



# Módulos e Pacotes

Live de Python # 287



## 1. O que precisamos saber antes?

Namespaces e Imports

## 2. Módulos

Eles também são objetos

## 3. Customização de acesso

PEP 562

## 4. Pacotes

Regulares e de namespaces



[apoia.se/livedepython](https://apoia.se/livedepython)



[pix.dunossauro@gmail.com](mailto:pix.dunossauro@gmail.com)



[patreon.com/dunossauro](https://patreon.com/dunossauro)



Ajude o projeto <3



Adriana Cavalcanti, Alexandre Harano, Alexandre Lima, Alexandre Takahashi, Alexandre Villares, Alfredo Braga, Alysso Oliveira, Andre Makoski, Andre Paula, Antonio Filho, Apc 16, Apolo Ferreira, Arthur Santiago, Aslay Clevisson, Augusto Domingos, Aurelio Costa, Bed, Belisa Arnhold, Bernardo Fontes, Bernarducs, Biancarosa, Brisa Nascimento, Bruno Batista, Bruno Bereoff, Bruno Freitas, Bruno Ramos, Bruno Russian, Brunu, Carlos Gonçalves, Celio Araujo, Christian Fischer, Claudemir Cruz, Cleiton Fonseca, Controlado, Curtos Treino, Daniel Aguiar, Daniel Bianchi, Daniel Brito, Daniel Souza, Daniel Wojcickoski, Danilo Boas, Danilo Silva, David Couto, David Kwast, Diego Guimarães, Edgar, Eduardo Pizorno, Elias Soares, Emerson Rafael, Érico Andrei, Everton Silva, Fabio Barros, Fábio Belotto, Fabio Faria, Fabiokleis, Felipe Adeildo, Felipe Augusto, Felipe Corrêa, Fernanda Prado, Fernandocelmer, Fichele Marias, Francisco Aclima, Frederico Damian, Fulvio Murenu, Gabriel Lira, Gabriel Mizuno, Gabriel Paiva, Gabriel Simonetto, Geilton Cruz, Geisler Dias, Giordane Oliveira, Giovanna Teodoro, Giuliano Silva, Guibeira, Guilherme Felitti, Guilherme Ostrock, Guilherme Piccioni, Gustavo Suto, Harold Gautschi, Heitor Fernandes, Hellyson Ferreira, Helton, Helvio Rezende, Henri Alves, Henrique Andrade, Henrique Machado, Henriquesebastiao, Herian Cavalcante, Hiago Couto, Igor Taconi, Ivan Santiago, Janael Pinheiro, Jean Melo, Jean Victor, Jeferson Vitalino, Jefferson Antunes, Jefferson Silva, Jerry Ubiratan, Jhonata Medeiros, Jlx, Joao Rocha, Jonas Araujo, Jonatas Leon, Joney Sousa, Jorge Silva, Jose Barroso, Jose Edmario, Joseíto Júnior, Jose Mazolini, José Predo), Josir Gomes, Jplay, Jrborba, Juan Felipe, Juliana Machado, Julio Franco, Julio Silva, Kaio Peixoto, Lara Nápoli, Leandro O., Leandro Pina, Leandro Vieira, Leonan, Leonardo Adelmo, Leonardo Mello, Leonardo Nazareth, Lilian Pires, Lisandro Pires, Lucas Carderelli, Lucas Castro, Lucas Mello, Lucas Mendes, Lucas Moura, Lucas Nascimento, Lucas Schneider, Luciano Ratamero, Luis Ottoni, Luiz Duarte, Luiz Martins, Luiz Paula, Luiz Perciliano, Marcelo Araujo, Marcelo Fonseca, Marcelo Grimberg, Marcio Freitas, Marcos Almeida, Marcos Oliveira, Maria Santos, Marina Passos, Mateusamorim96, Mateus Lisboa, Matheus Vian, Mírian Batista, Mlevi Lsantos, Murilo Carvalho, Naomi Lago, Ocimar Zolin, Otávio Carneiro, Patrick Felipe, Pedro Henrique, Peterson Santos, Phmmdev, Pytonyc, Rafael Ferreira, Rafael Fontenelle, Rafael Lopes, Rafael Romão, Raimundo Ramos, Ramayana Menezes, Renan, Renan Sebastião, Rene Pessoto, Renne Rocha, Ricardo Silva, Ricardo Viana, Richard Sousa, Rinaldo Magalhaes, Rodrigo Barretos, Rodrigo Santana, Rodrigo Vieira, Rogério Nogueira, Rui Jr, Samanta Cicilia, Santhiago Cristiano, Sergio Nascimento, Sherlock Holmes, Shirakawa, Tenorio, Téo Calvo, Tharles Andrade, Thiago Araujo, Thiago Lucca, Thiago Paiva, Tiago, Tiago Emanuel, Tomás Tamantini, Valdir, Varlei Menconi, Vinicius Meneses, Vinicius Silva, Vinicius Souza, Vinicius Stein, Vladimir Lemos, Williamslews, Willian Lopes, Zeca Figueiredo, Zero! Studio



Obrigado você



# O escopo dessa live!



Essa live não é sobre  
programação modular!



Um breve aviso



Essa live não é sobre programação modular!

Essa live não é sobre  
distribuição de pacotes  
(wheels, eggs, ...)



