



Sistema de imports

Live de Python #269



1. Namespaces

Um passo pra trás

2. Imports I

Como os imports são feitos?

3. Módulos

Parte crucial, embora não veremos tudo

4. Imports II

Um pouco mais do baixo nível da operação



apoia.se/livedepython



pix.dunossauro@gmail.com



patreon.com/dunossauro



Ajude o projeto <3



Ademar Peixoto, Adriano Casimiro, Alexandre Harano, Alexandre Lima, Alexandre Silva, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Allan Kleitson, Alysson Oliveira, Andre Paula, Antônio Filho, Apc 16, Aslay Clevisson, Aurelio Costa, Bernarducs, Bruno420g, Bruno Almeida, Bruno Barcellos, Bruno Batista, Bruno Freitas, Bruno Ramos, Bruno Santos, Caio Nascimento, Carlos Ramos, Carlos Roberto, Christian Fischer, Clara Battesini, Controlado, Cristian Firmino, Daniel Aguiar, Daniel Bianchi, Daniel Real, Daniel Wojcickoski, Danillo Ferreira, Danilo Boas, Danilo Silva, David Couto, David Kwast, Dead Milkman, Denis Bernardo, Dgeison Peixoto, Diego Guimarães, Dino, Edgar, Eduard0ml, Elton Guilherme, Emerson Rafael, Ennio Ferreira, Erick Andrade, Érico Andrei, Everton Silva, Fabio Barros, Fábio Barros, Fabiokleis, Fabio Valente, Fabricio Biazotto, Felipe Augusto, Felipe Rodrigues, Fernanda Prado, Fernandocelmer, Filipe Monteiro, Frederico Damian, Fsouza, Gabrieimoreira, Gabriel Espindola, Gabriel Mizuno, Gabriel Paiva, Gabriel Ramos, Gabriel Simonetto, Geilton Cruz, Giovanna Teodoro, Giuliano Silva, Graziela Pauluka, Guibeira, Guilherme, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Guilherme Silva, Gustavo Suto, Haelmo Almeida, Harold Gautschi, Heitor Fernandes, Helvio Rezende, Henrique Andrade, Henrique Machado, Henrique Sebastião, Hiago Couto, Igor Taconi, Jairo Lenfers, Janael Pinheiro, Jean Victor, Jefferson Antunes, Jlx, Joao Rocha, Joelson Sartori, Jonatas Leon, Jônatas Silva, Jorge Silva, Jose Barroso, Jose Edmario, José Gomes, Joseito Júnior, Jose Mazolini, Jose Terra, Josir Gomes, Jrborba, Juan Felipe, Juliana Machado, Julio Batista-silva, Julio Franco, Júlio Gazeta, Júlio Sanchez, Kaio Peixoto, Leandro Silva, Leandro Vieira, Leonan Ferreira, Leonardo Mello, Leonardo Nazareth, Lucas Carderelli, Lucas Lattari, Lucas Martins, Lucas Mello, Lucas Mendes, Lucas Nascimento, Lucas Schneider, Lucas Simon, Luciano Filho, Luciano Ratamero, Luciano Teixeira, Luis Leite, Luiz Carlos, Luiz Duarte, Luiz Lima, Luiz Paula, Mackilem Laan, Marcelo Araujo, Marcelo Fonseca, Marcelo Silva, Marcio Freitas, Marcio Novaes, Marcio Silva, Marco Gandra, Marco Mello, Marcos Almeida, Marcos Gomes, Marcos Oliveira, Marina Passos, Mateus Lisboa, Matheus Silva, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Murilo Viana, Natália Araújo, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Gomes, Pedro Henrique, Peterson Santos, Philipe Vasconcellos, Phmmdev, Pytonyc, Rafael, Rafael Araújo, Rafael Faccio, Rafael Lino, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Renato José, Renato Moraes, Rene Pessoto, Renne Rocha, Ricardo Combat, Ricardo Silva, Ricardo Viana, Richard Costa, Rinaldo Magalhaes, Rjribeiro, Rodrigo Barretos, Rodrigo Oliveira, Rodrigo Quiles, Rodrigo Santana, Rodrigo Vaccari, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Selmison Miranda, Shinodinho, Téó Calvo, Terramotta, Thiago Araujo, Thiago Borges, Thiago Lucca, Thiago Paiva, Tiago Ferreira, Tiago Henrique, Tony Dias, Tyrone Damasceno, Valdir, Valdir Tegen, Varlei Menconi, Viniciusccosta, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vitor Silva, Vladimir Lemos, Wagner Gabriel, Washington Teixeira, Williangl, Willian Lopes, Wilson Duarte, Zeca Figueiredo, Zecarlos Junior



Obrigado você



O sistema de imports



Se você me perguntar qual a parte mais complicada e mal compreendida da linguagem, certamente te direi que é o sistema de imports. Por muitas vezes nos pegamos estudando diversas partes da linguagem, porém, não me recordo de nenhum caso onde alguém efetivamente me disse que estava estudando como os imports funcionavam no python. Usamos eles a todo momento, porém, raramente o estudamos a fundo.

Sistema de imports



Isso acaba acarretando diversos cenários que não são necessários:

1. Colocamos **`__init__.py`** em todos os pacotes. Precisa?
2. **`ModuleNotFoundError`**: O que isso significa?
3. Casos de **imports circulares**
4. Adicionamos novos caminhos no **`sys.path`**
5. Fazemos imports **relativos**? Ou imports **absolutos**?

