**Introdução da aula**



Qual é o foco da aula?

Nesta aula, iremos conhecer mais sobre os módulos e suas aplicações.

Objetivos gerais de aprendizagem

Ao longo desta aula, você irá:

* relatar as classificações do módulos;
* identificar os conceitos de biblioteca request;
* demonstrar Módulo em Python.

**Situação-problema**

No Brasil, existe um órgão responsável por gerar as estatísticas da atividade econômica no país. Para tal tarefa, as atividades são classificadas em grupos; por exemplo, as atividades do grupo 262 referem-se à fabricação de equipamentos de informática e periféricos.

"A CNAE, Classificação Nacional de Atividades Econômicas, é a classificação oficialmente adotada pelo Sistema Estatístico Nacional na produção de estatísticas por tipo de atividade econômica, e pela Administração Pública, na identificação da atividade econômica em cadastros e registros de pessoa jurídica." (API CNAE, 2017, [s.p.])

Como desenvolvedor em uma empresa de consultoria de software, você foi alocado em um projeto com base no qual o cliente deseja automatizar a extração dos dados do CNAE e gerar um relatório. Os dados estão disponíveis neste [endereço](https://servicodados.ibge.gov.br/api/v2/cnae/classes" \t "_blank). Você deve extraí-los e gerar as seguintes informações:

* Quantas atividades distintas estão registradas?
* Quantos grupos de atividades existem?
* Quantas atividades estão cadastradas em cada grupo?
* Qual grupo ou quais grupos possuem o maior número de atividades vinculadas?

**O que são módulos?**



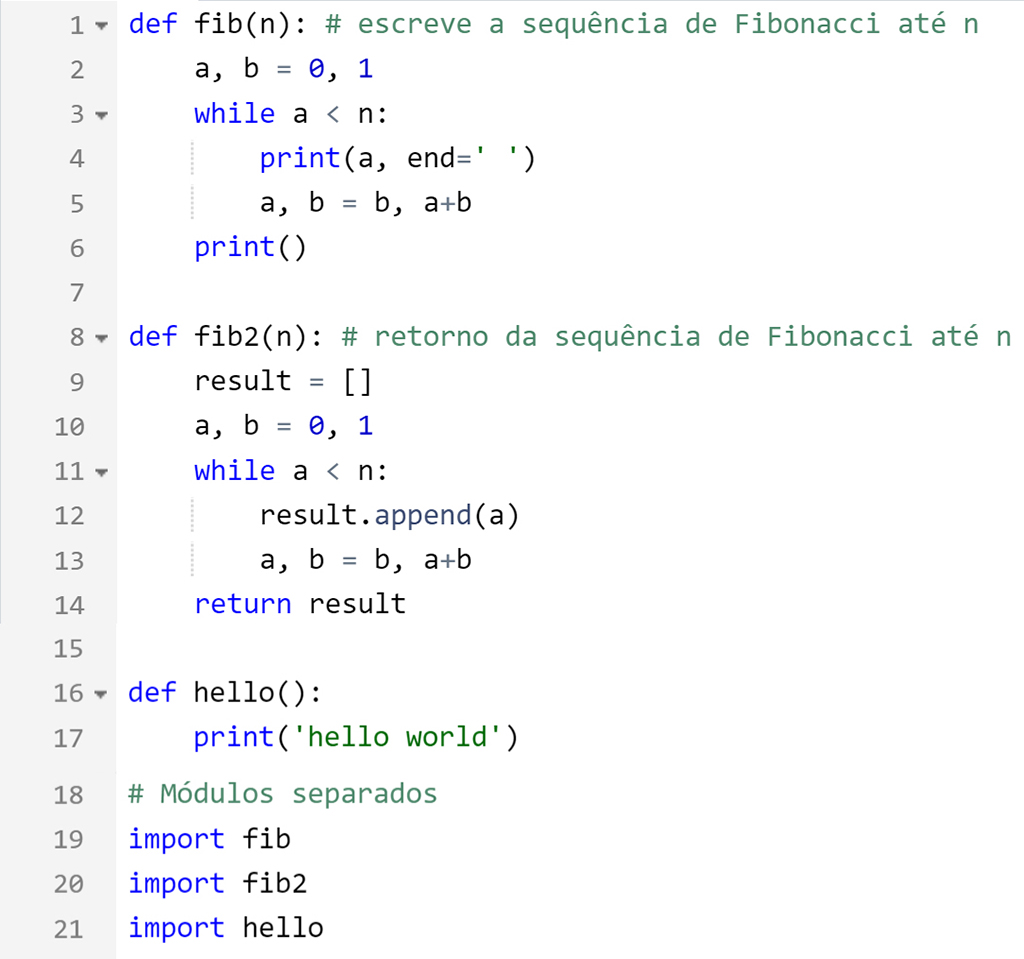
Um módulo pode ser uma biblioteca de códigos, o qual possui diversas funções (matemáticas, sistema operacional, etc.) as quais possibilitam a reutilização de código de uma forma elegante e eficiente.

Implementamos algoritmos, nas diversas linguagens de programação, para automatizar soluções e acrescentar recursos digitais, como interfaces gráficas e processamento em larga escala. Uma solução pode começar com algumas linhas de códigos, mas, em pouco tempo, passa a ter centenas, milhares e até milhões delas. Nesse cenário, trabalhar com um único fluxo de código se torna inviável, razão pela qual surge a necessidade de técnicas de implementação para organizar a solução.

**Módulos e bibliotecas em Python** - Uma opção para organizar o código é implementar funções, contexto em que cada bloco passa a ser responsável por uma determinada funcionalidade. Outra forma é utilizar a orientação a objetos e criar classes que encapsulam as características e os comportamentos de um determinado objeto. Conseguimos utilizar ambas as técnicas para melhorar o código, mas, ainda assim, estamos falando de toda a solução agrupada em um arquivo Python (.py). Considerando a necessidade de implementar uma solução, o mundo ideal é: fazer a separação em funções ou classes e ainda realizar a separação em vários arquivos .*py,* o que é chamado de *modular uma solução* (PSF, 2020b). Segundo a documentação oficial do Python, é preferível implementar, em um arquivo separado, uma funcionalidade que você possa reutilizar, criando assim um módulo.

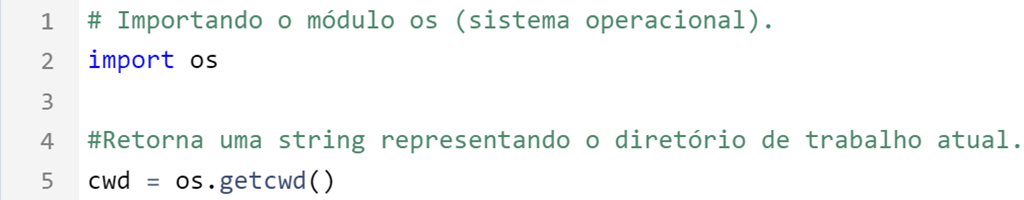
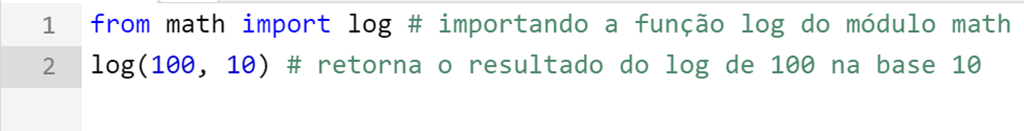
Um módulo é um arquivo contendo definições e instruções Python. O nome do arquivo é o nome do módulo acrescido do sufixo .py. — (PSF, 2020b, [s.p.])