Outro aviso :)



Antes  
de  
tudo

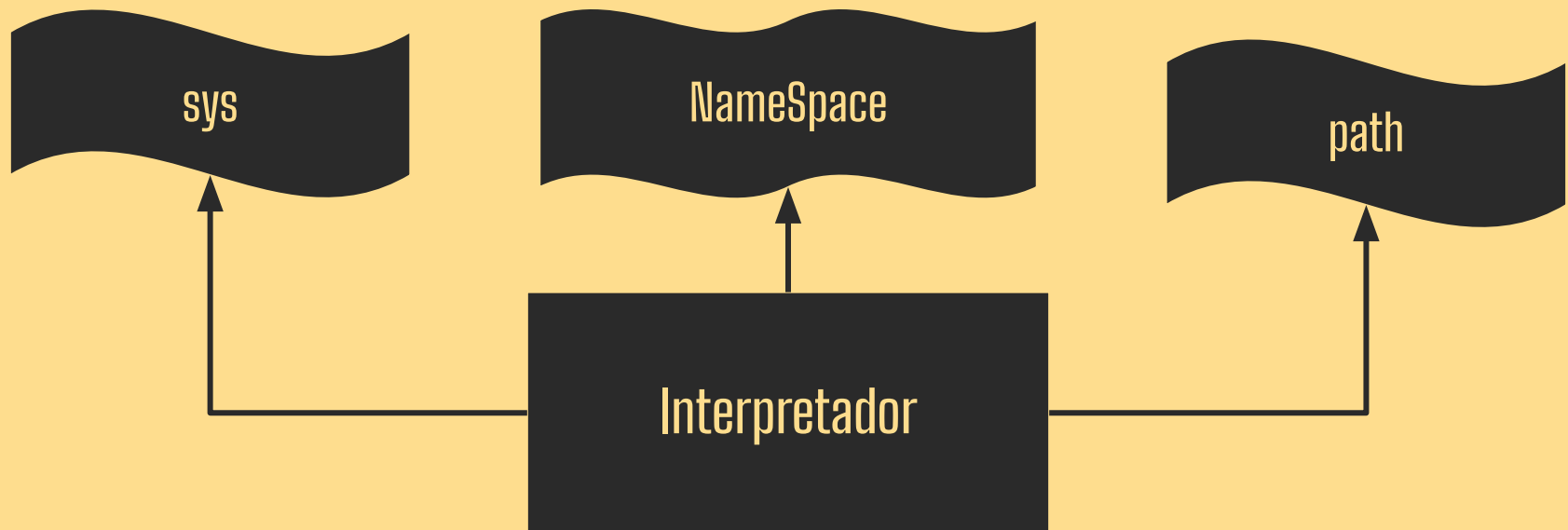
backoff



# Namespaces



Quando o interpretador é iniciado, diversas configurações são iniciadas por padrão



Você digita `python` no shell e tem isso

# O arquivo `.py` [xpto.py]



Todo o código que escrevemos, na maioria esmagadora dos casos, é segmentado em arquivos `.py`

```
CONSTANTE = 42
variável = ''
type T = int | float

def func(x: int):
    a = 10 # variável do escopo de func
    return a + x
```

# Namespaces



O resultado disso é um dicionário em memória, armazenado em `__dir__()`:

```
>>> import xpto
>>> xpto.__dir__()
[
    # Nomes presentes no módulo
    '__name__', '__doc__', '__package__', '__loader__', '__spec__', '...'
    # Nomes que definimos
    'CONSTANTE', 'variável', 'T', 'func'
]
```

# Escopos



Podemos ver que **a** é um nome que não aparece nesse escopo, pois ele pertence a outro escopo.

O escopo local da função **xpto**



Já falamos sobre isso em: live 238

# Imports



Sempre que importamos um arquivo, um objeto de módulo é criado

```
>>> import xpto  
>>> type(xpto)  
<class 'module'>
```

# Finders / Loaders



O sistema de import vai buscar o módulo nos paths em **sys.path** e usar os **finders** para reconhecer os módulos.

Caso ele encontre, vai usar a classe de **Loader** associada ao tipo de módulo e fazer o carregamento.

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

# Finders / Loaders



O sistema de import vai buscar o módulo nos paths em **sys.path** e usar os **finders** para reconhecer os módulos.

Caso ele encontre, vai usar a classe de **Loader** associada ao tipo de módulo e fazer o carregamento.



Já falamos sobre isso em: live 269

O básico necessário

Mód  
ulos



# O que são módulos



Como tudo, ~~ou quase tudo~~, módulos também são objetos. Você pode criar um módulo em runtime usando **type.ModuleType**:

```
>>> from types import ModuleType
>>> modulo = ModuleType('Módulo')

>>> type(modulo)
<class 'module'>

>>> modulo.__name__
'Módulo'
```

# O que o python vê como módulos?



Basicamente, o sistema de imports transforma qualquer dessas coisas em um objeto de módulo:

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

Falamos sobre isso na live de imports :)

# De volta ao básico



```
CONSTANTE = 42
variável = ''
type T = int | float

def func(x: int):
    a = 10 # variável do escopo de func
    return a + x
```

# De volta ao básico



```
CONSTANTE = 42
```

```
variável = ''
```

```
type T = int
```

```
def func(x):
```

```
    a = 10
```

```
    return a
```

```
>>> import xpto
```

```
>>> type(xpto)
```

```
<class 'module'>
```