O escopo dessa live!



Estamos aqui hoje!



live 238



Precisamos de uma live?

Não tenho intenção de
cobrir **todas** as nuances do
sistema de imports.

Mas me proponho a fazer outra live aprofundando especificamente nas entranhas
importações. Querem?



Disclaimer



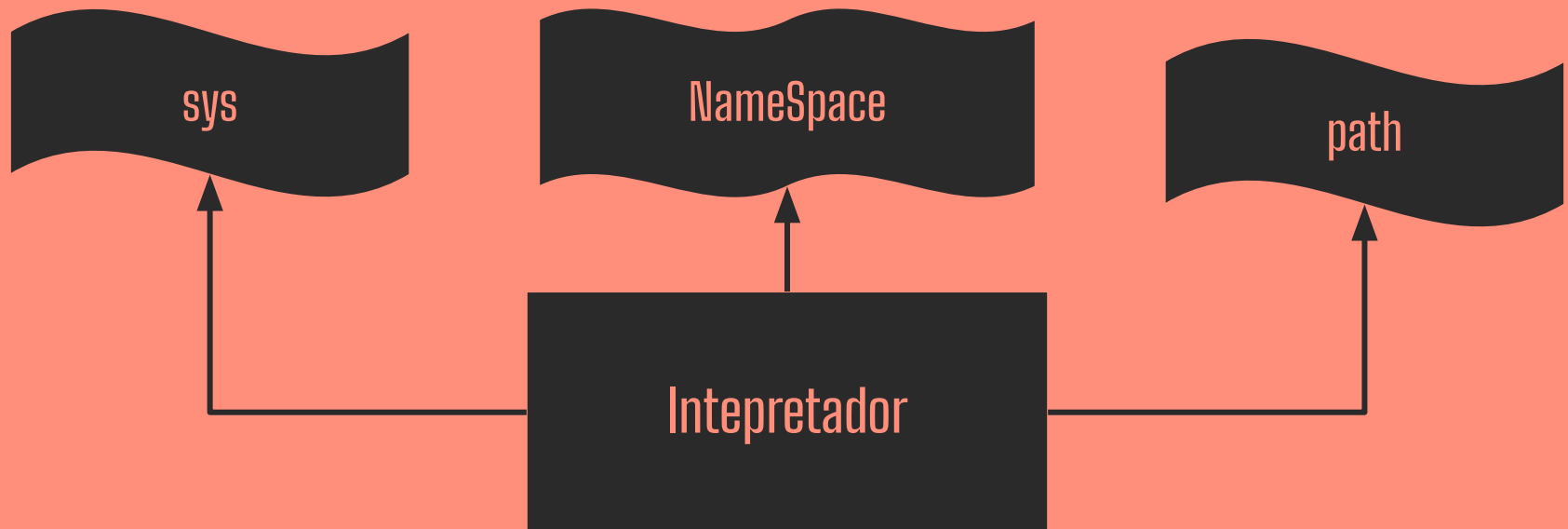
Um passos para
trás

Name
spaces

Tudo são detalhes



Quando o interpretador é iniciado, diversas configurações são iniciadas por padrão CPython 3.13.0b1 (para restringir nosso escopo)



Você digita Python no shell e ganha +- isso.

Namespaces



É o espaço dos "nomes", onde todas as "variáveis" tem uma referência no interpretador.

```
>>> globals()  
{'__annotations__': {},  
  '__builtins__': <module 'builtins' (built-in)>,  
  '__cached__': None,  
  '__doc__': None,  
  '__file__': '/home/dunossauro/Live269/aqui.py',  
  '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x732b2bc91f30>,  
  '__name__': '__main__',  
  '__package__': None,  
  '__spec__': None}
```


0 mundo dos nomes



Namespace



```
a = 1  
b = 2  
c = 3  
d = 4
```

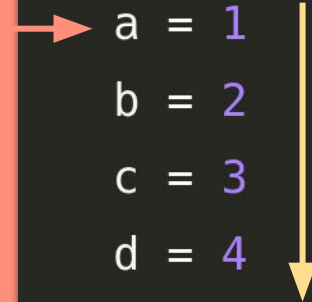
A yellow arrow pointing downwards, indicating the flow of execution or the sequence of variable assignments.

0 mundo dos nomes



Namespace

a:1

A diagram illustrating the relationship between a namespace and a code block. On the left, a dark blue square contains the text 'a:1'. A red arrow points from this square to a code block on the right. The code block is a dark blue rectangle with a light blue border and a title bar containing a minus sign, a square icon, and a close 'X' icon. Inside the code block, the following code is written:

```
a = 1  
b = 2  
c = 3  
d = 4
```

