Como definir uma função em Python

```
1 def nome_funcao(lista_parametros):
2   """Como se define uma funcao em Python"""
3   return valor de retorno
```

Soma

- **Documentação / Comentário** : """Esta e a funcao soma que dados os valores de x e y retorna o valor de x + y"""
- **Nome da Função**: soma
- **Parâmetros**: x,y
- **Valor de Retorno**: : x+y

```
1 def soma(x,y):
2   """Esta e a funcao soma que dados os valores de x e y
3   retorna o valor de x + y"""
4   return x+y
```

Função

Argumentos Default: Permitem que valores default sejam utilizados quando nenhum valor é especificado em um certo parâmetro.

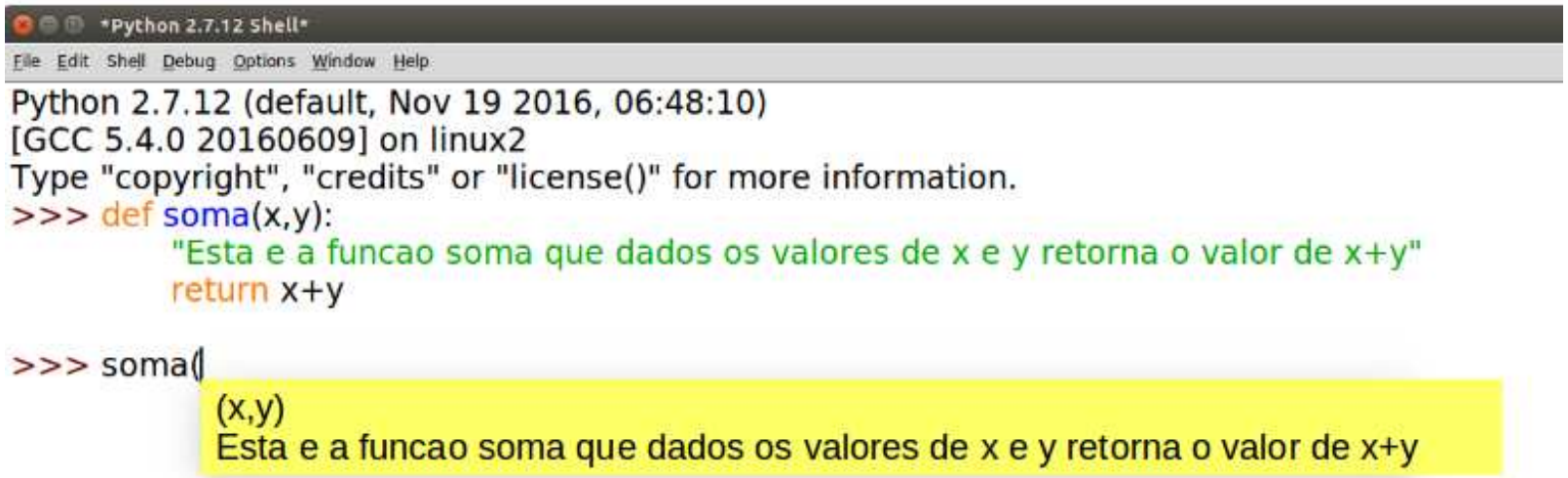
Formato

$$\text{def nome-funcao}(arg_0, \cdots, arg_N, arg_{N+1} = default_1, \cdots, arg_M = default_M)$$

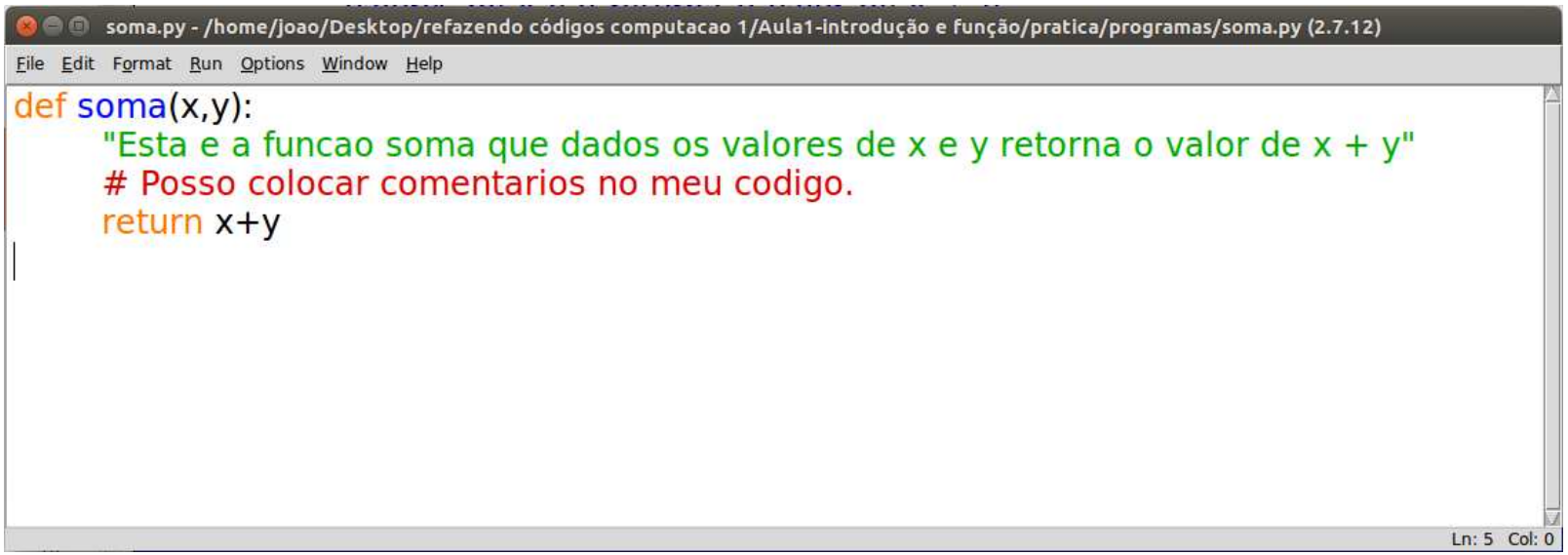
$$\cdots$$

- $arg_0, \cdots, arg_N$: *Argumentos sem valores default.*
- $arg_{N+1} = default_1, \cdots, arg_M = default_M$: *Argumentos com valores default. Devem ser sempre os últimos argumentos.*