# Especificação do módulo



Todo objeto de módulo é contemplado por uma especificação  
**ModuleSpec:**

```
>>> xpto.__spec__
ModuleSpec(
  # __name__
  name='xpto',
  # __loader__
  loader=<_frozen_importlib_external.SourceFileLoader object at 0x7a11fda5cdd0>,
  # __file__
  origin='/home/dunossauro/live_modulos_e_pacotes/xpto.py'
)
```

# Dunders do módulo



E o spec nos dá algumas variáveis que serão preenchidas no namespace:

- **\_\_name\_\_**: Nome do módulo, geralmente o nome do arquivo sem o .py
- **\_\_package\_\_**: Identifica se o módulo é um pacote
- **\_\_loader\_\_**: Quem é o loader desse pacote
- **\_\_file\_\_**: Nome absoluto do arquivo
- **\_\_doc\_\_**: Que é criado vazio caso o módulo não tem uma docstring

# Dunders de módulo que definimos



Alguns nomes **especiais** podem ser usados como metadados em módulos, como:

- **\_\_author\_\_**: Nome que diz de quem é a autoria do módulo
- **\_\_version\_\_**: Usado para versionamento dinâmico
- **\_\_all\_\_**: O que será importado via `*star*` (from modulo **import \***)

Incrivelmente, embora muito comuns, nenhum desses metadados aparece na documentação do python. Mas todos são citados na PEP 8.

## Autoria do módulo / pacote



Uma variável de metadado. Diz a quem interessar que quem escreveu o módulo foi o `__author__`.

Não existe nenhuma formatação recomendada. É free-style!

```
__author__ = 'Fausto, o Mago!'
```



# Versionamento do pacote



O **\_\_version\_\_** costuma ser usado no contexto de bibliotecas, ele pode determinar a versão de um pacote dinamicamente:

```
__version__ = '0.4.7b3'
```

# Versionamento do pacote



O **\_\_version\_\_** costuma ser usado no contexto de bibliotecas, ele pode determinar o número da versão de uma biblioteca. Vamos ver como fazer isso mais tarde:

Voltaremos nisso  
mais tarde!

```
__version__ = '0.4.7b3'
```

# \_\_all\_\_



Embora com algumas aparições na documentação, mas não nos módulos, nem nos imports... A ideia do `__all__` é mostrar quais nomes são públicos e quais nomes são privados em um módulo.

"Se `__all__` não estiver definido, o conjunto de nomes públicos inclui todos os nomes encontrados no espaço de nomes do módulo que não começam com um caractere sublinhado ('\_')."

[https://docs.python.org/pt-br/3.13/reference/simple\\_stmts.html#the-import-statement](https://docs.python.org/pt-br/3.13/reference/simple_stmts.html#the-import-statement)

## \_\_all\_\_



Ou seja, qualquer nome fora do **\_\_all\_\_**, caso ele exista, os nomes fora do all são **considerados** privados! Mas... isso é uma convenção. Tudo continua sendo acessível...

```
def x(): ...  
def y(): ...  
  
__all__ = ['x']
```

# \_\_all\_\_



Ou seja, qualquer nome fora do **\_\_all\_\_**, caso ele exista, os nomes fora do all são **considerados** privados! Mas... isso **é uma convenção**. Tudo continua sendo acessível...

```
CONSTANTE = 42
variável = ''
type T = int | float
```

```
def func(x: int):
    a = 10 # variável do escopo de func
    return a + x
```

```
__all__ = ['variável']
```

```
>>> dir(xpto)
['CONSTANTE', 'T', 'func', 'variável']
```

# Import start (\*)



O único caso real em que isso é levado em consideração "real" é no **import \***. Onde, só virão para o namespace atual os nomes listados em **\_\_all\_\_**

```
def x(): ...
```

```
def y(): ...
```

```
__all__ = ['x']
```

```
from modulo import *
```

# Docstrings



Outro metadado bastante comum em módulos são as docstrings. Armazenados em **`__doc__`**.

```
"""Docstring de módulo."""
```

```
CONSTANTE = 42
```

```
variável = ''
```

```
type T = int | float
```

```
def func(x: int):
```

```
    a = 10 # variável do es
```

```
    return a + x
```

```
__all__ = ['variável']
```

```
>>> xpto.__doc__  
'Docstring de módulo.'
```

