

Visão

COMPUCIONAL

Módulo Comunicabilidade em
Python



Aurélio Freire de Aquino - Engenheiro III

Autor da apostila

Aurélio Freire de Aquino - Engenheiro III

Instrutor do curso

Luana Dias Pena Forte – Assistente Pedagógica

Vitória Fernanda de Souza – Estagiária Pedagógica

Revisão da apostila

Fit Instituto de Tecnologia

Manaus, Agosto de 2023



Autor



Aurélio Aquino é formado em Engenharia da Computação pela UFAM – Universidade Federal do Amazonas e possui pós-graduação *lato sensu* em *Lean Manufacturing* e MBA em Gestão de Projetos Ágeis pelo IDAAM. Também foi aluno na California State University de Sacramento – Califórnia por um ano. Trabalha a mais de 5 anos na área de automação industrial, atuando nas áreas de robótica, visão computacional e desenvolvimento de *software*. Atualmente é instrutor de visão computacional no FIT – Instituto de Tecnologia da Amazônia.



APRESENTAÇÃO

A presente apostila é um instrumento teórico que complementa o curso de capacitação de **Comunicabilidade em Python**, promovido pelo Fit - Instituto de Tecnologia da Amazônia, para colaboradores do Fit Instituto de Tecnologia, Flextronics e comunidade. Nela, teremos uma visão mais aprofundada dos conceitos, equipamentos e soluções usadas na indústria em soluções utilizando visão computacional.

Serão 15 horas de treinamento no qual apresentaremos desde os conceitos de redes básicas até o desenvolvimento de soluções web com visão computacional, utilizando a biblioteca OpenCV, sockets, comunicação serial e Web. É extremamente recomendável ao aluno que, ao final da leitura de cada seção, realize os exercícios propostos, e acesse os materiais indicados nas referências bibliográficas, para aprofundar a leitura desse material e complementar o que foi lido aqui.

A apostila está dividida em três seções, iniciando com a Seção 1 com introdução a redes e sockets. A Seção 2 abordará conceitos de comunicação serial. E para finalizar, a Seção 3 apresenta conceitos de comunicação web com APIs REST.

Desejo a você, prezado aluno, que tenha um excelente curso!!

Boa Leitura !!



Sumário

1	Introdução a Redes e Sockets	5
1.1	Sockets em Python	6
1.2	Servidor Socket	8
1.3	Cliente Socket	10
1.4	Exercício Chat	11
1.5	Tratamento de Erros	11
1.6	Exercício Coordenadas da Face	13
2	Comunicação Serial com Arduino	14
2.1	Conexão Serial em Python	14
2.2	Exercício LED on/off	15
2.3	Exercício LEDs e Cores	15
3	APIs REST e Comunicação Web	16
3.1	Flask	16
3.2	Requests	20
3.3	Códificação Base 64	21
3.3.1	Imagem para String Base64	22
3.3.2	String Base64 para Imagem	22
3.4	Exercícios Imagem da Webcam em Rede	23
	Conclusão	24
	Referências	25
	CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION	
	CONTROL	26



1 Introdução a Redes e Sockets

Antes de programar sockets, fazendo nossos programas se comunicarem, precisamos entender alguns conceitos básicos de protocolos de redes de computadores.

Os protocolos de rede são conjuntos de regras e convenções que permitem a comunicação eficiente entre dispositivos em uma rede. Um dos conjuntos de protocolos mais amplamente utilizados é o TCP/IP, que é a base da Internet e de muitas redes locais. O TCP/IP é dividido em camadas, cada uma responsável por uma função específica na comunicação de rede.

