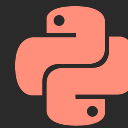




Escopos e Namespaces

Live de Python #238



1. Definição de nomes

Como criamos nomes? Vulgo variáveis.

2. Namespaces

Os lugares onde os nomes são armazenados.

3. Escopos

A relação na qual os nomes são acessados.

4. Tempo de vida

Quando nomes deixam de existir.



picpay.me/dunossauro



apoia.se/livedepython



pix.dunossauro@gmail.com



Ajude o projeto <3



Ademar Peixoto, Adilson Herculano, Adriano Ferraz, Alemao, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alex Lima, Alynne Ferreira, Alysson Oliveira, Ana Carneiro, Andre Azevedo, Andre Mesquita, Aquiles Coutinho, Arnaldo Turque, Aslay Clevisson, Aurelio Costa, Bernardo At, Bernardo Fontes, Bruno Almeida, Bruno Barcellos, Bruno Barros, Bruno Freitas, Bruno Lopes, Bruno Ramos, Caio Nascimento, Christiano Moraes, Damianth, Daniel Freitas, Daniel Wojcickoski, Danilo Boas, Danilo Segura, Danilo Silva, David Couto, David Kwast, Davi Govinho, Davi Souza, Delton Porfiro, Denis Bernardo, Diego Farias, Diego Guimarães, Dilenon Delfino, Diogo Paschoal, Diogo Silva, Edgar, Eduardo Silveira, Eduardo Tolmasquim, Elias Silva, Emerson Rafael, Eneas Teles, Erick Andrade, Érico Andrei, Everton Silva, Fabiano Tomita, Fabio Barros, Fábio Barros, Fabio Castro, Fábio Thomaz, Fabricio Patrocinio, Felipe Rodrigues, Fernanda Prado, Fernando Celmer, Firehouse, Flávio Meira, Francisco Neto, Francisco Silvério, Gabriel Espindola, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geandreson Costa, Geizielder, Gilberto Abrao, Giovanna Teodoro, Giuliano Silva, Guilherme Felitti, Guilherme Gall, Guilherme Silva, Guionardo Furlan, Gustavo Pereira, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Helvio Rezende, Hugo Cosme, Igor Riegel, Italo Silva, Janael Pinheiro, Jean Victor, Joelson Sartori, Jônatas Oliveira, Jônatas Silva, Jon Cardoso, Jorge Silva, José Gomes, Joseíto Júnior, Jose Mazolini, Juan Felipe, Juan Gutierrez, Juliana Machado, Julio Franco, Júlio Gazeta, Julio Silva, Kaio Peixoto, Kálita Lima, Kaneson Alves, Leandro Miranda, Leandro Silva, Leo Ivan, Leonardo Mello, Leonardo Nazareth, Leon Solon, Luancomputacao Roger, Lucas Adorno, Lucas Carderelli, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Lucas Simon, Lucas Valino, Luciano Filho, Luciano Ratamero, Luciano Teixeira, Luis Alves, Luis Eduardo, Luiz Duarte, Luiz Lima, Luiz Paula, Luiz Perciliano, Mackilem Laan, Marcelo Campos, Marcio Moises, Marco Mello, Marcos Gomes, Maria Clara, Marina Passos, Mateus Lisboa, Mateus Ribeiro, Mateus Silva, Matheus Silva, Matheus Vian, Mauricio Nunes, Mírian Batista, Mlevi Lsantos, Murilo Viana, Nathan Branco, Nicolas Teodosio, Otávio Carneiro, Patricia Minamizawa, Patrick Felipe, Paulo Tadei, Pedro Henrique, Pedro Pereira, Peterson Santos, Priscila Santos, Pydocs Pro, Pytonyc, Rafael Lopes, Rafael Romão, Rafael Veloso, Raimundo Ramos, Ramayana Menezes, Regis Santos, Renato Oliveira, Rene Bastos, Ricardo Silva, Riverfount, Rjribeiro, Robson Maciel, Rodrigo Barretos, Rodrigo Freire, Rodrigo Oliveira, Rodrigo Quiles, Rodrigo Ribeiro, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Ronaldo Silveira, Rui Jr, Samanta Cicilia, Téó Calvo, Thaynara Pinto, Thiago Araujo, Thiago Borges, Thiago Curvelo, Thiago Souza, Tiago Minuzzi, Tony Dias, Tyrone Damasceno, Uadson Emile, Valcilon Silva, Valdir Tegon, Vcwild, Vicente Marcal, Vinicius Stein, Vladimir Lemos, Walter Reis, William Vitorino, Willian Lopes, Wilson Duarte, Wilson Neto, Zeca Figueiredo



Obrigado você



Definição de

Nomes

Definição de nomes | Parte 1



Nomes se referem a objetos, nomes são atribuídos (ou ligados) [ou vinculados] por meio de algumas construções sintáticas.

