

Zander - Flamboyant (2016)

# Live de Python #65

Programação orientada a objetos #4

# Ajude a Live de Python

[apoia.se/livedepython](https://apoia.se/livedepython)

picPay: @livedepython

Desconto 30% novatec: MENDESPY

# Roteiro

- Sobrecarga de operadores
- Operadores Unários
  - -, +, ~, ...
- Operadores infixos
  - +, -, /, \*, ..
- Operadores *inplace*
  - +=, -=, \*=, ...
- Operadores que não podem ser sobrescritos

[https://docs.python.org/3/reference/lexical\\_analysis.html](https://docs.python.org/3/reference/lexical_analysis.html)

<https://docs.python.org/3/reference/datamodel.html>

# Quando essa live acabar

Você é **OBRIGADO** a assistir as lives:

- [32 - collections.abc / Collections #5](#)
- [Rapidinha Pythonica #3 - Programação declarativa com Python](#)
- [43 - Gerenciadores de contexto](#)
- [59 - Objetos assíncronos](#)

# Sobrecarga de operadores

“Operadores aritméticos são frequentemente usados para mais de um propósito. Por exemplo, + geralmente é usado para especificar adição de inteiros e adição de ponto flutuante. Algumas linguagens também o usam para catenação de strings.

```
In [1]: 2 + 2
Out[1]: 4

In [2]: 'Live ' + 'de ' + 'Python'
Out[2]: 'Live de Python'
```

Esse **uso múltiplo** de um operador é chamado de **sobrecarga do operador**”

# Sobrecarga de operadores

“A sobrecarga de operadores permite interoperação entre objetos definidos pelo usuário e os operadores”. (*Fluent Python / Luciano Ramalho*)

E digo mais, operações entre tipos nativos com os “nossos tipos”

```
In [1]: class Dois:
...:     val = 2
...:     def __add__(self, val):
...:         return self.val + val
...:

In [2]: Dois() + 2
Out[2]: 4
```

# Operadores unários

Existem 3 tipos de operadores unários em python, que necessariamente foram projetados para trabalhar com números:

- `+`: Operador para números positivos ex: `+20`, `+50`
- `-`: Operador para números negativos ex: `-20`, `-30`
- `~`: Operador de bitwise para  $(-x - 1)$  ex: `~19 == -20`

```
In [1]: +20
```

```
Out[1]: 20
```

```
In [2]: -20
```

```
Out[2]: -20
```

```
In [3]: ~20
```

```
Out[3]: -21
```

# Operadores unários

Existem 3 tipos de operadores unários em python, que necessariamente foram projetados para trabalhar com números:

- `+`: Operador para números positivos ex: `+20`, `+50`
- `-`: Operador para números negativos ex: `-20`, `-30`
- `~`: Operador de bitwise para  $(-x - 1)$  ex: `~19 == -20`

Em caso de expressões, estes operadores serão invocados antes da expressão

```
In [1]: +20
```

```
Out[1]: 20
```

```
In [2]: -20
```

```
Out[2]: -20
```

```
In [3]: ~20
```

```
Out[3]: -21
```



# Operadores unários

```
In [32]: class Número:
...:     def __init__(self, n):
...:         self.val = n
...:     def __neg__(self):
...:         print('chamei neg')
...:         return Número(-self.val)
...:     def __pos__(self):
...:         print('chamei pos')
...:         return Número(+self.val)
...:
```

Como é possível notar, os operadores unários, são invocados antes

```
In [33]: -Número(2) + +Número(2)
chamei neg
chamei pos
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-33-e7df77a3ba7f> in <module>()
----> 1 -Número(2) + +Número(2)
```

```
TypeError: unsupported operand type(s) for +: 'Número' and 'Número'
```

# Operadores unários

```
In [32]: class Número:
...:     def __init__(self, n):
...:         self.val = n
...:     def __neg__(self):
...:         print('chamei neg')
...:         return Número(-self.val)
...:     def __pos__(self):
...:         print('chamei pos')
...:         return Número(+self.val)
...:
```

Como é possível notar, os operadores unários, são invocados antes

```
In [33]: -Número(2) + +Número(2)
chamei neg
chamei pos
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-33-e7df77a3ba7f> in <module>()
----> 1 -Número(2) + +Número(2)
```

```
TypeError: unsupported operand type(s) for +: 'Número' and 'Número'
```

