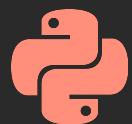


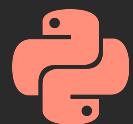
Funções

Live de Python #250 - #251 - #252

- **Fornecer um glossário** a quem estiver começando
- **Fundamentação teórica** a quem já tem bagagem
- Explorar lugares **não tão comuns**
- Ser um **hub de outras lives**



Objetivos dessa live

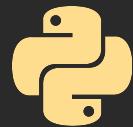


- **Fornecer um glossário** a quem estiver começando
- **Fundamentação teórica** a quem já tem bagagem
- Explorar lugares **não tão comuns**
- Ser um **hub de outras lives**

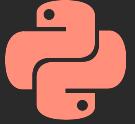
PS: Vamos no modo lento, **perguntem com força**, podemos dividir essa live em duas partes e **terminar esse assunto na próxima live!**



PS!



Roteiro



1. Subprogramas

O que são? Quais suas formas? Como são chamadas? Quais seus tipos?

2. Cabeçalho

Parâmetros de N formas, argumentos, anotações, fluxo, ...

3. Corpo

Subprogramas, funções, geradores, subgeradores, ...

4. O objeto

Funções como objeto, funções de ordem superior, funções em estruturas, ...



apoia.se/livedepython



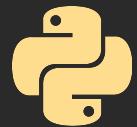
pix.dunossauro@gmail.com



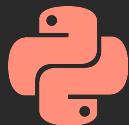
patreon.com/dunossauro



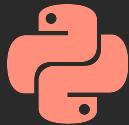
Ajude o projeto <3



Ademar Peixoto, Adilson Herculano, Adriano Ferraz, Alemao, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alex Lima, Alfredo Braga, Alisson Souza, Alysson Oliveira, Andre Azevedo, Andre Mesquita, Andre Paula, Aquiles Coutinho, Arnaldo Turque, Aslay Clevisson, Aurelio Costa, Ayr-ton, Bernardo At, Bruno Almeida, Bruno Barcellos, Bruno Freitas, Bruno Lopes, Bruno Ramos, Caio Nascimento, Carlos Ramos, Christian Semke, Claudemir Firmino, Cristian Firmino, Damianth, Daniel Freitas, Daniel Wojcickoski, Danilo Boas, Danilo Segura, Danilo Silva, David Couto, David Kwast, Davi Goivinho, Davi Souza, Dead Milkman, Delton Porfiro, Denis Bernardo, Diego Farias, Diego Guimarães, Dilenon Delfino, Dino, Diogo Paschoal, Edgar, Edilson Ribeiro, Eduardo Silveira, Eduardo Tolmasquim, Emerson Rafael, Erick Andrade, Érico Andrei, Everton Silva, Fabiano, Fabiano Tomita, Fabio Barros, Fábio Barros, Fabio Correa, Fábio Thomaz, Fabricio Biazzotto, Fabricio Patrocínio, Felipe Augusto, Felipe Rodrigues, Fernanda Prado, Fernando Celmer, Firehouse, Flávio Meira, Francisco Neto, Francisco Silvério, Gabriel Espindola, Gabriel Mizuno, Gabriel Paiva, Gabriel Ramos, Gabriel Simonetto, Geandreson Costa, Geizielder, Giovanna Teodoro, Giuliano Silva, Guilherme Felitti, Guilherme Gall, Guilherme Piccioni, Guilherme Silva, Guionardo Furlan, Gustavo Suto, Haelmo, Haelmo Almeida, Harold Gautschii, Heitor Fernandes, Helvio Rezende, Henrique Andrade, Higor Monteiro, Italo Silva, Janael Pinheiro, Jean Victor, Jefferson Antunes, Joelson Sartori, Jonatas Leon, Jônatas Silva, Jorge Silva, José Gomes, Joséito Júnior, Jose Mazolini, Josir Gomes, Juan Felipe, Juan Gutierrez, Juliana Machado, Julio Franco, Júlio Gazeta, Júlio Sanchez, Julio Silva, Kaio Peixoto, Kálita Lima, Leandro Silva, Leandro Vieira, Leonardo Mello, Leonardo Nazareth, Lucas Carderelli, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Pavelski, Lucas Schneider, Lucas Simon, Luciano Filho, Luciano Ratamero, Luciano Teixeira, Luis Eduardo, Luiz Duarte, Luiz Lima, Luiz Paula, Mackilem Laan, Marcelo Araujo, Marcelo Campos, Marcio Moises, Marco Mello, Marcos Gomes, Marina Passos, Mateus Lisboa, Mateus Ribeiro, Mateus Silva, Matheus Silva, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Murilo Viana, Natan Cervinski, Nathan Branco, Ocimar Zolin, Otávio Carneiro, Patricia Minamizawa, Patrick Felipe, Pedro Henrique, Pedro Pereira, Peterson Santos, Pytonyc, Rafael Faccio, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Regis Santos, Renato José, Renato Oliveira, Renê Barbosa, Rene Bastos, Rene Pessoto, Ricardo Silva, Riverfount, Rjribeiro, Robson Maciel, Rodrigo Barretos, Rodrigo Oliveira, Rodrigo Quiles, Rodrigo Vaccari, Rodrigo Vieira, Rui Jr, Samanta Cicilia, Samuel Santos, Selmison Miranda, Téo Calvo, Thiago Araujo, Thiago Borges, Thiago Curvelo, Thiago Souza, Tony Dias, Tyrone Damasceno, Valdir, Valdir Tegon, Vinicius Stein, Walter Reis, Willian Lopes, Wilson Duarte, Yros Aguiar, Zeca Figueiredo



Obrigado você



Sub
progra
mas

O que são?

Funções



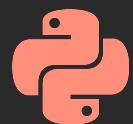
Na programação, funções são casos especiais de **subprogramas**.

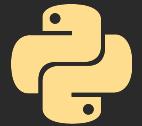


"Um **subprograma** é uma **abstração de processo, encapsulada, reutilizável**, que geralmente tem um **nome** e pode ser **parametrizável**."



Subprograma [definição pessoal]





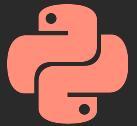
Subprograma

Um **subprograma** é uma

- **abstração de processo** (uma lista de passos)
- **encapsulada** (com sua lógica isolada),
- **reutilizável** (pode ser chamada diversas vezes),
- que *geralmente* tem um **nome** e
- pode ser **parametrizável**.

Exemplo de subprograma

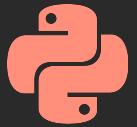
[<https://github.com/dunossauro/videomaker-helper/blob/main/vmh/audio.py#L125-L145>]



```
def extract_audio(  
    video_file: str, output_file: str, eq: bool = True  
) -> Path | tuple[Path, ...]:  
  
    audio: AudioSegment = AudioSegment.from_file(video_file)  
    audio.export(output_file, format='wav')  
  
    if eq:  
        logger.info(f'-eq enabled')  
        process_audio(output_file, 'eq_' + output_file)  
        return Path(output_file), Path('eq_' + output_file)  
  
    return Path(output_file)
```

Exemplo de subprograma

[<https://github.com/dunossauro/videomaker-helper/blob/main/vmh/audio.py#L125-L145>]



```
def extract_audio(nome, video_file: str, output_file: str, eq: bool = True) > Path | tuple[Path, ...]:  
    audio: AudioSegment = AudioSegment.from_file(video_file)  
    audio.export(output_file, format='wav')  
  
    if eq:  
        logger.info(f'-eq enabled')  
        process_audio(output_file, 'eq_' + output_file)  
        return Path(output_file), Path('eq_' + output_file)  
  
    return Path(output_file)
```

Parâmetros

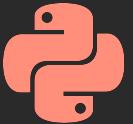
Passos



Subprograma

Um **subprograma** é uma

- **abstração de processo** (uma ~~lista de passos~~)
- **encapsulada** (com sua lógica isolada),
- **reutilizável** (pode ser chamada diversas vezes),
- que *geralmente* tem um **nome** e
- ~~pode ser **parametrizável**.~~



Chamada e Reutilização

As **chamadas** de subprogramas são feitas a partir do seu nome, seguido por () .