Executando funções interativamente



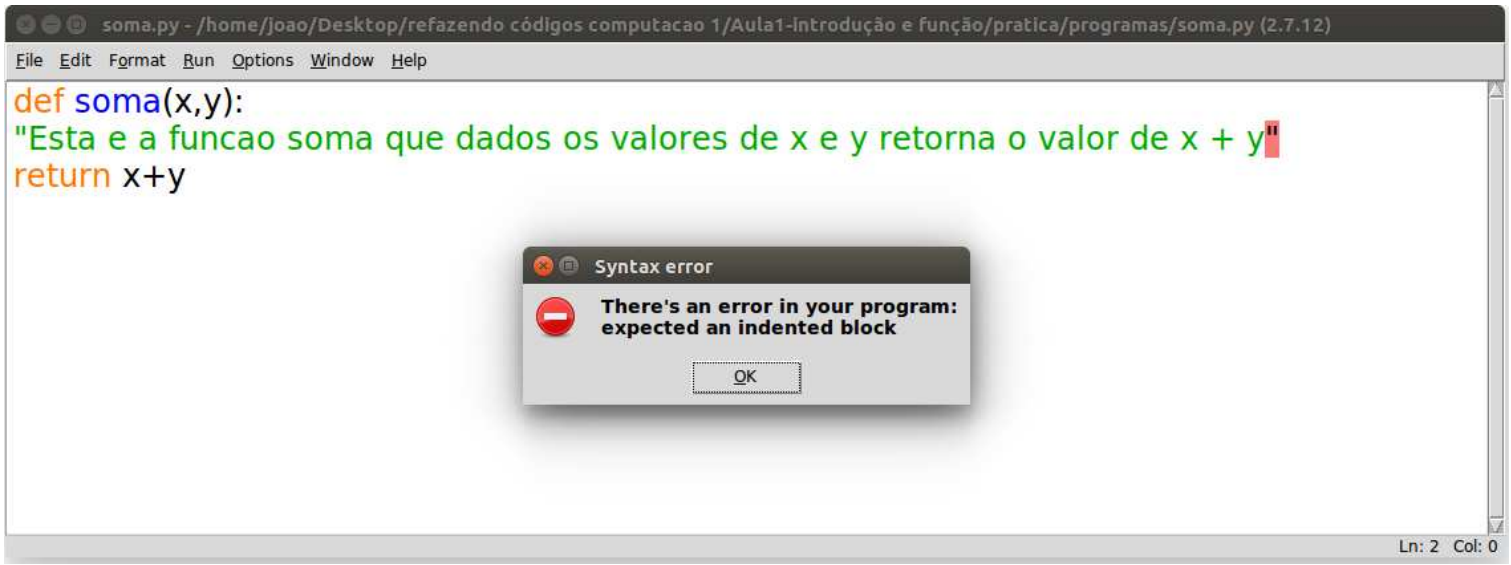
Executando funções interativamente



Usamos o símbolo # no início do comentário. Embora não obrigatório, documentar as funções que você faz é fortemente recomendado!

Editor IDLE

A identação é parte da sintaxe do Python. É através dela que podemos construir estruturas de código, como as funções.



Mensagens de Erro

Ao tentar interpretar o código que escrevemos, o Python avisa quando alguma coisa não foi compreendida através das mensagens de erro.
É importante ler as mensagens para saber onde estamos errando.

Erros Frequentes

- **SyntaxError**: erros de sintaxe. Alguma palavra foi escrita incorretamente, ou algum símbolo foi esquecido (por exemplo, o : ao final da definição de uma função)
- **IndentationError**: Alguma linha teve sua identação alterada manualmente (e erroneamente).

```
>>> def mult(x,y):
return x*y
File "<pyshell#9>", line 2
    return x*y
         ^
IndentationError: expected an indented block
```

Entrada e Saída de dados - Interação com o Usuário

Escrevendo informações com o print

- print(exp1,exp2,exp3) → próximo print na linha seguinte.
- print(exp1,exp2,exp3,end="") → próximo print na mesma linha.

```
1 print("Eu")
2 print("saio", "no bloco")
3 print("Suvaco do Cristo")
```

Eu
saio no bloco
Suvaco do Cristo

```
1 print("Eu")
2 print("saio", "no bloco",end="")
3 print("Suvaco do Cristo")
```

Eu
saio no bloco Suvaco do Cristo

Entrada e Saída de dados - Interação com o Usuário

Escrevendo informações com o print

- "\n" é usado para quebra de linha quando usado dentro de uma string.

```
>>> print("Meu nome e Jose \n e eu tenho 10 anos")
Meu nome e Jose
e eu tenho 10 anos

>>> print("A soma de 2 e 3 e : \n", soma(2,3))
A soma de 2 e 3 e :
5

>>> print("A soma de 2 e 3 e : ", soma(2,3))
A soma de 2 e 3 e : 5
```