# Operadores unários

```
In [32]: class Número:
...:     def __init__(self, n):
...:         self.val = n
...:     def __neg__(self):
...:         print('chamei neg')
...:         return Número(-self.val)
...:     def __pos__(self):
...:         print('chamei pos')
...:         return Número(+self.val)
...:
```

Como é possível notar, os operadores unários, são invocados antes

```
In [33]: -Número(2) + +Número(2)
chamei neg
chamei pos
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-33-e7df77a3ba7f> in <module>()
----> 1 -Número(2) + +Número(2)

TypeError: unsupported operand type(s) for +: 'Número' and 'Número'
```

# Operadores infixos

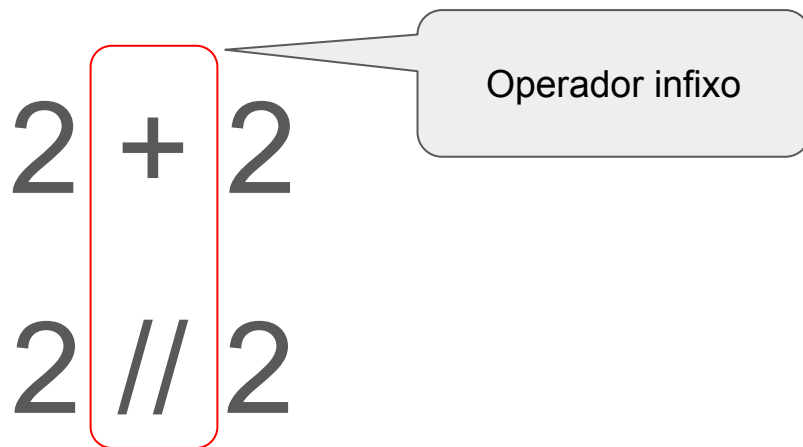
Operadores infixos são aqueles que ficam “entre” objetos.

$2 + 2$

$2 // 2$

# Operadores infixos

Operadores infixos são aqueles que ficam “entre” objetos.



# Operadores infixos

Existe uma gama grande de operadores infixos em Python

## 2.5. Operators

The following tokens are operators:

+	-	*	**	/	//	%	@
<<	>>	&		^	~		
<	>	<=	>=	==	!=		

[https://docs.python.org/3/reference/lexical\\_analysis.html#operators](https://docs.python.org/3/reference/lexical_analysis.html#operators)

# Operadores infixos

E pra cada um desses operadores temos um **dunder** específico

Operador	Método
+	<code>__add__</code>
-	<code>__sub__</code>
*	<code>__mul__</code>
/	<code>__floordiv__</code>
//	<code>__truediv__</code>
<	<code>__lt__</code>
<=	<code>__le__</code>

Operador	Método
==	<code>__eq__</code>
<<	<code>__lshift__</code>
>>	<code>__rshift__</code>
%	<code>__mod__</code>
&	<code>__and__</code>
	<code>__or__</code>
...	<code>__...__</code>

# Operadores infixos

Tá, chega de enrolação, vamos fazer ...

```
In [1]: class Somável:  
...:     def __add__(self, OUTRO_VALOR):  
...:         print('Olha só, eu sei somar')  
...:
```

```
In [2]: Somável() + 7  
Olha só, eu sei somar
```

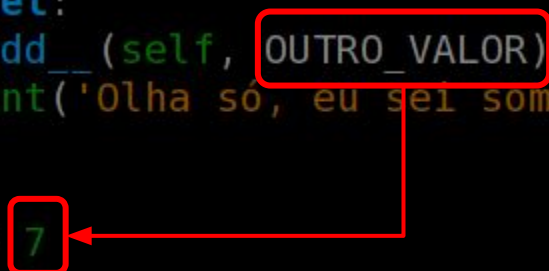


# Operadores infixos

Tá, chega de enrolação, vamos fazer ...

```
In [1]: class Somável:
...:     def __add__(self, OUTRO_VALOR):
...:         print('Olha só, eu sei somar')
...:

In [2]: Somável() + 7
Olha só, eu sei somar
```

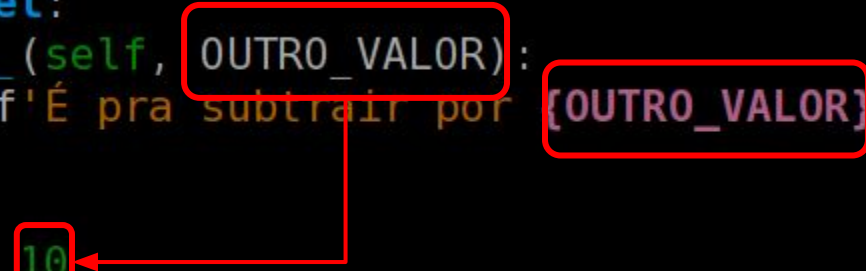
A red box highlights the parameter 'OUTRO\_VALOR' in the method definition of the first code block. A red line extends from this box, ending in an arrow that points to the number '7' in the second code block, illustrating how the value '7' is passed as an argument to the '\_\_add\_\_' method.