- Atribuição **x = 1**
- Definição de classes **class MinhaClasse:**
- Definições de funções **def minha_função():**
- Alvos [target] de alguma atribuição
 - Variáveis de iteração **for variável in []:**
 - Assinaturas de gerenciadores de contexto **with open('algo.txt') as file:**
 - Assinaturas de exceção **except Exception as e:**

O que são nomes afinal?



Nome é um sinônimo para **identificador**, são formas de identificar um endereço de memória, fazer com que fique mais fácil de nos comunicar com ele.

Nomes em python podem ser definidos com qualquer palavra que não seja uma **palavra reservada** e também não ser iniciado por um número.

Python é sensível a variações de minúsculas e maiúsculas: `var != VAR`

Estilo e nomes



Python segue algumas regras de estilo (PEP-8) para criação de nomes:

- **variáveis e nomes_de_funções:**
 - letras minúsculas
 - separadas por _
- **CONSTANTES:**
 - Letras maiúsculas separadas por _
- **Classes e Tipos:**
 - Pascal Case: Toda palavra é iniciada com letra maiúscula:
 - MinhaClasse, MeuTipo
- **Variância:**
 - PascalCase, com um _ explicando o tipo de variância
 - T_contra, K_co

Palavras reservadas



2.3.1. Palavras-chave

Os seguintes identificadores são usados como palavras reservadas ou *palavras-chave* do idioma e não podem ser usados como identificadores comuns. Eles devem ser soletrados exatamente como está escrito aqui:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

https://docs.python.org/pt-br/3.12/reference/lexical_analysis.html#keywords

Variáveis



Toda definição de variável é composta por diversos atributos, como:

- **Nome / Apelido:** Que damos a um identificador
- **Endereço / Identificador:** O local da memória onde o objeto está
- **Valor:** Valor alocado ao endereço de memória
- **Tipo:** Tipo do dado associado ao valor

```
>>> a = 1

>>> id(a)
140236375389160

>>> type(a)
int
```

Definindo nomes



```
1  a = 7
2
3  def minha_função_bacana(): ...
4
5  class MinhaClasseBacana: ...
```



Nomes como alvos (targets)

Na forma de alvos temos algumas definições um pouco diferentes de criar nomes como já vimos. Pois os nomes são criados para um contexto específico.

Em um gerenciador de contexto, temos a definição de um nome usando a combinação das palavras reservadas **with** e **as**:

```
with open('arquivo.txt') as arquivo:  
    arquivo.read()
```

Já fizemos uma live sobre gerenciadores de contexto: Live 43

Nomes como alvos (targets)



Para o contexto de exceções temos uma regra que combina as palavras `except` e `as`:

```
try:
    1 / 0
except Exception as e:
    print(e) # division by zero
```

Já fizemos uma live sobre Exceptions: Live 60

Alvos como targets



Para os loops de for, temos a atribuição também ocorrendo de forma diferente:

```
for variável in [1, 2, 3, 4]:  
    print(variável)
```

Removendo nomes



Um operador pouco usado em python, mas que ajuda a excluir (desvincular) nomes criados é o **`del`**:

```
x = 123
```

```
del x
```

Faz com que o nome `x` não exista mais!

Definição de nomes | Parte 2



Porém, existem outras formas de dar nomes a objetos com alguns recursos diferentes e/ou mais modernos da linguagem:

- Em um import `from collections import *`
- **Python 3.8**
 - Atribuição de um nome em uma expressão `(x := 1)`
- **Python 3.10**
 - Na captura de padrão de pattern matching (`match case`)
- **Python 3.12**
 - Em uma definição de tipo `type Vetor = tuple[float, float]`
 - Em uma definição de listas de parâmetros `def função_massa[T](args: list[T]) -> T:`

No sistema de imports



Quando importamos código de outro arquivo ou de bibliotecas, estamos criando novos nomes na nossa execução do sistema:

```
1  from collections import Counter
2  # Counter agora é um nome que podemos usar
3
4  Counter([1, 1, 1, 1, 2])
5  # Counter({1: 4, 2: 1})
```

Ainda não temos uma live explicando o sistema de imports, vocês tem interesse?