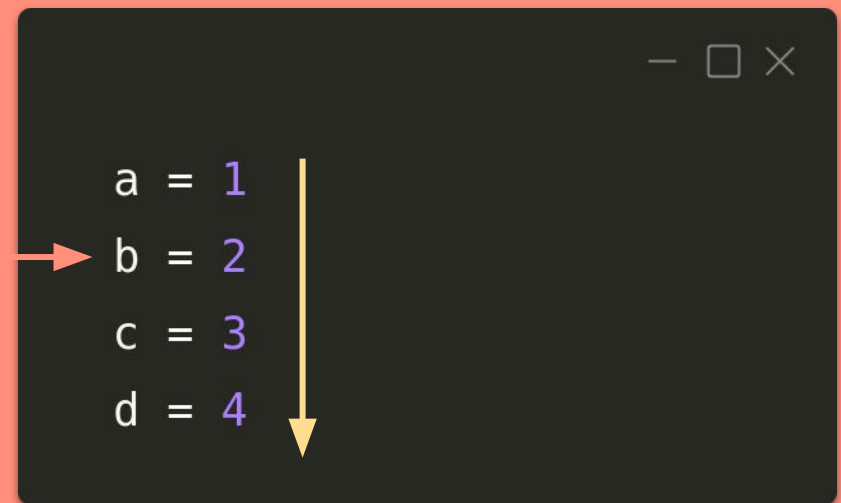
A yellow arrow points from the right side of the code block down to the bottom of the slide.

O mundo dos nomes



Namespace

a:1, b:2

A dark-themed code editor window with standard window controls (minimize, maximize, close) in the top right corner. It contains four lines of Python code: 'a = 1', 'b = 2', 'c = 3', and 'd = 4'. A red arrow points from the left towards the line 'b = 2'. A yellow arrow points downwards from the line 'a = 1' to the line 'd = 4'.

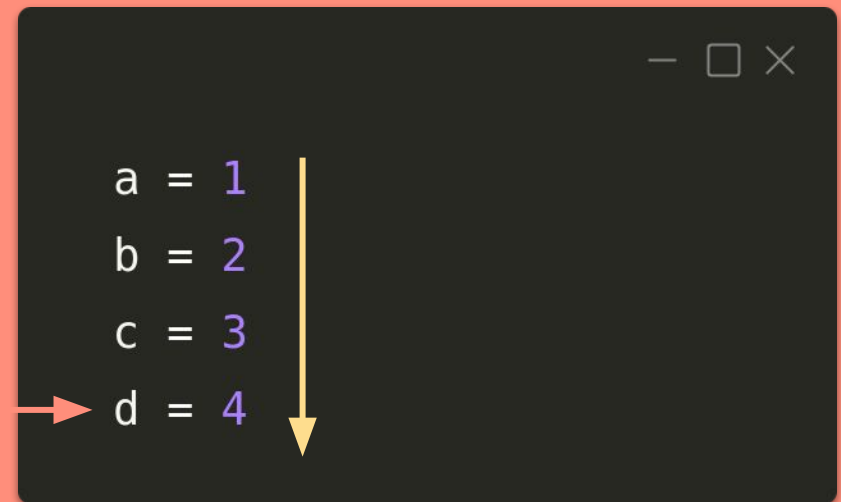
```
a = 1  
b = 2  
c = 3  
d = 4
```

0 mundo dos nomes



Namespace

a:1, b:2,
c:3,d:4

A dark-themed code editor window with a title bar containing a minus sign, a square icon, and a close 'X' button. It contains four lines of Python code: 'a = 1', 'b = 2', 'c = 3', and 'd = 4'. The numbers 1, 2, 3, and 4 are highlighted in blue. A red arrow points to the variable 'd' on the fourth line, and a yellow arrow points downwards from the right side of the code block.

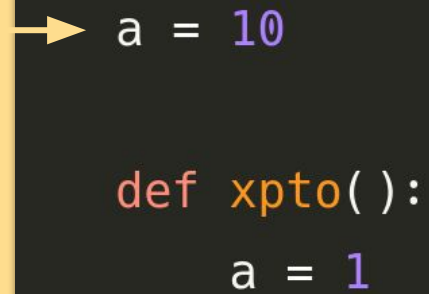
```
a = 1  
b = 2  
c = 3  
d = 4
```

Global vs Local



Namespace global

a:10



```
a = 10

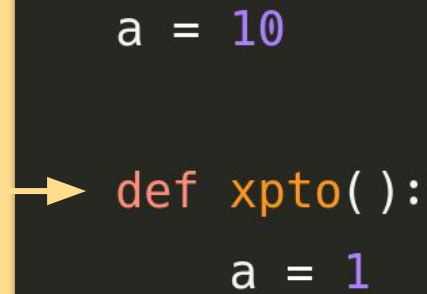
def xpto():
    a = 1
```


Global vs Local



Namespace global

a:10,
xpto:func



```
a = 10  
→ def xpto():  
    a = 1
```

Global vs Local



Namespace global

a:10,
xpto:func

Namespace local (xpto)

a:1



```
a = 10  
  
def xpto():  
    a = 1  
  
xpto()
```

— □ ×

```
a = 10
```

```
def xpto():  
    a = 1  
    print(locals())  
    print(globals())
```

xpto()

— □ ×

```
{'a': 1}  
{'__name__': '__main__',  
  '__doc__': None,  
  '__package__': None,  
  '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7b2b7cfa5f30>,  
  '__spec__': None,  
  '__annotations__': {},  
  '__builtins__': <module 'builtins' (built-in)>,  
  '__file__': '/home/dunossauro/Live269/aqui.py',  
  '__cached__': None,  
  'a': 10,  
  'xpto': <function xpto at 0x7b2b7ce1f1a0>}
```

