A biblioteca padrão do Python

Como já vimos durante o curso, o Python nos fornece diversas funções pré-definidas, de forma que podemos executar ações, como determinar o valor máximo em uma lista de números, sem precisar implementar o código novamente.

Apesar dessas funções pré-definidas serem interessantes, elas estão longe de atenderem a todas as nossas necessidades, como programadores. Caso tivéssemos que implementar todas as outras funções para gerar nossos programas, o trabalho seria inviável.

É por isso que há um conceito na Engenharia de Software chamado **reuso de software**. Essa linha de pesquisa trata de técnicas e ações que podem ser aplicadas para reaproveitar códigos de um projeto (ou uma parte do projeto) em outro lugar. Sempre devemos nos preocupar em como conseguir reaproveitar, ao máximo, o nosso código implementado.

A organização que faz a curadoria do desenvolvimento do Python também pensou nessa situação, e elaborou uma forma do programador Python não só reaproveitar código já implementado por outras pessoas, mas também desenvolver suas próprias funções e distribui-las para outros programadores na comunidade.

Módulos, pacotes e bibliotecas

Para esta aula, precisamos definir os termos **módulo**, **pacote** e **biblioteca**:

- Módulo: é qualquer arquivo Python (.py), que possui funções, variáveis e estruturas de dados. Normalmente chamamos de módulo o script Python que pode ser reutilizado externamente, ou seja, um script Python qualquer pode "chamar" uma função definida no módulo.
- Pacote: é um conjunto de módulos dentro de um **namespace**. Namespaces são coleções que possuem nomes únicos não apenas para módulos Python, mas também para qualquer objeto (funções, variáveis, etc.).
- Biblioteca: é um conjunto de funcionalidades que permitem a criação de várias tarefas sem a necessidade de se criar mais código. Conceitualmente, não há muita diferença entre uma biblioteca e um pacote, mas normalmente chamamos de biblioteca um conjunto de pacotes.

Na próxima aula vamos falar melhor sobre como criarmos os nossos próprios módulos e pacotes. Na aula de hoje, vamos falar um pouco mais sobre a biblioteca mais importante da linguagem, a **biblioteca padrão**.

A biblioteca padrão

A biblioteca padrão do Python nada mais é do que o conjunto de pacotes e módulos que são entregues em toda distribuição Python. Ou seja, sempre que você instala o Python no seu

computador, você já tem acesso a inúmeras funções e objetos já construídos, sem precisar instalar nada.

A biblioteca padrão está descrita nessa página da documentação oficial. O número de pacotes à nossa disposição é enorme, e cada pacote tem uma variedade de funções e métodos que torna inviável e improdutivo abordar tudo durante as aulas.

O que veremos hoje é apenas uma visão geral dos módulos que eu considero essenciais para, basicamente, qualquer projeto de médio/grande porte. Normalmente as pessoas escrevem código com a documentação aberta, portanto comece a desenvolver o hábito de abrir o site com a documentação oficial quando começar a programar. Sempre que surgir uma dúvida do tipo "será que essa função já foi feita por alguém?", pesquise na documentação. Em boa parte das vezes, a resposta a essa pergunta será "sim".

Como usar um módulo

Ok, já visualizamos na documentação quais módulos estão disponíveis para o nosso uso, mas como vamos chamar as funções desses módulos no nosso código?

Podemos fazer isso, basicamente, de três formas:

- Utilizando a instrução `import`: Nesse caso, o interpretador passa a identificar todo e qualquer objeto (variáveis, funções, classes, etc.) contido naquele módulo. Esse é o modo mais básico de se utilizar um módulo, e na maioria das vezes é o mais comum também.

```
import math

print(math.floor(5.785))
```

```
5
```

- Utilizando a instrução `from <modulo> import <objeto>`: Com essa instrução, o interpretador importa apenas o(s) objeto(s) listado(s) no comando. Isso é muito útil quando temos um módulo muito grande, e precisamos usar apenas uma função ou uma classe. No entanto, é preciso tomar cuidado com ambiguidade de funções. O módulo `os.path`, por exemplo, possui uma função `join()`, que tem o mesmo nome que a função `join()` que une uma lista de strings em uma só. Importar objetos dessa forma pode retornar resultados inesperados pelo interpretador.

```
from math import ceil

ceil(7.825)
```

```
8
```