# Módulos podem ser recarregados em runtime!



Uma das características mais legais de trabalhar com módulos (usando programação modular) é poder reescrever e fazer reload em módulos.

```
>>> import xpto
>>> xpto.__doc__
'Docstring de módulo. v1'

# alterando o módulo

>>> from importlib import reload
>>> reload(xpto)
<module 'xpto' from '../live_modulos_e_pacotes/xpto.py'>

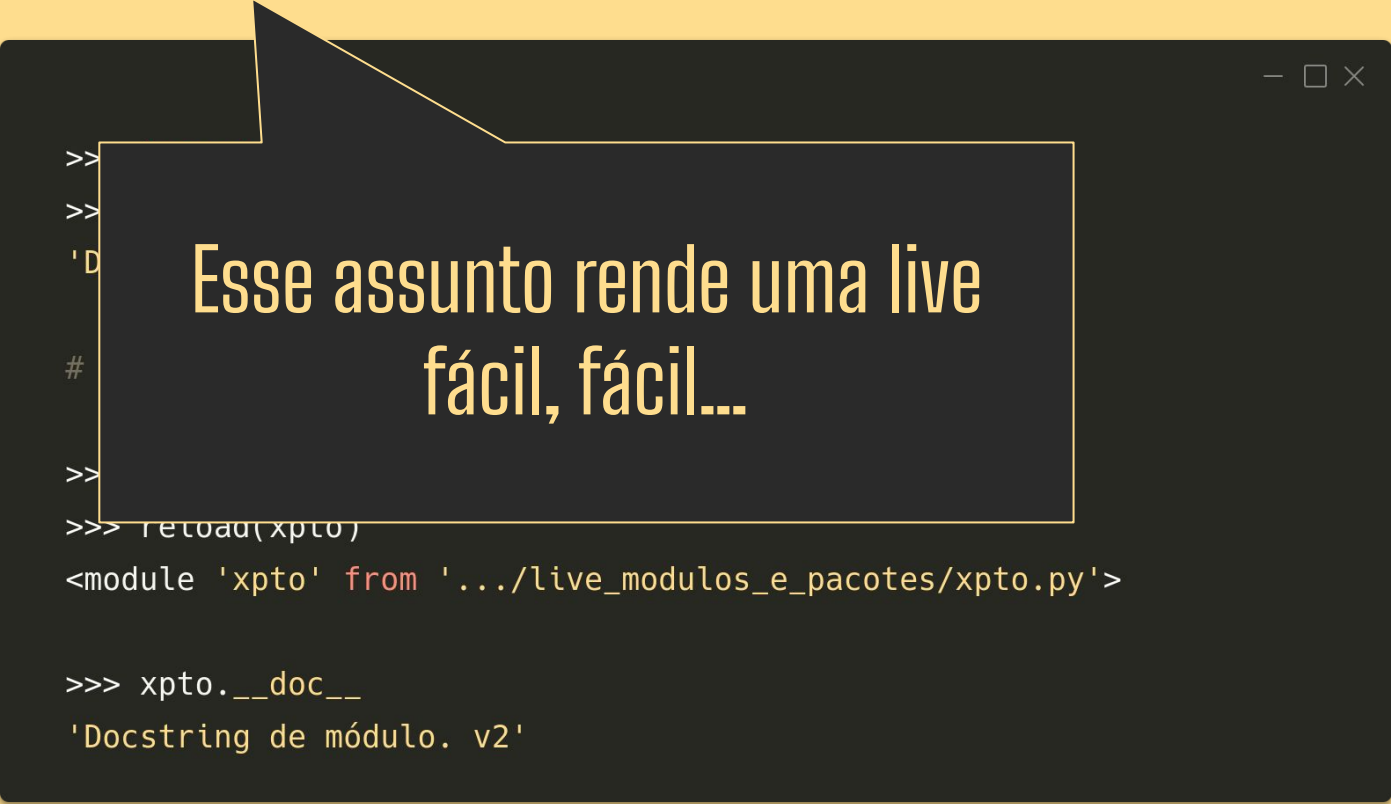
>>> xpto.__doc__
'Docstring de módulo. v2'
```



# Módulos podem ser recarregados em runtime!



Uma das características mais legais de trabalhar com módulos (usando **programação modular**) é poder reescrever e fazer reload em módulos.



```
>>>
>>>
'D
#
>>>
>>> reload(xpto)
<module 'xpto' from '../live_modulos_e_pacotes/xpto.py'>

>>> xpto.__doc__
'Docstring de módulo. v2'
```

Cus  
tomi  
zação

De acesso a nomes

# Acesso a nomes em módulos



Um problema bastante comum no uso de módulos é que conforme o sistema avança, as APIs mudam, coisas são removidas, interfaces são refeitas, ...

O módulo precisa se manter "tolerante" ao tempo e às alterações. Fazendo com que coisas ainda estejam disponíveis mesmo quando interfaces novas foram criadas.

Esse problema é tratado pela **PEP 562**, que nos permite customizar o acesso a atributos.

# Buscando nomes com `__getattr__`



Um problema comum é procurar por um nome que não existe mais no módulo. Uma coisa que já existiu no passado.

Por exemplo:

```
# exemplo_getattr.py
novo_nome = 'YAY!'

def __getattr__(nome: str):
    if nome == 'nome_antigo':
        return novo_nome
```

# Buscando nomes com `__getattr__`



Um problema comum é procurar por um nome que não existe mais no módulo. Uma coisa que já existiu no passado.

Por exemplo:

```
# exemplo_getattr
```

```
novo_nome = 'YAY!'
```

```
def __getattr__(nome: str):  
    if nome == 'nome_antigo':  
        return novo_nome
```

```
>>> from exemplo_getattr import nome_antigo  
>>> nome_antigo  
'YAY!'
```

# \_\_getattr\_\_



Quando um nome não existe no módulo, **\_\_getattr\_\_** é chamado e dentro dessa função customizada, ações podem ser tomadas antes de retornar um nome ou um erro.

