxo, decidindo qual caminho o código seguirá.

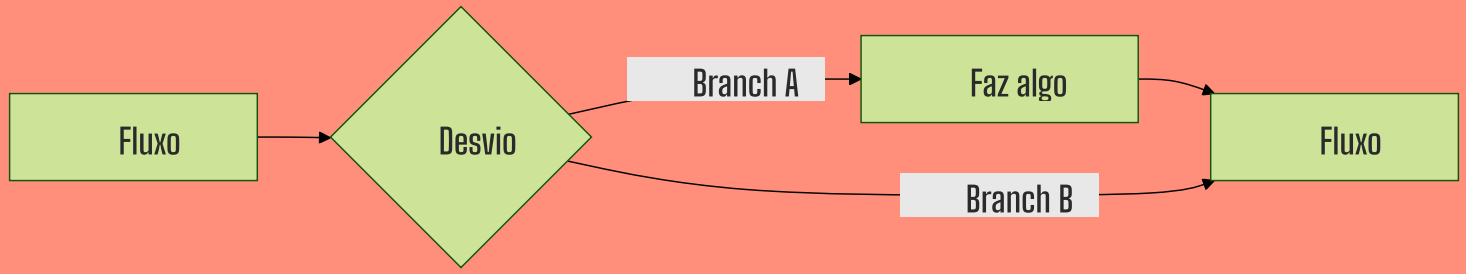


A condição é uma expressão em Python que veremos em breve.

Branches

Quando falamos sobre o **fluxo com condições**, os **branches** (ou **caminhos**, ou **arestas**) são as diferentes opções para onde o fluxo pode seguir, dependendo da condição.

Em um gráfico de controle de fluxo, esses **branches** são representados pelas **conexões entre os nós** (condições ou ações).



as formas do

IF

IF

Agora que remos uma ideia geral do controle de fluxo, podemos nos aprofundar no `if`, objetivo desse nosso encontro. O `if` aparece de duas formas na linguagem:

1. Como **expressão**:

```
result_true if cond else result_false
```

2. Como **statement**

```
if cond:
    # bloco
```

Expressão `if`

A expressão `if`, introduzida na PEP 308, é usada quando você precisa atribuir um valor com base em uma condição:

```
# sintaxe
result_true if cond else result_false

# Exemplo
status = "Ativo" if is_active else "Inativo"
```

Se a condição for verdadeira, o valor antes do `if` será retornado, caso não, o valor após o `else`.

É o "ternário" do python. Foram discutidas diversas formas ao longo do tempo e acabamos ficando com essa forma.

`if` como parte de expressões

Embora não sejam partes integrantes do `if`. O `if` se encontra em outras formas de expressões. Como as compreensões:

```
# list comp
l = [val for val in lista if cond]

# set comp
s = {val for val in lista if cond}

# dict comp
d = {chave: valor for chave, valor in dicionario if cond}

# gen exp
g = (val for val in lista if cond)
```

`if` statement

A forma mais comum de encontrar o `if` por aí é em sua forma de bloco. Algo como:

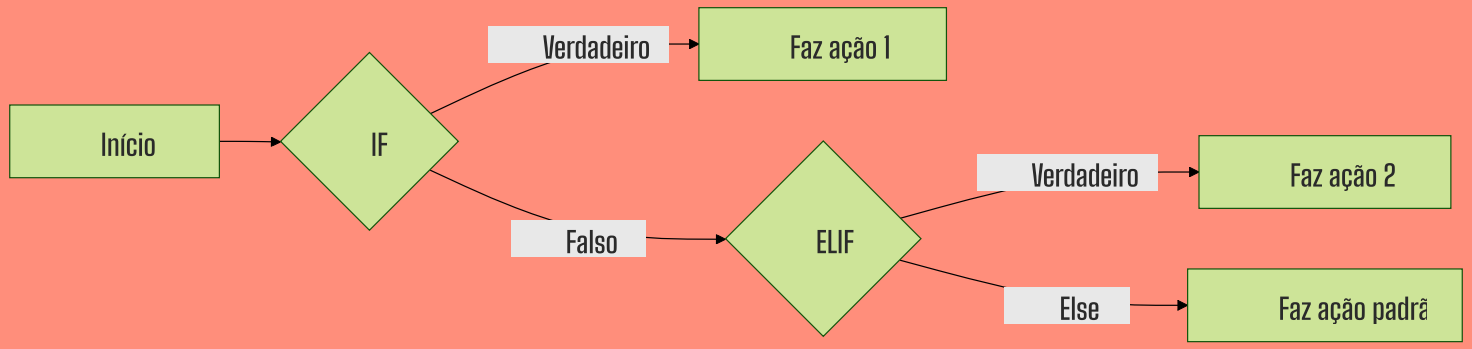
```
if cond:
   # bloco
```

Sua gramática é um pouco mais detalhada e permite mais operações:

```
if_stmt ::= "if" assignment_expression ":" suite
            ("elif" assignment_expression ":" suite)*
            ["else" ":" suite]
```