- Utilizando a instrução `from <modulo> import *`: Essa situação permite que o programador utilize qualquer função do módulo, como no primeiro caso. A diferença é que você não precisa inserir o nome do módulo antes do nome da função. **Utilize essa instrução com muito cuidado!** Ao eliminar a necessidade de se usar o nome do módulo, o risco de conflitos entre módulos e ambiguidade de funções aumenta exponencialmente

```
from os import *

print(getcwd())
```

```
/content
```

Ainda é possível definir "apelidos" para módulos e objetos importados, de forma a simplificar a escrita ao longo do código e/ou eliminar possíveis ambiguidades. Para fazer isto, basta usar a instrução `as`.

```
from math import factorial as fact

print(fact(10))
```

```
3628800
```

Uma breve apresentação de alguns módulos

Vamos então aos módulos que serão discutidos na aula. A lista abaixo apresenta esses módulos, com links para as páginas da documentação que detalham as funções e os demais objetos de forma mais completa.

- math
- random
- re
- datetime
- time
- os
- os.path
- sys
- subprocess
- tkinter

Para cada módulo, será apresentada uma breve descrição do seu uso e algumas das principais funções utilizadas.

math

O módulo `math` inclui diversas funções matemáticas, normalmente mais otimizadas que as funções pré-definidas. Normalmente essa função é usada quando queremos fazer operações

simples. Para operações mais complexas é mais recomendado adotar pacotes customizados, como a `numpy`, que possuem funções complexas e altamente eficientes.

```
import math

print(math.ceil(7.678))  # arredonda para o menor inteiro maior que o valor
print(math.floor(7.678)) # arredonda para o maior inteiro menor que o valor
print(math.factorial(7)) # calcula o fatorial de um número

lista = [0.1] * 10
print(math.fsum(lista))  # calcula uma soma precisa de números flutuantes

print(math.isinf(55))    # informa se um número passado é infinito
print(math.sqrt(16))     # calcula a raiz quadrada de um número

print(math.exp(2))       # calcula o exponencial de um número
print(math.log(8, 2))    # calcula o logaritmo de um número. Se o segundo
                         # parâmetro for vazio, calcula o log neperiano

print(math.sin(3))       # calcula o seno de um ângulo, dado em radianos
                         # funções análogas: cos, tan, acos, asin, atan
print(math.degrees(3))   # converte radianos para graus
print(math.radians(180)) # converte graus para radianos

print(math.pi)           # número pi
print(math.e)            # número de Euller
print(math.inf)          # um número floar que representa infinito
print(-math.inf)          # um número floar que representa infinito negativo
```

```
8
7
5040
1.0
False
4.0
7.38905609893065
3.0
0.1411200080598672
171.88733853924697
3.141592653589793
3.141592653589793
2.718281828459045
inf
-inf
```

random

Este módulo implementa diversas funções que geram valores pseudo-aleatórios. Utilizamos esse termo porque, em computação, nada é realmente aleatório. O computador precisa de um parâmetro inicial (chamado de **semente** ou **seed**) para poder calcular um valor, com base em algum algoritmo. A questão é que essas funções possuem algoritmos que fazem com que o valor calculado **pareça** aleatório, para o ser humano. Na prática, se tivermos acesso ao seed de um cálculo, é possível reproduzir a obtenção do valor com a aplicação do mesmo algoritmo.

Quem acompanha jogos em geral, ou o mercado de eSports em particular, talvez já tenha escutado falar do termo **RNG**. Muitas pessoas se aproveitam justamente dessa pseudo-aleatoriedade dos valores dentro dos jogos para explorar fraquezas dentro do jogo. Leia esse artigo para ter mais informações sobre como RNG ocorre nos eSports.

```
import random

print(random.seed(10))           # define um seed específico, ao invés de adotar
                                 # o valor definido pelo Python
print(random.randint(4, 9))      # retorna um valor entre os inteiros
                                 # especificados

lista = [2, 7, 9, 12, 11, 30]
print(random.choice(lista))      # escolhe um elemento de uma lista não vazia
print(random.choices(lista, k=3))# escolhe k elementos de uma lista não vazia
                                 # esses elementos podem ser repetidos

random.shuffle(lista)            # embaralha uma lista
print(lista)

print(random.sample(lista, k=3)) # escolhe uma amostra de elementos únicos
                                 # de uma lista

print(random.random())           # retorna um float entre 0.0 (inclusive) e 1.0
                                 # (exclusive)
```

```
None
8
2
[9, 12, 7]
[11, 30, 2, 7, 9, 12]
[2, 11, 30]
0.9523992311419316
```

re

Este módulo contém funções que operam em cima de **expressões regulares**, ou **regex**. Expressões regulares são strings que registram padrões de texto, e que podem ser usadas para se reconhecer onde esses padrões ocorrem em outras strings.