Isso pode ser usado em qualquer forma dinâmica de import. O ideal é que um erro seja retornado quando não existir o nome ou algo equivalente...

```
def __getattr__(nome: str):  
    if nome == 'nome_antigo':  
        return novo_nome  
    else:  
        raise AttributeError(  
            f'{nome} não existe no módulo {__name__}'  
        )
```

## Ponto de atenção



Para que a chamada de `__getatt__` seja feita, o nome chamado não pode estar disponível no módulo. Caso contrário, ele será retornado antes de `__getattr__` ser chamado:

```
>>> import exemplo_getattr
>>> exemplo_getattr.novo_nome
'YAY!'
>>> exemplo_getattr.nome_antigo
'YAY!'
```

# Namespaces e `__path__`



## Nota para nerds...

Um efeito complexo da relação entre módulos e diretórios de módulos (pacotes), vai ser refletido no sistema de imports.

Quando a estrutura **from \_\_ import \_\_** é chamada, por otimização o sistema quer saber se o que existe é um pacote, isso é definido pelo nome **`__path__`**.

Logo, toda chamada de from import resultará em uma chamada inicial de `__path__`

[https://github.com/python/cpython/blob/v3.14.0a7/Lib/importlib/\\_bootstrap.py#L1465](https://github.com/python/cpython/blob/v3.14.0a7/Lib/importlib/_bootstrap.py#L1465)



# Listando nomes via `__dir__`



Outro problema que pode acontecer é exibir nomes que não existem no namespace do pacote, eles podem ter sido removidos, alterados, ...

Nesse momento entra o `__dir__` do módulo, uma forma de mostrar dinamicamente os nomes contidos no namespace do módulo.

```
novo_nome = 'YAY!'

deprecated_names = {'nome_antigo': novo_nome}
__all__ = ['novo_nome']

def __dir__():
    return __all__ + list(deprecated_names.keys())
```

## \_\_dir\_\_



A implementação do **\_\_dir\_\_** depende somente de uma função. O ideal é que ela retorne uma lista contendo todos os nomes existentes no pacote + os que você gostaria de exibir dinamicamente, combinando as respostas com **\_\_getattr\_\_**.

```
>>> import exemplo_dir
>>> dir(exemplo_dir)
['nome_antigo', 'novo_nome']
```

# Casos de uso



Existem diversos motivos para ter nomes dinâmicos em pacotes. Os mais comuns são:

- Nova interface para algo (função, classe, ...)
- Remoção de nomes que existiam em versões anteriores
- Fazer swap entre chamadas
  - Retornar outro nome quando um nome deprecado é chamado
- Exibir mensagens de warning
  - "Isso será removido na versão x.y.z"
  - "Use a função xpto..."
  - Ou qualquer outro tipo de mensagem

# Exemplo de warning / swapping



```
from warnings import warn

novo_nome = 'YAY!'

deprecated_names = {'nome_antigo': novo_nome}
__all__ = ['novo_nome']

def __getattr__(name: str):
    if name in deprecated_names:
        swapped_name = deprecated_names[name]
        warn(f'{name!r} está deprecado, use {swapped_name!r}')
        return swapped_name
    else:
        raise AttributeError(f'{name!r} não existe no módulo {__name__}')

def __dir__():
    return __all__ + list(deprecated_names.keys())
```

# Exemplo de warning / swapping



```
from warnings import warn

novo_nome = 'YAY!'

deprecated_names = {'nome_antigo': novo_nome}
__all__ = ['novo_nome']

def __getattr__(name: str):
    if name in deprecated_names:
        swapped_name = deprecated_names[name]
```

```
>>> from exemplo_warning import nome_antigo
/home/dunossauro/live_modulos_e_pacotes/slides/exemplo_warning.py:12:
UserWarning: 'nome_antigo' está deprecado, use 'YAY!'
```

Os temidos

Pa  
co  
tes

# Pacotes



Pacotes não são nada mais que diretórios (pastas) com módulos python internos. O que formam um módulo único, nomeados pelo nome do diretório.

```
pacote
├── modulo.py
├── modulo_a.py
└── modulo_b.py
```

# Pacotes



Pacotes não são nada mais que diretórios (pastas) com módulos python internos. O que formam um módulo único, nomeados pelo nome do diretório.

pacote

├─ modulo.py

├─ modulo\_a.py

└─ modulo\_b.py

```
>>> import pacote
```

```
>>> type(pacote)
```

```
<class 'module'>
```