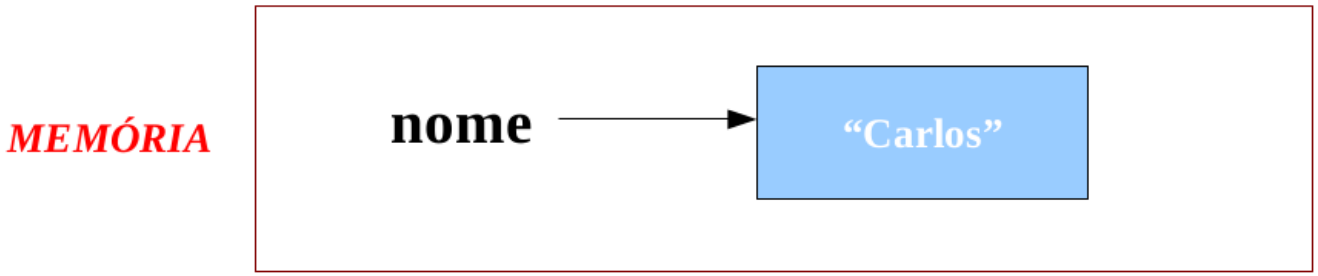
Variáveis e Atribuição

- **Atribuição**: O símbolo = é usado para atribuir um valor a uma variável.

$$var = valor$$

$$var1, var2, ..., varN = valor1, valor2, ..., valorN$$

```
...
nome = "Carlos"
return "Olá "+ nome
```



Atribuindo Valores a Variáveis

ATENÇÃO

Qual a diferença entre as funções abaixo ?

```
1 def teste1():
2   a = 10
3   a, b = 3 * a, a
4   return a, b
```

```
1 def teste2():
2   a = 10
3   a = 3 * a
4   b = a
5   return a, b
```

Atribuindo Valores a Variáveis

ATENÇÃO

Qual a diferença entre as funções abaixo ?

```
1 def teste1():
2   a = 10
3   a, b = 3 * a, a
4   return a, b
```

```
1 def teste2():
2   a = 10
3   a = 3 * a
4   b = a
5   return a, b
```

O lado direito da atribuição é sempre avaliado antes que a atribuição seja feita.

Variáveis – Escopo

- **Escopo**: onde a variável existe e onde ela deixa de existir.
- As variáveis definidas dentro de uma função são ditas **variáveis locais**, porque não podem ser acessadas fora da função.

```
1 def produtoSomaDiferenca(a,b):
2   x = a + b
3   y = a - b
4   return x*y
```

- As variáveis x e y são locais, pois só existem dentro da função. Depois que a função é executada, elas são destruídas.
- Dizemos que a função é o escopo de x e y.
- Tentar chamá-las fora da função ocasionaria um erro.

Atribuindo Valores a Variáveis

Uma variável é criada com um comando de atribuição: *variavel* = *valor*

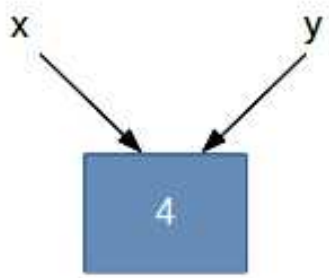
```
1 >>> x = 4
```

Um alias é um identificador que se refere a uma variável existente. É criado com uma atribuição *variavel* = *outra_variavel* já existente

```
1 >>> y = x
```

A variável *y* é um alias para a variável *x*. Portanto, *y* possui o mesmo valor e aponta para a mesma posição na memória que *x*.

```
1 >>> y
2     4
```



Tipos Numéricos

- **Números Inteiros**: Int

  Os inteiros (int) têm precisão fixa ocupando tipicamente uma palavra de memória

  Em PC's são tipicamente representados com 32 bits (de -2^{31} a $2^{31}-1$)

- **Ponto Flutuante**: Float
  Constantes têm que possuir um ponto decimal ou serem escritas em notação científica com a letra "e" (ou "E") precedendo a potência de 10

  10 int
  10.0 float

- **Números Complexos**: Complex
  Representados com dois números de ponto flutuante: um para a parte real e outro para a parte imaginária.
  Constantes são escritas como uma soma sendo que a parte imaginária tem o sufixo j ou J

  $(2+3j)$ $(7j)$ $(5+0j)$

Módulo *math*

Módulo que permite que o programador realize certos cálculos matemáticos.
Para usar uma função que está definida em um módulo, primeiro a função deve importar o módulo usando o comando *import*:

```
1 >>> import math
```

Após ter importado o módulo, a função pode chamar as funções daquele módulo da seguinte forma:

$$NomeDoModulo.NomeDaFuncao(arg_0, \cdots, arg_n)$$

Exemplo

```
1 >>> math.sqrt(81)
2     9.0
```

- **Módulo**: math
- **Função**: sqrt
- **Parâmetro**: 81

Módulo *math*

Módulo que permite que o programador realize certos cálculos matemáticos.
Para usar uma função que está definida em um módulo, primeiro a função deve importar o módulo usando o comando *import*:

```
>>> import math
```

Podemos importar parte dos módulos:

- **from math import *** : importa todos os elementos do módulo *math*
- **from math import** *nome-função* : importa apenas a função nome-função.