# Operadores infixos

Tá, chega de enrolação, vamos fazer ...

```
In [1]: class Subtraível:
...:     def __sub__(self, OUTRO_VALOR):
...:         print(f'É pra subtrair por {OUTRO_VALOR}?')
...:

In [2]: Subtraível() - 10
É pra subtrair por 10?
```



# Agora você está grandinho pra saber



Uma boa prática para objetos que sobrecarregam operadores é retornar um novo objeto.

Pense que operadores em expressões devem sempre ser resolvidos.

Exemplo:

`MeuTipo() + 2 + 10`

`MeuTipo.__add__(2)`



`????.__add__(10)`

# Agora você está grandinho pra saber



```
class Somável:  
    def __add__(self, OUTRO_VALOR):  
        print('Olha só, eu sei somar')
```

Somável() + 2 + 10

```
In [3]: result = Somável() + 10  
Olha só, eu sei somar  
  
In [4]: type(result)  
Out[4]: NoneType  
  
In [5]: hasattr(result, '__add__')  
Out[5]: False
```

Somável.\_\_add\_\_(2)



None.\_\_add\_\_(10)

Agora você está grandinho pra saber



```
In [6]: result + 2
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-6-6bb0aaccb514> in <module>()  
----> 1 result + 2
```

```
TypeError: unsupported operand type(s) for +: 'NoneType' and  
'int'
```

Agora você está grandinho pra saber



```
In [9]: class Somável:
...:     def __add__(self, OUTRO_VALOR):
...:         print('Olha só, eu sei somar')
...:         return Somável()
...:
...:
```

```
In [10]: Somável() + 2 + 10
```

```
Olha só, eu sei somar
```

```
Olha só, eu sei somar
```

```
Out[10]: <__main__.Somável at 0x7f6e1e5a3cf8>
```

# Operadores infixos

Tá, ok. Era só isso? **NÃO**. Em python tem uma peculiaridade interessante. A **comutatividade** não é verdadeira por definição. Hã?

**Dois() + 2** é diferente de **2 + Dois()**

# Operadores infixos

Tá, ok. Era só isso? **NÃO**. Em python tem uma peculiaridade interessante. A **comutatividade** não é verdadeira por definição. Hã?