# Sentindo calafrios...



```
>>> dir(pacote)
['__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__']
```

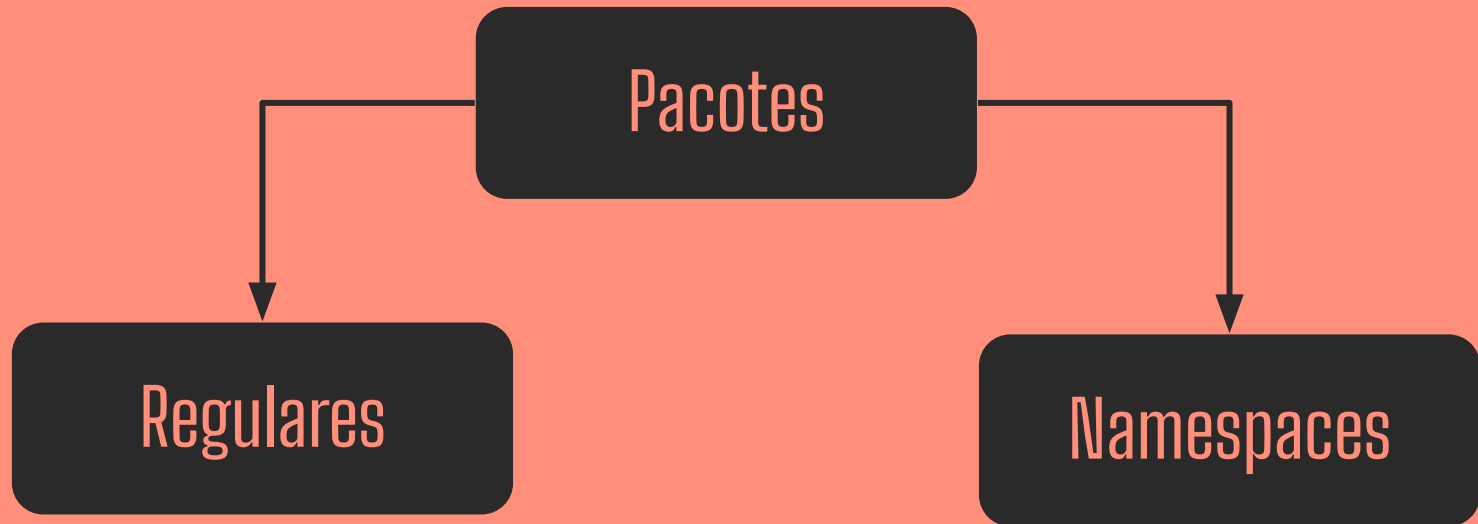
Cadê os módulos que deveriam estar aqui?

Olha ele aí...

# Pacotes



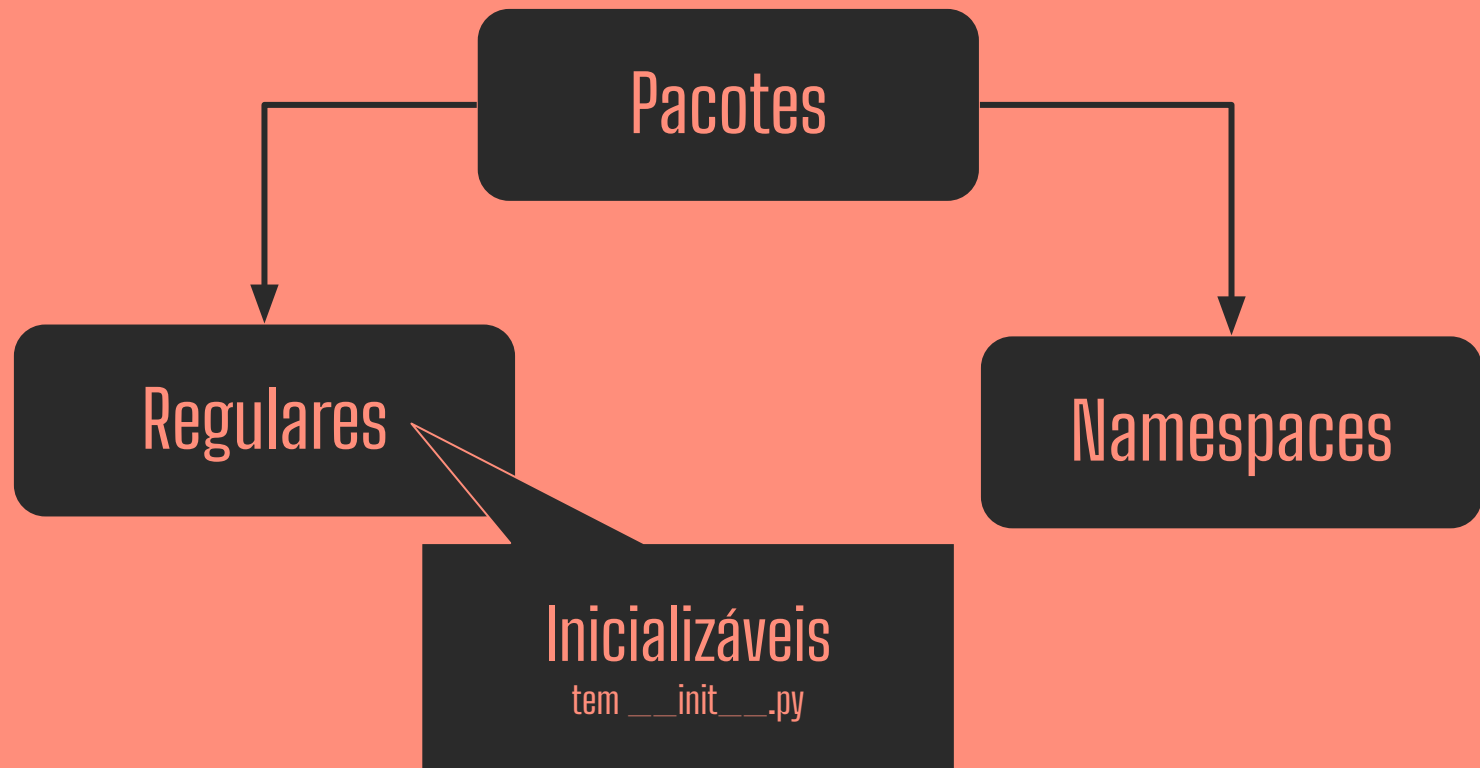
Existem dois tipos de pacotes possíveis em python