São reutilizáveis, pois podem ser chamadas diversas vezes

```
- □ ×  
path_1 = extract_audio('video_1.mp4', 'result_1.wav', eq=False)  
path_2 = extract_audio('video_2.mp4', 'result_2.wav', eq=False)  
path_3 = extract_audio('video_3.mp4', 'result_3.wav', eq=False)  
path_4, path_eq_4 = extract_audio('video_4.mp4', 'result_4.wav', eq=True)
```

* chamada é um termo técnico, em inglês refere-se a **Callable** [falaremos mais disso lá na frente]

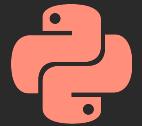


Subprograma

Um **subprograma** é uma

- **abstração de processo** (uma ~~lista de passos~~)
- **encapsulada** (com sua lógica isolada),
- ~~reutilizável~~ (pode ser chamada diversas vezes),
- que geralmente tem um **nome** e
- pode ser ~~parametrizável~~.

Explicar aqui abstração e encapsulamento!



Tipos de subprogramas

Existem duas categorias distintas de subprogramas:

- Procedimentos** [procedures]: **Não** retornam valores
- Funções**: Retornam valores

```
- □ ×  
  
def envia_email/remetente, destinatário, assunto, corpo:  
    mensagem = f"""  
        subject: {assunto}  
  
        {corpo}"""  
  
    with smtplib.SMTP_SSL(servidor, porta, contexto) as servidor:  
        server.login/remetente, senha)  
        server.sendemail/remetente, destinatário, mensagem)
```

Exemplo de um procedimento

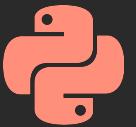
Procedimentos



A ideia por trás dos procedimentos é executar uma tarefa e pode retornar seus valores através da manipulação do escopo global

The thumbnail features a man with curly hair and a beard looking surprised, with his hand raised. The title 'ESCOPOS E NAME SPACES' is overlaid in large white letters. The video details are: 'Escopos e Namespaces | Live de Python #238', '3,3 mil visualizações • Transmitido há 5 meses', and 'Eduardo Mendes'. A description below reads: 'Nessa live iremos como funcionam os relacionamentos entre os nomes'. The Python logo is in the bottom right corner of the thumbnail.

Função



Já a função se baseia somente no estado dos parâmetros recebidos via argumentos durante a chamada.

Parâmetros

```
def extract_audio(video_file, output_file, eq=True): ...
```

- □ ×

```
path_1 = extract_audio('video_1.mp4', 'result_1.wav', eq=False)
```

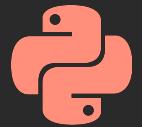
Argumentos



Efeitos colaterais

Dizemos que quando um subprograma altera variáveis [estado] fora do seu escopo interno ele gera um efeito colateral na aplicação

```
>>> lista = []
>>> def aumenta_lista(val):
...     lista.append(val)
...
>>> aumenta_lista(1)
>>> lista
[1]
>>> aumenta_lista(2)
>>> lista
[1, 2]
```



Funções puras

Dizemos que uma função que **não gera efeitos colaterais** é uma **função pura**. Pois ela manipula somente os parâmetros, os transforma e retorna um resultado

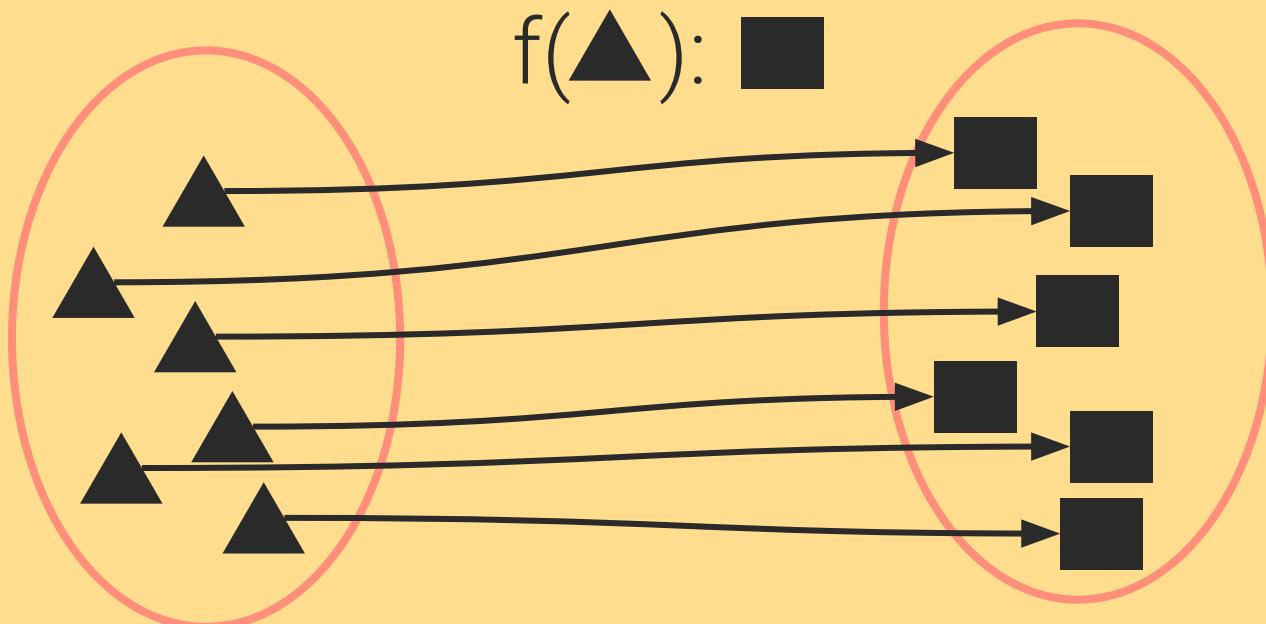
— □ ×

```
def soma(x, y):  
    return x + y
```

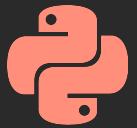


Transformação

A ideia por trás de uma função é transformar um dado em outro



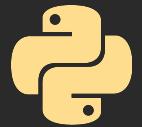
Falando em funções



Existem duas **palavras reservadas** para criar uma função em Python:

lambda

def



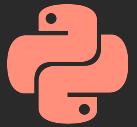
Funções lambda

A ideia por trás de uma função lambda é que ela **não tenha um nome**, seja **anônima**. Funções lambda em python são limitadas a uma única expressão. Onde `lambda` é o statement que define a função, `x` é o parâmetro dessa função e `x + 1` é o corpo da função. O resultado da expressão é o que será retornado

- □ ×

```
>>> lambda x: x + 1
<function <lambda> at 0x7f39742c3380>
>>> (lambda x: x + 1)(10)
11
```

Def



Já o statement `def` define uma função **nomeada**, que pode ser chamada no futuro e ser reutilizada.

- □ ×

```
def soma_mais_1(x):  
    return x + 1  
  
resultado = soma_mais_1(10)
```



Partes de um subprograma

Costumamos dividir os subprogramas em duas partes

```
def extract_audio(  
    video_file: str, output_file: str, eq: bool = True  
) -> Path | tuple[Path, ...]:
```

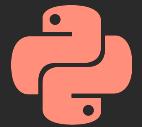
Cabeçalho

```
    audio: AudioSegment = AudioSegment.from_file(video_file)  
    audio.export(output_file, format='wav')
```

Corpo

```
        if eq:  
            logger.info(f'-eq enabled')  
            process_audio(output_file, 'eq_' + output_file)  
        return Path(output_file), Path('eq_' + output_file)
```

```
    return Path(output_file)
```



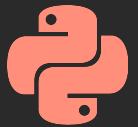
Às vezes temos Docstring :)

```
def extract_audio(video_file: str, output_file: str, eq: bool = True) -> Path | tuple[Path, ...]:  
    """Extract audio from video.  
  
    Args:  
        video_file: Video to extract audio  
        output_file: Path to save audio file  
        eq: Enable default equalization  
  
    Returns:  
        A audio Path  
    """  
    audio: AudioSegment = AudioSegment.from_file(video_file)  
    audio.export(output_file, format='wav')  
  
    if eq:  
        logger.info(f'-eq enabled')  
        process_audio(output_file, 'eq_' + output_file)  
        return Path(output_file), Path('eq_' + output_file)  
  
    return Path(output_file)
```

Nome, tipos de
parâmetros,
anotações,

...