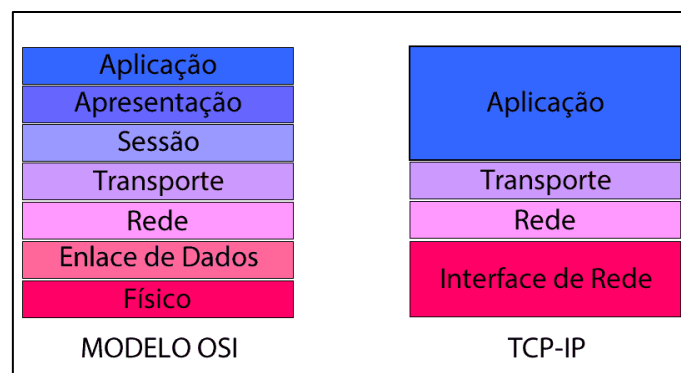


Figura 1 - Modelo OSI e TCP/IP em Camadas. Fonte: Citisystems.

A camada de rede, também conhecida como camada de internet, é responsável por rotear pacotes de dados pela rede, usando endereços IP para identificar os dispositivos e determinar o melhor caminho para enviar os dados. O protocolo IP (Internet Protocol) é o principal protocolo dessa camada.

A camada de transporte fornece mecanismos para a entrega confiável e ordenada dos dados entre os dispositivos. O protocolo TCP (Transmission Control Protocol) é um exemplo de protocolo de transporte amplamente utilizado. Ele garante que os pacotes de dados sejam entregues sem erros, em ordem e sem perdas.

Essas camadas, juntamente com outras camadas, como a camada de interface de rede e a camada de aplicação (HTTP, FTP, SMTP, etc.), formam a estrutura do TCP/IP. Com essa divisão em camadas, é possível construir redes



complexas e interconectadas, permitindo a comunicação eficiente e confiável entre dispositivos em todo o mundo.

Na comunicação de rede, é fundamental entender os conceitos de endereços IP, portas e sockets. O endereço IP é uma identificação única atribuída a cada dispositivo em uma rede, permitindo que os pacotes de dados sejam direcionados corretamente. Existem dois tipos de endereços IP: IPv4, que consiste em quatro conjuntos de números separados por pontos, como 192.168.0.1, e IPv6, que usa uma notação hexadecimal e permite um espaço de endereço maior.

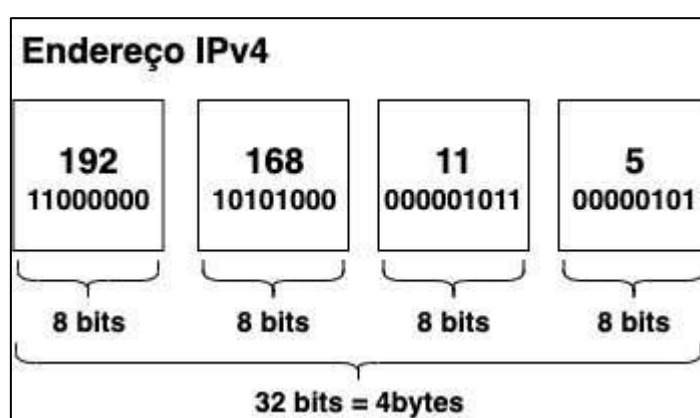


Figura 2 - Endereço IPv4. Fonte: DicionarioTec.

As portas são canais virtuais que permitem que diferentes aplicativos ou serviços se comuniquem em um mesmo dispositivo. Elas são identificadas por números inteiros e são usadas para direcionar o tráfego de rede para o aplicativo correto. As portas são divididas em duas categorias: portas bem conhecidas (0 a 1023), atribuídas a serviços específicos, como HTTP (80) e FTP (21), e portas dinâmicas ou registradas (1024 a 65535), que são usadas por aplicativos temporariamente. As portas dinâmicas serão utilizadas em nossas programações.

1.1 Sockets em Python

Sockets são interfaces de programação que permitem que os aplicativos se comuniquem por meio da rede. Um socket é a combinação de um endereço IP, uma porta e um protocolo de transporte (como TCP ou UDP).