# Pacotes



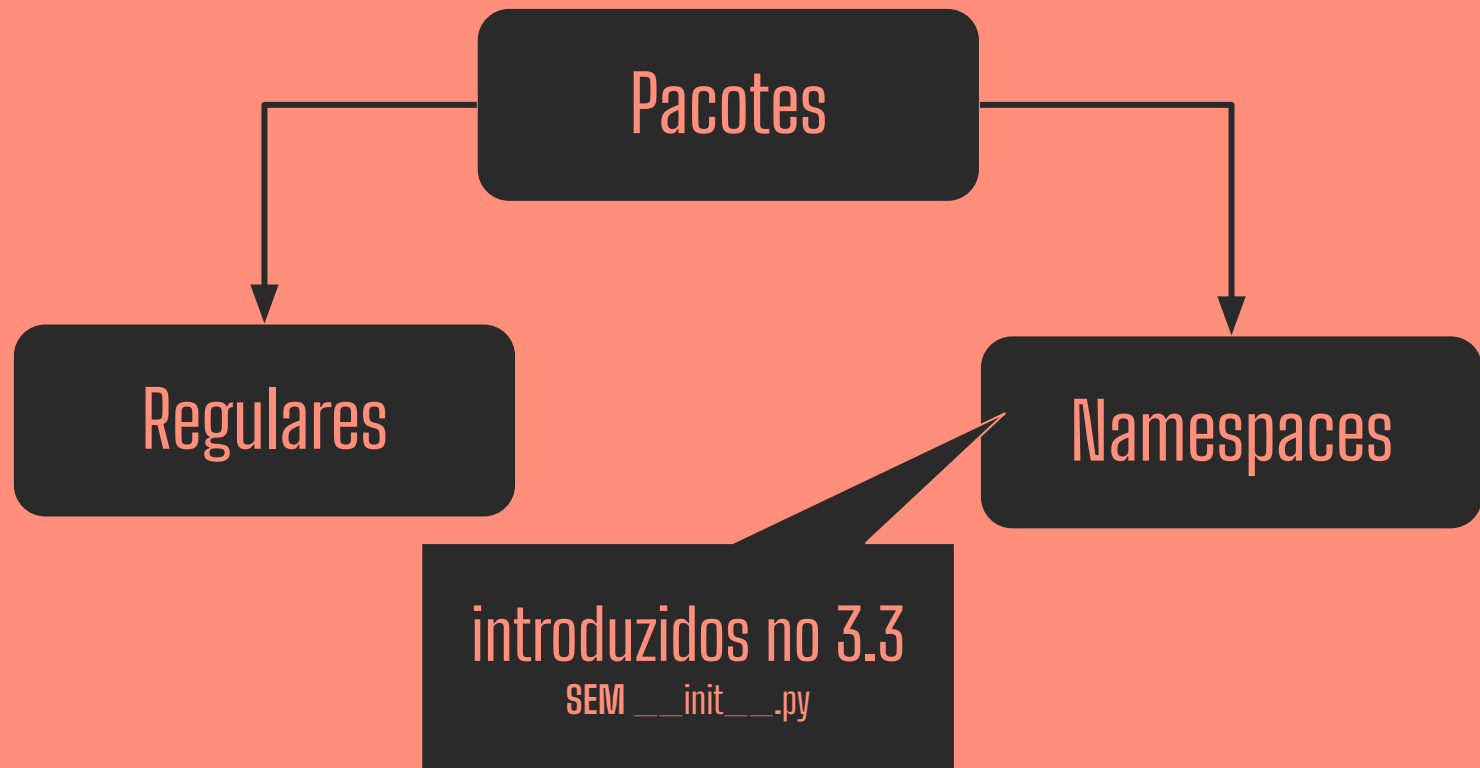
Existem dois tipos de pacotes possíveis em python



# Pacotes



Existem dois tipos de pacotes possíveis em python



# Pacote regular



O que define um pacote regular é somente sua característica de poder ser inicializado. Ou seja, dentro do diretório principal existe um arquivo **`__init__.py`**

O arquivo é responsável pela inicialização do pacote, ou seja, sempre que algo do pacote for importado (`import from`) ou o próprio pacote, `__init__` será executado.

## Exemplo básico



```
# pacote/__init__.py  
print('Olá, seu módulo foi inicializado...')
```



```
>>> import pacote  
Olá, seu módulo foi inicializado...
```

# namespaces / hooks



Como qualquer coisa que esteja em `__init__.py` será executada, o céu é o limite, algumas coisas que você pode testar depois:

- Fazer imports de dentro do pacote: simplificando o uso
- Executar "hooks": Qualquer código que queira pode ser executado
  - Fazer checagem do sistema
  - Carregar objetos em memória
  - Customizar o namespace
- Customizar o acesso: `__getattr__` e `__dir__`
- Montar uma "fachada" para o módulo
- ...