Cabe
çalho



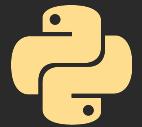
Cabeçalho

O **cabeçalho** da função é a **interface** com que o programa, a unidade chamadora, vai interagir com o subprograma. Ele é formado por:

- Nome
- Argumentos, opcional
- Anotação de tipo de retorno, opcional
- Definição de tipo genérico, opcional

Cabeçalho

```
def extract_audio(  
    video_file: str, output_file: str, eq: bool = True  
) -> Path | tuple[Path, ...]:
```

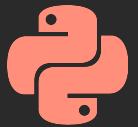


Nomes de subprogramas

Existem boas práticas e também algumas limitações para criação de nomes:

1. **Não podem iniciar** com números e caracteres especiais
2. Devem usar sempre letras minusculas (PEP-8)
3. Devem separar palavras com underline [snake_case] (PEP-8)
4. *Em alguns casos, a biblioteca padrão apresenta nomes em *mixedCase*, pois foram definidas antes de 2001, criação da PEP-8

```
- □ ×  
  
# correto  
def minha_função(): ...  
  
# errado PEP-8  
def MinhaFunção(): ...  
def minhaFunção(): ...  
def Minha_Função(): ...  
  
# Erro de sintaxe  
def 2func(): ...
```



Parâmetros

A forma tradicional de fornecer dados a subprogramas é com a passagem de argumentos durante a chamada do mesmo.

Parâmetros são uma espécie de variáveis internas usadas durante a definição de um subprograma.

— □ ×

```
def soma(x, y):  
    return x + y
```

* Na literatura você pode encontrar a nomenclatura parâmetros reais para argumentos e parâmetros formais para parâmetros!



Tipos de parâmetros

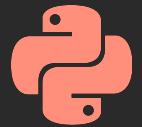
Existem diversos tipos de parâmetros que podemos usar em python:

- Posicionais: Que dependem que quem usa a interface saiba qual a ordem de chama dos parâmetros
- Chave-Valor [nomeados]: Que podem ser chamados independe da ordem, mas é necessário saber o nome do parâmetro

```
- □ X  
# Chave - valor  
>>> soma(y=1, x=2)  
3
```

```
# Posicional  
>>> soma(1, 2)  
3
```

```
0 1  
- □ X  
def soma(x, y):  
    return x + y
```



Posicionais e chave-valor

Por padrão, os parâmetros em python são "híbridos", pode ser chamados tanto como posicionais, quanto nomeados. Tendo somente a restrição de que **após a aparição de um parâmetro nomeado, todos os seguintes devem ser nomeados também**

```
In [17]: soma(1, y=2)
```

```
Out[17]: 3
```

```
In [18]: soma(x=1, 2)
```

```
Cell In[18], line 1
```

```
    soma(x=1, 2)
```

```
    ^
```

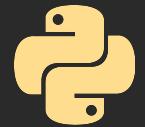
```
SyntaxError: positional argument follows keyword argument
```



Restrições em parâmetros

Os parâmetros têm mecanismos de restrição para que parâmetros sejam exclusivamente posicionais ou exclusivamente nomeados. Os parâmetros / e * respectivamente:

```
- □ ×  
← →  
-> >  
>>> def soma_3_valores(x, /, y, *, z):  
....:     return x + y + z
```



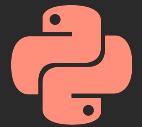
Restrições em parâmetros

Os parâmetros têm mecanismos de restrição para que parâmetros sejam exclusivamente posicionais ou exclusivamente nomeados. Os parâmetros / e * respectivamente:

```
>>> soma_3_valores(1, 2, z=3)  
6
```

```
>>> soma_3_valores(1, y=2, z=3)  
6
```

```
>>> def soma_3_valores(x, /, y, *, z):  
....:     return x + y + z
```

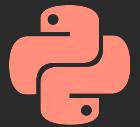


Empacotamento de parâmetros

Em alguns momentos pode existir uma limitação por conta dos parâmetros, não sabemos quantos vamos receber, ou propositalmente esperar um número incontável de argumentos. Para isso temos o empacotamento [star args em inglês]

– □ ×

```
def somatório(*args):  
    acumulador = 0  
    for val in args:  
        acumulador += val  
    return acumulador
```



Empacotamento de parâmetros

Em alguns momentos pode existir uma limitação por conta dos parâmetros, não sabemos quantos vamos receber, ou propositalmente esperar um número incontável de argumentos. Para isso temos o empacotamento [star args em inglês]

```
def somatório(*args):  
    acumulador = 0  
    for val in args:  
        acumulador += val  
    return acumulador
```

```
>>> somatório(1, 2, 3, 4, 5, 6)  
21  
>>> somatório(1, 2, 3, 4, 5)  
15  
>>> somatório(1, 2)  
3
```



Empacotamento de parâmetros

Também podemos empacotar parâmetros nomeados usando dois **

– □ ×

```
def cadastro(nome, telefone, **kwargs):
    print(f'{nome=}, {telefone=}, {kwargs=}')
```

KWARGS = Key-Word arguments = Argumentos de chave-valor



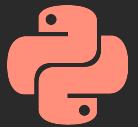
Empacotamento de parâmetros

Também podemos empacotar parâmetros nomeados usando dois **

```
def cadastro(nome, telefone, **kwargs):
    print(f'{nome=}, {telefone=}, {kwargs=}')
```

```
>>> cadastro(
...     'eduardo',
...     '99999999',
...     endereço='rua dos bobos 0',
...     telefone_residencial='11111111'
... )

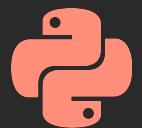
nome='eduardo', telefone='99999999',
{'endereço': 'rua dos bobos 0',
 'telefone_residencial': '11111111'}
```



Subprogramas como argumento

Uma propriedade especial de funções em python é o fato delas serem objetos também. O quer dizer que as funções também pode ser passadas como argumentos. Chamamos isso de **Função de ordem superior** [High order function]. Aqui as funções anônimas brilham.

```
- □ ×  
  
>>> verbos = ['comer', 'amar', 'saber', 'compor', 'cair']  
  
>>> sorted(verbos)  
['amar', 'cair', 'comer', 'compor', 'saber']  
  
>>> sorted(verbos, key=lambda x: x[-2:])  
['amar', 'comer', 'saber', 'cair', 'compor']
```



Subprogramas como argumento

Uma p
objeto
como
order f



O que são decoradores? | Live de Python #213

58 mil visualizações • Transmitido há 1 ano



Eduardo Mendes

Já se perguntou como funcionam os decoradores em Python? O que você

```
>>> verbos = ['comer', 'amar', 'saber', 'compor', 'cair']
```

```
>>> sorte  
['amar',
```

```
>>> sorte  
['LIVE DE PYTHON #136',  
 'FUNCTOOLS',  
 '1:43:55']
```



Live de Python #136 - Functools

2,9 mil visualizações • Transmitido há 3 anos



Eduardo Mendes

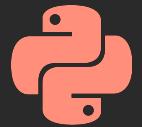
O canal é mantido por uma iniciativa de financiamento



Anotação de parâmetros

Uma frente recente na linguagem (3.5+) tem aderido às dicas de tipo, para explicar a aplicações externas quais são os tipos esperados pelos parâmetros. A sintaxe definida é `parâmetro: tipo`:

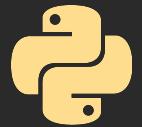
```
- □ ×  
  
def simulação_de_juros(valor: float, taxa: float, parcelas: int):  
    """Simulação de taxa de juros simples."""  
    template = "Juros em {}x parcelas: {}. Valor final: {}"  
  
    juros = valor * taxa * parcelas  
    valor_final = valor + juros  
  
    return template.format(parcelas, juros, valor_final)
```



