

Python I - Fundamentos do Python

Luiza
<CODE>

Ementa:

- Tipos básicos
 - int
 - float
 - bool
 - str
- Variáveis
- Comentários
- Operadores
 - aritméticos
 - comparativos
 - de atribuição
 - lógicos

Tipos Básicos

int

O tipo inteiro é um tipo composto por caracteres numéricos (algarismos) inteiros. É um tipo usado para um número que pode ser escrito sem um componente decimal, podendo ter ou não sinal, ou seja: ser positivo ou negativo. Por exemplo, 20, 7, 0, -3 e -500 são números inteiros.

Exemplos:

```
tipos básicos > type_int.py > ...  
1  idade = 27  
2  ano = 1995  
3  
4  print(type(idade))  
5  print(type(ano))  
6
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
<class 'int'>  
<class 'int'>
```

float

É um tipo composto por caracteres numéricos (algarismo) decimais.

O famoso ponto flutuante é um tipo usado para números racionais (números que podem ser representados por uma fração) informalmente conhecido como “número quebrado”.

Exemplos:

```
tipos básicos > type_float.py > ...  
1  altura = 1.75  
2  peso = 60.0  
3  
4  print(type(altura))  
5  print(type(peso))  
6
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
<class 'float'>  
<class 'float'>
```

bool

Tipo de dado lógico que pode assumir apenas dois valores: falso ou verdadeiro (**False** ou **True** em Python).

Na lógica computacional, podem ser considerados como 0 ou 1.

Exemplos:

```
tipos básicos > type.bool.py > ...
1  semana = False
2  feriado = True
3
4  print(type(semana))
5  print(type(feriado))
6
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
<class 'bool'>
<class 'bool'>
```


str

É um conjunto de caracteres dispostos numa determinada ordem, geralmente utilizada para representar palavras, frases ou textos.

Exemplos:

```
tipos básicos > type_str.py > ...
1  nome = 'Marcia'
2  profissao = 'Desenvolvedora de Software'
3
4  print(type(nome))
5  print(type(profissao))
6
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
<class 'str'>
<class 'str'>
```

tipos básicos >  all_types.py > ...

```
1 nome = 'Alyce'
2 idade = 13
3 altura = 1.75
4 feriado = True
5
6 print(type(nome))
7 print(type(idade))
8 print(type(altura))
9 print(type(feriado))
10
```

return

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
```


Variáveis

variáveis

Uma **variável** é um nome que se refere a um valor. Um comando de atribuição cria uma nova variável e lhe dá um valor.

Variáveis são usadas para guardarmos valores que serão reutilizados mais tarde no programa.


Exemplos:

```
variaveis > 📄 velocidade_int.py > ...  
1  velocidade_do_carro = 200  
2  print(velocidade_do_carro)  
3
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  
  
200
```

variáveis

```
variaveis >  altura_float.py > ...  
1 sua_altura = 1.62  
2 print(sua_altura)  
3
```

return

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

1.62

variáveis

```
variaveis > batata_bool.py > ...  
1  gosta_de_batata = True  
2  print(gosta_de_batata)  
3
```

return

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

True

variáveis

```
variaveis > 📄 frase_str.py > ...  
1 frase_boas_vindas = 'Sejam bem vindas ao LuizaCode'  
2 print(frase_boas_vindas)  
3
```

return

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Sejam bem vindas ao LuizaCode

Comentários

comentários

É uma linha que você escreve no código fonte, porém esse comentário não é interpretado na linguagem de programação. Ele serve para que você possa adicionar alguma informação a mais no código.

* Não é necessário escrever para tudo, apenas naquilo que realmente é importante. Em um código auto explicativo, por exemplo, não há necessidade.

Os comentários começam com o caractere cerquilha '#' e estende até o final da linha

Exemplos:

```
# Sejam bem vindas ao LuizaCode  
# Essa é uma mensagem de boas vindas da 5 edição  
# Aula inicial de Python com a Professora Milene
```

comentários

Exemplo de um print sem o comentário:

```
comentarios > 🐘 ola.py > ...  
1  ola = 'Olá, futuras Devas maravilhosas'  
2  print(ola)  
3
```

return

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Olá, futuras Devas maravilhosas

comentários

Exemplo de um print com o comentário:

```
comentarios > 🐍 ola.py > ...  
1  # mensagem de olá para as meninas do luizacode  
2  ola = 'Olá, futuras Devas maravilhosas'  
3  print(ola)
```

return

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Olá, futuras Devas maravilhosas

Operadores

operadores aritméticos

Esses operadores são utilizados para criar expressões matemáticas comuns, como soma, subtração, multiplicação e divisão.

Veja quais estão disponíveis no Python:

Operador	Nome	Função
+	Adição	Realiza a soma de ambos operandos
-	Subtração	Realiza a subtração de ambos operandos
*	Multiplicação	Realiza a multiplicação de ambos operandos
/	Divisão	Realiza a divisão de ambos operandos
//	Divisão inteira	Realiza a divisão entre operandos e a parte decimal de ambos operandos
%	Módulo	Retorna o resto da divisão de ambos operandos
**	Exponenciação	Retorna o resultado da elevação da potência pelo outro

operadores aritméticos

Vejamos agora a utilização de cada operador aritmético mencionado no slide anterior:

```
operadores > aritmeticos.py > ...
1  quatro = 4
2  dois = 2
3
4  soma = quatro + dois
5  print(soma) # resultado: 6
6
7  subtração = quatro - dois
8  print(subtração) # resultado: 2
9
10 multiplicacao = quatro * dois
11 print(multiplicacao) # resultado: 8
12
13 divisao = quatro / dois
14 print(divisao) # resultado: 2
15
16 divisão_interna = quatro // dois
17 print(divisão_interna) # resultado: 2
18
19 modulo = quatro % dois
20 print(modulo) # resultado 0
21
22 exponenciacao = quatro ** dois
23 print(exponenciacao) # resultado: 16
24
```


operadores comparativos

Como o nome já diz, esses Operadores são usados para **comparar** dois valores:

Operador	Nome	Função
==	Igual a	Verifica se um valor é igual ao outro
!=	Diferente de	Verifica se um valor é diferente ao outro
>	Maior que	Verifica se um valor é maior que o outro
>=	Maior ou igual	Verifica se um valor é maior ou igual ao outro
<	Menor que	Verifica se um valor é menor que o outro
<=	Menor ou igual	Verifica se um valor é menor ou igual ao outro

operadores comparativos

Vamos ver exemplos da utilização de cada operador de comparação mencionado no slide anterior.

Para facilitar o entendimento, todas as operações estão retornando um valor igual a **True**, para que vocês entendam como cada condição é aceita:

```
operadores > comparativos.py > ...
1  variavel = 5
2
3  if variavel == 5:
4      print('Os valores são iguais')
5
6
7  if variavel != 7:
8      print('O valor da variável não é igual a 7')
9
10
11 if variavel > 2:
12     print('O valor da variável é maior que 2')
13
14
15 if variavel >= 5:
16     print('O valor da variável é maior ou igual a 5')
17
18
19 if variavel < 7:
20     print('O valor da variável é menor que 7')
21
22
23 if variavel <= 5:
24     print('O valor da variável é menor ou igual a 5')
25
```

return

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

Os valores são iguais
O valor da variável não é igual a 7
O valor da variável é maior que 2
O valor da variável é maior ou igual a 5
O valor da variável é menor que 7
O valor da variável é menor ou igual a 5
```

operadores de atribuição

Esses Operadores são utilizados no momento da **atribuição** de valores à variáveis e controlam como a atribuição será realizada.

Veja quais Operadores de Atribuição estão disponíveis em Python:

Operador	Equivalente a
=	$x = 1$
+=	$x = x + 1$
-=	$x = x - 1$
*=	$x = x * 1$
/=	$x = x / 1$
%=	$x = x \% 1$

operadores de atribuição

operadores > atribuiçao.py > ...

```
1  # operador +=
2  numero = 5
3
4  numero += 7
5  print(numero) # resultado: 12
6
7
8  # operador -=
9  numero = 5
10
11 numero -= 3
12 print(numero) # resultado: 2
13
14
15 # operador *=
16 numero = 5
17
18 numero *= 2
19 print(numero) # resultado: 10
20
21
22 # operador /=
23 numero = 5
24
25 numero /= 4
26 print(numero) # resultado: 1.25
27
28
29 # operador %=
30 numero = 5
31
32 numero %= 2
33 print(numero) # resultado: 1
34
```

* O operador **%** é chamado módulo pois nada mais é que o resto da divisão.
No exemplo ao lado: 5 dividido por 2 dá 2 de resultado e sobra 1. Por isso **numero %= 2** será **1!**

operadores lógicos

Python nos disponibiliza três tipos de Operadores Lógicos: o **and**, o **or** e o **not**.

Operador	Definição
and	Retorna True se ambas afirmações forem verdadeiras
or	Retorna True se uma das afirmações for verdadeira
not	Retorna falso se o resultado for verdadeiro

operadores lógicos

Exemplo da utilização de cada um:

```
operadores > logicos.py > ...
```

```
1  num1 = 7
2  num2 = 4
3
4  # exemplo operador and
5  if num1 > 3 and num2 < 8:
6      print('As duas condições são verdadeiras')
7
8
9  # exemplo operador or
10 if num1 > 4 or num2 <= 8:
11     print('Uma ou duas das condições são verdadeiras')
12
13 # exemplo operador not
14 if not (num1 < 30 and num2 < 8):
15     print('Inverte o resultado da condição entre os parâmetros')
16
```


Parte 2



Thais Ribeiro

AI Engineer

thais.ribeiro@luizalabs.com

www.thaisplicando.com

@thaisplicandoo

Thais
Qlicando

CRIAÇÃO DE
PIXEL ART

START



Lu do Magalu ✓



HOJE

🔒 As mensagens são protegidas com a criptografia de ponta a ponta e ficam somente entre você e os participantes desta conversa. Nem mesmo o WhatsApp pode ler ou ouvi-las. Clique para saber mais.