**Dois() + 2** é diferente de **2 + Dois()**

```
In [11]: Somável() + 2
Olha só, eu sei somar
Out[11]: <__main__.Somável at 0x7f6e1e5a37b8>

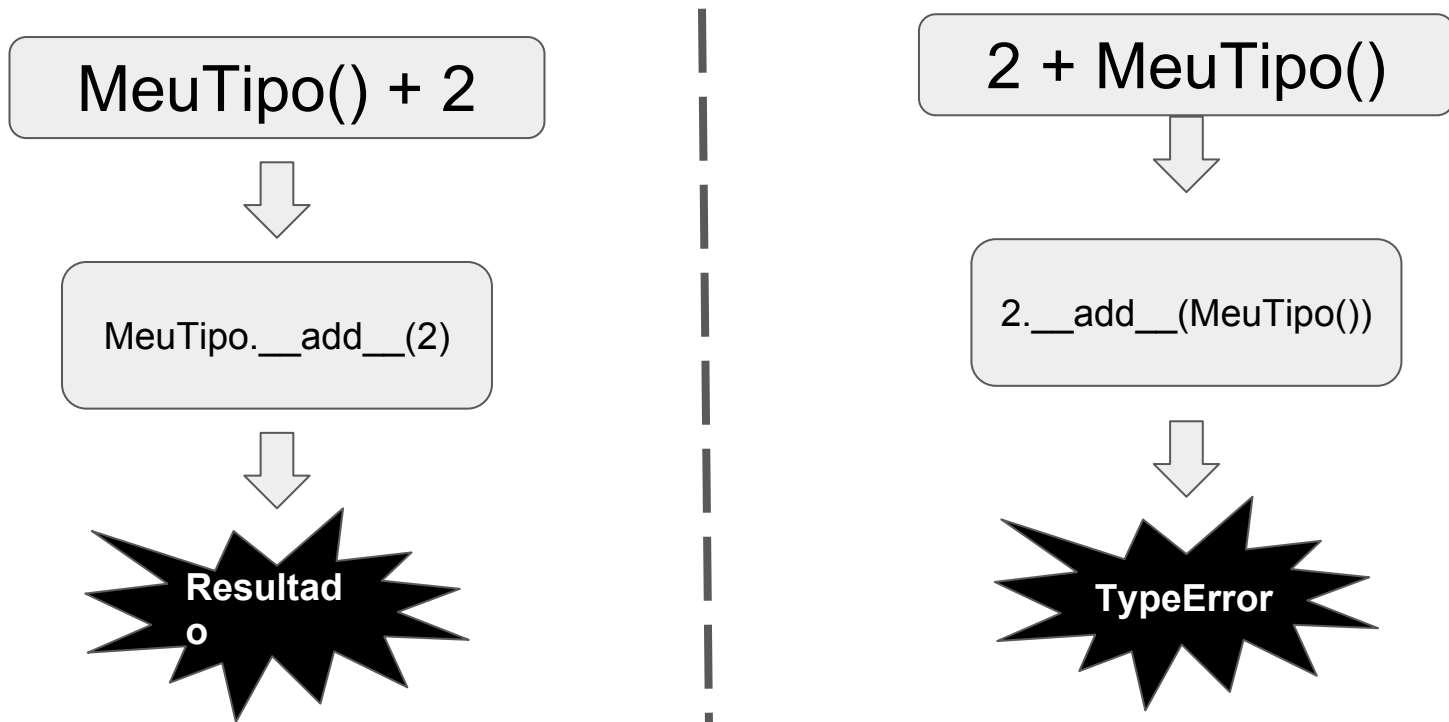
In [12]: 2 + Somável()
-----
TypeError                                Traceback (most recent call last)
<ipython-input-12-6ef3b5be3552> in <module>()
----> 1 2 + Somável()

TypeError: unsupported operand type(s) for +: 'int' and 'Somável'
```