Exemplos

```
>>> from math import *

>>> from math import sin
```

Tipos de Dados - Booleano (bool)

- Assume apenas dois valores: verdadeiro (True) ou falso (False)
- É o tipo de dado resultante das operações de comparação.

```
>>> 3>2
    True

>>> 10 <= 5
    False
```

Relações e Expressões Booleanas

- Operadores: $>$, $<$, $==$ (igual), $!=$ (diferente), $>=$, $<=$

ATENÇÃO

- $X == Y$: operador relacional $\Rightarrow$ X É IGUAL A Y
- $X = Y$: operador de atribuição $\Rightarrow$ ATRIBUIR A X O VALOR DE Y

Relações e Expressões Booleanas

- **Relações**: $>$, $<$, $==$ (igual), $!=$ (diferente), $>=$, $<=$
- **Operadores Lógicos**: not (negação), and (e), or (ou)
- **Expressões Booleanas**: Retornam como resultado de sua avaliação os valores verdadeiro (True) ou falso (False)

Exp 1	Exp 2	Exp 1 and Exp 2	Exp 1 or Exp 2	not Exp 1
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

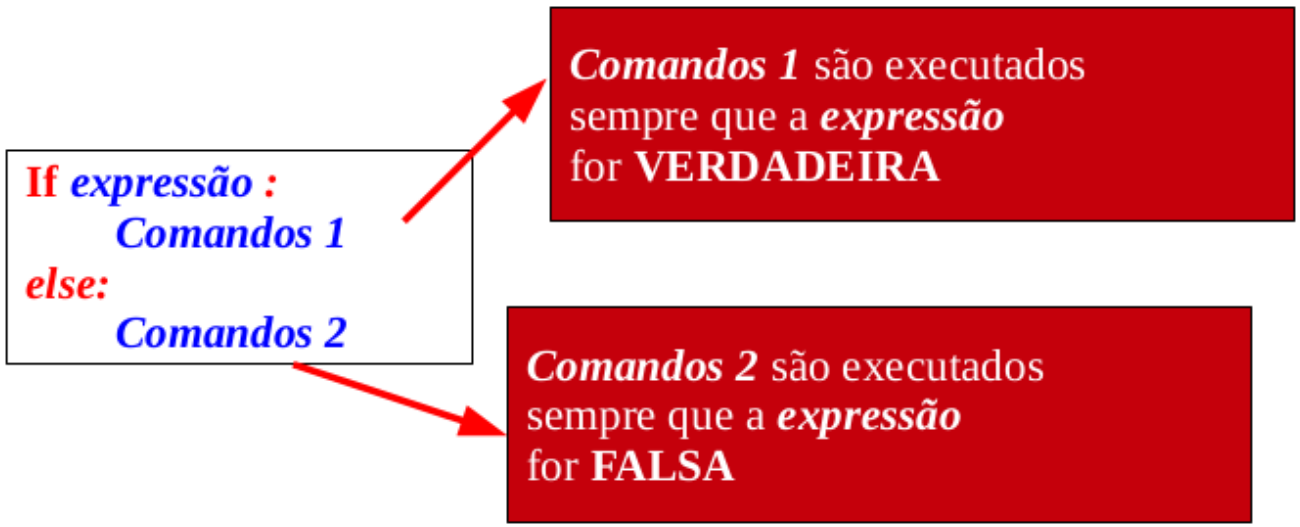
Exp1 e **Exp2** podem ser dados booleanas ou podem ser expressões booleanas compostas de operadores relacionais e operadores lógicos.

Relações e Expressões Booleanas

Ordem de Precedência: do maior para o de menor precedência

1. $**$
2. $*, /, //, \%$
3. $+, -$
4. $<, <=, >, >=, !=, ==$
5. **not**
6. **and**
7. **or**

Estrutura Condicional Composta



Estrutura Condicional

Faça uma função que, dado um número inteiro X passado como parâmetro, retorna a string *"X é positivo"* caso X seja um número positivo, e *"X não é positivo"* caso contrário.

```
1 def positivo(X):
2   """Funcao que recebe um numero inteiro e determina se ele e positivo.
3   Parametro de Entrada: int
4   Valor de Retorno : str"""
5
6   if X > 0 :
7     return str(X) + " e positivo"
8   else:
9     return str(X) + " nao e positivo"
```

Estrutura Condicional

```
if < expressão >:
    < comandos 1 >
else:
    < comandos 2 >
```

A **expressão** na estrutura condicional é
do tipo booleano - verdadeira (True) ou falsa (False).

Estrutura Condicional

else: ... if ⇔ elif ...:

```
1  def PosNegZero(X):
2    """Funcao ..."""
3
4    if X > 0 :
5      return str(X) + " e positivo"
6    else:
7      if X < 0 :
8        return str(X) + " e negativo"
9      else:
10       return str(X) + " e zero"
```

```
1  def PosNegZero(X):
2    """Funcao ..."""
3
4    if X > 0 :
5      return str(X) + " e positivo"
6    elif X < 0 :  # ESTA LINHA MUDOU !
7      return str(X) + " e negativo"
8    else:
9      return str(X) + " e zero"
```

Estrutura de Repetição *while*

```
while <condição>:
    <sequência de comandos>
```

- A <**condição**> é uma expressão ou dado do tipo booleano (True ou False), tal como os testes usados com o comando IF.
- Estrutura também conhecida como **laço de repetição** ou **"loop"**: o bloco de comandos é sequencialmente repetido tantas vezes quanto o teste da condição for verdadeiro.
- Somente quando a condição se torna falsa a próxima instrução após o bloco de comandos associado ao **while** é executada (fim do laço).

Estrutura de Repetição *while*

```
while <condição>:
    <sequência de comandos>
```

- Se a <condição> da estrutura while já for falsa desde o início, o bloco de <sequência de comandos> associado a ela nunca é executado.
- Deve haver algum processo dentro do bloco de <sequência de comandos> que torne a condição falsa e a repetição seja encerrada, ou um erro GRAVE ocorrerá: sua função ficará rodando para sempre!!