Oi 00:32 ✓✓

👋 Oi, como posso te ajudar?

Digita aqui pra mim o que vc precisa? Ou então, é só **tocar no menu** e escolher um assunto 🤖

00:32

☰ Escolher assunto

Olá pessoas incríveis da internet, tudo bem com vocês? Trago hoje mais um artigo explorando o mundo de inteligência artificial, onde vamos aprender a sumarizar textos com a ajuda de NLP (Natural Language Processing). Para começarmos, precisamos entender que sumarização é basicamente a construção de um resumo, onde dado um texto, se produz outro texto que condense o sentido do texto original, focando apenas nos pontos principais.

tualmente, usamos textos sumarizados para facilitar a análise dos dados que recebemos, por exemplo, uma grande organização usa desse processo para trazer partes importantes de feedbacks para gerar insights, outro exemplo é quando precisamos resumir partes importantes de um determinado texto para apresentar em links de matérias, blogs.

pesar de parecer trivial, é uma atividade bastante comum que geralmente é feita de forma manual, isso torna o processo mais tedioso e lento, então como podemos resolver isso? Com os avanços tecnológicos, hoje conseguimos fazer

Veja o conteúdo completo em:

luizalabs.com

luizalabs
magalu

Ementa

1. Conversão de tipos
2. Listas
3. Tuplas
4. Funções
 - Map
 - Filter
 - Reduce
5. Tipos de parâmetros (args, kwargs)
6. Decorator
7. Lambda



Recados Importantes

Ficou com Dúvidas?

- Fique à vontade para perguntar a qualquer momento nos nossos canais do slack e/ou buscar ajuda com os mentores/professores.



Conversão de Tipos

Conversão de tipos

A **conversão** de dados em **Python**, são o uso de **funções definidas** que **convertem** diretamente um **tipo de dado** em outro, o que é útil na programação diária e competitiva.

Existem dois tipos de conversão aqui:

- **Implícita**
- **Explícita**



Conversão Implícita

Na **conversão implícita**, acontece do **interpretador converter automaticamente** um tipo de dado em outro sem qualquer envolvimento do usuário.

Vamos ver isso acontecendo com o exemplo abaixo:

```
1 a = 15
2 print(f'A variável a é do tipo: {type(a)}')
3 # Output: A variável a é do tipo: <class 'int'>
4
5 b = 10.6
6 a = a + b
7 print(f'Valor da soma: {a}')
8 # Output: Valor da soma: 25.6
9
10 print(f'A variável a é do tipo: {type(a)}')
11 # Output: A variável a é do tipo: <class 'float'>
```

Conversão Explícita

Diferente da **conversão Implícita**, esse tipo de conversão é feita **manualmente** pelo usuário de acordo com seus requisitos, existem várias formas de conversão que se pode fazer nesse tipo, usando as funções do Python:

- **int(a, base):** essa função converte qualquer tipo de dado em inteiro. 'Base' especifica a base em que a string está se o tipo de dados for uma string
- **float():** função usada para converter qualquer tipo de dados em um float, literalmente.
- **str():** usada para converter números inteiros em strings
- **dict():** usada para converter tuplas em dicionário
- **ord():** função usada para converter qualquer caractere em um inteiro
- **hex():** função que converte número inteiro em uma string hexadecimal*
- **oct():** converte um número inteiro em octal
- **tupla:** converte para uma tupla

...

Conversão Explícita

Aqui temos alguns exemplos de conversão sendo feitas no código, contudo vocês conseguem ver uma lista de tipos de conversão na documentação oficial:

<https://docs.python.org/3/library/functions.html>

```
1  a = 1
2  print(type(a))
3  # output: <class 'int'>
4
5  a = str(a)
6  print(type(a))
7  # output: <class 'str'>
8
9  b = ['a', 'b', 'c', 'd']
10 print(type(b))
11 # output: <class 'list'>
12
13 b = tuple(b)
14 print(type(b))
15 # output: <class 'tuple'>
16 print(b)
17 # output: ('a', 'b', 'c', 'd')
```

Listas

Listas

Uma lista, nada mais é que uma **coleção ordenada de valores**, onde cada valor é identificado por **índices**. Em Python representamos uma lista separando os dados por vírgula, dentro de colchetes.