O sistema de busca de nomes



```
$ python -m dis aqui.py
```

```
0          RESUME          0

1          LOAD_NAME        0 (sum) # <--- A instrução que interessa
          PUSH_NULL
          BUILD_LIST         0
          LOAD_CONST         0 ((1, 2, 3))
          LIST_EXTEND        1
          CALL                1
          POP_TOP
          RETURN_CONST        1 (None)
```

```
# aqui.py
sum([1, 2, 3])
```

https://github.com/python/cpython/blob/d25954dff5409c8926d2a4053d3e892462f8b8b5/Python/generated_cases.c.h#L4575



```
{'a': 1}
{'__name__': '__main__',
 '__doc__': None,
 '__package__': None,
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x7b2b7cfa5f30>,
 '__spec__': None,
 '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__file__': '/home/dunossauro/Live269/aqui.py',
 '__cached__': None,
 'a': 10,
 'xpto': <function xpto at 0x7b2b7ce1f1a0>}
```

Módulo, take 1



módulo

Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um espaço de nomes contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de importação.

<https://docs.python.org/pt-br/3/glossary.html#term-module>

Módulo, take 1



```
>>> globals()['__builtins__']  
<module 'builtins' (built-in)>  
  
>>> type(globals()['__builtins__'])  
<class 'module'>
```

Módulo, o namespace



```
>>> globals()['__builtins__'].__dict__
{
  # Erros
  'ArithmeticError': <class 'ArithmeticError'>,
  'TypeError': <class 'TypeError'>,
  # Constantes
  'Ellipsis': Ellipsis,
  'False': False,
  # Atributos de módulo
  '__name__': 'builtins',
  '__package__': '',
  '__spec__': ModuleSpec(name='builtins', loader=<class '_frozen_importlib.BuiltinImporter'>, origin='built-in'),
  # Funções builtin
  'abs': <built-in function abs>,
  'zip': <class 'zip'>
}
```

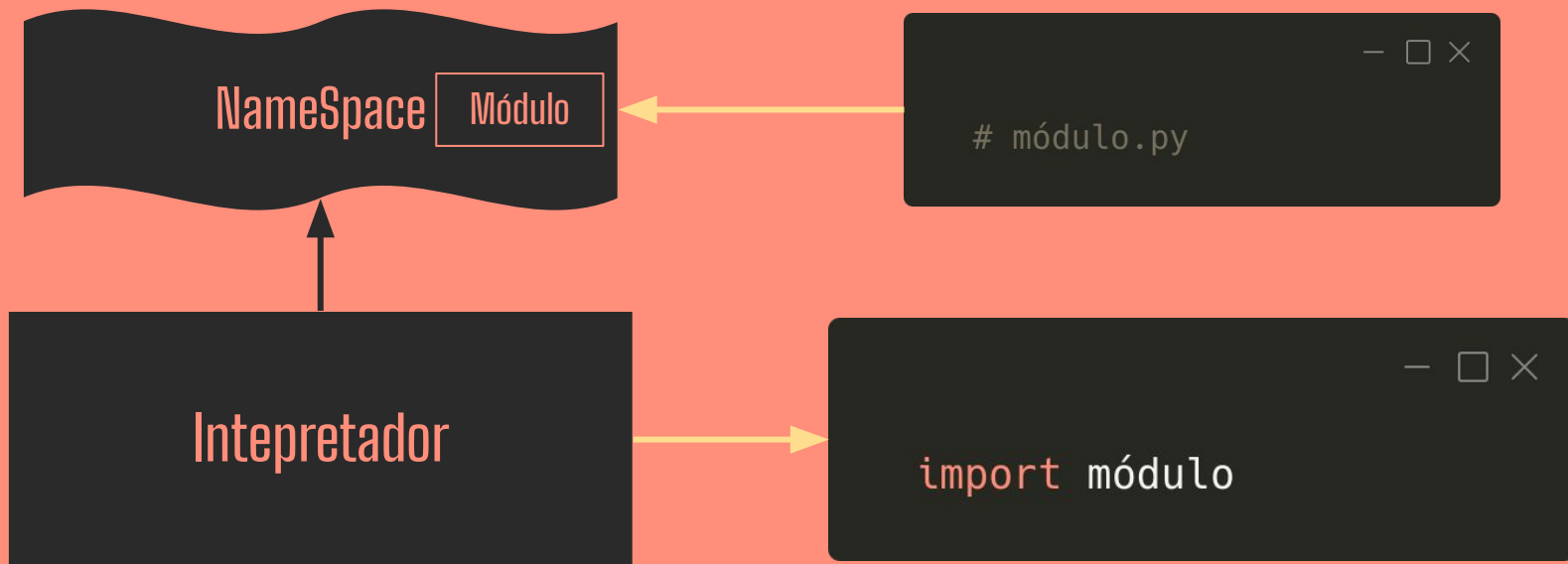

O básico necessário

Imports
I

A instrução de import



Basicamente, a instrução de import é sobre "trazer" algo que não está no namespace local, para o namespace local



Imports vs Namespaces



```
import functools
print(globals())
```

```
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': '/home/dunossauro/Live269/aqui.py',
 '__loader__': <_frozen_importlib_external.SourceFileLoader object at 0x721564ba5f30>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'functools': <module 'functools' from '/.pyenv/versions/3.13.0a6/lib/python3.13/functools.py'>}
```