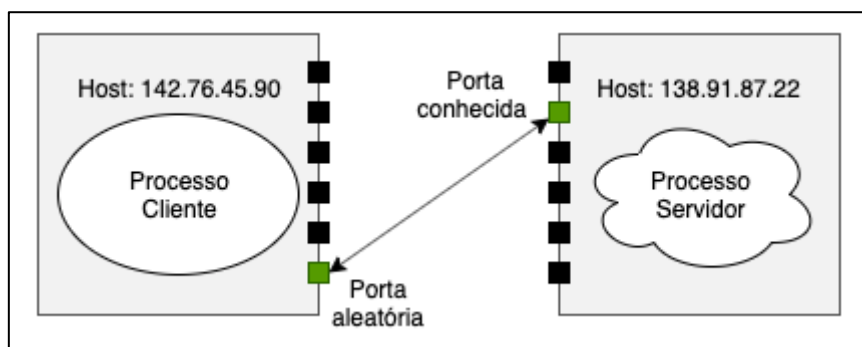


Figura 3 - Conexão de sockets. Fonte: TreinaWeb.

Os sockets podem ser usados tanto para comunicação cliente-servidor quanto para comunicação ponto a ponto entre dois dispositivos. Entenda que um programa agirá como processo servidor e o outro agirá como processo cliente, mesmo que seja o mesmo computador.

Em Python, a biblioteca **socket** fornece suporte para a criação de sockets, permitindo que os programas se conectem a outros dispositivos em uma rede e enviem ou recebam dados.

```
import socket
```

A biblioteca possui a classe **socket** para inicializar um objeto socket para comunicação. Alguns dos parâmetros são listados abaixo (todos são opcionais):

1. Família de endereços (socket.AF_INET ou socket.AF_INET6):
 - socket.AF_INET: Indica que o socket usará o protocolo IPv4 (endereços IPv4 com quatro conjuntos de números separados por pontos).
 - socket.AF_INET6: Indica que o socket usará o protocolo IPv6 (endereços IPv6 em formato hexadecimal).
2. Tipo de socket (socket.SOCK_STREAM ou socket.SOCK_DGRAM):
 - socket.SOCK_STREAM: Cria um socket de fluxo TCP, que é uma conexão confiável e bidirecional.
 - socket.SOCK_DGRAM: Cria um socket de datagrama UDP, que é uma conexão não confiável e sem estado, útil para comunicação rápida e eficiente sem garantia de entrega.
3. Protocolo (socket.IPPROTO_TCP, socket.IPPROTO_UDP ou outros):



- `socket.IPPROTO_TCP`: Indica que o protocolo de transporte será o TCP.
- `socket.IPPROTO_UDP`: Indica que o protocolo de transporte será o UDP.
- Além desses, existem outros protocolos disponíveis, como `socket.IPPROTO_ICMP` para o protocolo ICMP usado por ping.

```
servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

1.2 Servidor Socket

Depois de criar o socket, é necessário usar a função **bind**. Tal função é usada para associar um socket a um endereço IP e uma porta específicos no servidor. Isso informa ao sistema operacional qual endereço IP e porta o socket deve usar para aguardar conexões de entrada.

Em um servidor, você cria um socket e, em seguida, chama o método **bind** para vinculá-lo a um endereço e porta específicos. Isso permite que o servidor escute conexões que chegam a esse endereço e porta.

```
servidor.bind(('127.0.0.1', 8000))
```

Logo em seguida, utilize o método **listen** para aceitar conexões de processos clientes. O parâmetro (opcional) desta função é o número máximo de conexões pendentes.

```
servidor.listen(10)
```

Agora precisamos aceitar as conexões dos processos clientes. Para isso, usamos o método **accept**. Este método retorna dois valores: o socket com a conexão do processo cliente e uma tupla com o endereço e porta do processo cliente.

