Módulos e pacotes em Python

Na última aula falamos sobre a utilização da biblioteca padrão do Python, e discutimos sobre módulos pré-definidos, como `sys`, `os` e `random`.

Na aula de hoje, vamos ver sobre como é possível trabalhar com módulos que não estão pré-definidos na biblioteca padrão. Para este caso, vamos trabalhar com dois cenários: uso de módulos e pacotes desenvolvidos por terceiros, e criação dos nossos próprios pacotes Python.

Sobre pacotes externos

Uma das grandes vantagens que temos ao utilizar o Python como linguagem de programação, é que a comunidade online ao redor dela é vasta. Esse fato permite que qualquer desenvolvedor possa implementar seu próprio pacote Python, e distribui-lo para outros usuários.

Através dessa prática, surgiram inúmeros pacotes importantíssimos para o desenvolvimento de aplicações Python modernas, como `scikit-learn`, `pyqt`, `pandas` e `numpy`. Todos esses pacotes foram elaborados e publicados por outros desenvolvedores Python, que fazem a manutenção e evolução do código, além de garantirem uma boa documentação sobre como utilizá-lo.

No entanto, essa facilidade da linguagem, de possibilitar que qualquer um desenvolva e publique seu próprio pacote, apresenta um risco de que pessoas mal intencionadas coloquem à disposição códigos maliciosos, que podem trazer vírus e prejudicar usuários menos atentos.

Como forma de proteger os desenvolvedores dessas ameaças, o *Python Software Foundation* (PSF), entidade que mantém e evolui a linguagem, desenvolveu o **Python Package Index**, ou PyPI, um repositório de pacotes Python desenvolvidos pela comunidade, que foram analisados e aprovados pela entidade. Dessa forma, temos uma segurança maior quanto à integridade dos pacotes que iremos utilizar.

De qualquer forma, isso não significa que todos os pacotes disponibilizados no PyPI sejam seguros. Ocasionalmente vemos notícias de pacotes maliciosos que podem infectar a sua máquina. Por isso, é importante sempre pesquisar sobre o pacote antes de instalá-lo na sua máquina!

Como instalar pacotes externos

Instalando o pip

Para instalar pacotes externos em Python, utilizamos uma ferramenta chamada **pip**. Essa ferramenta normalmente já vem incluída ao instalarmos a distribuição Python. Para conferir se o seu computador possui o pip instalado, abra o prompt de comando, o power shell ou o terminal e execute o seguinte comando:

```
pip --version
```

Caso o computador retorne uma versão do pip, é porque você possui a ferramenta instalada. Se algum erro surgir, tente reinstalar o Python, utilizando as instruções contidas nos slides da disciplina. Caso você seja usuário de Mac, o Python já deve ter vindo instalado junto com o sistema operacional, e o pip não deve ter sido instalado. Veja nos slides da disciplina pelas instruções de como instalar.

Instalando um pacote externo

Para instalar um pacote externo, basta usar o pip no seu terminal de preferência. Insira o seguinte comando:

```
pip install <nome_do_pacote>
```

O pip deve, então, instalar o pacote na instância Python que estiver configurada no seu computador.

Como usar um pacote externo

O uso de pacotes externos é idêntico ao de módulos da biblioteca padrão. Basta utilizar a instrução `import` ou uma alternativa que falamos na última aula.

Lembre-se que é importantíssimo utilizar a documentação do pacote como referência ao utilizá-lo em algum projeto! Pesquise no google pelo nome do pacote que, normalmente, a documentação surge em um dos primeiros resultados.

Trabalhando com módulos e pacotes em Python

Um dos recursos mais poderosos em Python é a possibilidade de desenvolvermos nossos próprios módulos e pacotes.

Pense no seguinte cenário: você está desenvolvendo um jogo, mas a linguagem que está usando obriga que tudo precise ser escrito em um único arquivo. Você tem que implementar, nesse único arquivo, não só toda a lógica do jogo, mas também a animação de todos os objetos em cena, a física do mundo do jogo, a mecânica de pontuação do jogo e a simulação da inteligência artificial por trás. Esse código pode chegar a milhares, ou até mesmo, a milhões de linhas.