Podemos criar uma expressão regular, por exemplo, para validar se um CPF está armazenado no padrão `xxx.xxx.xxx-xx`. Ou então para identificar se um determinado texto possui palavras específicas. O número de possibilidades é enorme, e essa é uma ferramenta extremamente útil em diversos projetos Python.

Ao contrário dos outros módulos, não vamos entrar nos detalhes das funções que podem ser usadas. Esse é um módulo um pouco mais avançado, e que precisa de um estudo mais aprofundado.

Além da documentação do módulo, recomendo as seguintes referências para você poder estudar mais sobre o assunto:

- Esse artigo com dicas para usar regex
- Esse site que permite a criação e teste de regex
- Este artigo com uma breve introdução ao assunto

datetime

Este módulo fornece classes para manipulação de datas e tempos. É possível fazer operações matemáticas entre dias e horas para, por exemplo, calcular prazos para entrega de pedidos ou datas de vencimento de contratos.

Este módulo, ao contrário dos que já vimos, utiliza o conceito de **classes**, do paradigma orientado a objetos. Vamos falar um pouco sobre isso ao final do curso, mas abaixo é possível encontrar alguns exemplos de uso dessas classes.

```
from datetime import date, datetime

hoje = date.today()       # retorna o dia de hoje
print(hoje)

print(hoje.day)           # retorna o dia da variável hoje
print(hoje.month)         # retorna o mês da variável hoje
print(hoje.year)          # retorna o ano da variável hoje
print(hoje.weekday())     # retorna o dia da semana da variável hoje,
                          # sendo a segunda-feira o valor 0

# Estabelece um dia com os parâmetros ano, mês e dia
dia_aleatorio = date(2018, 7, 20)

print(dia_aleatorio)

# Converte uma data para um formato definido
print(hoje.strftime("%A, %d %B %Y"))

agora = datetime.now()    # a classe datetime tem os mesmos métodos e atributos
                          # da classe date, porém também tem informações sobre
                          # horas, minutos e segundos

print(agora)
print(agora.hour)
print(agora.minute)
print(agora.second)
```

```
2021-10-09
9
10
2021
5
2018-07-20
Saturday, 09 October 2021
2021-10-09 22:06:59.015654
22
```

```
6
```

time

Este módulo lida com várias funções relacionadas à manipulação de tempo. Não vamos entrar muito no detalhe deste módulo, mas vamos abordar duas funções que eu, particularmente, uso com bastante frequência: `sleep()` e `time()`.

```
import time

agora = time.time()       # marca o tempo no momento em que o interpretador
                          # executou essa instrução
time.sleep(2)             # para a execução do programa durante o tempo
                          # definido, em segundos

print(f"Passaram-se {time.time() - agora} segundos.")
```

```
Passaram-se 2.002316474914551 segundos.
```

os

Este módulo lida com funções que acessam e alteram o sistema operacional do computador em que o script Python está sendo executado. Ele é essencial para a implementação correta de projetos que rodam na máquina do usuário, para identificar informações como o diretório em que o script está sendo executado, os números dos processos da execução dos programas, variáveis de ambiente definidas, etc.

Neste módulo é importante ressaltar que você estará mexendo no sistema do computador: criando pastas, removendo arquivos, etc. É sempre bom criar testes (`if`) no seu código garantindo que esses arquivos e diretórios existem (ou não existem, caso esteja tentando criar) antes de remover.