```
cliente, endereço = servidor.accept()
```



A partir da conexão com o processo cliente podemos enviar e receber pacotes em formato de bytes. Para fazer essa troca de pacotes utilizamos os métodos **send** e **recv** com o objeto de conexão cliente.

No caso da função **send**, devemos gerar uma string em formato de bytes. Podemos utilizar o método **encode** indicando a codificação da string. O código ficaria da seguinte forma:

```
mensagem = 'ola cliente'
mensagemBytes = mensagem.encode('ascii')
cliente.send(mensagemBytes)
```


Já a função **recv** irá retornar os dados que o processo cliente enviar e necessita de um parâmetro do tipo inteiro, ao qual indica o buffer com o número de bytes a serem recebidos. O número de bytes a serem recebidos geralmente é uma potência de 2, como 1024 ou 4096. Os dados retornados são bytes que podem ser convertidos em string novamente com o método **decode** com a indicação da codificação desejada.

```
mensagemBytes = cliente.recv(1024)
mensagem = mensagemBytes.decode('ascii')
```

Ao finalizar a conexão, fechamos a comunicação com o método **close**.

```
cliente.close()
```

No exemplo abaixo, temos um código completo que abre e fecha a conexão para cada pacote trocado:



```

1  import socket
2
3  IP = "127.0.0.1"
4  porta = 8000
5
6  servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  servidor.bind((IP, porta))
8  servidor.listen()
9  while True:
10     cliente, endereco = servidor.accept()
11     msg = 'ola, eu sou o servidor'
12     msgBytes = msg.encode('ascii')
13     cliente.send(msgBytes)
14     msgBytes = cliente.recv(1024)
15     print(msgBytes.decode('ascii'))
16     cliente.close()

```

Figura 4 - Código de exemplo. Fonte: Autoria própria.

1.3 Cliente Socket


Para criar o processo cliente, iremos criar um socket da mesma forma que o processo servidor: utilizando a classe **socket** da biblioteca socket.

```
conexao = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Para conectar, utiliza-se o método **connect** com o mesmo parâmetro que o método **bind**: uma tupla com o endereço IP e a porta da conexão.

```
conexao.connect(('127.0.0.1', 8000))
```

A partir da conexão estabelecida com o servidor, utilize os métodos **send**, **recv** e **close** da mesma maneira que no processo servidor para enviar dados, receber dados e fechar a conexão. Um exemplo completo do processo cliente para conectar-se com o processo servidor do exemplo anterior segue abaixo.



```

1  import socket
2
3  IP = "127.0.0.1"
4  porta = 8000
5
6  while True:
7      conexao = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8      conexao.connect((IP, porta))
9      msgBytes = conexao.recv(1024)
10     print(msgBytes.decode('ascii'))
11     msg = 'ok'
12     conexao.send(msg.encode('ascii'))
13     conexao.close()

```

Figura 5 - Código de exemplo. Fonte: Autoria própria.

1.4 Exercício Chat

1. Faça um chat simples com outro colega no laboratório. Faça o papel de ambos os processos servidor e processo cliente.

1.5 Tratamento de Erros

Em Python, **try** e **except** são construções usadas para lidar com exceções (erros) que podem ocorrer durante a execução de um bloco de código. Quando você suspeita que um trecho específico do código pode gerar uma exceção, você pode colocá-lo dentro de um bloco **try**. Se uma exceção é lançada dentro desse bloco, o programa imediatamente sai do bloco **try** e entra no bloco **except**, onde você pode tratar o erro de forma apropriada. A sintaxe geral é a seguinte:

```

try:
    # Bloco de código que pode gerar uma exceção
except TipoDeExcecao:
    # Bloco de código para lidar com a exceção

```

Por exemplo, considere o seguinte código que lê um número inserido pelo usuário e tenta convertê-lo para inteiro, como comentamos quando falamos de variáveis no módulo anterior:

```

try:
    numero = int(input("Digite um número inteiro: "))

```



```
except ValueError:  
    print("Erro: Valor inválido. Por favor, insira um número inteiro.")
```

Se o usuário digitar um valor que não pode ser convertido para inteiro, o programa lançará uma exceção do tipo **ValueError**. O bloco **except** irá capturar essa exceção e exibirá uma mensagem de erro apropriada, permitindo que o programa continue a execução em vez de ser interrompido abruptamente.