Estrutura de Repetição *while*

A função abaixo apresenta algum problema?

```
1 def exemplo3():
2
3 """Parametro de entrada: nao tem
4 Valor de retorno: int"""
5
6   x = 10
7   while x > 8:
8     x = x+ 2
9   return x
```

- Sendo X igual a 10, o teste $X > 8$ é inicialmente verdadeiro.
- Enquanto a condição for verdadeira, apenas o comando $X = X + 2$ será executado. Porém incrementar a variável X não altera a validade da condição $X > 8$.
- Logo, a repetição segue indefinidamente! (Loop infinito)

Estrutura de Repetição *while*

Estrutura que permite a repetição de um conjunto de comandos. Até o momento vimos o *while*:

```
while <condição>:
    <sequência de comandos>
```

- Com *while* podemos implementar qualquer algoritmo que envolva repetição.
- **DICA**: o *while* é mais recomendado quando não se sabe ao certo quantas vez a repetição será feita, pois a condição é um teste booleano qualquer e não necessariamente uma contagem.

Estrutura de Repetição *for*

- A função range(...) pode ter 1, 2 ou 3 argumentos:
  - range(numero): faz com que a variável do **for** assuma valores de 0 a numero-1

    `for x in range(10):` → x recebe 0,1,2,...,9
  - range(inf,sup): faz com que a variável do **for** assuma valores de inf a sup-1

    `for x in range(3,8):` → x recebe 3,4,5,6,7
  - range(inf, sup, inc): faz com que a variável do **for** assuma valores de inf a sup-1 com incremento de inc

    `for x in range(3,8,2):` → x recebe 3,5,7

Estrutura de Repetição

IMPORTANTE: diferença de uso entre *while* e *for*:

- **while**: decisão sobre repetir ou não baseia-se em teste booleano. Risco de loop infinito. :-(
- **for**: Contagem automática do número de repetições.

Strings

- **Caracteres** são símbolos. Podem ser letras, números, caracteres especiais, e até o espaço em branco é um caractere.
  Exemplo: 'a', '9', '#', ' '.

- Uma *string* é uma sequência de caracteres.

```
>>> a = 'abcd'
>>> b = "1234"
>>> c = "#$5a"
>>> d = ''
>>> e = '  '
```

- Comprimento de uma string: número de caracteres que ela contém.

```
>>> s = '123456'
>>> len(s)
    6
```

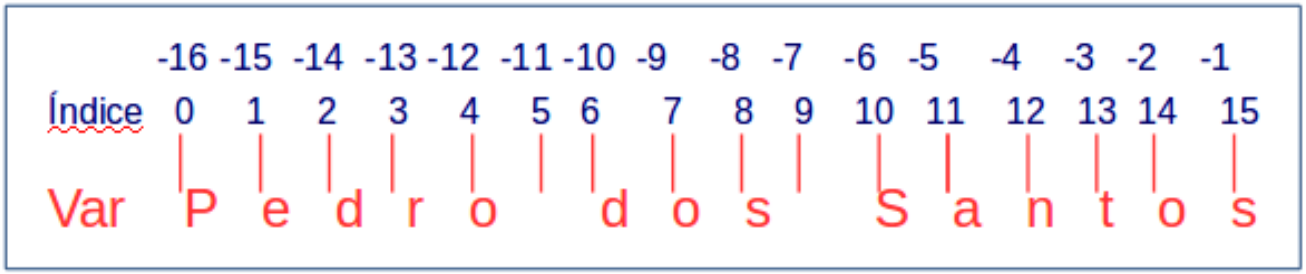
Strings - Recapitulando

- **Representação**: s = “12346” ou s = ‘123456’
- **len(s)** : retorna o tamanho de uma string.
- **Operador +**: concatena strings. Ex: ‘ab’ + ‘cd’ = ‘abcd’
- **Operador ***: repete strings. Ex: ‘a’*5 = ‘aaaaa’
- **Fatias (Slices)**: [start:end:step]

Strings - Índices

- Todo caractere de uma string é **indexado**, começando do primeiro caractere (**índice 0**) à esquerda.
- **Notação**: string[indice]

Exemplo: var = "Pedro dos Santos"

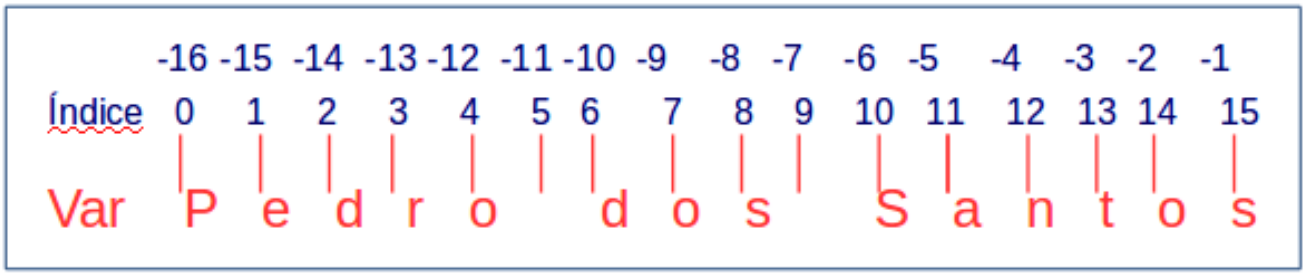



```
1 >>> var[2]
2     'd'
3 >>> var[9]
4     ' '
5 >>> var[15]
6     's'
```

Strings - Índices

- A string também pode ser indexada da direita para a esquerda, começando no **índice -1**.
- **Notação**: string[indice]

Exemplo: var = "Pedro dos Santos"




```
1 >>> var[-14]
2     'd'
3 >>> var[-7]
4     ' '
5 >>> var[-1]
6     's'
```

Strings - Fatiamento

Incremento: podemos usar incremento / decremento para selecionar os elementos de uma string.