Para verificar se um arquivo ou diretório existem, veja a referência do módulo `os.path`.

```
import os

dir_atual = os.getcwd()       # retorna o diretório de trabalho que está sendo
                              # executado
print(dir_atual)
print(os.listdir(dir_atual))      # lista todas as entradas de um diretório

os.mkdir("exemplo")               # cria um diretório
os.makedirs("exemplo2/exemplo3")  # cria um diretório, incluindo os diretórios
                                  # intermediários
os.chdir("sample_data")           # muda o diretório de trabalho
os.chdir("..")                    # volta para o diretório pai

os.remove("sample_data/README.md")  # remove um arquivo, caso ele exista
                                    # sobe uma exceção caso o arquivo não
                                    # exista

os.rmdir("exemplo")                 # remove um diretório
```

```
os.rmdir("exemplo")                    # remove um diretório
os.removedirs("exemplo2/exemplo3")  # remove um diretório, incluindo os
                                       # diretórios intermediários
```

```
/content
['.config', '.ipynb_checkpoints', 'sample_data']
```

os.path

O módulo `os` possui um submódulo tão grande e tão específico, que até mesmo a documentação oficial da biblioteca padrão do Python considera como um módulo à parte. É o módulo `os.path`. Nele encontramos funções específicas para manipulação de nomes de arquivos e diretórios.

Com esse módulo, podemos obter extensões de arquivos, separar um nome de arquivo do diretório em que ele está localizado, unir duas ou mais subpastas, etc.

Sempre que for montar nomes de diretórios, utilize o `os.path` ao invés de criar os diretórios em strings. Uma alternativa ao `os.path` é o `pathlib`, que possui funções um pouco mais sofistifcadas para manipulação de caminhos no computador. Eu, particularmente, prefiro usar o `pathlib`, mas o `os.path` é mais comum na comunidade por estar desde o Python 2.

```
import os.path

dir_atual = os.getcwd()

# Junta um diretório a um ou mais sub-diretórios
sample = os.path.join(dir_atual, "sample_data")
arquivo = os.path.join(sample, "mnist_test.csv")

print(os.path.isdir(sample))       # indica se um diretório existe
print(os.path.isfile(arquivo))     # indica se um arquivo existe

# Indica a última parte do caminho passado
nome_arquivo = os.path.basename(arquivo)
print(nome_arquivo)
print(os.path.basename(sample))

# Separa o caminho no último ponto (.) encontrado
print(os.path.splitext(nome_arquivo))

# Separa o caminho no último diretório ou arquivo
print(os.path.split(sample))

# Indica o nome do diretório pai do caminho apresentado
print(os.path.dirname(sample))

# Retorna o tamanho do caminho passado
print(os.path.getsize(arquivo))
```

```
True
True
mnist_test.csv
sample_data
```

```
('mnist_test', '.csv')
('/content', 'sample_data')
/content
18289443
```

sys

Este módulo lida com funções específicas para a operação do interpretador e das funções que interagem diretamente com o interpretador.

Não vamos ver muitas funções para este módulo, já que muitas delas precisam de um conhecimento mais aprofundado de como funciona o interpretador do Python "por baixo dos panos".

```
import sys

print(sys.executable)    # indica o caminho em que o Python está sendo executado
print(sys.version)       # indica a versão do interpretador Python executado

# Sai da execução do Python, passando um código de status para o programa que
# chamou o interpretador.
# A linha abaixo está comentada para não sairmos do código...
# sys.exit(-2)
```

```
/usr/bin/python3
3.7.12 (default, Sep 10 2021, 00:21:48)
[GCC 7.5.0]
```

subprocess

Para este módulo, veremos apenas uma função, chamada `subprocess.run()`. Essa função está presente desde o Python 3.5, e substitui a função `os.system()` para executar programas dentro do próprio Python.

Não vamos entrar no mérito de como esses programas são executados. Para rodar um programa usando essa função, basta passar uma lista com os mesmos parâmetros que seriam utilizados na chamada pela linha de comando.

```
import subprocess

# O parâmetro capture_output é usado para armazenar o resultado da execução
# do subprocesso.
processo = subprocess.run(["ls", "-l"], capture_output=True)

# O output é dado num tipo de dado chamado bytes, que não vemos durante o curso.
# Para converter para string, basta chamar a função decode()
print(processo.stdout.decode())
```

```
total 4
drwxr-xr-x 1 root root 4096 Oct  9 22:33 sample_data
```

tkinter

Assim como o `re`, não vamos entrar no detalhe deste módulo. No entanto, ele é importante de se conhecer, principalmente para quem quer começar a trabalhar com interfaces gráficas usando Python.

Este módulo oferece diversos objetos específicos para a criação de janelas, caixas de diálogo e outras interfaces gráficas com o usuário. Para estudar mais sobre o assunto, seguem três dicas de conteúdos:

- Este vídeo com uma aplicação prática bem simples
- Este tutorial aplicado
- Este guia passo-a-passo