Você também pode ter vários blocos **except** para tratar diferentes tipos de exceções de forma específica:

```
try:  
    # Algum código que pode gerar exceções  
except TipoDeExcecao1:  
    # Tratamento para TipoDeExcecao1  
except TipoDeExcecao2:  
    # Tratamento para TipoDeExcecao2  
except Exception:  
    # Tratamento para outras exceções não especificadas
```

O bloco **except Exception** é usado para capturar qualquer exceção não especificada nos blocos **except** anteriores. Pode ser utilizado sozinho como a captura de erros em geral.

```
try:  
    cliente.connect((IP, porta))  
except Exception:  
    print("Erro ao conectar")
```

Ou ainda, renomear a exceção para utilizar a mensagem do ocorrido.

```
try:  
    cliente.connect((IP, porta))  
except Exception as erro:  
    print(erro)
```



1.6 *Exercício Coordenadas da Face*

1. Faça um programa servidor que detecte rostos na webcam assim que um processo cliente abra conexão. O programa cliente deve receber a coordenada encontrada e imprimir seu valor.



2 Comunicação Serial com Arduino

A comunicação serial em Python é uma forma de trocar dados entre o computador e dispositivos externos através de uma conexão serial. Essa conexão é estabelecida por meio de uma porta serial (também conhecida como porta COM no Windows) e é amplamente utilizada para interagir com dispositivos como microcontroladores, sensores, módulos GPS, impressoras e outros periféricos.

Para realizar a comunicação serial em Python, é necessário usar a biblioteca **serial**, que oferece suporte para trabalhar com portas seriais. Para instalar a biblioteca use **pip install pyserial**, então use no seu código:

```
import serial
```

2.1 Conexão Serial em Python

Para abrir uma conexão com algum dispositivo serial, utiliza-se a classe **Serial** passando dois parâmetros: o nome da porta serial e a taxa de transmissão.

```
conexao = serial.Serial('COM1', baudrate=9600)
```

A porta serial e a taxa de transmissão devem ser ajustadas de acordo com o dispositivo que você está usando.

Para enviar dados para o dispositivo, utilizamos o método **write** passando a mensagem em formato de bytes:

```
msg = 'ola dispositivo'
conexao.write(msg.encode('ascii'))
```

Para receber dados do dispositivo, utilizamos o método **read** passando o número de bytes a serem obtidos, ou também o método **read_all** para obter todos os bytes disponíveis:



```
data = conexao.read(50)
data = conexao.read_all()
msg = data.decode('ascii')
```

Sempre que você terminar a comunicação, é importante fechar a conexão serial para liberar o recurso da porta serial, assim como na comunicação de sockets. Isso é feito com o método **close**:

```
conexao.close()
```

2.2 *Exercício LED on/off*

1. Faça um programa que envie mensagens para um Arduino para acionar um LED. As mensagens devem ser “on” e “off”.

2.3 *Exercício LEDs e Cores*

1. Faça um programa que receba um comando do Arduino para tirar uma foto de um objeto e devolva uma mensagem com a cor do objeto. O comando terá a mensagem “foto” e as cores aceitas são “azul”, “vermelho” e “verde”.

3 APIs REST e Comunicação Web

API REST é uma interface de programação de aplicativos (API) que segue os princípios da arquitetura REST. Uma API REST permite que diferentes sistemas e aplicativos se comuniquem e interajam uns com os outros por meio do protocolo HTTP de maneira padronizada e consistente. Ela utiliza os métodos HTTP, como GET, POST, PUT e DELETE, para realizar operações em recursos, e as informações são trocadas em formatos de dados padronizados, como JSON ou XML. A API REST é projetada para ser simples, escalável, flexível e fácil de usar, permitindo a criação de serviços web que podem ser facilmente consumidos por clientes diversos.