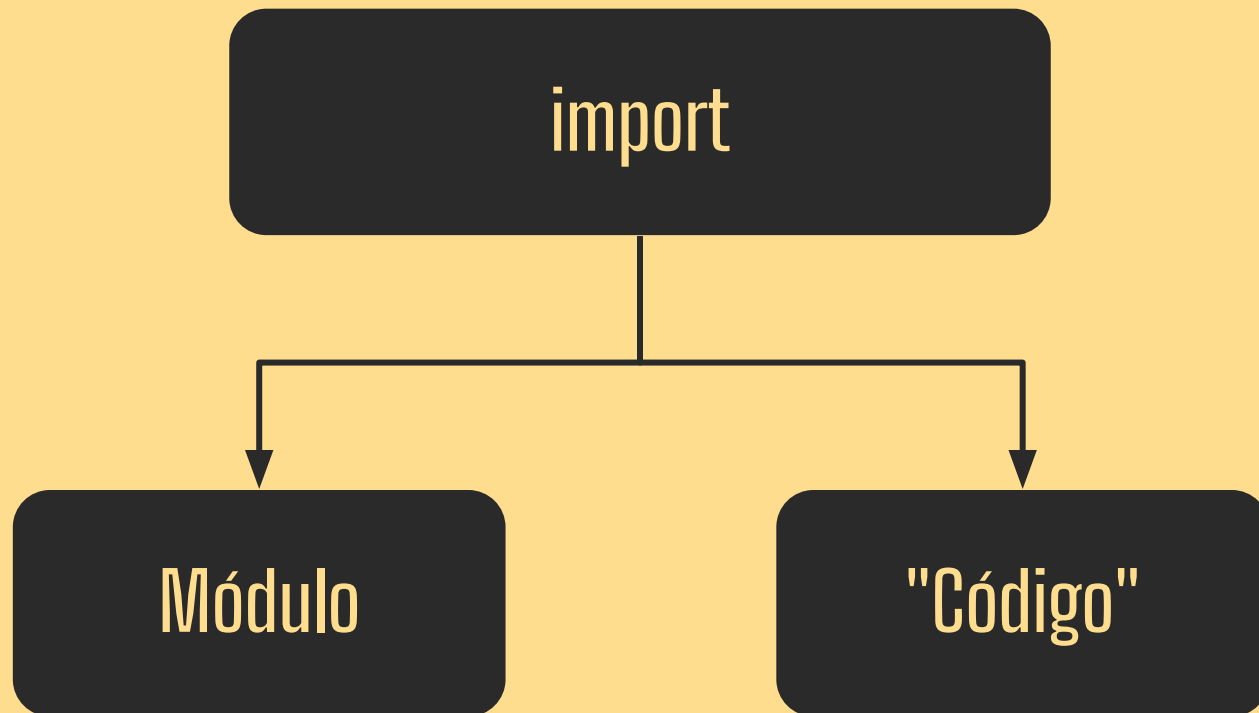
from A import B



A instrução **from** importa um "pedaço", um objeto, do namespace do pacote. Por exemplo:

```
>>> from functools import cache
>>> globals()
{'__name__': '__main__',
 '__doc__': None, '__package__': None,
 '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
 '__spec__': None, '__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 'cache': <function cache at 0x788fe6a41120>}
```

Porém, nem sempre



Módulos podem ter submódulos



```
>>> import urllib
>>> from urllib import request
>>> from urllib.request import urlopen
>>> pprint(globals())
{
    ...
    'request': <module 'urllib.request' from '/3.13.0b1/lib/python3.13/urllib/request.py'>,
    'urllib': <module 'urllib' from '/3.13.0b1/lib/python3.13/urllib/__init__.py'>,
    'urlopen': <function urlopen at 0x7fa1e8dfaa20>
}
```

Outra forma de importar



Essa forma que vimos agora, é a forma "estática" de importação.

Mas também podemos usar a **importlib**, que provém mecanismos dinâmicos de importação

```
>>> import importlib
>>> importlib.import_module('aqui')
<module 'aqui' from '/home/dunossauro/Live269/aqui.py'>
```

Módulos

0 que são?
Do que se alimentam?

Mas afinal, o que são módulos?



Basicamente o python reconhece 6 coisas como módulos:

1. **Arquivos .py**: arquivos padrão
2. **Arquivos .pyc**: bytecode
3. **Diretórios**: pastas no sistema
4. **Extensões em C**: .so por exemplo
5. **Builtins**: módulos escritos em C e disponíveis no interpretador
6. **Frozen**: módulos escritos em python "compactados" no interpretador

Começando pelo começo (.py)



```
.  
└─ aqui.py  # <- Nosso arquivo  
└─ modulo.py  # <- o que queremos
```

Começando pelo começo (.py)



```
.  
├─ aqui.py # <- Nosso arquivo  
└─ modulo.py # <- o que queremos
```

```
# modulo.py  
def soma(x, y):  
    return x + y
```

```
# aqui.py  
import modulo  
modulo.soma(1, 2) # 3
```

Module



```
— □ ×  
  
>>> import functools  
>>> type(functools)  
<class 'module'>
```

`.pyc`



Os bytecodes também são reconhecidos como módulos.

```
python -m compileall modulo.py
```

```
Compiling 'modulo.py'...
```

```
python -i __pycache__/modulo.cpython-313.pyc
```

```
>>> soma
```

```
<function soma at 0x7dab8a48f6a0>
```

Extensões C



Extensões C, como .so podem ser importadas diretamente pelo sistema de imports.