A figura abaixo ilustra essa ideia, com base na qual uma solução original implementada em um único arquivo .py é transformada em três módulos que, inclusive, podem ser reaproveitados, conforme aprenderemos.

Contínuo versus módulo. Fonte: elaborada pela autora.

Falamos em módulo, mas em Python se ouve muito o termo *biblioteca*. O que será que eles têm em comum?

 Na verdade, **um módulo pode ser uma biblioteca de códigos**! Observe a figura abaixo, na qual temos o módulo *math*, que possui diversas funções matemáticas, e o módulo os, que possui funções de sistema operacional, como capturar o caminho (*getcwd*), listar um diretório (*listdir*), criar uma nova pasta (*mkdir*), dentre inúmeras outras. Esses módulos são bibliotecas de funções pertinentes a um determinado assunto (matemática e sistema operacional), as quais possibilitam a reutilização de código de uma forma elegante e eficiente.

Módulo como biblioteca. Fonte: elaborada pela autora.

**Como utilizar um módulo** - Para utilizar um módulo é preciso importá-lo para o arquivo. Essa importação pode ser feita de maneiras distintas:

1. **import**moduloXXText

    1.2  **import**moduloXX **as**apelido

1. **from**moduloXX **import**itemA, itemB

Utilizando as duas primeiras formas de importação (1 e 1.2), **todas**as funcionalidades de um módulo são carregadas na memória. A diferença entre elas é que, na primeira, usamos o nome do módulo e, na segunda, atribuímos a este um apelido (*as = alias*). Na outra forma de importação (2), somente funcionalidades **específicas**de um módulo são carregadas na memória.

A forma de importação também determina a sintaxe para utilizar a funcionalidade. Observe os códigos a seguir.

In [1]: import math

math.sqrt(25)

math.log2(1024)

math.cos(45)

Out[1]:0.5253219888177297

In [2]: import math as m

m.sqrt(25)

m.log2(1024)

m.cos(45)

Out[2]: 0.5253219888177297

In [3]: from math import sqrt, log2, cos

sqrt(25)

log2(1024)

cos(45)

Out[3]: 0.5253219888177297

Na entrada 1, usamos a importação que carrega todas as funções na memória. Observe (linhas 3 a 5) que  precisamos usar a seguinte sintaxe: *nomemodulo.nomeitem*

Na entrada 2, usamos a importação que carrega todas as funções na memória, mas, no caso, demos um apelido para o módulo. Veja (linhas 3 a 5) que, para usála, precisamos colocar o apelido do módulo: *apelido.nomeitem*

Na entrada 3, usamos a importação que carrega funções específicas na memória. Veja (linhas 3 a 5) que, para usá-la, basta invocar a função.

\_\_\_\_\_\_\_

**🔁 Assimile**

Todos os import devem ficar no começo do arquivo.

Ainda segundo a documentação do [site](https://www.python.org/dev/peps/pep-0008/" \l "imports" \t "_blank), é uma boa prática declarar primeiro as bibliotecas-padrão (módulos *built-in)*, depois as bibliotecas de terceiros e, por fim, os módulos específicos criados para a aplicação. Cada bloco deve ser separado por uma linha em branco.

**Classificação dos módulos (bibliotecas)**



Podemos classificar os módulos (bibliotecas) em três categorias, cada uma das quais vamos estudar:

1. **Módulos built-in**: embutidos no interpretador.
2. **Módulos de terceiros**: criados por terceiros e disponibilizados via PyPI.
3. **Módulos próprios**: criados pelo desenvolvedor.

Módulos **built-in** - Ao instalar o interpretador Python, também é feita a instalação de uma biblioteca de módulos, que pode variar de um sistema operacional para outro.

 Alguns módulos estão embutidos no interpretador; estes possibilitam acesso a operações que não são parte do núcleo da linguagem, mas estão no interpretador seja por eficiência ou para permitir o acesso a chamadas do sistema operacional. — (PSF, 2020b, [s.p.])

Como estão embutidos no interpretador, esses módulos não precisam de nenhuma instalação adicional.

São vários os módulos built-in disponíveis. No [endereço](https://docs.python.org/3/py-modindex.html" \t "_blank)você encontra a lista com todos os recursos disponíveis. Vamos explorar alguns deles.

Módulo ***random***- Random é um módulo *built-in* usado para criar números aleatórios. Vamos explorar as funções:

* *random.randint(a, b*): retorna um valor inteiro aleatório, de modo que esse número esteja entre a, b.
* *random.choice(seq)*: extrai um valor de forma aleatória de uma certa sequência.
* *random.sample(population, k)*: retorna uma lista com k elementos, extraídos da população.

In [4]: import random

print(random.randint(0, 100))

print(random.choice([1, 10, -1, 100]))

print(random.sample(range(100000), k=12))

80

100

[18699, 46029, 49868, 59986, 14361, 27678, 69635, 39589, 74599,

6587, 61176, 14191]

Módulo ***OS*** - *OS*é um módulo built-in usado para executar comandos no sistema operacional. Vamos explorar as funções:

*os.getcwd()*: retorna uma string com o caminho do diretório de trabalho.

*os.listdir(path='.')*: retorna uma lista com todas as entradas de um diretório. Se não for especificado um caminho, então a busca é realizada em outro diretório de trabalho.

*os.cpu\_count()*: retorna um inteiro com o número de CPUs do sistema.

*os.getlogin()*: retorna o nome do usário logado.

*os.getenv(key)*: retorna uma string com o conteúdo de uma variável de ambiente especificada na key.

*os.getpid()*: retorna o id do processo atual.

In [5]: import os

os.getcwd()

os.listdir()

os.cpu\_count()

os.getlogin()

os.getenv(key='path')

os.getpid()

Out[5]:6476

Módulo ***RE*** - O módulo *re*(regular expression) fornece funções para busca de padrões em um texto. Uma expressão regular especifica um conjunto de strings que corresponde a ela. As funções neste módulo permitem verificar se uma determinada string corresponde a uma determinada expressão regular. Essa técnica de programação é utilizada em diversas linguagens de programação, pois a construção de *re*depende do conhecimento de padrões. Vamos explorar as funções:

* *re.search(pattern, string, flags=0)*: varre a string procurando o primeiro local onde o padrão de expressão regular produz uma correspondência e o retorna. Retorna None se nenhuma correspondência é achada.
* *re.match(pattern, string, flags=0)*: procura por um padrão no começo da string. Retorna None se a sequência não corresponder ao padrão.
* *re.split(pattern, string, maxsplit=0, flags=0)*: divide uma string pelas ocorrências do padrão.

Para entendermos o funcionamento da expressão regular, vamos considerar um cenário onde temos um nome de arquivo com a data: meuArquivo\_20-01-2020.py. Nosso objetivo é guardar a parte textual do nome em uma variável para a usarmos posteriormente. Vamos utilizar os três métodos para fazer essa separação. O *search()*faz a procura em toda string, o *match()* faz a procura somente no começo (razão pela qual, portanto, também encontrará neste caso) e o *split()* faz a transformação em uma lista. Como queremos somente a parte textual, pegamos a posição 0 da lista.