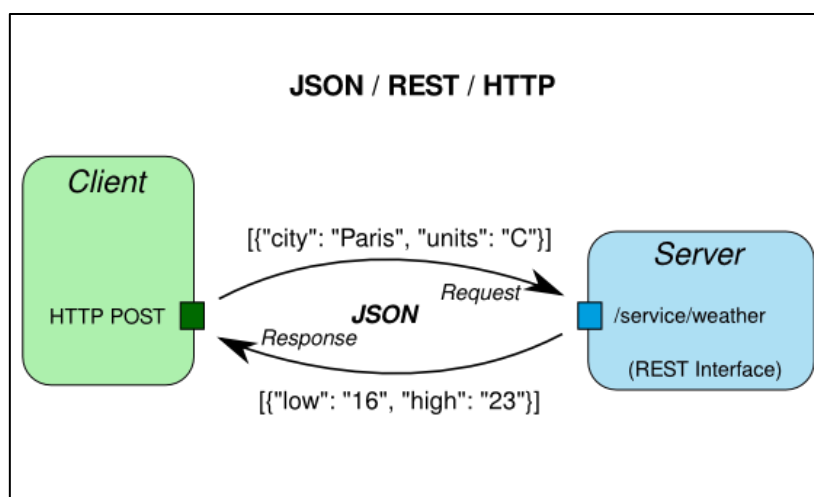


Figura 6 - Arquitetura REST. Fonte: Medium.

3.1 Flask

O Flask é um popular framework leve e flexível para desenvolvimento web em Python. Ele foi criado para facilitar a construção de aplicativos web de forma rápida e simples, permitindo que desenvolvedores criem aplicativos web com facilidade e elegância.

Graças à sua simplicidade, flexibilidade e natureza minimalista, o Flask é uma excelente opção para desenvolvedores que desejam criar aplicativos web em Python sem o peso de um framework mais complexo. Ele é amplamente utilizado para construção de APIs RESTful, aplicativos web e projetos de todos os tipos que requerem interações via HTTP.



Figura 7 - Logo do framework Flask. Fonte: Flask Pallets Projects.

Uma API RESTful (Representational State Transfer) é um estilo de arquitetura de desenvolvimento web, que define um conjunto de padrões para a criação de serviços web que se comunicam por meio do protocolo HTTP. Essa abordagem permite que diferentes sistemas e aplicativos se comuniquem e compartilhem dados de maneira eficiente e escalável. Principais características de uma API RESTful:

- Utilização de métodos HTTP: uma API RESTful utiliza os métodos padrão do protocolo HTTP, como GET, POST, PUT e DELETE, para realizar operações em recursos. Cada método é utilizado para representar uma ação específica, como obter informações (GET), criar novos recursos (POST), atualizar recursos existentes (PUT) ou excluir recursos (DELETE).
- Recursos e URIs: os recursos são elementos identificáveis que podem ser manipulados pela API. Eles são representados por URIs (Uniform Resource Identifiers), que são endereços únicos que identificam cada recurso de forma única na API.
- Representação de dados: as informações dos recursos são representadas em formatos padrão, como JSON ou XML. Essas representações são enviadas como resposta ao cliente quando a API é consultada.
- Stateless (sem estado): cada requisição para a API contém todas as informações necessárias para ser processada, sem que o servidor precise manter o estado da sessão do cliente entre as requisições. Isso facilita a escalabilidade e a confiabilidade do serviço.



- Interação cliente-servidor: a API RESTful é baseada em uma arquitetura cliente-servidor, em que o cliente envia solicitações para o servidor e o servidor responde com os dados solicitados.

No código, importe a classe **Flask** e a variável **request** da biblioteca.

```
from flask import Flask, request
```

A classe **Flask** será utilizada para criar a aplicação flask que será o nosso serviço. uma instância de **Flask** deve ser inicializada com um nome. Vamos criar uma API para somar dois números.

```
app = Flask('Minha API de Somas')
```

Já a variável **request** conterá as informações enviadas pelo cliente. No nosso exemplo, usaremos os dados que vem do formato JSON (JavaScript Object Notation), ao qual é convertido no python para um dicionário.