Anotação de retorno

Em conjunto com esses metadados, também foi inserida a anotação para o retorno da função. Para isso descrevemos no cabeçalho o tipo de dado que uma função retorna. A sintaxe é `-> tipo`

```
- □ ×  
  
def simulação_de_juros(  
    valor: float, taxa: float, parcelas: int  
) -> str:  
    """Simulação de taxa de juros simples."""  
    template = "Juros em {}x parcelas: {}. Valor final: {}"  
  
    juros = valor * taxa * parcelas  
    valor_final = valor + juros  
  
    return template.format(parcelas, juros, valor_final)
```

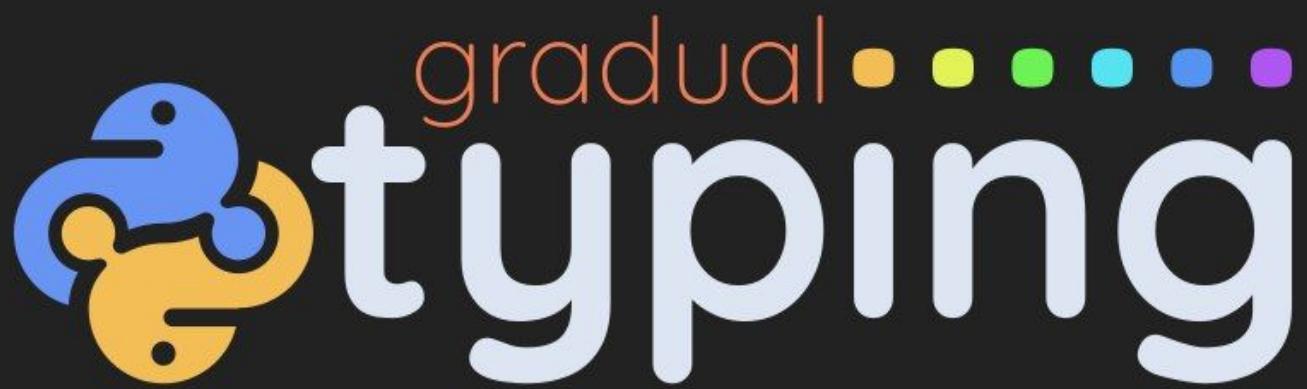


Variáveis de tipo

Na versão 3.12 do python foi introduzido o contexto para **variáveis de tipos genéricos** no cabeçalho dos subprogramas, adicionando uma variável genérica após o nome

— □ ×

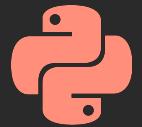
```
def soma[T](x: T, y: T) -> T:  
    return x + y
```



Em breve falaremos MUITO mais sobre isso!

A parte interna do
subprograma!

Corpo



Corpo [lambda]

O corpo dos subprogramas são responsáveis pela computação que ocorrerá quando o subprograma for chamado

— □ ×

```
lambda x: x + 1
```

```
lambda x: 10 if x < 10 else x
```



Corpo [lambda]

As **funções** lambda tem um comportamento diferente de funções tradicionais, elas não necessitam da palavra **return**. O resultado da expressão sempre será o retorno dela

— □ ×

```
lambda x: x + 1
```

```
lambda x: 10 if x < 10 else x
```



Corpo [def]

Já o corpo de subprogramas definidos com **def** podem conter N expressões e linhas dentro de seu corpo

```
def simulação_de_juros(  
    valor: float, taxa: float, parcelas: int  
) -> str:  
    """Simulação de taxa de juros simples."""  
    template = "Juros em {}x parcelas: {}. Valor final: {}"  
  
    juros = valor * taxa * parcelas  
    valor_final = valor + juros  
  
    return template.format(parcelas, juros, valor_final)
```



Função x Procedimento

O que pode diferenciar uma função de um procedimento em **def**s é o uso da palavra **return**.

– □ ×

```
def envia_email/remetente, destinatário, assunto, corpo):
    mensagem = f"""
        subject: {assunto}

        {corpo}"""

    with smtplib.SMTP_SSL(servidor, porta, contexto) as servidor:
        server.login/remetente, senha)
        server.sendemail/remetente, destinatário, mensagem)
```



Função x Procedimento

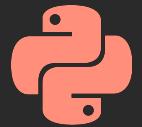
O que pode diferenciar uma função de um procedimento em **def**s é o uso da palavra **return**. Em outras linguagens de programação existe uma palavra usada no cabeçalho para dizer que não haverá retorno: **void**

- □ ×

```
def envia_email/remetente, destinatário, assunto, corpo):
    mensagem = f"""
        subject: {assunto}

        {corpo}"""

    with smtplib.SMTP_SSL(servidor, porta, contexto) as servidor:
        server.login/remetente, senha)
        server.sendemail/remetente, destinatário, mensagem)
```



Função x Procedimento

Um comportamento interessante em python é que não **existem procedimentos formais**. Toda def é em teoria uma função, pois caso não exista **return**. O interpretador retornará **None**

— □ ×

```
>>> def func( ):  
...     pass  
  
>>> result = func( )  
>>> print(result)  
None
```



Aninhamento

Um dos conceitos mais interessantes do corpo dos subprogramas é a possibilidade de definir subprogramas dentro de subprogramas.

– □ ×

```
# Exemplo de operações booleanas com funções
ID = lambda x: x

TRUE = lambda x: lambda y: x
FALSE = lambda x: lambda y: y

OR = lambda x: lambda y: x(TRUE)(y)
AND = lambda x: lambda y: x(y)(FALSE)
```



Aninhamento

Um dos conceitos mais interessantes do corpo dos subprogramas é a possibilidade de definir subprogramas dentro de subprogramas.

```
# Exemplo de operações boole
ID = lambda x: x

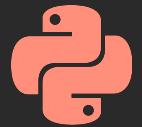
TRUE = lambda x: lambda y: x
FALSE = lambda x: lambda y:
          y

OR = lambda x: lambda y: x(T
AND = lambda x: lambda y: x(
```

```
— □ ×
```

```
# And
assert AND(TRUE)(TRUE) == TRUE
assert AND(TRUE)(FALSE) == FALSE
assert AND(FALSE)(FALSE) == FALSE
assert AND(FALSE)(TRUE) == FALSE

# OR
assert OR(FALSE)(TRUE) == TRUE
```

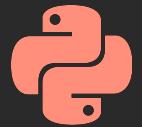


Aninhamento com defs

Também podemos aninhar subprogramas nomeados.

- □ ×

```
def add(x):  
    def inner(y):  
        return x + y  
    return inner
```



Aninhamento com defs

Também podemos aninhar subprogramas nomeados.

```
def add(x):  
    def inner(y):  
        return x + y  
    return inner
```

```
>>> add_1 = add(1)  
>>> add_1(2)  
3  
>>> add_1(4)  
5
```



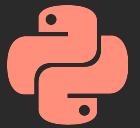
Dá até pra juntar :)

- □ ×

```
def add(x):  
    return lambda y: x + y
```

```
>>> add_1 = add(1)  
>>> add_1(2)  
3
```

Compartilhamento de escopo



Uma das grandes vantagens de criar subprogramas aninhados é que você pode atribuir **estados** definidos para os mesmos.

```
def add(x):  
    def inner(y):  
        return x + y  
    return inner
```

O valor de X fica
na "memoria" de
inner



Closure

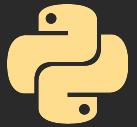
Damos o nome de **closure** para essa referência a variável que está na "memória" da função interna.

```
from inspect import getclosurevars

def add(x):
    return lambda y: x + y

>>> add_1 = add(1)
>>> getclosurevars(add_1)
ClosureVars(
    nonlocals={'x': 1}, globals={}, builtins={}, unbound=set()
)
```