In [6]: import re

string = 'meuArquivo\_20-01-2020.py'

padrao = "[a-zA-Z]\*"

texto1 = re.search(padrao, string).group()

texto2 = re.match(padrao, string).group()

texto3 = re.split("\_", string)[0]

print(texto1)

print(texto2)

print(texto3)

meuArquivo

meuArquivo

meuArquivo

Na linha 4, da entrada 6, construímos uma expressão regular para buscar por sequências de letras maiúsculas e minúsculas [a-zA-Z], que pode variar de tamanho 0 até N (\*). Nas linhas 6 e 7 usamos esse padrão para fazer a procura na string. Ambas as funções conseguiram encontrar; e, então, usamos a função *group()* da *re*para capturar o resultado. Na linha 8, usamos o padrão "\_" como a marcação de onde cortar a string, o que resulta em uma lista com dois valores – como o texto é a primeira parte, capturamos essa posição com o [0].

Módulo ***Datetime***- Trabalhar com datas é um desafio nas mais diversas linguagens de programação. Em Python há um módulo *built-in* capaz de lidar com datas e horas. O módulo datetime fornece classes para manipular datas e horas. Uma vez que esse módulo possui classes, então a sintaxe para acessar os métodos deve ser algo similar a: *modulo.classe.metodo()*. Dada a diversa quantidade de possibilidades de se trabalhar com esse módulo, vamos ver um pouco das classes *datetime*e *timedelta*.

In [7]: import datetime as dt

# Operações com data e hora

hoje = dt.datetime.today()

ontem = hoje - dt.timedelta(days=1)

uma\_semana\_atras = hoje - dt.timedelta(weeks=1)

agora = dt.datetime.now()

duas\_horas\_atras = agora - dt.timedelta(hours=2)

# Formatação

hoje\_formatado = dt.datetime.strftime(hoje, "%d-%m-%Y")

#https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior

ontem\_formatado = dt.datetime.strftime(ontem, "%d de %B de %Y")

# Converção de string para data

data\_string = '11/06/2019 15:30'

data\_dt = dt.datetime.strptime(data\_string, "%d/%m/%Y %H:%M")

Na entrada 7, usamos algumas funcionalidades disponíveis no módulo datetime. Repare que fizemos a importação com a utilização do apelido de dt, prática essa que é comum para nomes grandes.

Linha 4: usamos o método *today()* da classe datetime para capturar a data e a hora do sistema.

Linha 5: usamos a classe *timedelta*para subtrair 1 dia de uma data específica.

Linha 6: usamos a classe *timedelta*para subtrair 1 semana de uma data específica.

Linha 8: usamos o método *now()* da classe *datetime*para captura a data e hora do sistema.

Linha 9: usamos a classe *timedelta*para subtrair 2 horas de uma data específica.

Linhas 12 e 13: usamos o método *strftime()* da classe datetime para formatar a aparência de uma data específica. [Acesse o [endereço](https://docs.python.org/3/library/datetime.html" \l "strftime-strptime-behavior" \t "_blank)para verificar as possibilidades de formatação.]

Linha 17: usamos o método *strptime()* da classe datetime, para converter uma string em um objeto do tipo datetime. Essa transformação é interessante, pois habilita as operações que vimos.

Aproveite o [emulador](https://docs.python.org/3/py-modindex.html" \t "_blank)a seguir e teste os vários módulos que utilizamos até o momento. [Também é interessante acessar a documentação e explorar novas possibilidades.

**Módulo de terceiros**



Na documentação oficial da linguagem [Python](https://www.python.org/), você encontra, em um dos menus, a opção PyPI, que o levará para a [página](https://pypi.org/). *PyPI* é a abreviação para *Python Package Index*, que é um repositório para programas Python. Programadores autônomos e empresas podem, com isso, criar uma solução em Python e disponibilizar em forma de biblioteca no repositório PyPI, o que permite que todos usufruam e contribuam para o crescimento da linguagem. No próprio portal existe uma documentação explicando como distribuir sua solução com PyPI [em](https://bit.ly/3iNxbnt).

No momento em que este material está sendo produzido, o repositório PyPI conta com 237.653 projetos. Estamos falando de quase 300 mil bibliotecas prontas para usar. São tantas opções, que, se estudássemos uma biblioteca por dia, demoraríamos cerca de (237653 // 365) 651 anos para ver todas! Com tamanha diversidade, o caminho é, diante da necessidade de resolver um problema, buscar em fóruns e comunidades de programadores informações sobre bibliotecas que podem ser usadas para resolver o problema.

Para utilizar uma biblioteca do repositório PyPI, é preciso instalá-la. Para isso, abra um terminal no sistema operacional e digite: *pip install biblioteca* [*biblioteca*é o nome do pacote que deseja instalar. Por exemplo: *pip install numpy*.]

Como já deu para perceber, não é possível que consigamos estudar todas as bibliotecas. No entanto, vamos conhecer algumas delas. No dia a dia, exitem bibliotecas que têm sido amplamente utilizadas, como as para tratamento e visualização de dados, para implementações de inteligência artificial (*deep learning*e *machine learning*), para tratamento de imagens, para conexão com banco de dados, dentre outras. Veja algumas a seguir:

* **Bibliotecas para tratamento de imagens**
  + - **Pillow**: esta biblioteca oferece amplo suporte aos formatos de arquivo, uma representação interna eficiente e recursos de processamento de imagem bastante poderosos.
    - **OpenCV Python**: é uma biblioteca de código aberto licenciada por BSD que inclui várias centenas de algoritmos de visão computacional.
    - **Luminoth**: é um *kit*de ferramentas de código aberto para visão computacional. Atualmente, atua com a detecção de objetos, mas a ideia é expandi-la.
    - **Mahotas**: é uma biblioteca de algoritmos rápidos de visão computacional (todos implementados em C++ para ganhar velocidade) que opera com matrizes *NumPy.*

1. **Bibliotecas para visualização de dados**  
     
   * **Matplotlib**: é uma biblioteca abrangente para criar visualizações estáticas, animadas e interativas em Python.
   * **Bokeh**: é uma biblioteca de visualização interativa para navegadores modernos. Oferece interatividade de alto desempenho em conjuntos de dados grandes ou de streaming.
   * **Seaborn**: é uma biblioteca para criar gráficos estatísticos em Python.
   * Altair: é uma biblioteca declarativa de visualização estatística para Python.
2. **Bibliotecas para tratamento de dados**  
     
   * **Pandas**: é um pacote Python que fornece estruturas de dados rápidas, flexíveis e expressivas, projetadas para facilitar o trabalho com dados estruturados (em forma de tabela).
   * **NumPy**: além de seus óbvios usos científicos, a NumPy também pode ser usada como um eficiente recipiente multidimensional de dados genéricos.
   * **Pyspark**: Spark é um sistema de computação em cluster rápido e geral para Big Data.
   * **Pingouin**: é um pacote estatístico Python baseado em Pandas.
3. **Bibliotecas para tratamento de textos**  
     