Targets de casamento de padrões



Quando criamos a cláusula **case** na operação de **match** podemos criar novos nomes.

```
1 nome = 'Eduardo da Silva Sauro Jr.'
2
3 match nome.split():
4     case nome, sobrenome:
5         print(f'{nome=} {sobrenome=}')
6     case nome, nome_do_meio, sobrenome:
7         print(f'{nome=} {nome_do_meio=} {sobrenome=}')
8     case *nomes, 'Jr.' | 'Junior':
9         print(f'{nomes=} Jr.')
```

Já fizemos uma live sobre Pattern Matching: Live 171

Criação e parâmetros de tipo



Na PEP-695, agora existem definições e parâmetros de tipos, também são formas válidas de descrever um nome:

```
1  type Vetor = tuple[float, float]
2
3  def função_massa[T](args: list[T]) -> T: ...
4  # Tipo T genérico
```

Falaremos mais disso quando o python 3.12 chegar :heart:

Para onde os nomes
vão?

Name
spaces

O conceito de Namespace



Namespaces são lugares onde os nomes são armazenados durante a execução da nossa aplicação. Temos basicamente 4 namespaces:

- **L**ocal
- **E**nclosing
- **G**lobal
- **B**uilt-in (embutido)

Geralmente quando nos referimos aos 4 costumamos usar uma sigla **`LEGB`**.

Namespace global



Todos os nomes que definimos em nosso módulo, fazem parte do escopo **Global**. Por exemplo:

```
1  nome = 'Eduardo'
2  sobrenome = 'Mendes'
3
4  def minha_função_bacana(): ...
5
6  class ClasseShowDeBola: ...
7
8  for variável in [1, 2, 3]: ...
9
10 with open('arquivo.txt') as arquivo: ...
```

Acessando o escopo global



Se quisermos acessar as variáveis do escopo global, podemos usar uma função do python chamada **globals()**:

```
globals( )
```

Exemplo de resposta



Se entrarmos no terminal importando o nosso arquivo `python -i exemplo_00.py`, podemos fazer isso:

```
>>> from pprint import pprint

>>> pprint(globals())
{'ClasseShowDeBola': <class '__main__.ClasseShowDeBola'>,
 'arquivo': <_io.TextIOWrapper name='arquivo.txt' mode='r' encoding='UTF-8'>,
 'minha_função_bacana': <function minha_função_bacana at 0x7f1521398720>,
 'nome': 'Eduardo',
 'pprint': <function pprint at 0x7f15211cd940>,
 'sobrenome': 'Mendes',
 'variável': 3}
```


Assim, conseguimos ver todos os nomes definidos no namespace global.

Namespace local



Um pouco diferente no Global, o namespace local é referente a nomes definidos em blocos específicos de código. Um exemplo é em uma função. A função pode definir variáveis encapsuladas dentro de seu corpo. Esses nomes fazem parte do namespace local da função.

Podemos acessar os nomes definidos no namespace local usando a função **locals()**.

A dark-themed terminal window with a title bar containing a minus sign, a square icon, and a close 'X' icon. The text 'locals()' is displayed in a light blue monospace font.

```
locals( )
```

Namespaces e funções

```
def minha_função(a, b):  
    x = 7  
    z = 10  
    print('Namespace local da função:\n', locals())  
  
minha_função('primeiro argumento', 'segundo argumento')  
# Ecopo local da função:  
# {'a': 'primeiro argumento', 'b': 'segundo argumento', 'x': 7, 'z': 10}
```

Namespace embutido (Built-in)



Diferente dos namespaces em que são grupos os nomes definidos por nós, como global ou local.

O namespace embutido é referente aos nomes criados pelo próprio python, que podemos usar em nossa aplicação. Como as funções **sum()**, **len()**, **print()**, **type()**. Diferentes mensagens de erro, como **Exception**, **KeyError**, **ZeroDivisionError** e também valores padrões como **False**, **True** e **None**.

São os nomes que podemos usar sem precisar importar ou definir dentro da nossa aplicação.

Acessando os nomes embutidos



Quando pedimos o namespace global, o python nos respondeu alguns nomes que não definimos, um deles é o `__builtins__`. Existe uma função embutida chamada `dir()`, que serve para observarmos os atributos (nomes) contidos em outros objetos.

Podemos ver todos os nomes contidos no namespace embutido dessa forma:

```
>>> dir(__builtins__)  
# ['ArithmeticError', 'AssertionError',  
#   'AttributeError', 'BaseException' ...]
```

Namespace embutido



Podemos acessar todos esses valores importantes do módulo `builtins` também:

```
>>> import builtins
>>> dir(builtins)
# ['ArithmeticError', 'AssertionError',
#   'AttributeError', 'BaseException' ...]
```

Já já entraremos nesse detalhe...



Enclosing



Escopos

O lugar onde os
nomes são
definidos

O conceito de escopo



“O **escopo** de uma variável é a faixa de sentenças nas quais ela é visível. Uma variável é **visível** em uma sentença se ela pode ser referenciada ou atribuída nessa sentença.”