Elas são utilizadas para armazenar diversos itens em uma única variável e existem várias maneiras de serem criadas.

```
letters = ["a", "b", "c"]  
           0   1   2
```

Listas

Podemos criar uma lista de uma maneira bem simples, apenas envolvendo os elementos por colchetes:

```
1 lista = ['luizacode']  
2 print(type(lista))  
3 # output: <class 'list'>
```

Podemos criar uma lista vazia:

```
1 lista = []
```

Listas

Também é possível criar uma lista com vários itens e de diferentes tipos:



```
1 lista = ['luizacode', 5, 'python']
```

Podemos obter uma lista através de **compreensão de listas (List Comprehensions)***



```
1 [item for item in iteravel]  
2
```


Listas

Como acessar os dados de uma lista?

Todos os **itens** de uma lista são **indexados**, ou seja, para cada item da lista um índice é atribuído ao mesmo, para **acessar** esses itens é necessário que **especifiquemos** o **índice**, lembrando, o índice começa no 0.

Índice	0	1	2	3	4
Valores	Maçã	Banana	Jaca	Melão	Abacaxi

No código ficaria assim:

```
1 lista = ['Maçã', 'Banana', 'Jaca', 'Melão', 'Abacaxi']
2 print(f'Estou buscando a Fruta: {lista[2]}')
3 # output: Estou buscando a Fruta: Jaca
```

Listas

E para pegar o último item da lista?

Uma forma de buscar o último item, ou itens mais próximos ao fim, é usando indexação negativa.

A indexação negativa, vai buscar os itens do fim, logo, se eu quiser buscar o Abacaxi de uma forma mais simples, basta eu fazer assim:

```
1 lista = ['Maçã', 'Banana', 'Jaca', 'Melão', 'Abacaxi']  
2 print(f'Estou buscando a Fruta: {lista[-1]}')  
3 # output: Estou buscando a Fruta: Abacaxi
```

Listas

Lista dentro de lista?

Sim, é possível! Assim como no exemplo, para acessarmos a sub-lista é necessário que busquemos onde o índice da mesma está.



```
1 lista = ['Maçã', ['Banana', 'Jaca'], 'Melão', 'Abacaxi']
2
3 sublista = lista[1]
4 print(f'Acessando a sublista: {sublista}')
5 # output: Acessando a sublista: ['Banana', 'Jaca']
6
7 """
8     Acessando um item da sublista
9 """
10 print(f'Acessando um item da sublista: {sublista[0]}')
11 # output: Acessando um item da sublista: Banana
12
```

Listas

Percorrendo uma lista

Se precisarmos percorrer uma lista, a melhor forma é usarmos loop.

Por exemplo, dada uma lista grande de elementos, eu quero buscar somente aqueles que forem maior que 10:

```
1 lista = [0, 5, 8, 10, 35, 15, 7, 4, 12, 22, 3, 2, 9, 1]
2
3 lista_numeros_maior_10 = []
4
5 for i in lista:
6     if i > 10:
7         lista_numeros_maior_10.append(i)
8
9 print(f'Resultado da lista: {lista_numeros_maior_10}')
10 # output: Resultado da lista: [35, 15, 12, 22]
11 """
12     Você também pode fazer assim
13 """
14 lista_numeros_maior_10 = [i for i in lista if i > 10]
15 print(f'Resultado da lista: {lista_numeros_maior_10}')
16 # output: Resultado da lista: [35, 15, 12, 22]
```

Listas

É importante saber como manusear uma lista, em python temos vários métodos que estão disponíveis para nos ajudar nisso, confira comigo alguns deles:

- **list.append(x):** adiciona um item ao fim da lista
- **list.extend(iterable):** adiciona todos os itens do iterável ao fim da lista
- **list.insert(i, x):** insere um item em uma dada posição (i) dada pelo index
- **list.remove(x):** remove o primeiro elemento cujo o valor for “x”
- **list.pop(i):** remove o item da posição i da lista e retorna, caso o index não seja especificado, retorna o último elemento.
- **list.clear():** remove tudo da lista
- **list.index(x, start[, end]):** retorna o índice do primeiro elemento cujo valor seja x.
- **list.count(x):** retorna o número de vezes que x aparece na lista
- **list.sort(key=nome, reverse=False):** ordena os itens da lista
- **list.reverse():** reverte os elementos da lista
- **list.copy():** retorna uma lista com a cópia dos elementos da lista de origem.

...