   * **Punctuation**: esta é uma biblioteca Python que removerá toda a pontuação em uma string.
   * **NLTK**: o *Natural Language Toolkit*é um pacote Python para processamento de linguagem natural.
   * **FlashText**: este módulo pode ser usado para substituir palavras-chave em frases ou extraí-las.
   * **TextBlob**: é uma biblioteca Python para processamento de dados textuais.
4. **Internet, rede e cloud**  
     
   * **Requests**: permite que você envie solicitações HTTP/1.1 com extrema facilidade. Não há necessidade de adicionar manualmente *queries*de consulta aos seus URLs ou de codificar os dados PUT e POST: basta usar o método JSON.
   * **BeautifulSoup**: é uma biblioteca que facilita a captura de informações de páginas da web.
   * **Paramiko**: é uma biblioteca para fazer conexões SSH2 (cliente ou servidor). A ênfase está no uso do SSH2 como uma alternativa ao SSL para fazer conexões seguras entre scripts Python.
   * **s3fs**: é uma interface de arquivos Python para S3 (Amazon Simple Storage Service).
5. **Bibliotecas para acesso a bancos de dados**  
     
   * **mysql-connector-python**: permite que programas em Python acessem bancos de dados MySQL.
   * **cx-Oracle**: permite que programas em Python acessem bancos de dados Oracle.
   * **psycopg2**: permite que programas em Python acessem bancos de dados PostgreSQL.
   * **SQLAlchemy**: fornece um conjunto completo de padrões de persistência, projetados para acesso eficiente e de alto desempenho a diversos banco de dados, adaptado para uma linguagem de domínio simples e Python.
6. **Deep learning - Machine learning**  
     
   * **Keras**: é uma biblioteca de rede neural profunda de código aberto.
   * **TensorFlow**: é uma plataforma de código aberto de ponta a ponta para aprendizado de máquina, desenvolvido originalmente pela Google.
   * **PyTorch**: é um pacote Python que fornece dois recursos de alto nível: i) computação de tensor (como NumPy) com forte aceleração de GPU; e ii) redes neurais profundas.
   * **Scikit Learn**: módulo Python para aprendizado de máquina construído sobre o SciPy (SciPy é um software de código aberto para matemática, ciências e engenharia).
7. **Biblioteca para jogos - PyGame**  
     
   * **PyGame**: é uma biblioteca para a construção de aplicações gráficas e aplicação multimídia, utilizada para desenvolver jogos.

Além dessas categorias e bibliotecas citadas, como você já sabe, existem inúmeras outras. Você, como profissional desenvolvedor, deve buscá-las e estudar aquelas da área em que deseja atuar. A grande vantagem de usar bibliotecas é que elas encapsulam a complexidade de uma determinada tarefa, razão pela qual, com poucas linhas de códigos, conseguimos realizar tarefas complexas.

Aprenderemos a trabalhar com banco de dados em uma outra aula e teremos uma unidade inteira dedicada ao estudo da biblioteca Pandas. Para conhecermos um pouco do poder das bibliotecas em Python, vamos falar um pouco sobre o pacote *requests*.

**Biblioteca requests**



A biblioteca requests habilita funcionalidades do protocolo HTTP, como o get e o post. Dentre seus métodos, o *get()* é o responsável por capturar informação da internet. A documentação sobre ela está disponível no [endereço](https://requests.readthedocs.io/pt_BR/latest/" \t "_blank). Essa biblioteca foi construída com o intuito de substituir o módulo *urllib2*, que demanda muito trabalho para obter os resultados. O método *get()*permite que você informe a URL de que deseja obter informação. Sua sintaxe é: requests.get('https://XXXXXXX'). Para outros parâmetros dessa função, como autenticação, cabeçalhos, etc., consulte a documentação.

Observe o código a seguir. Na entrada 8, importamos a biblioteca *requests*e, na linha 3, usamos o método *get()*para capturar um conteúdo de uma API do github e guardar na variável *info*. Ao fazer uma requisição podemos olhar algumas informações da extração pela propriedade *headers*.

In [8]: import requests

info **=** requests**.**get('https://api.github.com/events')

info**.**headers

Out [8]: {'date': 'Thu, 04 Jun 2020 22:09:33 GMT', 'content-type': 'application/json; charset=utf-8', 'server': 'GitHub.com', 'status': '200 OK', 'cache-control': 'public, max-age=60, s-maxage=60', 'vary': 'Accept, Accept-Encoding, Accept, X-Requested-With, Accept-Encoding', 'etag': 'W/"078bb18598ef42449c62d7d39a8f303a"', 'last-modified': 'Thu, 04 Jun 2020 22:04:33 GMT', 'x-poll-interval': '60', 'x-github-media-type': 'github.v3; format=json', 'link': '<https://api.github.com/events?page=2>; rel="next", <https://api.github.com/events?page=10>; rel="last"', 'access-control-expose-headers': 'ETag, Link, Location, Retry-After, X-GitHub-OTP, X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-GitHub-Media-Type, Deprecation, Sunset', 'access-control-allow-origin': '\*', 'strict-transport-security': 'max-age=31536000; includeSubdomains; preload', 'x-frame-options': 'deny', 'x-content-type-options': 'nosniff', 'x-xss-protection': '1; mode=block', 'referrer-policy': 'origin-when-cross-origin, strict-origin-when-cross-origin', 'content-security-policy': "default-src 'none'", 'content-encoding': 'gzip', 'X-Ratelimit-Limit': '60', 'X-Ratelimit-Remaining': '58', 'X-Ratelimit-Reset': '1591310892', 'Accept-Ranges': 'bytes', 'Transfer-Encoding': 'chunked', 'X-GitHub-Request-Id': 'EAF6:7629:700E8:A32A6:5ED9711D'}

In [9]: print(info**.**headers['date']) # Data de extração

print(info**.**headers['server']) # Servidor de origem

print(info**.**headers['status']) # Status HTTP da extração, 200 é ok

print(info**.**encoding) # Encoding do texto

print(info**.**headers['last-modified']) # Data da última modificação da informação

Thu, 04 Jun 2020 22:09:33 GMT

GitHub.com

200 OK

utf-8

Thu, 04 Jun 2020 22:04:33 GMT

A propriedade *headers*retorna um dicionário de informações. Veja que, na entrada 9, extraímos algumas informações dessa propriedade. Na linha 1, acessamos a data de extração; na linha 2, o servidor que foi acessado; na linha 3, o status da extração; na linha 4, a decodificação texto; e na linha 5, a data da última modificação da informação no servidor. Veja que essas informações podem ser usadas em um relatório!

Para acessar o conteúdo que foi extraído, podemos usar a propriedade *text*, que converte todo o conteúdo para uma string, ou então o método *json()*, que faz a conversão para uma lista de dicionários. Observe o código a seguir. Na entrada 10, temos o conteúdo como uma string e, na entrada 11, o conteúdo como uma lista de dicionários. Nesse caso, bastaria usar a linguagem Python para fazer os devidos tratamentos e extrair as informações!