Agora considerem que você mesmo, dois anos depois, precisa desenvolver uma expansão para esse jogo, e tem que incluir um novo personagem ao código. Você consegue imaginar o esforço que seria necessário para localizar a parte do código referente à movimentação dos personagens em um código com milhões de linhas? Não deve ser uma tarefa muito fácil.

É para prevenir esse cenário que o Python possui o recurso do desenvolvimento de módulos próprios. Devido a essa funcionalidade, é possível separar o código de forma que ele fique logicamente agrupado, organizado em arquivos que estejam relacionados entre si. Poderíamos ter, por exemplo, uma estrutura como a abaixo:

```
src/
  personagens/
    fulano.py
    cicrano.py
  fisica/
    movimento.py
    aceleracao.py
  tela/
    fundo.py
    quebraveis.py
    itens.py
  pontuacao/
    pontuacao.py
  npcs/
    inimigo1.py
    inimigo2.py
main.py
```

Percebam que essa estrutura indica, facilmente, onde precisamos incluir novos objetos ao evoluir o nosso código. A organização do código em módulos e pacotes é parte essencial de um bom desenvolvimento de software.

E como construímos pacotes?

Para a aula de hoje, vamos montar um exemplo puramente didático. Ele não vai fazer muita coisa, mas vamos aplicar esse conceito de módulos e pacotes com mais detalhes em uma aplicação futura.

Considere um cenário em que temos o seguinte:

```
jogo/
  __init__.py
  atores/
    __init__.py
    ator1.py
    ator2.py
  sons/
    __init__.py
    externo.py
    interno.py
```

```
    vozes.py
main.py
```

Veja que cada pacote e subpacote (`jogo`, `atores` e `sons`) possuem um arquivo `__init__.py` (com dois `_` antes e dois após o `init`). Esses arquivos são necessários para que o Python trate diretórios contendo o arquivo como pacotes. Isso previne que diretórios com um nome comum, como `string`, ocultem, involuntariamente, módulos válidos que ocorrem posteriormente no caminho de busca do módulo. Nos casos mais simples, `__init__.py` pode ser simplesmente um arquivo vazio, mas pode também executar código de inicialização do pacote ou outras ações.

Para os propósitos dessa aula, vamos focar apenas nos cenários em que não precisamos incluir nada nos arquivos `__init__.py`. Nesse caso, as versões mais recentes do Python costumam deixar como opcional a inclusão desses arquivos. Não obstante, é interessante incluir como boa prática de projeto.

Quem estiver usando o pacote `jogo`, pode importar módulos individuais, como por exemplo:

```
import jogo.atores.ator1
```

Isso carrega o submódulo `jogo.atores.ator1`. Ele deve ser referenciado com seu nome completo, como em:

```
jogo.atores.ator1.falar("Olá!")
```

Uma maneira alternativa para a importação desse módulo e o seu uso é:

```
from jogo.atores import ator1

ator1.falar("Olá!")
```

Dessa forma, o interpretador carrega o módulo `ator1` sem a necessidade de mencionar o prefixo do pacote no momento da utilização. Todas as outras formas de importação de módulos, submódulos e objetos que vimos na última aula também são aplicáveis aqui.

Para a importação estrela (`from jogo.atores.ator1 import *`) há a necessidade de alteração do `__init__.py`. Como essa é uma prática que já falamos que não é ideal, vamos pular essa explicação por aqui, mas a documentação do Python explica o seu uso.

Referências em um mesmo pacote

Considere que, no nosso exemplo, os atores programados utilizam sons definidos no subpacote `sons`. Nesse caso, podemos realizar uma **importação absoluta** ou uma **importação relativa**.

Na importação absoluta, se queremos utilizar o módulo `ator1`, por exemplo, precisamos importá-lo utilizando `from jogo.atores import ator1`.

Já na importação relativa, usamos pontos para indicar o pacote pai e o atual, envolvidos no import relativo. Do módulo `ator1`, por exemplo, pode-se usar:

```
from . import ator2  # ator2 é "irmão" de ator1
from ..sons import vozes # aqui estou subindo mais um nível antes de escolher o pacote sons
```

Existem outras formas de fazermos referências a módulos e pacotes, porém seria necessário entrarmos no detalhe do `__init__.py`, o que não devemos fazer nesse curso.