# Operadores infixos

Ok, vamos tentar entender a ordem em que o python resolve as expressões



# Operadores infixos

Ok, vamos tentar entender a ordem em que o python resolve as expressões

Meu tipo  
sabe ler  
tipos nativos

`MeuTipo() + 2`



`MeuTipo.__add__(2)`



**Resultado**

o

`2 + MeuTipo()`



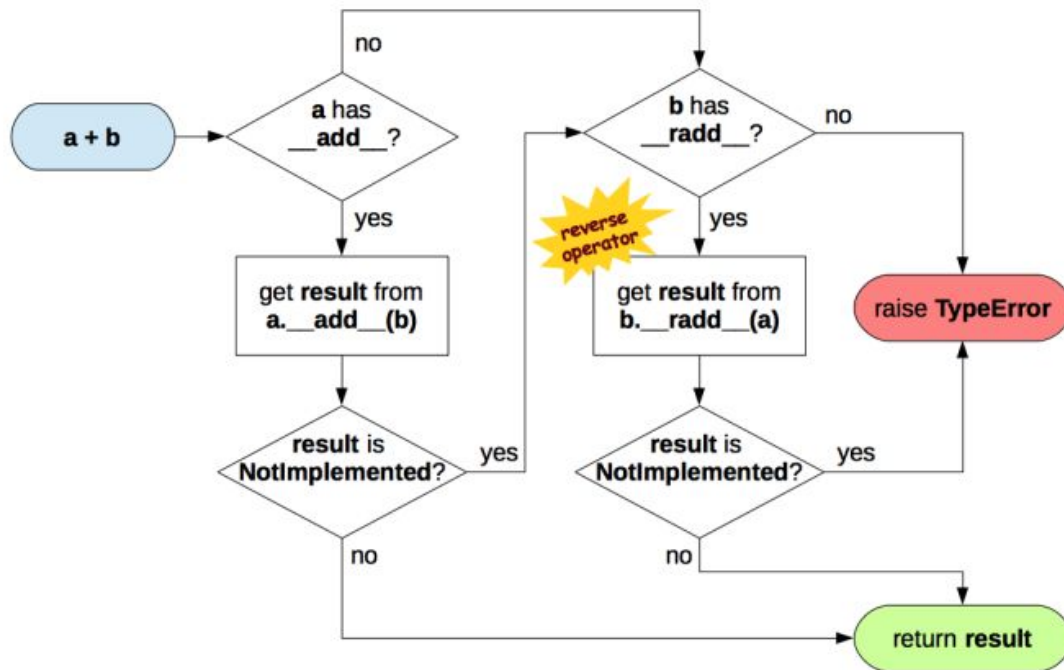
`2.__add__(MeuTipo())`



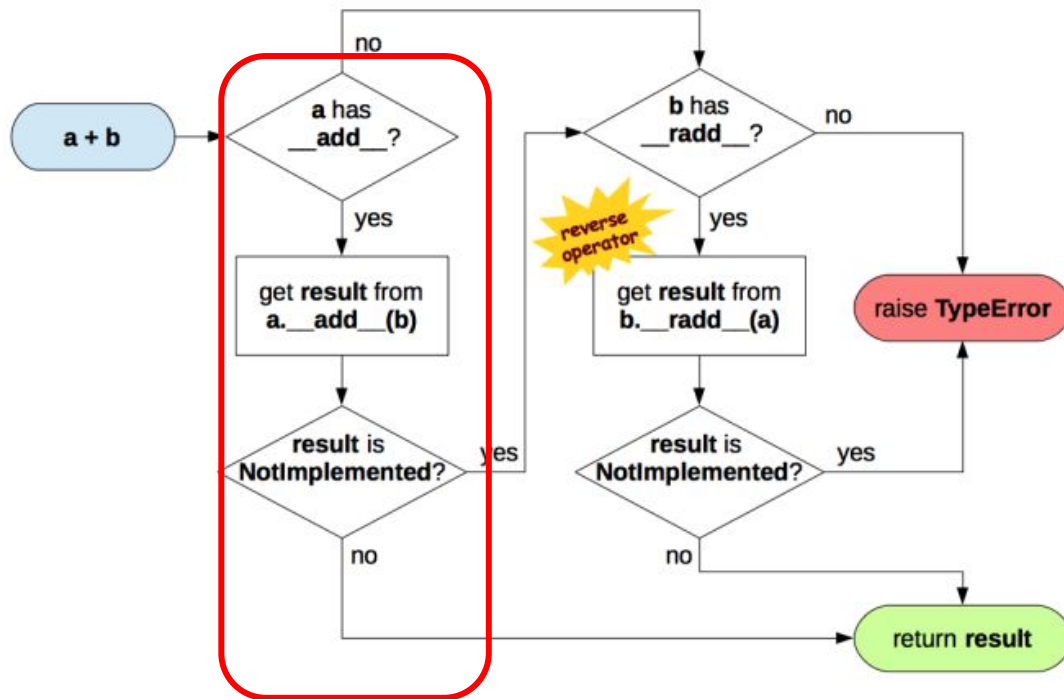
**TypeError**

Os tipos  
nativos não  
sabem ler  
meu tipo

# Operadores infixos



# Operadores infixos

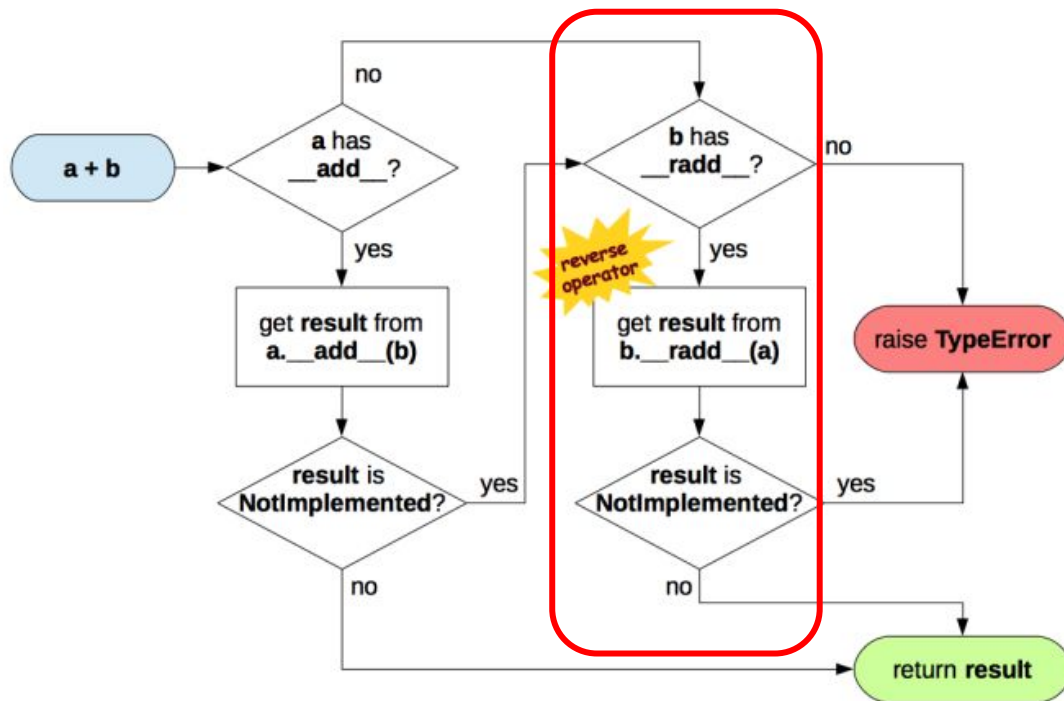


# Operadores infixos

E pra cada um desses operadores temos um **dunder** específico e um reverso

Operador	Método	Reverso
+	<code>__add__</code>	<code>__radd__</code>
-	<code>__sub__</code>	<code>__rsub__</code>
*	<code>__mul__</code>	<code>__rmul__</code>
/	<code>__floordiv__</code>	<code>__rfloordiv__</code>
//	<code>__truediv__</code>	<code>__rtruediv__</code>
<<	<code>__lshift__</code>	<code>__rlshift__</code>
>>	<code>__rshift__</code>	<code>__rrshift__</code>

# Operadores infixos



# Operador infixo com operação reversa

```
In [15]: class Somável:
...:     def __add__(self, OUTRO_VALOR):
...:         print('Olha só, eu sei somar')
...:         return Somável()
...:     def __radd__(self, OUTRO_VALOR):
...:         print('Olhá, já sei somar reverso')
...:         return Somável()
...:
```

```
In [16]: Somável() + 2
```

```
Olha só, eu sei somar
```

```
Out[16]: <__main__.Somável at 0x7f6e1e53e3c8>
```

```
In [17]: 2 + Somável()
```

```
Olhá, já sei somar reverso
```

```
Out[17]: <__main__.Somável at 0x7f6e1e567518>
```

# Operadores infixos (sem reverso)

Métodos de comparação não tem inverso, pois eles já têm a inversão

Operador	Método	Reverso
==	__eq__	?
!=	__ne__	?
<=	__le__	?
>=	__ge__	?
<	__lt__	?