Closure



Damos o nome de **closure** para essa referência a variável que está na "memória" da função interna.

```
from inspect import getclosurevars

def add(x):
    return lambda y: x + y

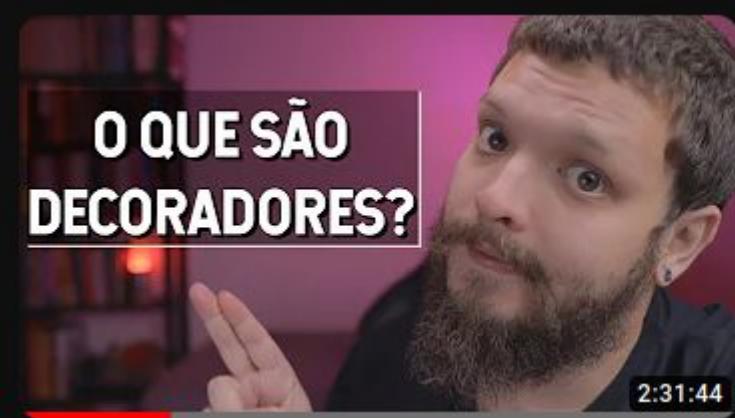
>>> add_1 = add(1)
>>> getclosurevars(add_1)
ClosureVars(
    nonlocals={'x': 1}, globals={}, builtins={}, unbound=set()
```

Dizemos que X é
uma variável livre

Closure



Damos
"memória"



O que são decoradores? | Live de Python #213

58 mil visualizações • Transmitido há 1 ano

Eduardo Mendes

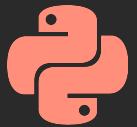
Já se perguntou como funcionam os decoradores em Python? O que você

```
fr
def add(x):
    return lambda y: x + y
```

```
>>> add_1 = add(1)
>>> getclosurevars(add_1)
ClosureVars(
    nonlocals={'x': 1}, globals={}, builtins={}, unbound=set()
```

Dizemos que X é
uma variável livre

Esses valores são armazenados no objeto



Vamos falar sobre isso lá na frente, mas esses valores estão nessas variáveis:

– □ ×

```
>>> add_1.__code__.co_freevars  
( 'x' , )
```

```
>>> add_1.__closure__  
(<cell at 0x7f7b56a635e0: int object at 0x7f7b5ab60d28>, )
```

Fluxo

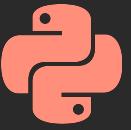


O fluxo segue como qualquer código imperativo, ele é lido linha a linha.
Mas nem sempre!

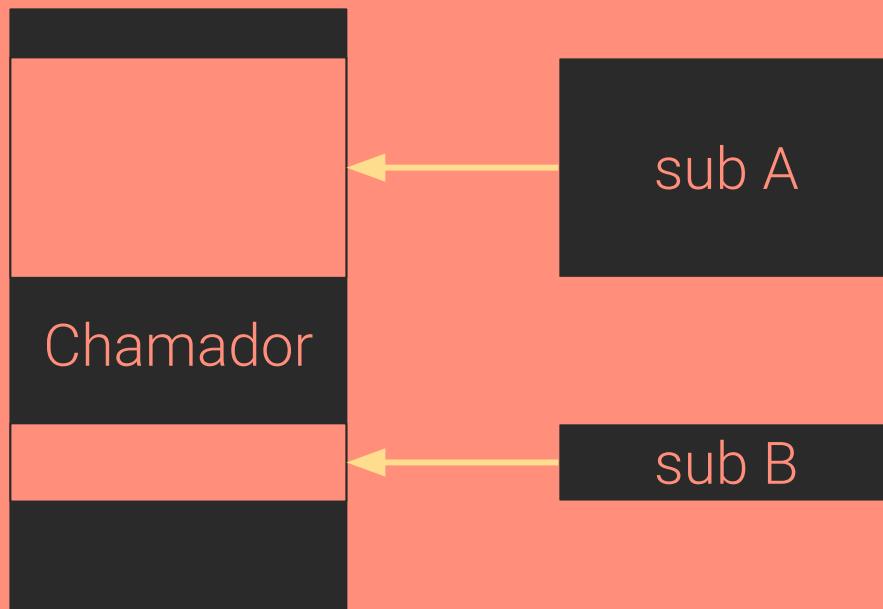
```
- □ ×  
  
def simulação_de_juros(  
    valor: float, taxa: float, parcelas: int  
) -> str:  
    """Simulação de taxa de juros simples."""  
    template = "Juros em {}x parcelas: {}. Valor final: {}"  
  
    juros = valor * taxa * parcelas  
    valor_final = valor + juros  
  
    return template.format(parcelas, juros, valor_final)
```



Fluxo



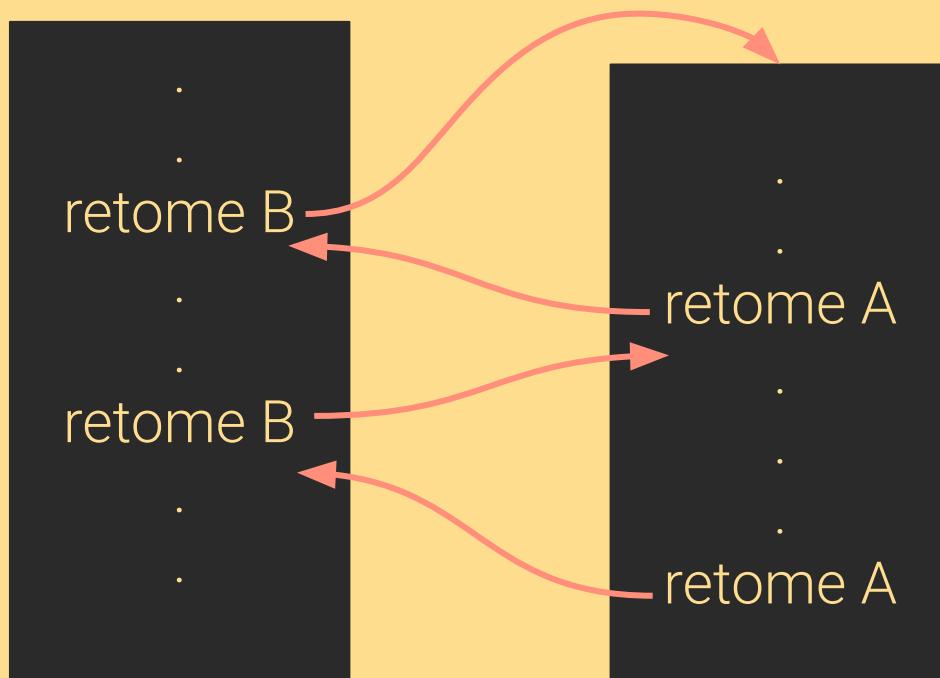
Uma das coisas que vimos até agora é quem um subprograma é sempre chamado por uma unidade chamadora, toma o controle e quando sua execução termina, o controle é retomado por quem fez a sua chamada.

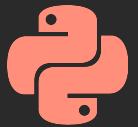


Corrotinas



As corrotinas são uma forma de um **modelo de controle de unidades simétrico**[*symmetric unit control model*]. Onde a responsabilidade é dividida com o subprograma em um sistema de **retomadas**[*resume*].