[start:end:step]: vai do índice *start* até *end* (sem ultrapassá-lo, com passo *step*)

Exemplo

```
>>> x= "abcde"
>>> x[0:-1:2]
    'ac'
>>> x[3:0:-1]
    'dcb'
```

Manipulação de Strings

- **lower()**: retorna a string com todos os caracteres maiúsculos convertidos para minúsculos.
- **upper()**: retorna a string com todos os caracteres minúsculos convertidos para maiúsculos.
- **Exemplo**

```
>>> str.upper("Esperanca")
    ESPERANCA

>>> str.lower("Pe de Laranja Lima")
    pe de laranja lima
```

Manipulação de Strings

- **str.count(umaString, elemento, inicio, fim)**: retorna quantas vezes o elemento aparece na string, procurando-se a partir da posição **inicio** e indo até a posição **fim**.
- **inicio** e **fim** são opcionais.
- **Exemplo**

```
>>> frase="macaco come banana"
>>> str.count(frase,"a", 2, 10)
>>> 1
```

Manipulação de Strings

- **str.index(umaString,elemento, inicio, fim)**: retorna o índice da primeira ocorrência de elemento na string, a partir da posição **inicio**, até a posição **fim**.
- **inicio** e **fim** são opcionais.
- **Exemplo**

```
>>> str.index("mariana", "a")
    1
>>> str.index("mariana", "a", 2)
    4
>>> str.index("mariana", "a",5, 7)
    6
>>> str.index('Mariana', 'ana')
    4
>>> str.index('Mariana', 'x')
    Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module>
    str.index('Mariana', 'x')
    ValueError: substring not found
```

Strings

- Elementos de uma string não aceitam o operador de atribuição.

```
>>> s = '123456'
>>> s[0] = '0'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    s[0]='0'
TypeError: 'str' object does not support item assignment
```

- Strings são, portanto, imutáveis. Ou seja, os dados contidos em uma string não podem ser alterados.

Tuplas

- Uma **tupla** é uma sequência heterogênea (permite que seus elementos sejam de tipos diferentes):

```
>>> a = (1,2,3,4)
>>> b = (1.0, 2, '3', 4+0j)
>>> c = 1,2,3,4
>>> d = (1,)
```

- Valores em uma tupla podem ser distribuídos em variáveis como uma atribuição múltipla:

```
>>> x = 1, 2, 3
>>> x
    (1, 2, 3)
>>> a, b, c = x
>>> a
    1
>>> b
    2
>>> c
    3
```

Tuplas

- **Tupla Vazia**: tupla sem elementos.
- **Tupla unitária**: contém um único elemento, que deve ser sucedido por uma vírgula.
- Os parênteses são opcionais se não provocarem ambiguidade.
- Um valor entre parênteses sem vírgula no final é meramente uma expressão.

Qual o tipo de dado da variável A em cada um dos casos abaixo:

```
>>> A = ()
    () # tupla vazia
>>> A = (10)
    10 # inteiro
>>> A = 10,
    (10,) # tupla unitaria
>>> A = (10,)
    (10,) # tupla unitaria
>>> A = 3*(10+3)
    39 # inteiro
>>> A = 3*(10+3,)
    (13, 13, 13) # tupla
```

Tuplas

- Tuplas são muito similares às strings em relação às operações.
- O tamanho de uma tupla é dado pela função len.

```
>>> x = (1,2,3)
>>> len(x)
    3
```

- **Indexação**: começando do 0 à esquerda, ou de -1 à direita.

```
>>> x[0]
    1
```

- **Fatiamento**: idêntico às strings.

```
>>> x[0:2]
    (1,2)   # NOVA TUPLA
```

Tuplas

- **Concatenação e Replicação**

```
>>> x*2
    (1,2,3,1,2,3)
>>> x + (5,4)
    (1,2,3,5,4)
```

- **Imutabilidade** : uma vez criada, uma tupla não pode ser alterada !

```
>>> x[0] = 9
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x[0]=9
TypeError: 'tuple' object does not support item assignment
```

Listas

- Tipo de dados mais versátil do Python.
- Uma lista é representada como uma sequência de valores entre colchetes e separados por vírgula.
- Os elementos de uma lista podem ser de tipos de dados diferentes.
- Listas são mutáveis !!!

Exemplo

```
>>> lista1 = ['calculo', 'fisica', 'computacao']
>>> lista2 = ['notas', 5.4, 'aprovado']
>>> lista2[1] = 6
>>> lista2
    ['notas', 6, 'aprovado']
```

Manipulação de Listas

ATENÇÃO

Algumas funções que manipulam listas não possuem valor de retorno:

- list.append
- list.extend
- list.insert
- list.remove
- list.reverse
- list.sort

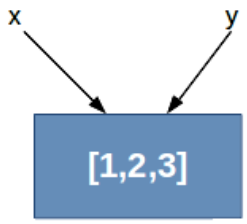
Enquanto outras possuem:

- list.pop
- list.count
- list.index

Alias e Tipos Mutáveis

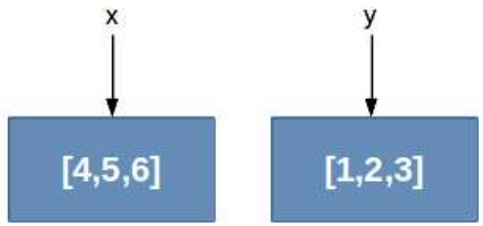
Relembrando - um *alias* acontece quando duas variáves se referem ao mesmo dado:

```
>>> y = [1,2,3]
>>> x = y
>>> x
[1,2,3]
>>> y
[1,2,3]
```