Um exemplo de um módulo compilado com **mypyc**:

```
mypyc pacote_em_c.py
...
copying build/lib.linux-x86_64-cpython-312/pacote_em_c.cpython-312-x86_64-linux-gnu.so -> .

rm pacote_em_c.py

python -c "import pacote_em_c"
Fui criado pelo mypyc!
```

Pacotes



Podemos associar pacotes em python com diretórios que contém código python. Uma "pasta" no sistema.

Existem dois tipos de pacotes:

- Pacotes **tradicionais**: Os que tem um arquivo `__init__.py`
- Pacotes de **namespaces**: Os que **não tem** um arquivo `__init__.py`

```
.
├── aqui.py
├── pacote
│   ├── __init__.py
│   └── modulo.py
└── pacote_namespace
    └── modulo.py
```

Precisa criar `__init__.py` em todos os diretórios?

Não



Respondendo à pergunta!



Pacotes tradicionais



```
parent/  
    __init__.py  
one/  
    __init__.py  
two/  
    __init__.py  
three/  
    __init__.py
```

O `__init__.py` é um arquivo que será executado **SEMPRE** que o módulo for importado. Uma espécie de **hook**.

Se você precisar disso, crie!

Existem diversas coisas interessantes de se fazer com pacotes, mas hoje não é o dia!

Vale uma live sobre módulos?

Imports relativos



0 motivo de
estarmos aqui!

Imports

||

Mas como? `__import__`



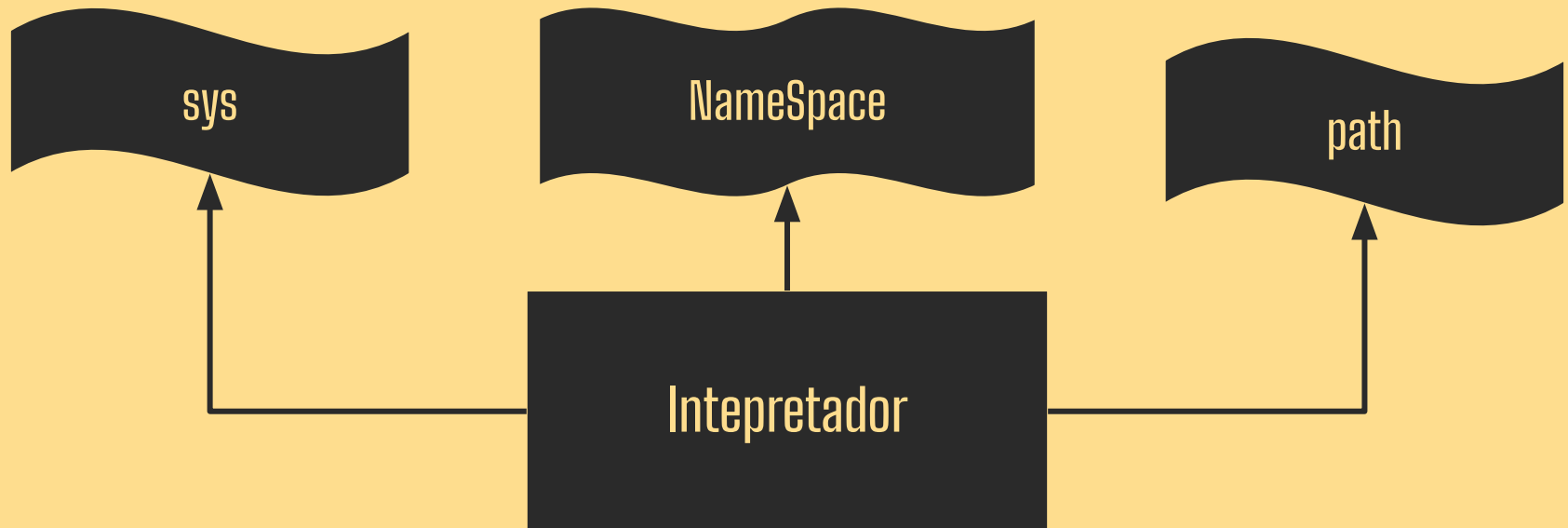
Basicamente, quando chamamos a instrução `import x` o python faz uma chamada na função `__import__` e adiciona o módulo importado no namespace local. Partindo disso, `LOAD_NAME`, pode ser chamado.

```
>>> globals()['functools'] = __import__('functools')
>>> functools
<module 'functools' from 'path/python3.13/functools.py'>
>>> functools.cache
<function cache at 0x7a653f938400>
```

Mas como o `__import__` importa?



Lembra da configuração padrão do interpretador?





O módulo sys tem um mapa dos módulos que já foram carregados durante a fase inicialização do interpretador [chamamos **bootstrap**].

