NÃO PODE FALTAR

BIBLIOTECAS E MÓDULOS EM PYTHON

Vanessa Cadan Scheffer

O QUE SÃO MÓDULOS?

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

Fonte: Shutterstock.

Deseja ouvir este material?

INTRODUÇÃO

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, passar 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 3.6 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.

Figura 3.6 | Contínuo *versus* módulo

```python
def fib(n): # escreve a sequência de Fibonacci até n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # retorno da sequência de Fibonacci até n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result

def hello():
    print('hello world')
# Módulos separados
import fib
import fib2
import hello
```

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 3.7, 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.

Figura 3.7a | Módulo como biblioteca

```python
from math import log # importando a função log do módulo math
log(100, 10) # retorna o resultado do log de 100 na base 10
```

Fonte: elaborada pela autora.
Figura 3.7b | Módulo como biblioteca

```
# Importando o módulo os (sistema operacional).
import os

#Retorna uma string representando o diretório de trabalho atual.
cwd = os.getcwd()
```

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`
2. `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 usá-la, 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.

BOA PRÁTICA

Todos os *import* devem ficar no começo do arquivo (https://bit.ly/2XWFFjR). Ainda segundo a documentação do site, 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://bit.ly/2FgUxmJ você encontra a lista com todos os recursos disponíveis. Vamos explorar alguns deles.

MÓDULO ***RANDOM***

Random é um módulo *built-in* usado para criar número 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://bit.ly/2E33mzR 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 a seguir e teste os vários módulos que utilizamos até o momento. [Também é interessante acessar a documentação e explorar novas possiblidades: https://bit.ly/3ajIgcJ.]

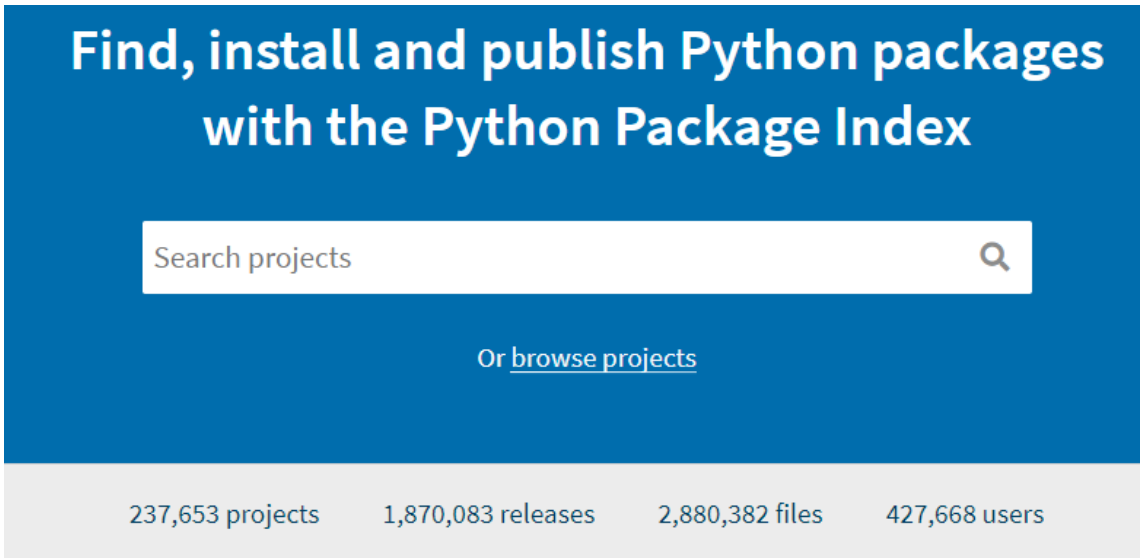
MÓDULOS DE TERCEIROS

Na documentação oficial da linguagem Python (https://www.python.org/), você encontra, em um dos menus, a opção PyPI, que p 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, conforme mostra a Figura 3.8. 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*.]

Figura 3.8 | Repositório PyPI



Fonte: Pypi. Disponível em: https://pypi.org/.

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:

1. **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.*
2. **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.
3. **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.
4. **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.

5. **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).
6. **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.
7. **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).
8. **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 procotolo 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/. 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/. 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.