Corrotinas

As corrotinas em python são atualmente divididas em **duas categorias**. As corrotinas **clássicas** e as corrotinas **assíncronas**.

```
- □ ×  
  
def corrotina( ):  
    print('Iniciei o subprograma')  
    print('Agora delegarei ao chamador')  
    yield  
  
    print('Retomando a corrotina')  
    print('Agora delegarei ao chamador')  
    yield  
  
    print('Retomando a corrotina')  
    print('Agora delegarei ao chamador')
```

Corrotinas

– □ ×

```
def corrotina():
    print('Iniciei o subprograma')
    print('Agora delegarei ao chamador')
    yield
    print('Agora delegarei ao chamador')
    yield
    print('Agora delegarei a')
```

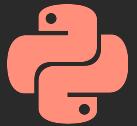
– □ ×

```
>>> c = corrotina()
>>> next(c) # delega para corrotina
'Iniciei o subprograma'
'Agora delegarei ao chamador'
```

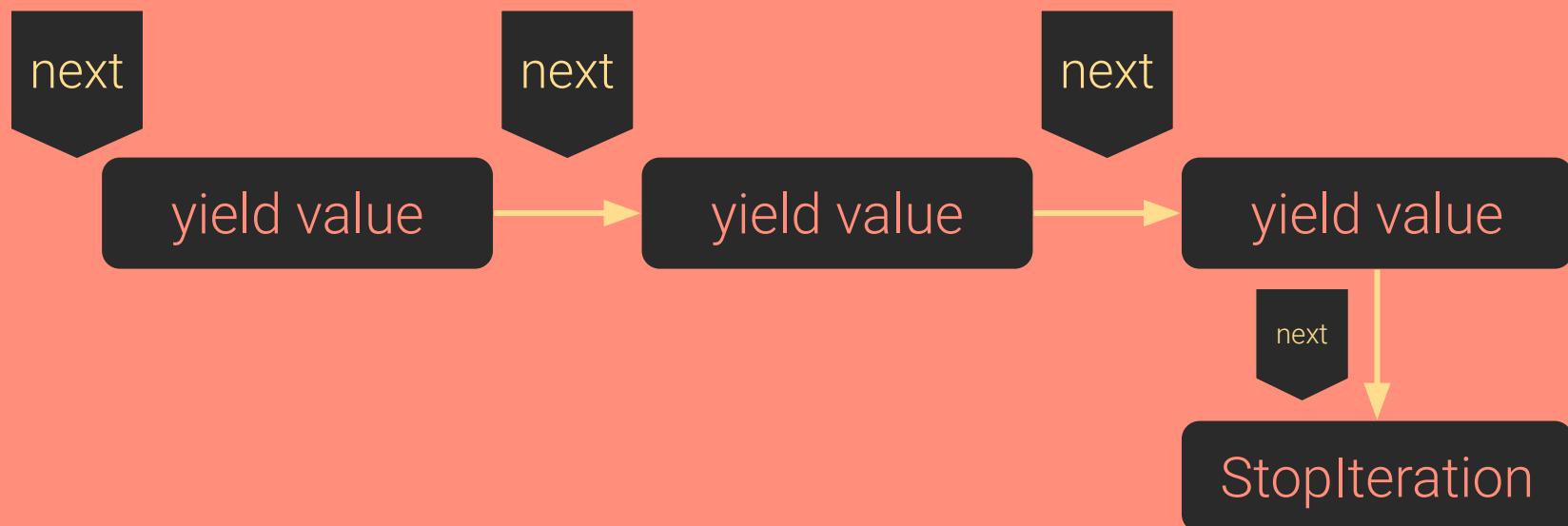
```
>>> next(c) # delega para corrotina
'Retomando a corrotina'
```

```
>>> next(c) # delega para corrotina
'Retomando a corrotina'
```

Geradores



Um dos tipos mais tradicionais de corوتina são os **geradores***. Geradores são criadores de dados com a condição simétrica de parada. Eles são a implementação do protocolo de **iteração** usando subprogramas.



Geradores



Iteráveis em python são coisas "desmontáveis", que conseguimos pegar as partes. Da forma como fazemos em um for.

— □ ×

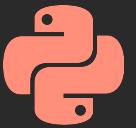
```
>>> for val in [1, 2, 3]:  
    print(val)
```

1

2

3

Geradores



A ideia desse tipo de subprograma é **gerar valores**.

```
def gerador():  
    yield 1  
    yield 2  
    yield 3
```

- □ ×

```
>>> for val in gerador():
```

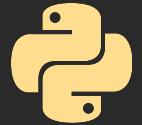
```
    print(val)
```

```
1
```

```
2
```

```
3
```

- □ ×



Exemplo de gerador

Geradores podem ser infinitos e por conta com o controle simétrico
podem ser continuados

– □ ×

```
def gerador(start=0, step=1):  
    while True:  
        yield start  
        start += step
```

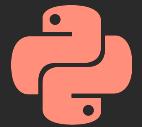
Exemplo de gerador



Geradores podem ser infinitos e podem ser continuados

Essa condição de só calcular quando for chamado, por conta da simetria, se chama **Avaliação preguiçosa** [lazy evaluation]

```
def gerador(start, step=1):  
    while True:  
        yield start  
        start += step
```



Sub Geradores

Uma das condições especiais de geradores é que eles podem **delegar** uma tarefa a outro gerador. Chamamos de sub geradores.

— □ ×

```
def subgerador( ):
    yield from gerador()
```

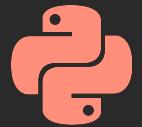
```
>>> g = subgerador()
```

```
>>> next(g)
```

```
0
```

```
>>> next(g)
```

```
1
```



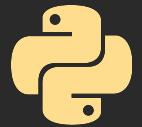
Sub Geradores

Uma das condições especiais de geradores é que eles podem **delegar** uma tarefa a outro gerador. Chamamos de sub geradores.

```
def subgerador():
    yield from ...

>>> g = subgerador()
>>> next(g)
0
>>> next(g)
1
```

The thumbnail features a man with curly hair and a beard, looking thoughtful with his hand on his chin. The title 'YIELD GERADORES CORROTINAS E AFINS' is displayed in a stylized font. The video duration '1:23:10' is shown in the bottom right corner. Below the thumbnail, the video details are: 'Live de Python #151 - Desvendando o yield e as funções geradoras', '8,1 mil visualizações • Transmitido há 2 anos', and the channel name 'Eduardo Mendes' with a small profile picture.



Corrotinas

Embora os geradores sejam corrotinas, quando nos referimos especificamente a corrotinas, queremos aproveitar a condição de parada para delegar uma tarefa específica ao subprograma.

Podemos usar o **yield como expressão** e passar valores usando o método `send`:

```
def corrotina():
    print('Começou')
    valor = yield
    print(f'Recebi: {valor}')
```



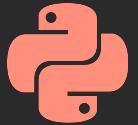
Corrotinas

Embora os geradores sejam
especificamente a corrotina
para delegar uma tarefa es-

Podemos usar o **yield con-**
send:

```
def corrotina():
    print('Começou')
    valor = yield
    print(f'Recebi: {valor}')
```

```
>>> c = corrotina()
>>> next(c) # Primeiro resultado
'Começou!'
>>> c.send(35)
'Recebi 35'
```



Um exemplo legal!

Exemplo de média cumulativa adaptado do fluent python (16.3)

```
def média():
    total = 0.0
    contador = 0
    média = None
    while True:
        entrada = yield média
        total += entrada
        contador += 1
        média = total/contador
```

```
coro = média()
next(coro) # preparação
coro.next(10) # 10.0
coro.next(20) # 15.0
```

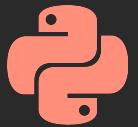


Corrotinas

Existe outro caso de uso além dos geradores com yield que são as corrotinas.