In [10]: texto\_str **=** info**.**text

print(type(texto\_str))

texto\_str[:100] # exibe somente os 100 primeiros caracteres

<class 'str'>

Out [10]:

'[{"id":"12535753096","type":"PushEvent","actor":{"id":5858581,"login":"grahamegrieve","display\_login'

In [11]: texto\_json **=** info**.**json()

print(type(texto\_json))

Texto\_json[0]

<class 'list'>

Out [11]: {'id': '12535753096',

 'type': 'PushEvent',

 'actor': {'id': 5858581,

  'login': 'grahamegrieve',

  'display\_login': 'grahamegrieve',

  'gravatar\_id': '',

  'url': 'https://api.github.com/users/grahamegrieve',

  'avatar\_url': 'https://avatars.githubusercontent.com/u/5858581?'},

 'repo': {'id': 178767413,

  'name': 'HL7/fhir-ig-publisher',

  'url': 'https://api.github.com/repos/HL7/fhir-ig-publisher'},

 'payload': {'push\_id': 5179281979,

  'size': 1,

  'distinct\_size': 1,

  'ref': 'refs/heads/master',

  'head': 'c76e1dbce501f23988ad4d91df705942ca9b978f',

  'before': '3c6af8f114145351d82d36f506093a542470db0a',

  'commits': [{'sha': 'c76e1dbce501f23988ad4d91df705942ca9b978f',

    'author': {'email': 'grahameg@gmail.com', 'name': 'Grahame Grieve'},

    'message': '\* add -no-sushi command',

    'distinct': True,

    'url': 'https://api.github.com/repos/HL7/fhir-ig-publisher/commits/c76e1dbce501f23988ad4d91df705942ca9b978f'}]},

 'public': True,

 'created\_at': '2020-06-04T22:04:33Z',

 'org': {'id': 21250901,

  'login': 'HL7',

  'gravatar\_id': '',

  'url': 'https://api.github.com/orgs/HL7',

  'avatar\_url': 'https://avatars.githubusercontent.com/u/21250901?'}}

\_\_\_\_\_\_\_

**📝Exemplificando**

Vamos utilizar a biblioteca *requests*para extrair informações da Copa do Mundo de Futebol Feminino, que aconteceu no ano de 2019. As informações estão disponíveis no [endereço](http://worldcup.sfg.io/matches), no formato chave:valor. Após extrair as informações, vamos gerar um relatório que contém informações de cada jogo no seguinte formato: (dia/mes/ano) - time 1 x time 2 = gols time 1 a gols time 2. Então, vamos lá!

In [12]: # primeiro passo extrair as informações com o request utilizando o método json().

import requests

import datetime as dt

jogos **=** requests**.**get('http://worldcup.sfg.io/matches')**.**json()

print(type(jogos))

<class 'list'>

Na entrada 12, foi realizada a extração com o *requests*e já convertemos o conteúdo para *json()*. Observe que, como resultado, temos uma lista de dicionários. Na linha 7, estamos extraindo as informações do primeiro dicionário da lista, ou seja, as informações do primeiro jogo. Nossa missão é criar uma lógica que extraia as informações de cada jogo, conforme solicitado, e gere um relatório. Então vamos a lógica para extrair.

In [13]: # segundo passo: percorrer cada dicionário da lista (ou seja, cada jogo) extraindo as informações

info\_relatorio **=** []

file **=** open('relatorio\_jogos.txt', "w") # cria um arquivo txt na pasta em que está trabalhando.

for jogo in jogos:

    data **=** jogo['datetime'] # extrai a data

    data **=** dt**.**datetime**.**strptime(data, "%Y-%m-%dT%H:%M:%SZ") # converte de string para data

    data **=** data**.**strftime("%d/%m/%Y") # formata

        nome\_time1 **=** jogo['home\_team\_country']

    nome\_time2 **=** jogo['away\_team\_country']

    gols\_time1 **=** jogo['home\_team']['goals']

    gols\_time2 **=** jogo['away\_team']['goals']

        linha **=** f"({data}) - {nome\_time1} x {nome\_time2} = {gols\_time1} a {gols\_time2}"

    file**.**write(linha **+** '\n') # escreve a linha no arquivo txt

    info\_relatorio**.**append(linha)

file**.**close() # é preciso fechar o arquivo

info\_relatorio[:5]

Out [13]: ['(07/06/2019) - France x Korea Republic = 4 a 0',

 '(08/06/2019) - Germany x China PR = 1 a 0',

 '(08/06/2019) - Spain x South Africa = 3 a 1',

 '(08/06/2019) - Norway x Nigeria = 3 a 0',

 '(09/06/2019) - Brazil x Jamaica = 3 a 0']

Na entrada 13, usamos uma estrutura de repetição para percorrer cada item do dicionário, extraindo as informações. Chamamos a atenção para as linhas 13 e 14, nas quais usamos duas chaves. Isso foi feito porque, dentro do dicionário, existe outro dicionário. Veja: 'home\_team': {**'country': 'France', 'code': 'FRA', 'goals': 4, 'penalties': 0**}. Dentro da chave *home\_team* existe um outro dicionário. Portanto, para acessar os gols, precisamos também acessar a chave interna *goals*, ficando então **jogo['home\_team']['goals']**.

Na entrada 13, usamos a função *built-in open()* para criar um arquivo chamado *relatorio\_jogos.txt*, no qual escreveremos informações – por isso o parâmetro "w". Na linha 18, escrevemos cada linha gerada no arquivo, concatenando com uma nova linha "\n" a cada informação gerada. Como passamos somente o nome do arquivo, ele será gerado na pasta onde estiver trabalhando.

**Matplotlib**



Matplotlib é uma biblioteca com funcionalidades para criar gráficos, cuja documentação está disponível no [endereço](https://matplotlib.org/" \t "_blank).  É composta por uma série de exemplos. Vamos utilizar a interface Pyplot para criar um gráfico simples baseado nas informações que salvamos sobre os jogos da Copa do Mundo de Futebol Feminino de 2019.

Fizemos a extração e criamos um relatório, salvando-o como *relatorio\_jogos.txt. A*gora vamos ler os dados que foram persistidos no arquivo, extrair somente as datas no formado *dd/mm* e contabilizar quantos jogos aconteceram em cada data. Em seguida, vamos usar o Pyplot para construir esse gráfico de contagem.

In [14]: # ler o arquivo salvo

file **=** open('relatorio\_jogos.txt', 'r')

print('file = ', file, '\n')

info\_relatorio **=** file**.**readlines()

file**.**close()

print("linha 1 = ", info\_relatorio[0])

file =  <\_io.TextIOWrapper name='relatorio\_jogos.txt' mode='r' encoding='cp1252'>

linha 1 =  (07/06/2019) - France x Korea Republic = 4 a 0

Para ler um arquivo, usamos a função *built-in open()*, passando como parâmetros o nome do arquivo e a opção 'r', que significa que queremos abrir o arquivo em modo leitura (r = read). A função *open()* retorna um objeto do tipo "\_io.TextIOWrapper", conforme podemos observar pelo print na linha 3. Para acessar o conteúdo do arquivo, precisamos usar a função*readlines()*, que cria uma lista, na qual cada elemento é uma linha do arquivo.

Após a criação da lista, podemos fechar o arquivo. Na linha 7, imprimimos o primeiro item da lista criada, que corresponde à primeira linha do arquivo de que fizemos a leitura. Para cada linha, queremos somente a parte que corresponde ao dia e mês: dd/mm, razão pela qual vamos criar uma nova lista que contém somente essas datas. Observe o código a seguir.