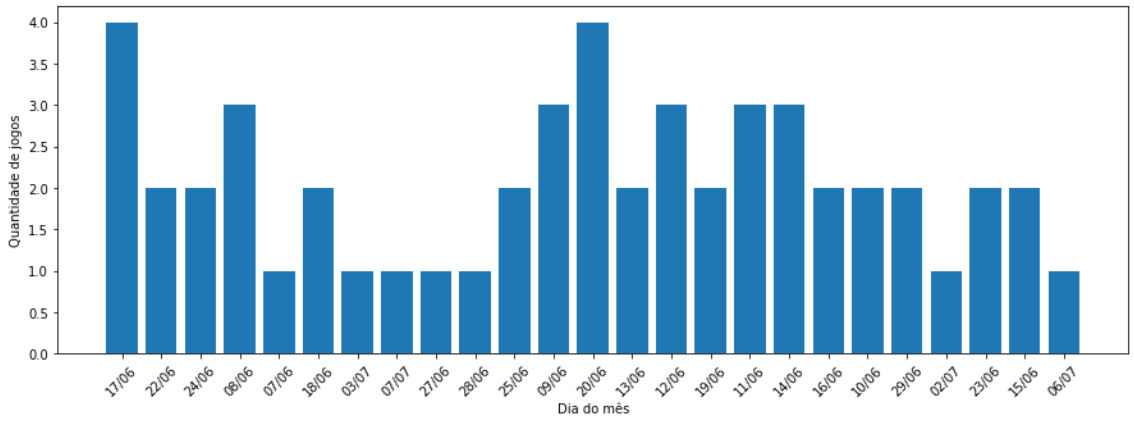
In [18]:
```
%matplotlib inline
import matplotlib.pyplot as plt

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()
```



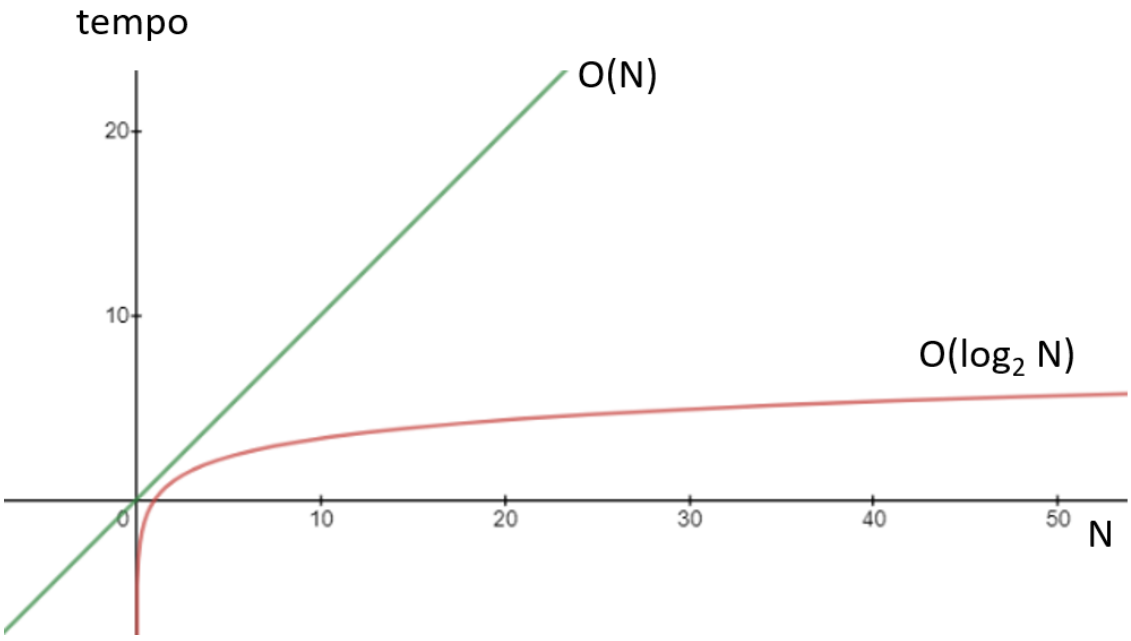
A quantidade de funcionalidades embutidas em uma biblioteca traz um universo de possibilidades ao desenvolvedor. Consulte sempre a documentação e os fóruns para acompanhar as novidades.

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 3.9, 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.

Figura 3.9 | Módulo em Python



Fonte: elaborada pela autora.

Para finalizar, vamos aprender como transformar módulos em scripts que podem ser chamados via linha de comando. Ao criar um arquivo *.py*, pode-se utilizar o terminal do sistema operacional para executá-lo; por exemplo: `python meu_programa.py`. Para que esse programa de fato execute, ele precisa ter a variável *__name__* com valor *"__main__"*, conforme mostra a Figura 3.10. Veja que dentro do *__name__ == "__main__"*, a função *main()* é invocada, fazendo com que o script execute.

Figura 3.10 |Módulos como scripts

Fonte: elaborada pela autora.

REFERÊNCIAS E LINKS ÚTEIS

API CNAE - Cadastro Nacional de Atividades Econômicas. **API e documentação**. Versão: 2.0.0. CNAE, 2017. Disponível em: https://bit.ly/3amaPGE. Acesso em: 30 jul. 2020.

LJUBOMIR, P. **Introdução à computação usando Python**: um foco no desenvolvimento de aplicações. Rio de Janeiro: LTC, 2016.

PSF - Python Software Fundation. **Modules**. 2020b. Disponível em: https://bit.ly/30SmzNC. Acesso em: 04 jun. 2020.

PyPI. Python Package Index. Página inicial. 2020. Disponível em: https://pypi.org/. Acesso em: 04 jun. 2020.

RAMALHO, L. **Fluent Python**. Gravenstein: O'Reilly Media, 2014.

Bons estudos!