>	__gt__	?



# Operadores infixos (sem reverso)

Métodos de comparação não tem inverso, pois eles já têm a inversão

Operador	Método	Reverso
==	__eq__	__ne__
!=	__ne__	__eq__
<=	__le__	__ge__
>=	__ge__	__le__
<	__lt__	__gt__
>	__gt__	__lt__

Vamos fazer uma lista  
bem louca?

# Operadores *inplace*

Você deve ter notado que durante essa longa jornada alguns operadores foram esquecido, não, eles não foram...

Operadores *inplace* são infixos também, mas ele tem um propósito um pouco diferente.

`Obj() += 5`

# Operadores *inplace*

*inplace* poderia ser traduzido com “No lugar”. Ou seja, a expressão não tem um resultado, ela modifica o objeto que tem o operado.

```
In [1]: n = 10  
In [2]: n += 5  
In [3]: n  
Out[3]: 15
```



```
In [5]: n + 5  
Out[5]: 20  
In [6]: n  
Out[6]: 15
```

# Operadores *inplace*

Operador	<i>inplace</i>
<code>+=</code>	<code>__iadd__</code>
<code>-=</code>	<code>__isub__</code>
<code>*=</code>	<code>__imul__</code>
<code>/=</code>	<code>__ifloordiv__</code>
<code>//=</code>	<code>__itruediv__</code>
<code>&lt;&lt;=</code>	<code>__ilshift__</code>
<code>&gt;&gt;=</code>	<code>__irshift__</code>

Por boas práticas operadores *inplace* nunca devem retornar nada, CLARO, estão modificando o próprio objeto

# Tudo Junto, agora!

Operador	Método	Reverso	<i>inplace</i>
+	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>
-	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>
*	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>
/	<code>__floordiv__</code>	<code>__rfloordiv__</code>	<code>__ifloordiv__</code>
//	<code>__truediv__</code>	<code>__rtruediv__</code>	<code>__itruediv__</code>
<<	<code>__lshift__</code>	<code>__rlshift__</code>	<code>__ilshift__</code>
>>	<code>__rshift__</code>	<code>__rrshift__</code>	<code>__irshift__</code>