In [15]: # Extrair somente a parta 'dd/mm' da linha

datas **=** [linha[1:6] for linha in info\_relatorio]

print(sorted(datas))

['02/07', '03/07', '06/07', '07/06', '07/07', '08/06', '08/06', '08/06', '09/06', '09/06', '09/06', '10/06', '10/06', '11/06', '11/06', '11/06', '12/06', '12/06', '12/06', '13/06', '13/06', '14/06', '14/06', '14/06', '15/06', '15/06', '16/06', '16/06', '17/06', '17/06', '17/06', '17/06', '18/06', '18/06', '19/06', '19/06', '20/06', '20/06', '20/06', '20/06', '22/06', '22/06', '23/06', '23/06', '24/06', '24/06', '25/06', '25/06', '27/06', '28/06', '29/06', '29/06']

Agora temos uma lista com todas as datas dos jogos e precisamos contar quantas vezes cada uma aparece. Assim vamos ter a quantidade de jogos por dia. Para fazer esse trabalho, vamos utilizar a operação *count()*disponível para os objetos do tipo sequência: sequencia.count(valor), que retorna quantas vezes o valor aparece na sequência.

Então, se fizermos datas.count('08/06'), temos que obter o valor 3. Uma vez que ´precisamos fazer isso para todas as datas, vamos, então, usar uma list comprehension para fazer essa iteração. Para cada data, vamos gerar uma tupla com dois valores: (data, count). Em nossa lista final, queremos ter uma linha para cada data, exemplo: ('08/06', 3). Então, para remover as duplicações, vamos utilizar o construtor*set()*. Observe o código a seguir:

In [16]:  datas\_count **=** [(data, datas**.**count(data)) for data in set(datas)]

print(datas\_count)

 [('17/06', 4), ('22/06', 2), ('24/06', 2), ('08/06', 3), ('07/06', 1), ('18/06', 2), ('03/07', 1), ('07/07', 1), ('27/06', 1), ('28/06', 1), ('25/06', 2), ('09/06', 3), ('20/06', 4), ('13/06', 2), ('12/06', 3), ('19/06', 2), ('11/06', 3), ('14/06', 3), ('16/06', 2), ('10/06', 2), ('29/06', 2), ('02/07', 1), ('23/06', 2), ('15/06', 2), ('06/07', 1)]

Com o passo anterior, temos uma lista de tuplas com a data e quantidade de jogos! Por uma questão de conveniência, vamos transformar essa lista em um dicionário usando o construtor dict(). Veja a seguir.

In [17]: datas\_count **=** dict(datas\_count)

print(datas\_count)

{'17/06': 4, '22/06': 2, '24/06': 2, '08/06': 3, '07/06': 1, '18/06': 2, '03/07': 1, '07/07': 1, '27/06': 1, '28/06': 1, '25/06': 2, '09/06': 3, '20/06': 4, '13/06': 2, '12/06': 3, '19/06': 2, '11/06': 3, '14/06': 3, '16/06': 2, '10/06': 2, '29/06': 2, '02/07': 1, '23/06': 2, '15/06': 2, '06/07': 1}

Essa transformação da lista para dicionário nos permite extrair as chaves (que são as datas) e os valores (que são as quantidades). Esses dois itens serão usados nos eixos *x* e *y* do gráfico.

Agora que já preparamos os dados, vamos utilizar a interface Pyplot da biblioteca *matplotlib* para criar nosso gráfico. Para que possamos ver um gráfico dentro de um notebook, temos que habilitar a opção %matplotlib inline e importar a biblioteca. Fazemos isso nas linhas 1 e 2 da entrada 18. Nas linhas 4 e 5, usamos as informações do nosso dicionário para definir os dados que serão usados nos eixos*x* e *y*. Na linha 7, configuramos o tamanho do nosso gráfico. Nas linhas 8 e 9 definimos os rótulos dos eixos; na linha 10, configuramos uma rotação para as datas que vão aparecer no eixo *x*; a linha 12 é onde de fato criamos o gráfico, escolhendo a opção de barras (bar) e passando as informações a serem plotadas. Na linha 14, usamos a função *show*para exibir nosso gráfico.

In [18]: **%**matplotlib inline

import matplotlib.pyplot as plt

eixo\_x **=** datas\_count**.**keys()

eixo\_y **=** datas\_count**.**values()

plt**.**figure(figsize**=**(15, 5))

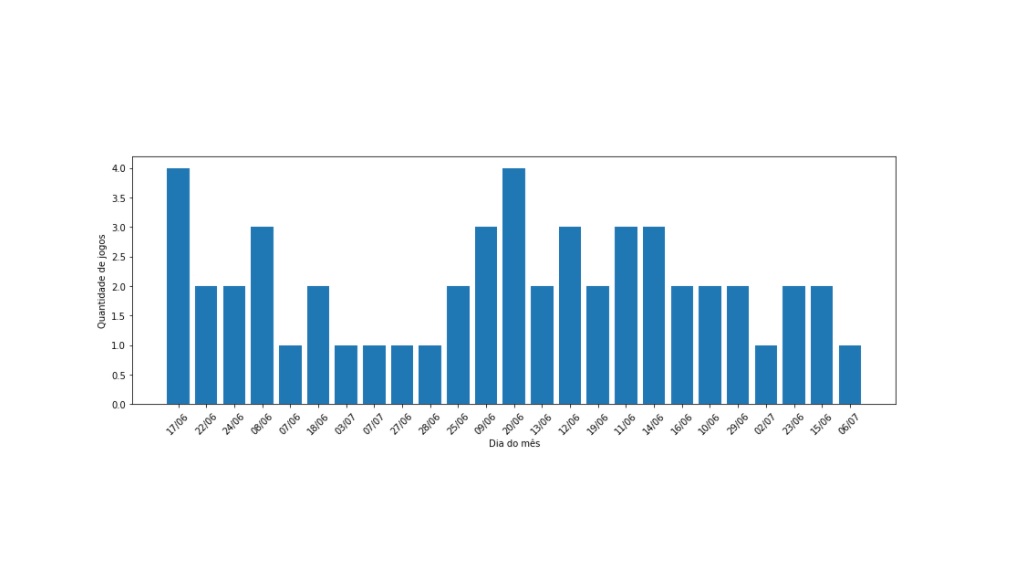
plt**.**xlabel('Dia do mês')

plt**.**ylabel('Quantidade de jogos')

plt**.**xticks(rotation**=**45)

plt**.**bar(eixo\_x, eixo\_y)

plt**.**show()