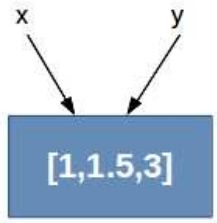
Qualquer nova atribuição feita a uma das variáveis quebrará o *alias* entre elas.

```
>>> x = [4,5,6]
>>> x
[4,5,6]
>>> y
[1, 2, 3]
```



Porém quando o dado é mutável (lista!!), o *alias* não é quebrado caso uma atribuição modifique uma parte do dado:

```
>>> y = [1,2,3]
>>> x = y
>>> x[1]=1.5
>>> x
[1, 1.5, 3]
>>> y
[1, 1.5, 3]
```



Listas - Cópias

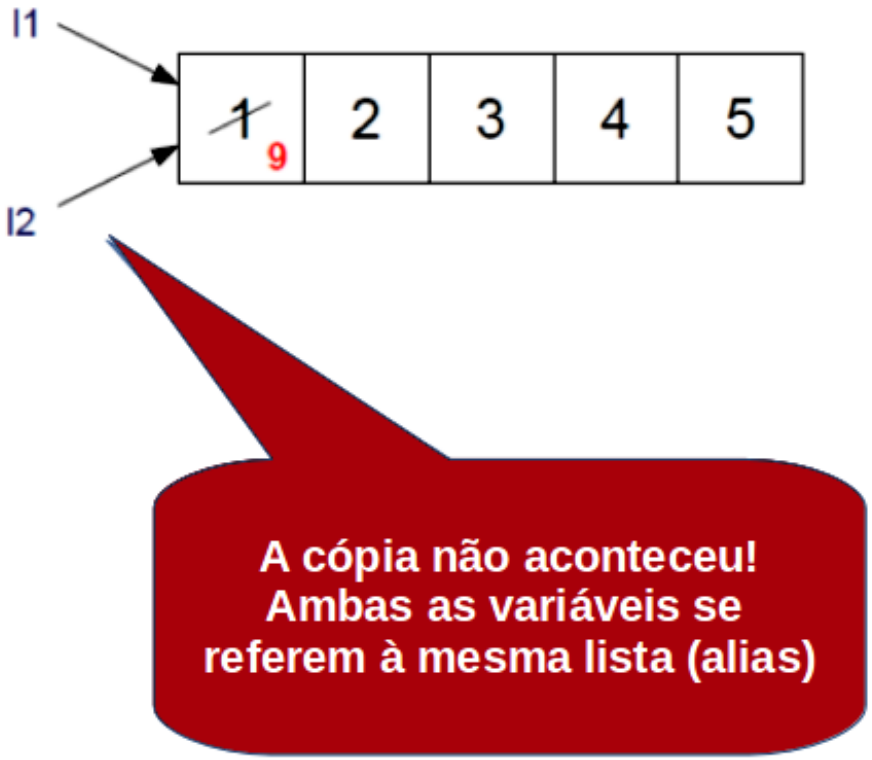
Cuidado quando fizer cópia de listas!

```
>>> l1 = [1,2,3,4,5]
>>> l2 = l1
>>> l1
[1,2,3,4,5]

>>> l2
[1,2,3,4,5]

>>> l2[0]=9
>>> l2
[9,2,3,4,5]

>>> l1
[9,2,3,4,5]
```



Listas - Cópias

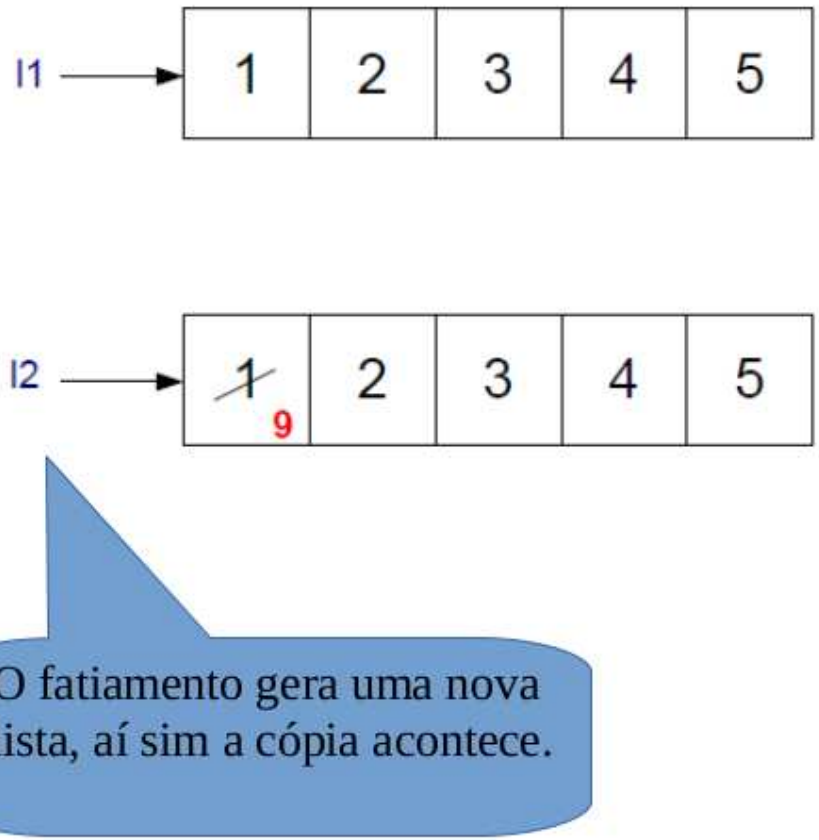
Cuidado quando fizer cópia de listas!

```
>>> l1 = [1,2,3,4,5]
>>> l2 = l1[:]
>>> l1
[1,2,3,4,5]

>>> l2
[1,2,3,4,5]

>>> l2[0]=9
>>> l2
[9,2,3,4,5]

>>> l1
[1,2,3,4,5]
```



Dicionário

Dicionários são estruturas para armazenar dados. Não são sequências como strings, listas e tuplas.