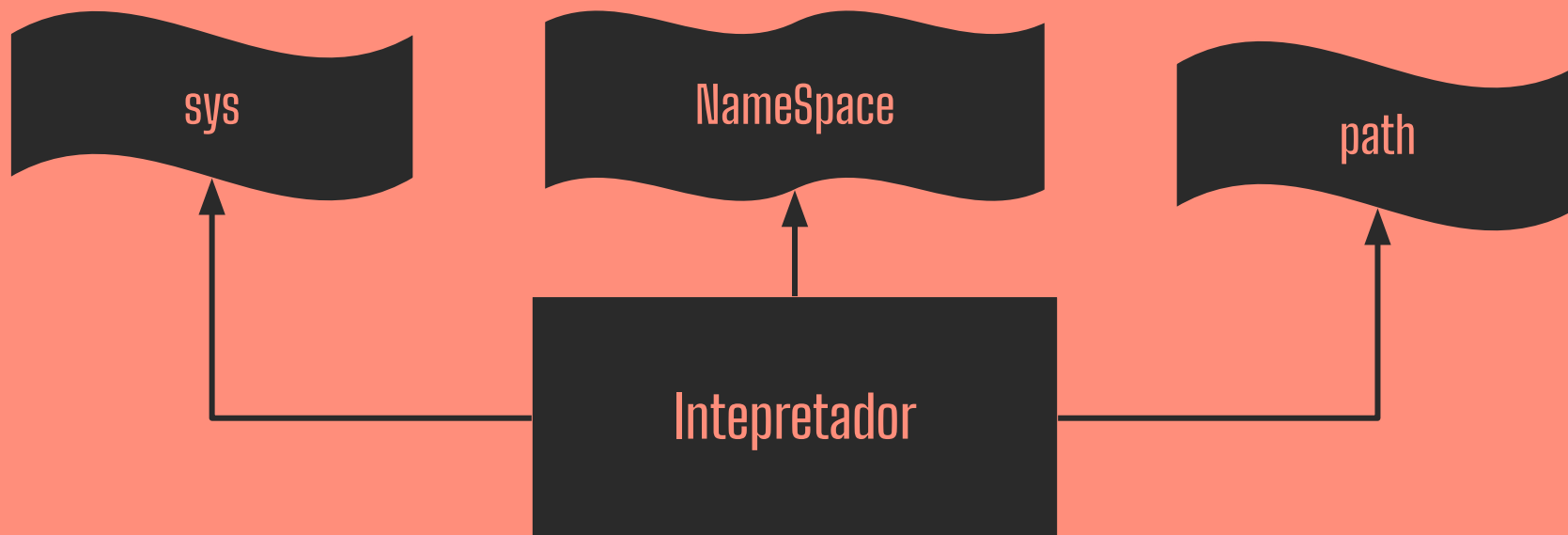
```
>>> sys.modules
```

Caso o que você queira esteja aqui, ele **copia o módulo para o namespace**

Mas e se não estiver em `sys.modules`?



Lembra da configuração padrão do interpretador?



Finders e Loaders / Localizadores e carregadores



A instrução `__import__` usa um conceito de **localizador** (finder) para encontrar os pacotes.

Por padrão, é usada variável **PYTHONPATH** ou o **local o interpretador é iniciado**.

Isso cria uma nova variável durante o bootstrap. **sys.path**:

```
>>> sys.path
['', # <- PYTHONPATH ou onde o interpretador foi iniciado (os.getcwd)
 '/home/dunossauro/.pyenv/versions/3.13.0b1/lib/python313.zip',
 '/home/dunossauro/.pyenv/versions/3.13.0b1/lib/python3.13',
 '/home/dunossauro/.pyenv/versions/3.13.0b1/lib/python3.13/lib-dynload',
 '/home/dunossauro/.pyenv/versions/3.13.0b1/lib/python3.13/site-packages'
]
```

Finders



Para todos os diretórios em `sys.path` (e mais alguns), foram criados objetos "**localizadores**". Durante a fase de bootstrap, esses objetos são adicionados em **`sys.path_importer_cache`**

```
>>> print(sys.path_importer_cache)
{'/3.13.0b1/lib/python3.13': FileFinder('/3.13.0b1/lib/python3.13'),
 '/3.13.0b1/lib/python3.13/_pyrepl': FileFinder('/3.13.0b1/lib/python3.13/_pyrepl'),
 '/3.13.0b1/lib/python3.13/collections': FileFinder('/3.13.0b1/lib/python3.13/collections'),
 '/3.13.0b1/lib/python3.13/encodings': FileFinder('/3.13.0b1/lib/python3.13/encodings'),
 '/3.13.0b1/lib/python3.13/importlib': FileFinder('/3.13.0b1/lib/python3.13/importlib'),
 '/3.13.0b1/lib/python3.13/lib-dynload': FileFinder('/3.13.0b1/lib/python3.13/lib-dynload'),
 '/3.13.0b1/lib/python3.13/re': FileFinder('/3.13.0b1/lib/python3.13/re'),
 '/3.13.0b1/lib/python3.13/site-packages': FileFinder('/3.13.0b1/lib/python3.13/site-packages'),
 '/3.13.0b1/lib/python3.13/urllib': FileFinder('/3.13.0b1/lib/python3.13/urllib'),
 '/3.13.0b1/lib/python313.zip': None,
 '/home/dunossauro/Live269': FileFinder('/home/dunossauro/Live269')}
```

Finders



Finders são objetos que sabem como especificar um módulo.