Sobre o módulo principal

Toda aplicação Python possui um módulo que é considerado principal, ou seja, é aquele que começa todo o procedimento previsto na aplicação. Veja que na nossa estrutura de exemplo temos um arquivo `main.py` direto na raiz do projeto, fora do pacote `jogo`.

Sempre que tivermos programando no módulo principal, devemos utilizar referências absolutas nos nossos imports! Essa é uma regra que está relacionada com o funcionamento dos módulos dentro de uma aplicação Python.

Usando o `if __name__ == "__main__":`

O Python inclui, em todo módulo ou objeto (classe, função ou variável), algumas variáveis "ocultas", que permitem a manipulação desses itens por parte do programador e também por parte do interpretador. Não vamos entrar nos detalhes dessas variáveis, mas uma delas é a `__name__`.

Essa variável indica o nome do módulo ou objeto que está sendo executado no momento. Caso o módulo seja o principal, ou seja, caso ele seja o arquivo Python originalmente chamado pelo usuário, o interpretador dará o nome `__main__` para ele.

Aí entra um recurso interessante e bem prático para usarmos em nossos projetos. Considere que implementamos um código no módulo `jogo.atores.ator1` e desejamos testar se esse código está correto, porém não quero incluir toda a lógica do meu projeto nesse teste. Para executar esse teste, podemos usar o terminal para navegar diretamente para esse arquivo e simplesmente executar esse arquivo (`python ator1.py`).

No entanto, se tivermos uma função `teste()` nesse módulo e essa função for chamada diretamente, quando executarmos nosso projeto completo, a função `teste()` também será executada!

É para isso que precisamos utilizar a estrutura `if __name__ == "__main__":`. Dessa forma, a função teste só será chamada caso o módulo seja o principal, ou seja, o usuário tenha executado diretamente o módulo `jogo.atores.ator1`.

O nosso código ficaria, portanto:

```
def falar():
  print("Olá! Sou o ator 1.")

def teste():
  falar()

if __name__ == "__main__":
  teste()
```

Este artigo possui mais detalhes sobre por que é interessante sempre usar essa estrutura nos

Aplicando os conceitos

Tendo apresentado todos os conceitos, vamos criar um projeto com a mesma estrutura dos pacotes `jogo`, `atores` e `sons` do nosso exemplo!

Crie os arquivos conforme a estrutura apresentada, e implemente os códigos conforme apresentado abaixo (deixe os arquivos `__init__.py` em branco).

Após implementar os códigos, execute o arquivo `main.py` e veja a conversa sendo realizada!

No exemplo, usamos os imports relativos e absolutos, mas é interessante manter um padrão para que o entendimento fique mais fácil para outras pessoas. Particularmente, eu prefiro utilizar imports absolutos a relativos.

Arquivo main.py

```
from jogo.atores import ator1
from jogo.sons import externo, interno

def main():
  externo.emitir_som()
  ator1.conversar()
  interno.emitir_som()

if __name__ == "__main__":
  main()
```

Arquivo jogo.atores.ator1

```
from . import ator2
from ..sons import vozes

def falar():
  print("Olá! Sou o ator 1.")
  vozes.emitir_som()
```

```
def conversar():
  falar()
  ator2.falar()
```

Arquivo jogo.atores.ator2

```
import jogo.sons.vozes

def falar():
  print("Olá! Sou o ator 2.")
  jogo.sons.vozes.emitir_som(False)
```

Arquivo jogo.sons.externos

```
def emitir_som():
  print("Você está ouvindo o barulho de vento")
```

Arquivo jogo.sons.internos

```
def emitir_som():
  print("Você está ouvindo um barulho de vozes em um espaço fechado.")
```

Arquivo jogo.sons.vozes

```
def emitir_som(voz_aguda=True):
  if voz_aguda:
    print("O tom de voz que você ouviu foi bem agudo.")
  else:
    print("O tom de voz que você ouviu foi bem grave.")
```