`if` statement

O que nos permite criar alguns desvios bastante complexos pela gramática:




```
if cond:
    ...
elif cond:
    ...
else:
    ...
```

`elif`

Dentro dessa condição. Acho que o **`elif`** se destaca como parte importante da estrutura de seleção, ao permitir que N caminhos sejam formados.

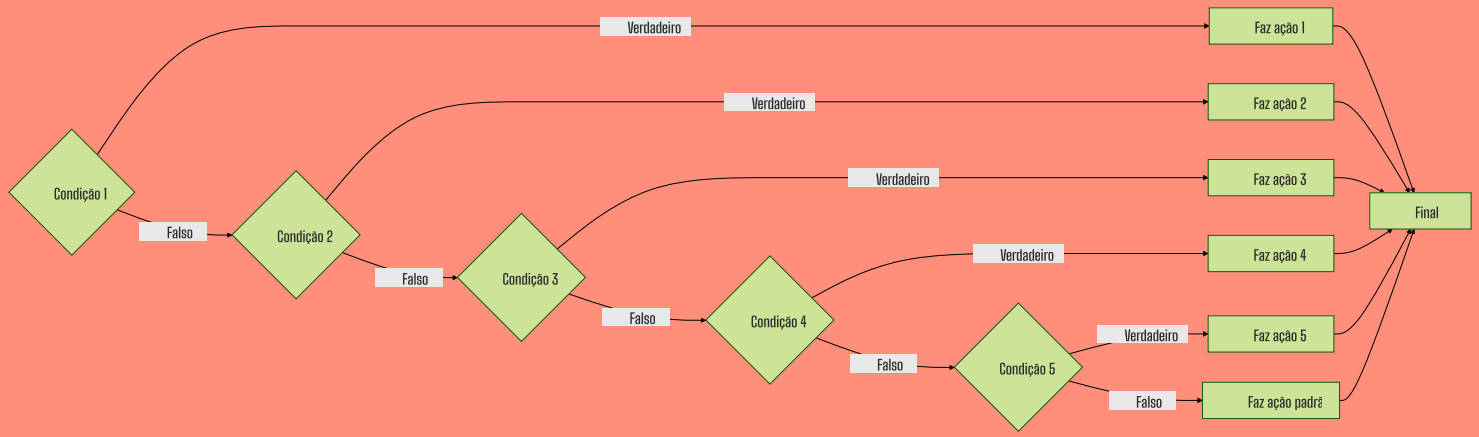
```
if cond:
    ...
elif cond:
    ...
elif cond:
    ...
elif cond:
    ...
```

Talvez com múltiplas seleções você deva usar **match/case**

elif

Ao mesmo tempo que é uma benção... é uma maldição.

Quanto mais desvios, menos previsibilidade e mais dificuldade pra testar o código.



Se possível, evite!

Expressões

Condicionais

Expressões condicionais

Vimos os desvios, um pouco de sintaxe, mas precisamos entender o que são as condições, afinal.

Gostaria de separá-las em categorias:

- **Expressões atômicas**
  - Que contém somente um objeto: `if a`
- **Expressões de comparação rica**
  - Que comparam dois objetos: `if a <comparação> b`
- **Expressões com operações**
  - Combinação de expressões: `if <exp> and | or <exp>`
- **Expressões de atribuição**
  - Que atribuem valores a variáveis: `if (var := <exp>) <exp>`

Expressões atômicas

A ideia de "atômico" vem de um único valor, um **único objeto**.

Por exemplo:

```
if '':
    ... # bloco

if 0:
    ... # bloco
```

Aqui ficam algumas perguntas:

- `P1`: O resultado dessa expressão usado no `if` é verdadeiro ou falso?
- `P2`: O que a linguagem considera para avaliar essa expressão atômica?
- `P3`: Como isso é feito em objetos criados por nós?