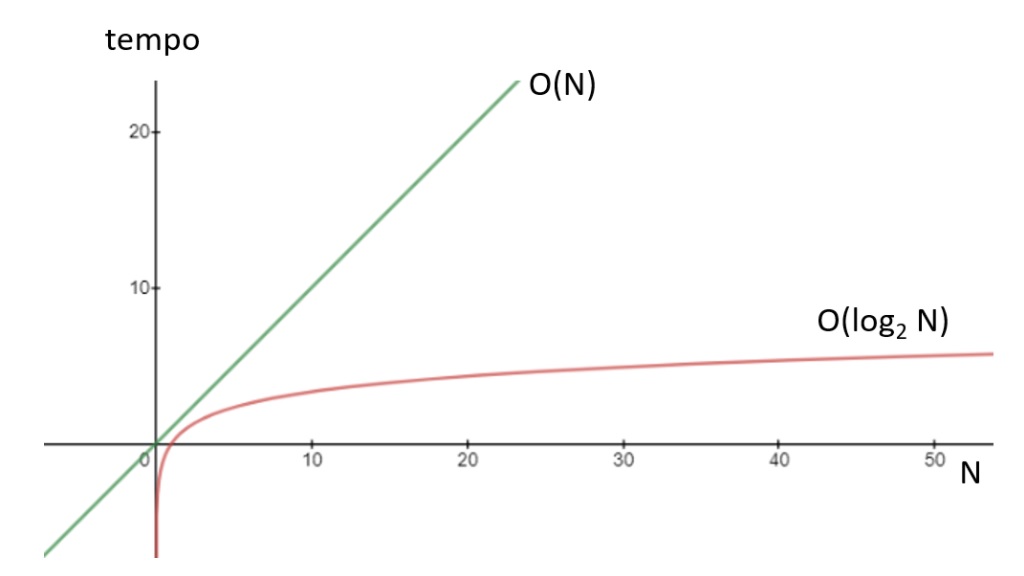


A quantidade de funcionalidades embutidas em uma biblioteca traz um universo de possibilidades ao desenvolvedor. Consulte sempre a documentação e os fóruns para acompanhar as novidades.

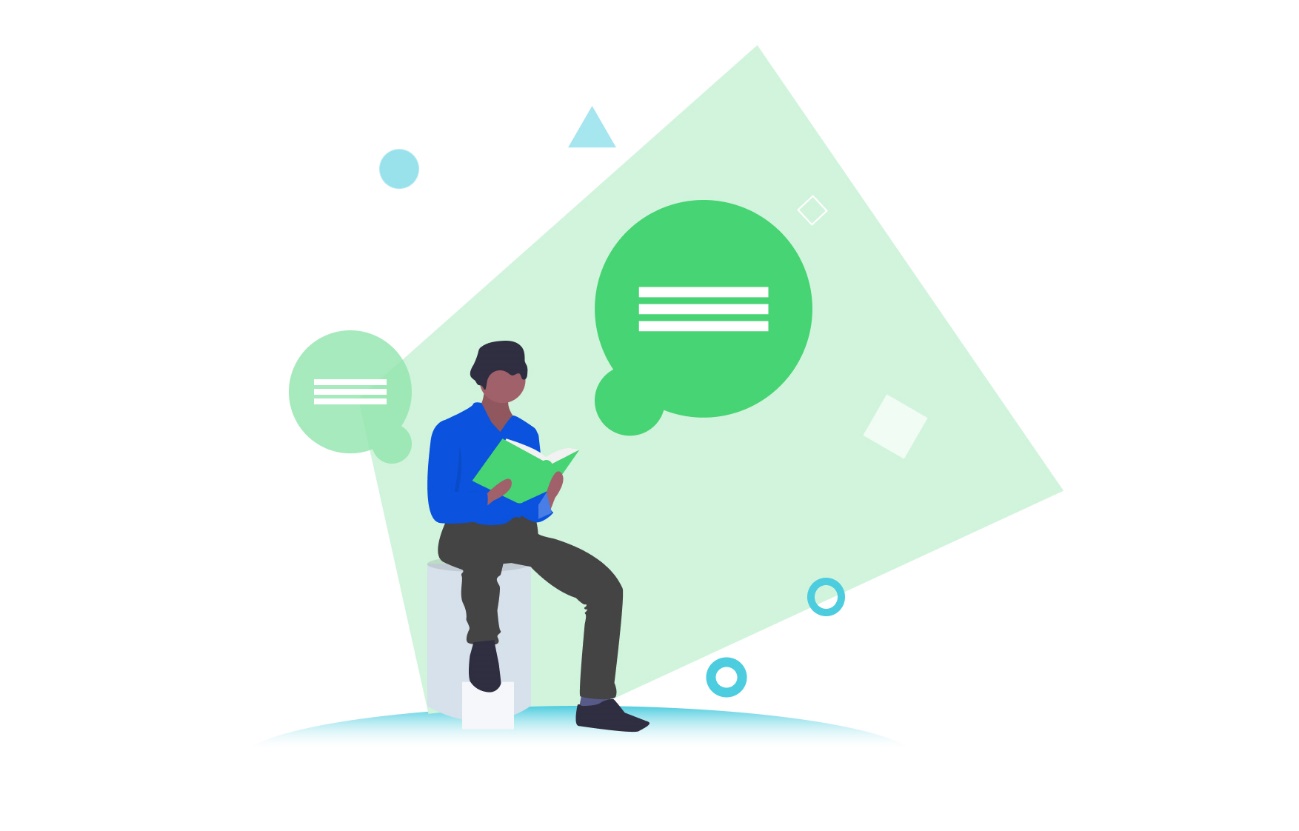
**Módulos próprios** - Os códigos podem ser organizados em diversos arquivos com extensão *.py*, ou seja, em módulos. Cada módulo pode importar outros módulos, tanto os pertencentes ao mesmo projeto, como os*built-in* ou de terceiros. A figura abaixo, ilustra a modularidade em Python. Criamos um módulo (arquivo Python) chamado *utils.py.* Esse módulo possui uma função que cria uma conexão *ssh*com um determinado servidor. Podemos entender um cliente *SSH*como um túnel de comunicação. Veja que no módulo precisamos usar a biblioteca *paramiko*para construir essa conexão.

A função *create\_ssh\_client* retorna um client, ou seja, a conexão em si. Em um outro módulo, chamado principal, importamos a função do módulo *utils*. É dentro do módulo principal que vamos utilizar a funcionalidade de conexão para copiar um arquivo que está em um servidor para outro local. **É importante ressaltar que, da forma pela qual fizemos a importação, ambos os arquivos .*py*precisam estar no mesmo nível de pasta**.

Se precisarmos usar o módulo *utils*em vários projetos, é interessante transformá-lo em uma biblioteca e disponibilizá-la via PyPI.

Módulo em Python. Fonte: elaborada pela autora.

**Conclusão**



Para automatizar o processo de extração dos dados do CNAE e gerar o relatório, vamos ter de usar bibliotecas. Para fazer a extração dos dados do CNAE, podemos usar a biblioteca *requests*. Para responder às perguntas, vamos precisar manipular listas e dicionários. Então vamos começar pela extração.

In [19]: import requests

dados **=** requests**.**get('https://servicodados.ibge.gov.br/api/v2/cnae/classes')**.**json() # resulta em uma lista de diconários

dados[0] # exibindo o primeiro registro de dados (primeiro dicionário da lista)

Out [19]: {'id': '01113',

 'descricao': 'CULTIVO DE CEREAIS',

 'grupo': {'id': '011',

  'descricao': 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

  'divisao': {'id': '01',

   'descricao': 'AGRICULTURA, PECUÁRIA E SERVIÇOS RELACIONADOS',

   'secao': {'id': 'A',

    'descricao': 'AGRICULTURA, PECUÁRIA, PRODUÇÃO FLORESTAL, PESCA E AQÜICULTURA'}}},

 'observacoes': ['Esta classe compreende - o cultivo de alpiste, arroz, aveia, centeio, cevada, milho, milheto, painço, sorgo, trigo, trigo preto, triticale e outros cereais não especificados anteriormente',

  'Esta classe compreende ainda - o beneficiamento de cereais em estabelecimento agrícola, quando atividade complementar ao cultivo\r\n- a produção de sementes de cereais, quando atividade complementar ao cultivo',

  'Esta classe NÃO compreende - a produção de sementes certificadas dos cereais desta classe, inclusive modificadas geneticamente (01.41-5)\r\n- os serviços de preparação de terreno, cultivo e colheita realizados sob contrato (01.61-0)\r\n- o beneficiamento de cereais em estabelecimento agrícola realizado sob contrato (01.63-6)\r\n- o processamento ou beneficiamento de cereais em estabelecimento não-agrícola (grupo 10.4) e (grupo 10.6)\r\n- a produção de biocombustível (19.32-2)']}

Agora que temos os dados guardados em uma lista de dicionários, podemos usar a função *built-in len()*para saber quantos elementos essa lista tem. Esse resultado será a quantidade de dicionários que representa a quantidade distintas de atividades.