Listas

Vamos ver funcionando na prática:

```
1 lista = [0, 5, 8, 10, 35, 15, 7, 4, 12, 22, 3, 2, 9, 1]
2 list_append = []
3
4 for i in lista:
5     # pegamos o elemento da lista e adicionamos mais 1
6     list_append.append(i+1)
7
8 print(f'Append: {list_append}')
9 # Append: [1, 6, 9, 11, 36, 16, 8, 5, 13, 23, 4, 3, 10, 2]
10
11 list_append.extend([0,0,0, 0])
12 print(f'Extend: {list_append}')
13 # Extend: [1, 6, 9, 11, 36, 16, 8, 5, 13, 23, 4, 3, 10, 2, 0, 0, 0, 0]
14
15 list_append.insert(8, 'meio')
16 print(f'Insert: {list_append}')
17 # Insert: [1, 6, 9, 11, 36, 16, 8, 5, 'meio', 13, 23, 4, 3, 10, 2, 0, 0, 0, 0]
```

Listas

```
1 list_append.remove('meio')
2 print(f'Remove: {list_append}')
3 # Remove: [1, 6, 9, 11, 36, 16, 8, 5, 13, 23, 4, 3, 10, 2, 0, 0, 0, 0]
4
5 list_append.pop(4)
6 print(f'Pop: {list_append}')
7 # Pop: [1, 6, 9, 11, 16, 8, 5, 13, 23, 4, 3, 10, 2, 0, 0, 0, 0]
8 # removeu o 36
9
10 print(f'Index: {list_append.index(11)}')
11 # Index: 3
12
13 list_append.sort()
14 print(f'Sort: {list_append}')
15 # Sort: [0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 13, 16, 23]
16
17 list_append.reverse()
18 print(f'Reverse: {list_append}')
19 # Reverse: [23, 16, 13, 11, 10, 9, 8, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0]
20
21 list_new = list_append.copy()
22 print(f'Copy: {list_new}')
23 # Copy: [23, 16, 13, 11, 10, 9, 8, 6, 5, 4, 3, 2, 1, 0, 0, 0, 0]
24
```

Listas

Entendo o que é List Comprehensions e porquê usá-las.

Compreensão de listas foi concebida na **PEP 202** e é uma forma **concisa** de **criar e manipular listas**. Basicamente é a forma mais rápida de **criar uma lista através de outra**. Python oferece diferentes formas de iteração, mas **list comprehensions** é mais **otimizada** para o interpretador python detectar um padrão previsível durante o loop, e como bônus ela é mais legível e economiza variáveis na contagem.

Veja um exemplo:

```
1 # Dada uma quantidade de n números, listar somente os números pares.
2 # O BOM
3 lista = [i for i in range(10) if i%2 == 0]
4 print(lista)
5 # output: [0,2,4,6,8]
6
7 # O NÃO TÃO BOM ASSIM
8 i = 0
9 lista = []
10 while i < 10:
11     if i%2 == 0:
12         lista.append(i)
13 print(lista)
14 # output: [0,2,4,6,8]
```


Tuplas

Tuplas

Python oferece diferentes **tipos de estruturas de dados**, entre elas as **listas** que acabamos de ver e as **tuplas**, que podem causar algumas dúvidas devido a **semelhança** entre elas.

Contudo, é necessário entender o conceito que envolve cada uma para utilizá-las da maneira correta.

Tupla, é uma estrutura parecida com listas, mas com a característica de ser **imutável**. Isso significa que quando uma tupla é criada, não há a possibilidade (entre aspas)* de adicionar, remover, ou alterar seus elementos. Geralmente são utilizadas para **adicionar tipos diferentes** de informações, podemos utilizar uma tupla para adicionar, por exemplo, a sigla do estado em uma posição e o nome em outra, tornando-a boa para ser usada quando queremos trabalhar com **informações diferentes** em uma **mesma variável**.

```
1 tupla = (('MG', 'Minas Gerais'), ('SP', 'São Paulo'))
```

Tuplas

Sua característica de **imutabilidade** oferece **segurança nas informações** armazenadas, sendo assim, uma tupla vai ser usada para armazenar uma sequência de dados que não serão modificados no decorrer do código.

Entretanto, ***ela não é totalmente imutável**, pois quando armazena outro objeto como uma lista, os dados armazenados nesse elemento podem ser modificados. Ah, mas é preciso entender que esse tipo de alteração não é feita na estrutura da tupla, apenas no conteúdo desse objeto, que é passado como referência.

Assim:

```
1 tupla = (('MG', 'Minas Gerais'), ('SP', 'São Paulo'), [0,1,2,3])
2
3 print(f'Remove o ultimo elemento da lista da tupla e retorna: {tupla
4 # Remove o ultimo elemento da lista da tupla e retorna: 3
5
6 print(f'Resultado da tupla com o objeto lista modificado: {tupla}')
7 # Resultado da tupla com o objeto lista modificado: (('MG', 'Minas G
8 erais'), ('SP', 'São Paulo'), [0, 1, 2])
9
```

Funções

Funções

Uma **função** é uma **sequência de comandos** que **executa uma tarefa**, sua principal finalidade é nos ajudar a **organizar programas em pedaços** que correspondam a como imaginamos uma solução do problema.