Avalização de condição

A primeira coisa que se deve pensar aqui é que toda a expressão deve ser resumida a um valor do tipo `bool`. Resultando em `True` ou `False`.

Para que isso seja feito, no caso atômico, o python vai fazer o que podemos simular com a função embutida `bool()`:

```
>>> bool('')
False

>>> bool(0)
False
```

Mas como ele sabe que é `True` ou `False`?

Avalização de condição

Quando a função `bool()` recebe um objeto, ela pode tomar caminhos diferentes, dependendo do tipo de dados.

A forma mais comum, para objetos "customizados" é chamar o método `__bool__`.

Fazendo que o objeto diga se, em seu estado atual, ele é `True` ou `False`. Ou seja, a responsabilidade é de quem implementa o objeto.

```
>>> bool(0)
False

>>> (0).__bool__()
False
```

Objetos customizados

Se quisermos que nosso objeto seja "condicionável", devemos implementar o método especial `__bool__`. Algo como:

```
class MeuObjeto:
    def __bool__(self):
        """ps: fazer uma validação que faça sentido xD"""
        print('Chamei __bool__')
        return True

>>> bool(MeuObjeto()
Chamei __bool__
True

>>> if MeuObjeto():
...  print('É True!')
Chamei __bool__
É True!!!
```

As exceções

As exceções à regra são as **sequências** e **mappings**, por padrão (cpython), não implementam `__bool__`. A validade delas é em relação ao tamanho. Ou seja, `__len__`:

```
// object.c - CPython
int PyObject_IsTrue(PyObject *v) {
    // ...
    else if (Py_TYPE(v)->tp_as_mapping != NULL &&
             Py_TYPE(v)->tp_as_mapping->mp_length != NULL)
        res = (*Py_TYPE(v)->tp_as_mapping->mp_length)(v);
    else if (Py_TYPE(v)->tp_as_sequence != NULL &&
             Py_TYPE(v)->tp_as_sequence->sq_length != NULL)
        res = (*Py_TYPE(v)->tp_as_sequence->sq_length)(v);
        // ...
}
```

As exceções

Ou seja, quando o tamanho delas for igual a zero, não tendo nenhum elemento, a resposta será `False`. Caso contrário, `True`:

```
>>> bool('')
False

>>> ('').__len__()
0

>>> bool([])
False

>>> ([]).__len__()
0

>>> (dict()).__len__()
0
```

Cheira mal...

Por isso, algumas comparações não fazem sentido *pythonicamente falando*.

Como aquele sotaque que a gente encontra por aí...:

```
if numero != 0:
    ... -> if numero

if len(sequencia) > 0:
    ... -> if sequencia

if var is False:
    ... -> if var

if bool(objeto) == True:
   ... -> if objeto
```

Entender as condições é **lindo de mais**.

Comparações ricas

Um dos casos mais comuns de encontrar em desvios condicionais são as **comparações** entre objetos:

```
if objeto_a > objeto_b: ...

if objeto_a < objeto_b: ...

if objeto_a == objeto_b: ...

if objeto_a != objeto_b: ...

if objeto_a >= objeto_b: ...

if objeto_a <= objeto_b: ...
```

Queremos saber como comparar `objeto_a` com `objeto_b`.

Comparações ricas

Todas as operações de comparação entre objetos, que constantemente usamos no `if`. São métodos **dunder** implemementados em objetos.

Operação	Métodos especiais
`>`	`__gt__`
`>=`	`__ge__`
`<`	`__lt__`
`<=`	`__le__`
`==`	`__eq__`
`!=`	`__ne__`

Isso pode ser customizado a depender do objeto. Ele só precisa implementar esses métodos.

Comparações ricas

Um exemplo **realmente** simples:

```
class MeuComparavel:
    def __init__(self, val: int):
        self.val = val

    def __eq__(self, other: int) -> bool:
        print('Será que sou igual?')
        return self.val == other
```

Poderia ser usado em uma comparação:

```
>>> MeuComparavel(3) == 3
Será que sou igual?
True
```

Comportamentos legais!