The screenshot shows a YouTube channel interface. On the left, there's a thumbnail for a video titled "Geradores e uma Introdução histórica à corrotinas com..." by Eduardo Mendes. On the right, there's a sidebar titled "Ordenar" with four video thumbnails listed:

- Live de Python #151 - Desvendando o yield e as funções geradoras** by Eduardo Mendes • 8,1 mil visualizações • Transmitido há 2 anos
- Live de Python #152 - Uma introdução histórica à corrotinas** by Eduardo Mendes • 6,5 mil visualizações • Transmitido há 2 anos
- Live de Python #153 - Uma introdução histórica à corrotinas PARTE 2** by Eduardo Mendes • 2,9 mil visualizações • Transmitido há 2 anos
- Live de Python #154 - Uma introdução histórica à corrotinas PARTE 3 (AsyncIO)** by Eduardo Mendes • 5 mil visualizações • Transmitido há 2 anos

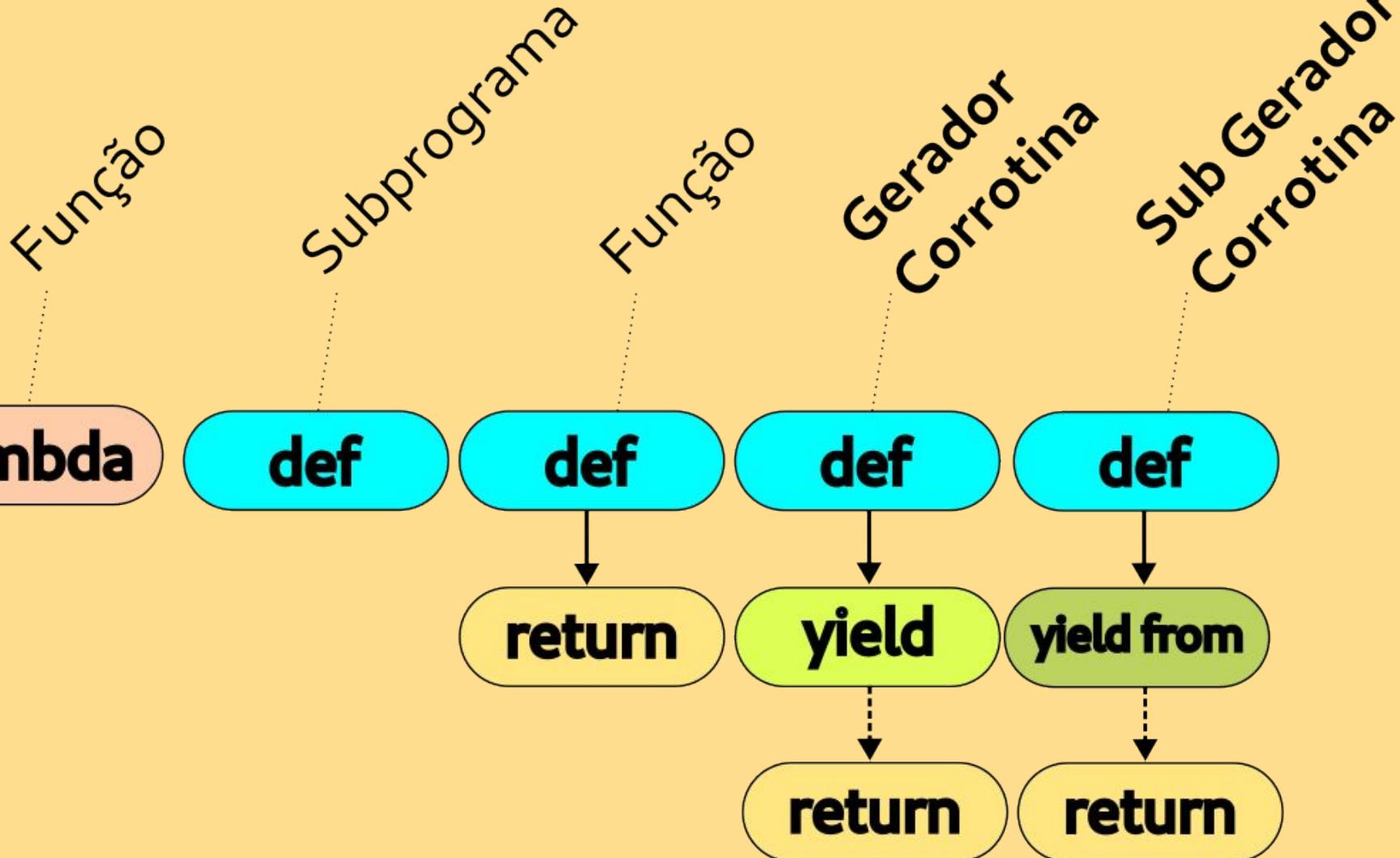


O corpo de um subprograma

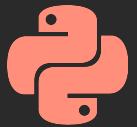
O corpo de um subprograma pode delimitar comportamentos completamente diferentes. Nesse ponto, chegamos em diferentes tipos de subprogramas. Palavras reservadas que podem ser encontradas em subprogramas:

- **yield**: Um subprograma em um gerador / corrotina
- **yield from**: Transforma o subprograma em um sub gerador
- **return**: Diz que o subprograma é uma função
- **_**: Caso nenhuma dessas coisas ocorra em nenhuma def, ela pode ser considerada um **procedimento**

Tipos de subprogramas



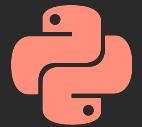
Corrotinas abrem espaço para assincronismo!



Além das corrotinas clássicas que vimos. Temos as **corrotinas assíncronas**

— □ ×

```
async def corrotina_async():
    valor = await outra_corrotina()
    return valor
```



Isso é assunto pra outro momento!



Trio: Concorrência estruturada | Live de Python #242

3,7 mil visualizações • Transmitido há 4 meses



Eduardo Mendes

Vamos explorar a biblioteca `Trío` em Python, um poderoso pacote para lidar com a progr



Requests assíncronos com HTTPX | Live de Python #234

5,4 mil visualizações • Transmitido há 6 meses

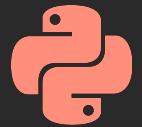


Eduardo Mendes

Nessa live vamos conversar sobre como fazer requests de forma assíncrona (com asy

Funções também
são

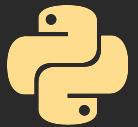
Objetos



Funções são objetos

Uma das propriedades interessantes do python, e de outras linguagens também, é que **funções são objetos**.

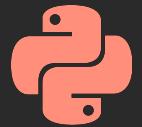
```
- □ ×  
calculadora = {  
    'soma': lambda x, y: x + y,  
    'sub': lambda x, y: x - y,  
    'mul': lambda x, y: x * y,  
    'div': lambda x, y: x / y,  
}
```



O que isso quer dizer?

Quer dizer que funções podem ser usadas como qualquer outro objeto.
Elas podem ser:

- Atribuídas a variáveis
- Inseridas em containers (listas, dicionários, etc.)
- Passadas como parâmetros
- Retornadas por funções
- ...



Um exemplo básico

Sempre gosto de iniciar com um exemplo simples, mas que ajuda a entender

— □ ×

```
def apply(func, val):  
    return func(val)
```

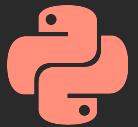
```
>>> apply(lambda x: x + 1, 2)  
3
```



Funções recebendo funções

Esse é o conceito por trás de diversas funções da biblioteca padrão do python, como:

- map
- filter
- sorted
- functools.reduce
- itertools.takewhile
- ...



Funções vs loops

Uma das formas mais comuns de usar funções como objeto é poder evitar o uso de loops como for e while. Transformando a maneira **imperativa** de se programar, para uma forma **declarativa**.

```
- □ X  
seq = [1, 2, 3, 4, 5]  
result = []  
  
for val in seq:  
    result.append(val ** 2)  
  
print(result)  
[1, 4, 9, 16, 25]
```

```
- □ X  
seq = [1, 2, 3, 4, 5]  
result = map(lambda x: x ** 2, seq)  
  
print(result)  
<map at 0x7f8a09d74760>
```

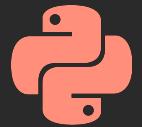
Gerador!