No python, além de podermos criar funções, temos a opção de usar **funções embutidas** que estão disponíveis para nos ajudar a escrever nosso programa e economizar tempo, pois estaremos utilizando funções existentes e não precisaremos criar.

Na documentação oficial: <https://docs.python.org/pt-br/3/library/functions.html> , temos a lista dessas funções, hoje vamos ver três tipos delas:

- **Map**
- **Filter**
- **Reduce**

Funções

Primeiramente, vamos focar nas três funções mais poderosas do Python, pois os três principais pilares da programação funcional é mapear, filtrar e reduzir

...

MAP, FILTER, REDUCE

```
[🐮, 🥔, 🐔].map(cook) = [🍔, 🍟, 🍗]  
[🍔, 🍟, 🍗].filter(veggie) = [🍟]  
[👨, 🚑, 🩺].reduce(build) = 🏥
```

Funções

map() - Syntax *map(função, iteráveis)*

Utilizamos esta função quando precisamos realizar uma operação específica nos itens da lista e **transformá-los** em outra coisa.

Ex:

[🐮, 🥔, 🐔].map(cook) = [🍔, 🍟, 🍗]

```
1  # Dada uma lista, retornar outra lista
2  # contendo a soma do elemento por ele mesmo.
3  lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
4  lista_somada = map(lambda x: x+x, lista)
5  print(list(lista_somada))
6
7  # Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Funções

filter() - Syntax *filter(função, iteráveis)*

Utilizamos o **filter()** quando precisamos **filtrar elementos** da lista.

Ex:

[🍔, 🍟, 🍗].filter(veggie) = [🍟]



```
1 # Dada uma lista, retornar apenas os números pares
2 lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 pares = filter(lambda x: x%2 == 0, lista)
4 print(list(pares))
5 # Output: [0, 2, 4, 6, 8]
```


Funções

reduce() - Syntax *reduce(função, iteráveis)*

Esta função **aplica uma operação** em todos os elementos da lista **reduzindo** a apenas um elemento.

Agora, diferente das outras duas, essa função não é diretamente do interpretador, precisamos importar o **módulo *functools***.

Ex:

[👤, 🚑, 💉].reduce(build) = 🏥

```
1  from functools import reduce
2
3  # Somar os itens da lista
4  lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5  soma = reduce(lambda x, y: x+y, lista, 0)
6  print(soma)
7  # Output: 45
```

Funções

reduce() - Syntax *reduce(função, iteráveis)*

Esta função **aplica uma operação** em todos os elementos da lista **reduzindo** a apenas um elemento.

Agora, diferente das outras duas, essa função não é diretamente do interpretador, precisamos importar o **módulo *functools***.

Ex:

[👤, 🚑, 💉].reduce(build) = 🏥

```
1 from functools import reduce
2
3 # Somar os itens da lista
4 lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 soma = reduce(lambda x, y: x+y, lista, 0)
6 print(soma)
7 # Output: 45
```

Funções


O que fazer para criar uma função?

A **sintaxe** de uma função é definida por três partes: **nome, parâmetros e corpo**, o qual agrupa uma **sequência de linhas** que representa algum comportamento. No código abaixo, temos um exemplo de declaração de função em Python que tem como objetivo imprimir o parâmetro passado.

Uma função **só é executada** quando **chamamos por ela**, observe que não só criamos a função *foo*, como logo abaixo a chamamos passando o parâmetro.

Execute o código abaixo e no canal do slack informe qual foi sua saída.

```
1 def foo(value):  
2     print(f'Olá, esse é o parametro: {value}')
```



```
3  
4     foo('LuizaCode')
```

Lições Extras

Vamos exercitar o que aprendemos de funções?

Entrada:

```
lista = [100, 248.90, 88, 124.90]
```

```
def desconto(preco):  
    return preco * (1 - 0.1)
```

- 1 - Dada uma lista com n valores, aplicar a função de desconto usando map()
- 2 - Retornar os valores maiores que 100, usando filter()
- 3 - Somar todos os valores da lista usando reduce()

Saída:

- 1 - [90.0, 224.010000000000002, 79.2, 112.410000000000001]
- 2 - [248.9, 124.9]
- 3 - 561.8



Tipos de Parâmetros

Parâmetros

Parâmetros são os nomes dados aos **atributos** que uma **função** pode **receber**, são eles que definem quais são os **argumentos** aceitos por uma função, podendo ou não ter um **valor padrão**.

Temos aqui um exemplo de uma função recebendo dois parâmetros, um de valor que é obrigatório e um de horas, que não é obrigatório porque tem um valor padrão, porém pode ser modificado se informado.