Mas pera aí... vamos fazer outro teste

```
class MeuComparavel:
    def __init__(self, val: int):
        self.val = val

    def __eq__(self, other: int) -> bool:
        print('Será que sou igual?')
        return self.val == other
```

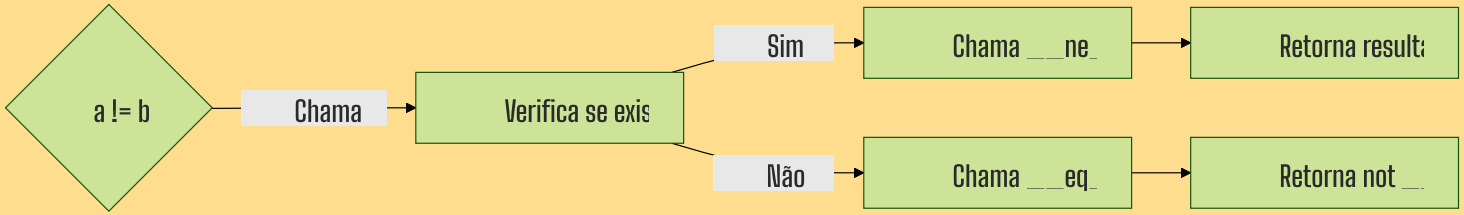
Poderia ser usado em uma comparação:

```
>>> MeuComparavel(3) != 3  # QQQQQ???????
Será que sou igual?
False
```

O comportamento é legal :)

Para qualquer comparação rica, caso o método não esteja implementado, mas o inverso esteja, ele vai fazer a negação do resultado.

Em nosso exemplo anterior, vimos algo com isso:



Não existia **__ne__**, então o python chamou o **__eq__** e inverteu o resultado.

Expressões com operações

Agora que entendemos as operações atômicas e as comparações ricas. Podemos usar uma combinação de tudo isso de forma booliana.

Com os operadores **`and`**, **`or`** e **`not`**.

Por exemplo:

```
if val and not val > 10:
    ...
```

Aqui nada muda radicalmente, podemos compor e negar expressões de forma mais eficiente.

Expressões com operações

A ideia é simplesmente compor expressões.

Operador	O que fazem?	Exemplo
`and`	Verifica se ambas as expressões são verdadeiras	`a and b`
`or`	Verifica se pelo menos uma expressão é verdadeira	`a or b`
`not`	Nega o resultado de uma expressão	`not a`

Como:

```
>>> not (True and True)  # `not` nega o resultado dos ()
False

>>> not bool('') or False    # `not` nega o True
True
```

Embora não pareça, temos um pequeno segredinho divertido aqui.

Curto-circuíto

Vamos brincar com funções:

```
def true():
    print('func: True')
    return True
```

```
def false():
    print('func: False')
    return False
```

A avaliação varia em relação ao resultado do primeiro termo.

```
>>> false() and true()  # `and`: o primeiro é false, o resultado é False
func: False
False

>>> true() or false() # `or`: o primeiro é true, o resultado é True
func: True
True
```

Expressões de atribuição

Outra forma comum, relativamente nova, de criar expressões foi introduzida na PEP 572. As **expressões de atribuição**.

São formas de criar novas "variáveis" no meio de expressões:

```
>>> (a := 1)
1
>>> a
1
```

O que raios isso faz?

1. Atribui o valor à variável
2. retorna o resultado da expressão

Expressões de atribuição

Isso pode ser usado nos fluxos em vários momentos. Por exemplo:

```
import re
texto = '123a456'

numeros = re.findall(r'\d+', texto)
if numeros and len(numeros) > 1:
    print(numeros) # [123, 456]
```

Poderíamos definir a variável na expressão do `if`:

```
if (numeros := re.findall(r'\d+', texto)) and len(numeros) > 1:
    print(numeros) # [123, 456]

# ou então, minha forma preferida :)
if len(numeros := re.findall(r'\d+', texto)) > 1:
    print(numeros) # [123, 456]
```

Dicas

Ou algo que o valha!

Por que as pessoas "criticam" o if?

As críticas ao uso excessivo de **`if`** geralmente giram em torno de dois fatores principais:

1. **Complexidade ciclomática**: quanto mais condições você adiciona, mais difícil se torna entender e testar o código, pois o número de caminhos de execução aumenta exponencialmente.
    - A medida de qual espaguete está o seu fluxo
2. **Previsibilidade e legibilidade**: blocos de código muito ramificados com **`if/elif`** podem se tornar difíceis de manter, já que os caminhos de execução ficam cada vez mais complexos e difíceis de seguir.
    - Isso impacta na leitura e durante os testes

Complexidade ciclomatica

Complexidade ciclomática é uma medida que quantifica a complexidade de um programa, com base no número de caminhos independentes no fluxo de controle.