# Ok... Vamos resolver um problema clássico!



Toda vez, todos os dias, toda hora... Alguém vem dizer "Estou com um import circular". Vamos resolver o problema com o `__init__.py`

```
# pacote/__init__.py
print('Olá, seu módulo foi inicializado...')

# Definindo batatas_fritas voadoras
batatas_fritas_voadoras = True

# Trazendo o nome ao namespace
from .modulo import xpto

# Só pq sim...
__all__ = ['xpto']
```

```
# pacote/modulo.py
from . import batatas_fritas_voadoras

def xpto():
    return batatas_fritas_voadoras
```



## Pausa dramática para se chocar no chat



```
>>> import pacote
Olá, seu módulo foi inicializado...
>>> pacote.xpto()
True
>>> from pacote import xpto
>>> xpto()
True
```

# Pacote executável



Outro arquivo que aparece nos pacotes é o `__main__.py`.

Esse arquivo dá ao pacote o superpoder de ser executado pela CLI do python. Usando o parâmetro **-m**.

Algo como:

A dark-themed terminal window with a title bar containing minimize, maximize, and close buttons. The command 'python -m http.server' is typed in white text, with 'server' highlighted in yellow.

```
python -m http.server
```

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

# \_\_main\_\_.py



Qualquer pacote com o arquivo pode ser executado via -m. Lembrando que se o pacote for regular, o \_\_init\_\_.py será executado antes

pacote

├─ \_\_init\_\_.py

├─ \_\_main\_\_.py

└─ modulo.py

```
# pacote/__init__.py
```

```
print('batatinhas')
```

```
# pacote/__main__.py
```

```
print('Ihooooooooooooooooooooo!')
```

```
python -m pacote
```

```
batatinhas
```

```
Ihooooooooooooooooooooo!
```

# Versionamento dinâmico [desviando do assunto]



Aqui é um bom lugar para relembrar o `__version__`, quando ele está no `__init__.py`, os backends de build podem usar ele de forma dinâmica.

```
# pyproject.toml
[project]
name = 'pacote'
dynamic = ['version']
# pacote.__init__.__version__
```

```
[tool.setuptools.dynamic]
version = {attr = "pacote.__version__"}
```

```
[tool.hatch.version]
path = "pacote/__init__.py"
```

# Pacote de namespace



Introduzidos no Python 3.3 (**2012**) os pacotes de namespace não têm a necessidade de serem inicializados.

Tá, mas o que muda? Em 99% dos casos **\*NADA\***! Vai dizer que nunca esqueceu de criar um `__init__.py`? E tudo seguiu normalmente...

(Pode olhar seu projeto agora, eu sei que você vai fazer isso)

# Casos de uso



A ideia mais complexa, daqueles 1% faltantes, é criar pacotes distribuídos. Por exemplo, aquele seu diretório que tem mais 20 diretórios dentro que tem mais 20 em cada um e assim vai...

O conceito permeia em torno de fazer pacotes independentes, mas que, no fundo, todos usem o mesmo namespace... (por isso o nome)

Mas como?



namespace\_demo

└─ pkg1

│ └─ meupacote

│ │ └─ mod1.py

│ └─ pyproject.toml

└─ pkg2

│ └─ meupacote

│ └─ pyproject.toml

└─ teste.py

namespace\_demo

└─ pkg1

│ └─ meupacote

│ │ └─ mod1.py

│ └─ pyproject.toml

└─ pkg2

│ └─ meupacote

│ └─ pyproject.toml

└─ teste.py

```
[project]
```

```
name = "meupacote-mod2"
```

```
version = "0.1"
```

```
description = "Parte 2 do namespace meupacote"
```

```
[tool.setuptools.packages.find]
```

```
where = ["."]
```

```
[build-system]
```

```
requires = ["setuptools>=61.0"]
```

```
build-backend = "setuptools.build_meta"
```



Deixei as instruções pra reproduzir aqui



`criar_namespace.py`



Por que raios todos os gerenciadores de projeto usam  
/src no lugar de flat?

Pelo fato de suportarem N pacotes no namespace



Isso responde a velha pergunta!



# Referências



- PEP 8: <https://peps.python.org/pep-0008/#module-level-dunder-names>
- PEP 562: <https://peps.python.org/pep-0562/>
- PEP 420: <https://peps.python.org/pep-0420/>
- Modulos: <https://docs.python.org/pt-br/3.13/tutorial/modules.html>
- Pacotes: <https://docs.python.org/pt-br/3.13/tutorial/modules.html#packages>
- Every dunder method: <https://www.pythonmorsels.com/every-dunder-method/>
- Discuss: <https://discuss.python.org/t/module-level-getattr-and-from-imports/32236/7>
- flat vs src: <https://packaging.python.org/en/latest/discussions/src-layout-vs-flat-layout/>

## Versionamento dinâmico:

- packing: <https://packaging.python.org/en/latest/guides/writing-pyproject-toml/#version>
- setuptools: [https://setuptools.pypa.io/en/latest/userguide/pyproject\\_config.html#dynamic-metadata](https://setuptools.pypa.io/en/latest/userguide/pyproject_config.html#dynamic-metadata)
- hatch: <https://hatch.pypa.io/1.12/version/#configuration>

## Links citados:

- Live de escopos e namespaces: <https://youtu.be/nWmPEgTwGMM>
- Live do sistema de imports: <https://youtu.be/a5R5dvim6TQ>
- Live do pyproject: <https://youtu.be/6p1HKaHrk0Y>



[apoia.se/livedepython](https://apoia.se/livedepython)



[pix.dunossauro@gmail.com](mailto:pix.dunossauro@gmail.com)



[patreon.com/dunossauro](https://patreon.com/dunossauro)



Ajude o projeto <3