Eles sabem "onde" estão, e qual o hook que precisa ser chamado para construir um módulo:

```
>>> finder = sys.path_importer_cache['/home/dunossauro/Live269']
>>> finder.path
'/home/dunossauro/Live269'
>>> finder.path_hook
<bound method FileFinder.path_hook of <class '_frozen_importlib_external.FileFinder'>>
>>> finder.find_spec('aqui')
ModuleSpec
```

ModuleSpec e Loaders



Especificações de módulo são objetos que sabem como ser "carregados" e seu caminho.

```
>>> finder.find_spec('aqui')
ModuleSpec(
  name='aqui',
  loader=<_frozen_importlib_external.SourceFileLoader object at 0x7ad363a7eb70>,
  origin='/home/dunossauro/Live269/aqui.py'
)
```

Loader



O loader é como o objeto se "cria" como um módulo

```
>>> import sys
>>> finder = sys.path_importer_cache['/home/dunossauro/Live269']
>>> spec = finder.find_spec('aqui')
>>> spec.loader.load_module()
<module 'aqui' from '/home/dunossauro/Live269/aqui.py'>
```

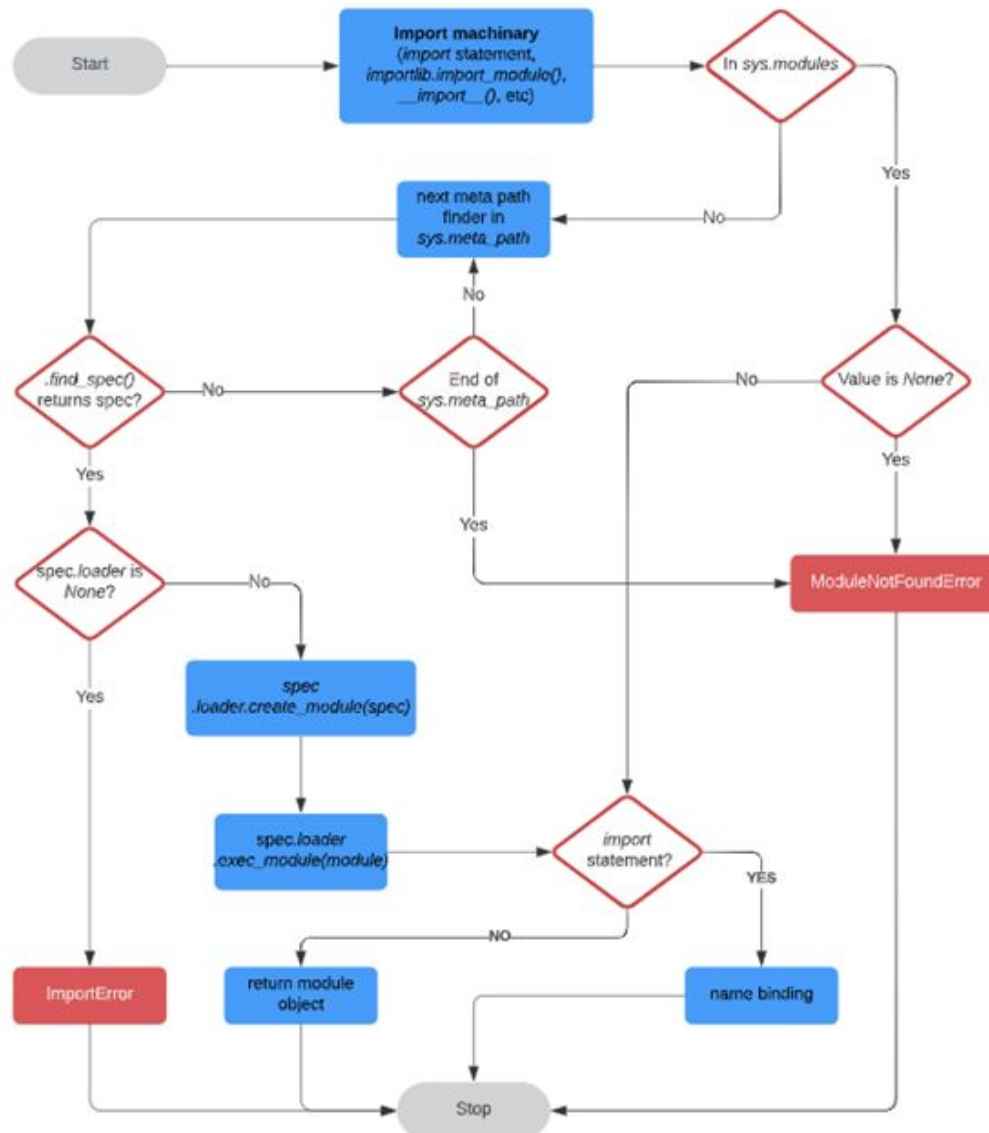
O passo após a criação



Depois que o módulo é criado, ele é adicionado em **sys.modules** para não fazer esse processo novamente.

O módulo é inserido no namespace local.

Python Import System



```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)

_init_module_attrs(spec, module)

if spec.loader is None:
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
    return sys.modules[spec.name]
```

<https://docs.python.org/pt-br/3/reference/import.html#loading>



apoia.se/livedepython



pix.dunossauro@gmail.com



patreon.com/dunossauro



Ajude o projeto <3