Formula:

$$V(G) = E - N + 2P$$

Onde:

- **V(G)** é o número de McCabe (complexidade ciclomática)
- **E** é o número de arestas no grafo de controle de fluxo
- **N** é o número de nós no grafo de controle de fluxo
- **P** é o número de componentes conectados (geralmente 1)

Quanto maior a complexidade, mais difícil é manter o código e testar todos os caminhos possíveis.

Complexidade ciclomatica

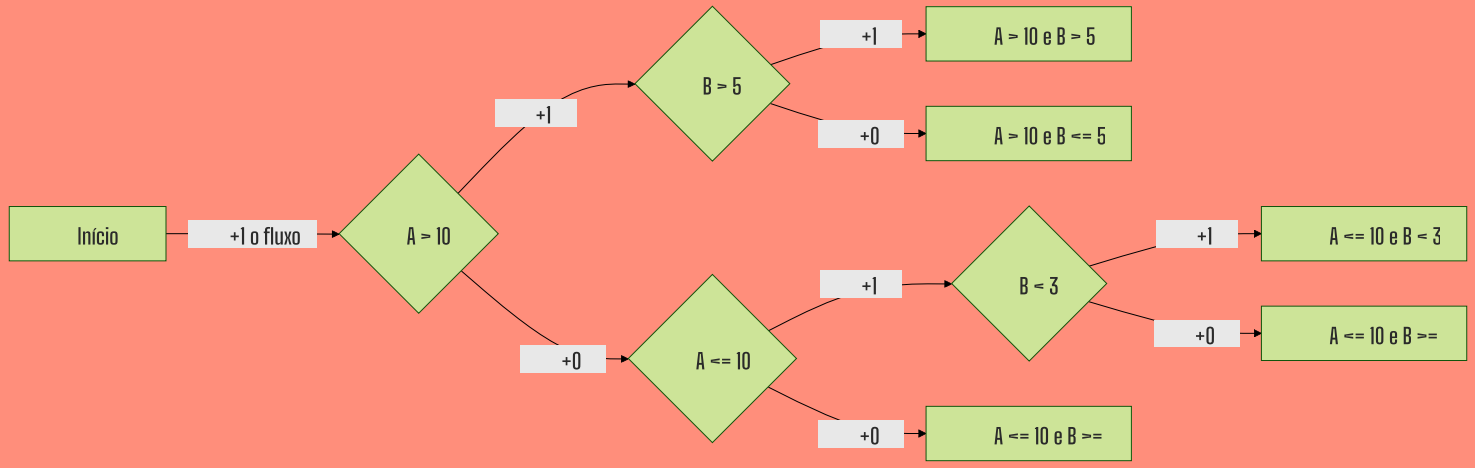
Aqui, você pode criar um exemplo simples de código com várias condições e, em seguida, calcular sua complexidade ciclomática.

```
def processa_valores(a, b):
    if a > 10:
        if b > 5:
            print("A > 10 e B > 5")
        else:
            print("A > 10 e B <= 5")
    elif a <= 10:
        if b < 3:
            print("A <= 10 e B < 3")
        else:
            print("A <= 10 e B >= 3")
```

Analizando

Felizmente, não precisamos fazer a conta :)

```
$ pipx run radon cc -s comp.py
comp.py
    F 1:0 processa_valores - A (5)
```



Radon pode te ajudar

Construct	Effect on CC	Reasoning
if	+1	An `if` statement is a single decision.
elif	+1	The `elif` statement adds another decision.
else	+0	The `else` statement does not cause a new decision. The decision is at the `if`.
for	+1	There is a decision at the start of the loop.
while	+1	There is a decision at the `while` statement.
except	+1	Each `except` branch adds a new conditional path of execution.
finally	+0	The finally block is unconditionally executed.
with	+1	The `with` statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	+1	The `assert` statement internally roughly equals a conditional statement.
Comprehension	+1	A list/set/dict comprehension of generator expression is equivalent to a for loop.
Boolean Operator	+1	Every boolean operator (and, or) adds a decision point.

https://radon.readthedocs.io/en/latest/intro.html#cyclomatic-complexity

Referências

- Robert Sebesta: Princípios de linguagens de programação, 11ed.
- Documentação do python:
  - Referência da linguagem
    - A instrução if
    - Expressões condicionais
    - Operações booleanas
    - Comparações
    - Expressões de atribuição
  - Modelo de dados:
    - Comparação rica
- PEPs
  - Rich Comparisons - PEP 207
  - Conditional Expressions - PEP 308
  - Assignment Expressions - PEP 572