Sebesta

O Conceito de escopos



O conceito de escopo é relativo ao local do código em que uma variável é acessível. Por exemplo, em uma função "em seu namespace local" podemos acessar um nome que não foi definido em seu namespace local.

```
variável_global = 10

def função():
    variável_local = 20
    return sum([variável_global, variável_local])

função() # 30
```

Os escopos em ação



Nessa operação, de qual namespace o escopo pegará o nome `x`?

```
1  x = 10
2
3  def função( ):
4      x = 20
5      return x
6
7  função( ) # ???
```

Continua "óbvio"



```
1  x = 10
2
3  def função( ):
4      x = 20
5      def função_aninhada( ):
6          x = 30
7          return x
8      return função_aninhada( )
9
10 função( )  # ???
```

Mas e agora?

```
1  # Escopo global
2  x = 10
3
4  def função():
5      # Escopo Enclosing
6      x = 20
7      def função_aninhada():
8          # Escopo local
9          return x
10     return função_aninhada()
11
12 função()  # ???
```

O escopo tem acesso livre de acesso??



Todos os nomes de namespaces superiores podem ser acessados pelo escopo mais restrito. Porém, não podem modificar os valores associados a esses nomes:

```
1  x = 10
2
3  def função():
4      x += 1
5      return x
6
7  função()
```

UnboundLocalError



Quando tentamos modificar o valor reestricto a outro namespace, isso não é possível. O escopo é somente livre para acessar, não para modificar!

```
1 Cell In[1], line 7
2     4     x += 1
3     5     return x
4 ----> 7 função( )
5
6 Cell In[1], line 4, in função()
7     3 def função():
8 ----> 4     x += 1
9     5     return x
10
11 UnboundLocalError: cannot access local
    variable 'x' where it is not associated with
    a value
```

Alterando valores em nomes de outros namespaces



Se precisarmos fazer isso, temos que usar as palavras reservadas da linguagem `global` e `nonlocal`:

Devemos ter um pouco de cuidado com isso, ao poder gerar efeitos colaterais.

```
1  x = 10
2
3  def função():
4      global x
5      x += 1
6      return x
7
8  função() # 11
```

Enclosing



Quando estamos em um escopo onde a variável que queremos modificar não está nem local, nem global, no caso das funções aninhadas, devemos usar `nonlocal` para realizar essas modificações.

```
1  x = 10
2
3  def função():
4      x = 20
5      def função_aninhada():
6          nonlocal x
7          x += 1
8          return x
9      return função_aninhada()
10
11 função() # 21
```


Tudo que nasce,
morre

Ciclo
de
vida

except e case



Em casos específicos, os nomes só existem durante a execução do bloco.
Por exemplo, em uma exceção:

```
1  def exemplo():
2      try:
3          print(locals())
4          1 / 0
5      except Exception as e:
6          print(locals())
7
8      print(locals())
9
10 exemplo()
```

A variável `e` só existe dentro do except



```
1  def exemplo():
2      try:
3          print(locals())
4          1 / 0
5      except Exception as e:
6          print(locals())
7
8      print(locals())
9
10  exe
```

```
{  
  'e': ZeroDivisionError('division by zero')  
}
```

0 caso do case



```
1  def exemplo(parametro):
2      match parametro.split():
3          case nome, sobrenome:
4              print(locals())
5          case nome:
6              print(locals())
7      print(locals())
8
9  exemplo('eduardo')
10 exemplo('eduardo mendes')
```

0 caso do case



No caso do `case`, assim como o `for` e `with`, os nomes ficam disponíveis no escopo:

```
1 def exemplo(parametro):
2     match parametro.split():
3         case nome, sobrenome:
4             print(locals())
5         case nome:
6             print(locals())
7
8     {'parametro': 'eduardo', 'nome': ['eduardo']}
9     {'parametro': 'eduardo', 'nome': ['eduardo']}
10    {'parametro': 'eduardo mendes', 'nome': 'eduardo', 'sobrenome': 'mendes'}
    {'parametro': 'eduardo mendes', 'nome': 'eduardo', 'sobrenome': 'mendes'}
```

O caso de compreensão



Quando criamos uma list comp ou algo do gênero, sua variável também é limitada ao seu escopo:

```
[x for x in [1, 2, 3, 4]]  
print(locals()) # x não estará presente
```



picpay.me/dunossauro



apoia.se/livedepython



pix.dunossauro@gmail.com



Ajude o projeto <3



Referências



- <https://docs.python.org/3.12/reference/executionmodel.html>
- Conceitos de linguagens de programação — Robert W. Sebesta (11a ed.)