```
1 def calcula_salario(valor, horas=220):  
2     return valor * horas  
3  
4 print(calcula_salario(35))  
5 # output: 7700
```

Parâmetros

***args e **kwargs**

As palavras **args** e **kwargs** são apenas usadas como **convenção**, poderia receber qualquer outro nome: `*params`, `**kparams`, por exemplo.

Afinal, o que são?

***args** é usado para **passar uma lista de argumentos variável** sem palavras chaves em forma de **tupla**, pois a função que recebe não necessariamente saberá quantos argumentos serão passados.

Exemplo:

```
1 def foo(*args):
2     print(f'conteudo: {args}')
3
4     for i in args:
5         print(i)
6
7 foo('Hello', 'Moças', 'LuizaCode')
```


Parâmetros

... e ****kwargs**:

É a abreviação de keyword arguments, ele permite passar um dicionário com inúmeras chaves para a função.

Isso deixa definido que tal função irá receber tais valores, pronto.

```
1 def foo(**kwargs):
2     print(f'O nome dela(e) é: {kwargs.get("nome")}')
3
4
5 foo(nome='Jhon',
6     idade='28',
7     pais='Brasil')
8
```

Decorator

Decorator

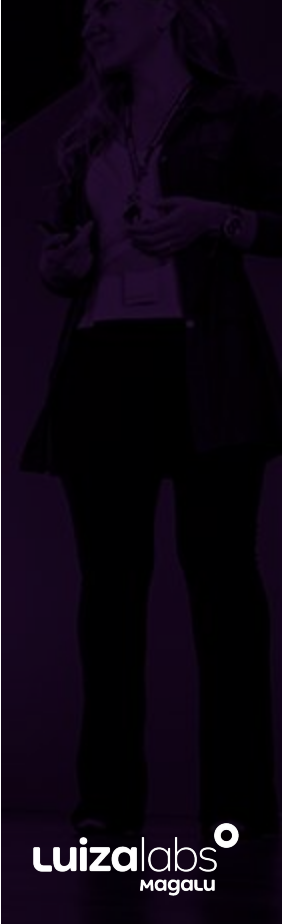
Lá na PEP 318, veio a proposta de melhoria na linguagem, que propôs o decorator, que nada mais é do que um método para envolver uma função, modificando o seu comportamento.

Dá um olhada no código abaixo, que tem a seguinte saída:

```
>>> Estou antes da execução da função passada como argumento  
>>> Sou um belo argumento  
>>> Estou depois da execução da função passada como argumento
```

```
1  def decorator(funcao):  
2      def wrapper():  
3          print ("Estou antes da execução da função passada como argumento")  
4          funcao()  
5          print ("Estou depois da execução da função passada como argumento")  
6  
7          return wrapper  
8  
9  def outra_funcao():  
10     print ("Sou um belo argumento!")  
11  
12     funcao_decorada = decorator(outra_funcao)  
13     funcao_decorada()
```

Luiza
<CODE>



luizalabs
magalu

Podemos usar o decorator com o **@**, que fará o mesmo papel de como se estivéssemos chamando a função da imagem anterior, mostrarei um exemplo útil de utilização de decorator, colocando o **@ em cima da função que será decorada**.

Luizalabs^o
magalu

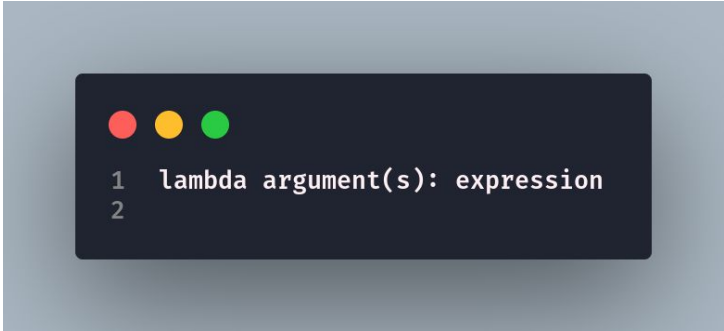
Lambda

Lambda

Não é lambida, é **LAMBDA**! É aqui que todo dev corre até entender de fato o que são, só que funções lambdas são como todas as outras: **funções normais**.

Exceto pelo fato de **não ter um nome para defini-la** e estar **contida em uma linha de código**.

A **sintaxe** é assim:



```
1 lambda argument(s): expression
2
```

Lambda

Uma função lambda funciona da seguinte forma: **você dá à função um valor (argumento)** e então **fornece a operação (expressão)**.

A palavra **lambda** vem na frente, depois vem ":" e o **argumento**.

Quais são os pós de usá-las?

- Bom para operações lógicas simples que são fáceis de entender, isso torna o código mais legível também
- Bom para funções que serão usadas apenas uma vez.

E os contras?

- Elas só podem executar uma única expressão, por isso tem que ser usada somente no simples
- Se a função abrange mais de uma linha, esquece, não funcionará
- Dependendo do contexto ela vai ser executada somente uma

Lambda

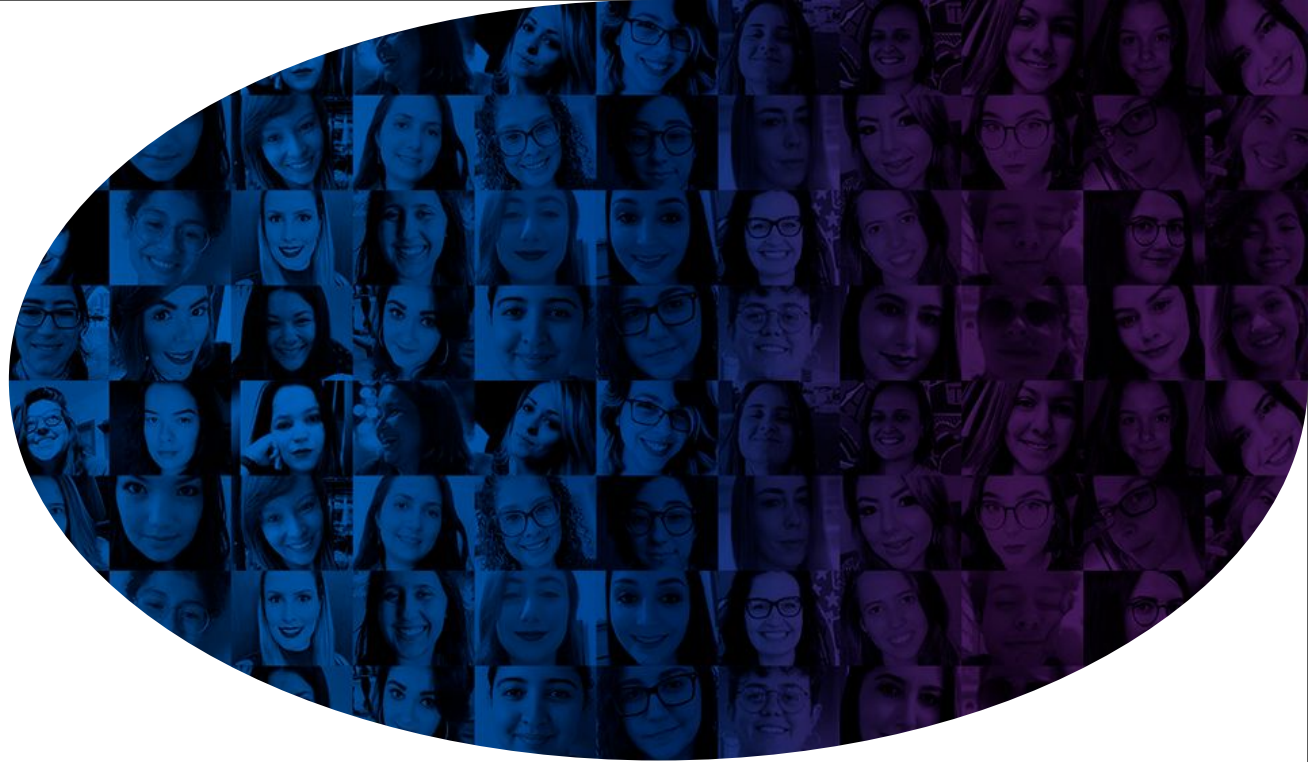
Usei em slides anteriores a função lambda para facilitar os resultados, nas compreensões de lista, nas funções de filter, map, reduce. Contudo, vou mostrar um exemplo que deixa claro como podemos reduzir uma função simples, que busca os valores pares de uma lista, em uma linha.

```
1 lista = [1,2,3,4,5,6,7,8,9]
2
3 # Assim funciona, mas ...
4 pares = []
5 def func_par():
6     for i in lista:
7         if i%2 == 0:
8             pares.append(i)
9     return pares
10
11 print(func_par())
12 # output: [2, 4, 6, 8]
13
14 # A melhor forma:
15
16 print(list(filter(lambda x: x%2==0, lista)))
17 # output [2, 4, 6, 8]
18
```

Lambda

Funções lambdas por não ser nomeadas, são funções que chamamos de **anônimas**.
Outro exemplo para fixar:

```
1  # dada a lista, retornar os valores elevado ao quadrado,  
2  # para elevar um número ao quadrado podemos fazer o valor ** 2  
3  lista = [1, 2, 3, 4, 5]  
4  
5  print(list(map(lambda x: x ** 2, lista)))  
6  
7  # output: [1, 4, 9, 16, 25]
```



Perguntas?

Magalu



#VemSerFeliz