In [20]: # Quantidade distintas de atividades, basta saber o tamanho da lista.

qtde\_atividades\_distintas **=** len(dados)

Para saber quantos grupos de atividades existem e já começar a preparar os dados para os próximos passos, vamos criar uma lista que percorre cada registro e extrai a informação do grupo. Dado um registro, essa informação está na chave interna 'descricao' da chave externa 'grupo'. Logo, para acessar, temos que usar a sintaxe: *dicionario['chave\_externa']['chave\_interna']*. Na entrada 21, criamos uma lista vazia na linha 3 e, dentro da estrutura de repetição, vamos extraindo a informação e guardando-a na lista.

In [21]: # Criar uma lista dos grupos de atividades, extraindo a descrição de cada registro

grupos **=** []

for registro in dados:

    grupos**.**append(registro['grupo']['descricao'])

grupos[:10]

Out [21]: ['PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'PRODUÇÃO DE LAVOURAS TEMPORÁRIAS',

 'HORTICULTURA E FLORICULTURA',

 'HORTICULTURA E FLORICULTURA',

 'EXTRAÇÃO DE MINERAIS METÁLICOS NÃO-FERROSOS']

Agora que temos uma lista com todos os grupos, para saber quantos grupos distintos existem, basta eliminar as duplicações e fazer a contagem. Na entrada 22, usamos o construtor *set()*para criar um conjunto de dados, sem repetições e sem alterar a lista com todos, uma vez que ainda vamos utilizá-la. O resultado do *set()* fazemos a contagem com a função *len()*, obtendo, então, a quantidade de grupos distintos.

In [22]: # A partir da lista, podemos extrair a quantidade de grupos de atividades

qtde\_grupos\_distintos **=** len(set(grupos)) # o construtor set cria uma estrutura de dados removendo as duplicações.

Agora vamos contar quantas atividades estão cadastradas em cada grupo. O código na entrada 23 faz esse trabalho. Usamos uma list comprehension para criar uma lista de tuplas. Cada tupla vai conter o grupo e a contagem de quantas vezes esse grupo aparece na lista de grupos: (grupo, grupos.count(grupo). Isso será feito para cada grupo distinto: for grupo in set(grupos).

In [23]: # Resultado é uma lista de tuplas. Cria uma nova lista com o grupo e a quantidade de atividades pertencentes a ele

grupos\_count **=** [(grupo, grupos**.**count(grupo)) for grupo in set(grupos)]

grupos\_count[:5]

Out [23]: [('TECELAGEM, EXCETO MALHA', 3),

 ('COMÉRCIO ATACADISTA DE PRODUTOS DE CONSUMO NÃO-ALIMENTAR', 8),

 ('ATIVIDADES DE ORGANIZAÇÕES ASSOCIATIVAS PATRONAIS, EMPRESARIAIS E PROFISSIONAIS',

  2),

 ('SEGURIDADE SOCIAL OBRIGATÓRIA', 1),

 ('FABRICAÇÃO DE ELETRODOMÉSTICOS', 2)]

Para sabermos qual grupo ou quais grupos possuem o maior número de atividades vinculadas, vamos transformar a lista de tuplas em um dicionário.

In [24]: # Por conveniência, transformamos a lista em um dicionário

grupos\_count **=** dict(grupos\_count)

Agora podemos criar uma nova lista que contém todos os grupos que possuem a contagem com o mesmo valor da quantidade máxima que encontramos. Ao usar dicionário, conseguimos acessar a chave e o valor, o que facilita o trabalho.

In [25]: # A partir do dicionário vamos descobrir qual (ou quais) grupos possuem mais atividades

valor\_maximo **=** max(grupos\_count**.**values())

grupos\_mais\_atividades **=** [chave for (chave, valor) in grupos\_count**.**items() if valor **==** valor\_maximo]

print(len(grupos\_mais\_atividades))

grupos\_mais\_atividades

1

Out [25]: ['REPRESENTANTES COMERCIAIS E AGENTES DO COMÉRCIO, EXCETO DE VEÍCULOS AUTOMOTORES E MOTOCICLETAS']

Para formalizar a entrega do componente de extração, que tal criar uma classe e um método com todo o código? Assim, quando for preciso extrair, basta instanciar a classe e invocar o método.

In [26]: import requests

from datetime import datetime

class ETL:

    def \_\_init\_\_(self):

        self**.**url **=** None

    def extract\_cnae\_data(self, url):

        self**.**url **=** url

        data\_extracao **=** datetime**.**today()**.**strftime("%Y/%m/%d - %H:%M:%S")

        # Faz extração

        dados **=** requests**.**get(self**.**url)**.**json()

         # Extrai os grupos dos registros

        grupos **=** []

        for registro in dados:

            grupos**.**append(registro['grupo']['descricao'])

        # Cria uma lista de tuplas (grupo, quantidade\_atividades)

        grupos\_count **=** [(grupo, grupos**.**count(grupo)) for grupo in set(grupos)]

        grupos\_count **=** dict(grupos\_count)  # transforma a lista em dicionário

        valor\_maximo **=** max(grupos\_count**.**values())  # Captura o valor máximo de atividades

        # Gera uma lista com os grupos que possuem a quantidade máxima de atividades

        grupos\_mais\_atividades **=** [chave for (chave, valor) in grupos\_count**.**items() if valor **==** valor\_maximo]

        # informações

        qtde\_atividades\_distintas **=** len(dados)

        qtde\_grupos\_distintos **=** len(set(grupos))

        print(f"Dados extraídos em: {data\_extracao}")

        print(f"Quantidade de atividades distintas = {qtde\_atividades\_distintas}")

        print(f"Quantidade de grupos distintos = {qtde\_grupos\_distintos}")

        print(f"Grupos com o maior número de atividades = {grupos\_mais\_atividades}, atividades = {valor\_maximo}")

        return None

In [27]: # Usando a classe ETL

ETL()**.**extract\_cnae\_data('https://servicodados.ibge.gov.br/api/v2/cnae/classes')

Dados extraídos em: 2020/06/04 - 19:09:39

Quantidade de atividades distintas = 673

Quantidade de grupos distintos = 285

Grupos com o maior número de atividades = ['REPRESENTANTES COMERCIAIS E AGENTES DO COMÉRCIO, EXCETO DE VEÍCULOS AUTOMOTORES E MOTOCICLETAS'], atividades = 9