Funções vs Loops

— □ ×

```
from itertools import takewhile  
  
seq = [1, 2, 3, 4, 5]  
result = takewhile(lambda x: x < 3, seq)  
  
list(result) # [1, 2]
```



O retorno de funções

Outro ponto interessante é o fato de poder retornar funções. Casos usados por closures e decoradores. Um bom exemplo é a função partial em conjunto com o módulo operator

- □ ×

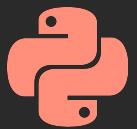
```
from functools import partial  
from operator import lt
```

```
lt_100 = partial(lt, 100)
```

```
>>> lt_100(10)
```

```
False
```

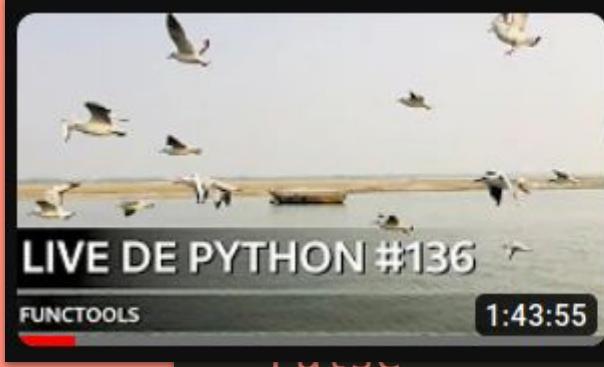
O retorno de funções



Outro ponto interessante é o fato de poder retornar funções. Casos usados por closures e decoradores. Um bom exemplo é a função partial em conjunto com o módulo operator

- □ ×

```
from functools import partial  
from operator import lt
```



Live de Python #136 - Functools

Eduardo Mendes • 2,9 mil visualizações • Transmitido há

O canal é mantido por uma iniciativa de financiamento co
@dunossauro ----- Código: <https://github.com/dunossauro>

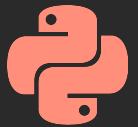


Juntando conceitos

Sabemos que uma função pode:

1. Ser colocada em um container (Objeto)
2. Receber uma função (HOF)
3. Retornar uma função (HOF)
4. Ser definida dentro de outra (Nested)
5. Carregar um estado interno (Closure)

Podemos explorar um pouco mais a união desses conceitos!



Juxt

Juxt é uma função tradicional em linguagens de programação funcional. Ela é uma função que cria uma função que aplica uma lista de funções a um mesmo valor. Resultado em uma lista de valores aplicados as funções recebidas.

$$([f_1, f_2, f_3, f_4], \text{val}) \rightarrow [f_1(\text{val}), f_2(\text{val}), f_3(\text{val}), f_4(\text{val})]$$

Para simplificar, vamos imaginar um cenário de validação de dados. Para uma string ser válida, ela tem que ser maior 3, menor que 10 e ter o caractere @:

$$[\text{gt_3}, \text{lt_10}, \text{has_at}] \rightarrow [\text{gt_3}(\text{str}), \text{lt_10}(\text{str}), \text{has_at}(\text{str})]$$



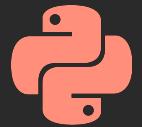
Pensando em uma implementação!

Uma básica, para entrarmos nos trilhos! **Não é reutilizável!**

— □ ×

```
gt_3 = lambda x: len(x) > 3
lt_10 = lambda x: len(x) < 10
has_at = lambda x: '@' in x

>>> [f('edu@edu') for f in (gt_3, lt_3, has_at)]
[True, True, True]
```



Tentando uma HOF

Podemos reutilizar, mas o dado e as funções tem que ser passadas juntas.

- □ ×

```
def applics(val, *funcs):  
    return [f(val) for f in funcs]
```

```
>>> applics('edu@edu', gt_3, lt_3, has_at)  
[True, True, True]
```



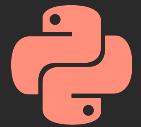
Se adicionarmos a closure, vamos ter um validador independente. Além de criarmos uma fábrica deles!

```
def juxtapose(*funcs):
    def inner(val):
        return [f(val) for f in funcs]
    return inner

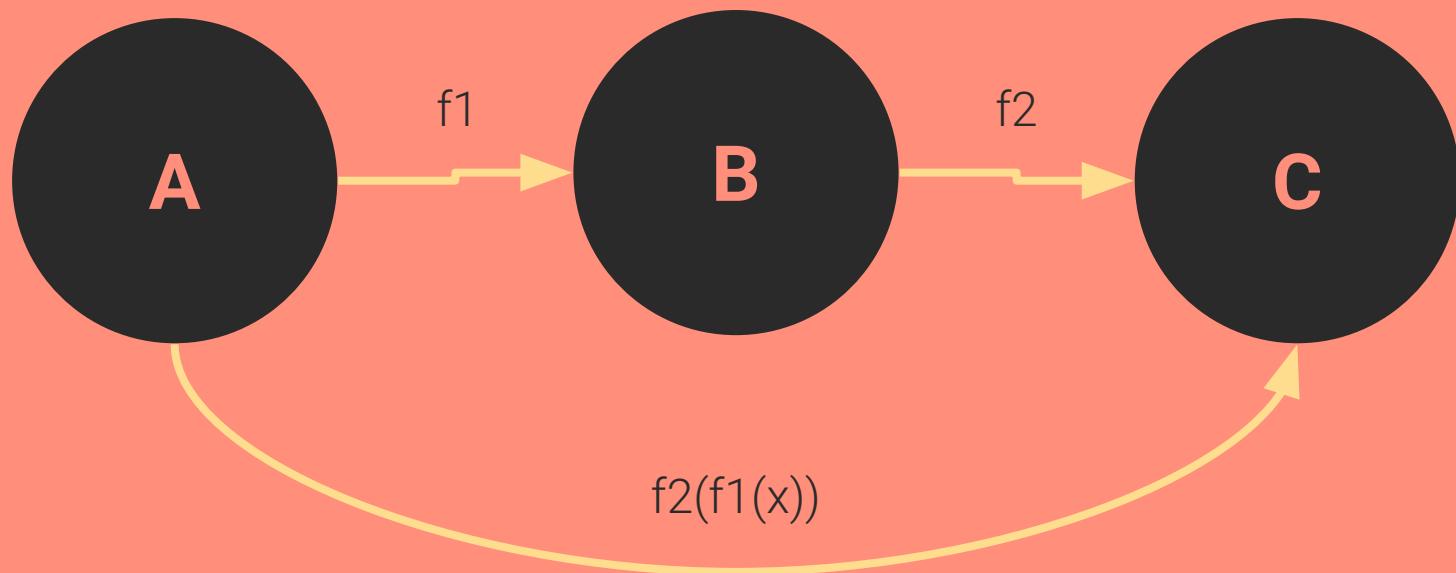
>>> valida_str = juxtapose(gt_3, lt_3, has_at)
>>> valida_str('edu@edu')
[True, True, True]

>>> valida_str('')
[False, True, False]
```

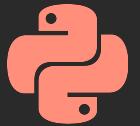
Aplicando o mesmo conceito, mas diferente



A aplicação de funções de forma consecutiva, uma composição de funções:



Aplicando o mesmo conceito, mas diferente



```
def compose(*funcs):
    def inner(data):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner
```



Exemplo de uso

- □ ×

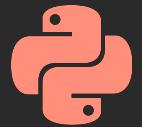
```
from functools import partial
from re import findall
from urllib.request import urlopen

urls = partial(findall, 'https://(.*)\\'')

def fetch(url):
    return urlopen(url).read().decode()

links = compose(urls, fetch)

>>> links('https://dunossauro.com')
['https:', 'https:...']
```



Puro suco das funções

— □ ×

```
def zip_with(func):
    def inner(seq_a, seq_b, /):
        yield from map(func, seq_a, seq_b)
    return inner

>>> zip_add = zip_with(lambda x, y: x + y)
>>> list(zip_add('abcd', 'dcba'))
['ad', 'bc', 'cb', 'da']
```



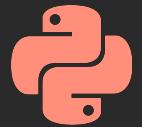
Classes também podem!

Trabalhando com objetos imutáveis

- □ ×

```
class Immutable:  
    def __init__(self, *data):  
        self.data = data  
  
    def map(self, func):  
        return Immutable(  
            *list(map(func, self.data)))  
  
    def __repr__(self):  
        vals = ', '.join(str(x) for x in self.data)  
        return f'Immutable({vals})'
```

```
>>> (Immutable(1, 2, 3, 4)  
...:     .map(lambda x: x + 2)  
...:     .map(lambda x: x**2))  
Immutable(9, 16, 25, 36)
```



Uma característica interessante

Como dissemos a alguns slides atrás, em python, pelo fato de "tudo ser objeto", qualquer objeto que implemente a interface para ser chamável, pode se comportar como subprograma:

— □ ×

```
class Subprograma:  
    def __call__(self):  
        print('Olá, eu sou um subprograma!')
```



apoia.se/livedepython



pix.dunossauro@gmail.com



patreon.com/dunossauro



Ajude o projeto <3