São **mapeamentos** formados por pares de *chave* – *valor*.

$$\textit{Chave}1 \rightarrow \text{Conteúdo1}$$
$$\textit{Chave}2 \rightarrow \text{Conteúdo2}$$
$$\textit{Chave}3 \rightarrow \text{Conteúdo3}$$
$$\ldots$$

Representam uma coleção não ordenada de **valores** onde cada valor é referenciado através de sua **chave**.

Notação: { chave1: conteúdo1, chave2: conteúdo2, ..., chaveN: conteúdoN }

Dicionário

```
>>> Caderno = {"Carlos":"2222-2223","Andre":"2121-9092","Jose":"9999-9291"}

>>> "Jorge" in Caderno
False

>>>Caderno["Jorge"] = "8586-9091"

>>>Caderno["Jorge"]
'8586-9091'

>>>Caderno
{'Andre':'2121-9092','Jorge':'8586-9091','Jose':'8799-0405',
'Carlos':'2222-2223'}
```

- **Para adicionar um novo par chave:valor**
  Perceba que, diferentemente de listas, atribuir a um elemento de um dicionário não requer que uma posição exista previamente.
- O dicionário não fornece garantia de que as chaves estarão ordenadas, mas a ordem em que os elementos aparecem não é importante, pois os valores são acessados somente através de suas respectivas chaves, e não de suas posições.

Manipulação de Dicionário

Como saber quais são as chaves e os valores de um dicionário?

- **dict.keys(<dicionário>)**: retorna a lista com todas as chaves do dicionário passado como parâmetro.
- **dict.values(<dicionário>)**: retorna a lista com todos os valores do dicionário passado como parâmetro.

```
>>> meses = {"janeiro":31,"fevereiro":28,"marco":31,"abril":30,"maio":31,
"junho":30,"julho":31,"agosto":31,"setembro":30,"outubro":31,"novembro":31,
"dezembro":31}

>>> list(dict.keys(meses))
['novembro','marco','julho','agosto','fevereiro','junho','dezembro','janeiro',
'abril','maio','outubro', 'setembro']

>>> list(dict.values(meses))
[31, 31, 31, 31, 28, 30, 31, 31, 30, 31, 31, 30]
```

Manipulação de Dicionário

- **dict.items(<dicionário>)**: retorna uma lista com todos os pares (chave, conteúdo) do dicionário passado como parâmetro.

```
>>> meses = {"janeiro":31, "fevereiro":28, "marco":31, "abril":30, "maio":31,
"junho":30, "julho":31, "agosto":31, "setembro":30,"outubro":31,
"novembro":31,"dezembro":31}

>>> list(dict.items(meses))
[('novembro',31),('marco',31),('julho',31),('agosto',31),('fevereiro',28),
('junho',30),('dezembro',31),('janeiro',31),('abril',30),('maio',31),
('outubro',31),('setembro',30)]
```

Manipulação de Dicionário

- **dict.get(<dicionário>,chave,[valor de retorno])**: Retorna o valor associado com a chave. Se *chave* não está no dicionário e se o *valor de retorno* é especificado, retorna o valor especificado.
  Se o *valor de retorno* não é especificado, retorna **None**.

```
>>> meses = {"janeiro":31, "fevereiro":28, "marco":31, "abril":30,"maio":31,
"junho":30, "julho":31, "agosto":31, "setembro":30, "outubro":31, "novembro":31,
"dezembro":31}

>>> dict.get(meses,"marco")
31

>>> dict.get(meses,"marco","Valor nao encontrado")
31

>>> dict.get(meses,"Marco")
None

>>> dict.get(meses,"Marco","Valor nao encontrado")
"Valor nao encontrado"
```

Manipulação de Dicionário

- **dict.clear(<dicionário>)**: apaga todos os itens do dicionário.
- **dict.copy(<dicionário>)**: cria e retorna uma cópia do dicionário.

```
>>> meses = {"janeiro":31, "fevereiro":28}

>>> novo = dict.copy(meses)

>>> novo
{'fevereiro': 28, 'janeiro': 31}

>>> novo["maio"] = 31

>>> novo
{'maio': 31, 'fevereiro': 28, 'janeiro': 31}

>>> meses
{'fevereiro': 28, 'janeiro': 31}

>>> dict.clear(meses)

>>> meses
{}
```

Manipulação de Dicionário

- **dict.copy(<dicionário>)**: cria e retorna uma cópia do dicionário. Os elementos mutáveis (listas ou dicionários) do novo dicionário são apenas referências aos elementos do dicionário original.

```
>>> meses = {"janeiro":30, "fevereiro":[28,29]}
>>> novo = dict.copy(meses)
>>> novo
    {'fevereiro': [28,29], 'janeiro': 30}
>>> novo["maio"]=31
>>> novo
    {'maio': 31, 'fevereiro': [28,29], 'janeiro': 30}
>>> meses
    {'fevereiro': [28,29], 'janeiro': 30}
>>> novo["janeiro"]=31
>>> novo
    {'maio': 31, 'fevereiro': [28,29], 'janeiro': 31}
>>> meses
    {'fevereiro': [28,29], 'janeiro': 30}
>>> list.pop(novo["fevereiro"])
    29
>>> novo
    {'maio': 31, 'fevereiro': [28], 'janeiro': 31}
>>> meses
    {'fevereiro': [28], 'janeiro': 30}
```