A próxima etapa é definir a rota **/soma** com o método POST. Isso significa que a rota **/soma** será acessível através do método HTTP POST e é usada para enviar dados para a API.

A sintaxe a seguir do "@" em Python é chamada de "decorator". O decorador **app.route** é um recurso específico do Flask que permite definir rotas para as funções de uma aplicação web. Ele é usado para associar URLs específicas às funções que serão executadas quando essas URLs são acessadas através de solicitações HTTP.

```
@app.route('/soma', methods=['POST'])
```

A função associada a essa rota será chamada no exemplo de **adicionarNumeros**.

```
def adicionarNumeros():  
    try:  
        num1 = request.json['num1']  
        num2 = request.json['num2']
```

```
        return {'msg': num1 + num2}, 200
    except KeyError as e:
        return {'erro': f'Campo {e} não encontrado'}, 400
    except Exception as e:
        return {'erro': f'{e}'}, 400
```

No código de exemplo, o bloco **try** está envolvendo a lógica principal da função **adicionarNumeros**, onde a soma dos números é realizada.

Primeiro, a função tenta obter os valores dos campos 'num1' e 'num2' do corpo da requisição JSON (vindos da solicitação POST) usando **request.json**. Se esses campos estiverem presentes, a soma dos valores de num1 e num2 será retornada como uma resposta JSON bem-sucedida com status 200.

Se algum dos campos 'num1' ou 'num2' não estiver presente no corpo da requisição (ou seja, uma exceção **KeyError** for lançada), o bloco **except KeyError** será executado. Nesse caso, uma resposta JSON com uma mensagem de erro indicando qual campo está faltando será retornada, e o status da resposta será definido como 400 (Bad Request). Caso ocorra outro erro qualquer, **Exception** retornará o erro genérico e o status 400 (Bad Request).

Para executar o serviço Flask, utilize o método **run** do objeto Flask. Este método tem parâmetros opcionais como **host** e **port**, ao qual podem ser configurados com o endereço da rede WiFi, LAN cabeada, etc. Caso nenhum parâmetro seja passado, o servidor será acionado em `http://127.0.0.1:5000`.

```
app.run()
```



```
1  from flask import Flask, request
2
3  app = Flask('Minha API de Somas')
4
5  @app.route('/soma', methods=['POST'])
6  def adicionarNumeros():
7      try:
8          num1 = request.json['num1']
9          num2 = request.json['num2']
10         return {'msg': num1 + num2}, 200
11     except KeyError as e:
12         return {'erro': f'Campo {e} não encontrado'}, 400
13     except Exception as e:
14         return {'erro': f'{e}'}, 400
15
16  app.run()
```

Figura 8 - Código de exemplo. Fonte: Autoria própria.

3.2 Requests

Para fazer requisições HTTP no Python, utiliza-se uma biblioteca chamada **requests**.

```
import requests
```

É possível fazer requisições do tipo get, post, patch ou delete por exemplo. No caso do exemplo do serviço Flask do exemplo anterior, devemos fazer uma requisição post para a URL <http://127.0.0.1:5000> na rota/soma. A função **post** recebe um parâmetro obrigatório com a string da URL e os dados da requisição (números a serem somados) são enviados em formato JSON usando o parâmetro opcional **json**.

```
url = 'http://127.0.0.1:5000/soma'
dados = {
    'num1': 10,
    'num2': 20
}
resposta = requests.post(url, json=dados)
```



A resposta contém um atributo chamado **status_code** contendo o status da requisição. A resposta também contém um método chamado **json** que retorna um dicionário com a resposta json recebida. Abaixo o exemplo completo para fazer uma requisição.

```
1  import requests
2
3  url = 'http://127.0.0.1:5000/soma'
4
5  data = {
6      'num1': 10,
7      'num2': 20
8  }
9
10 response = requests.post(url, json=data)
11
12 if response.status_code == 200:
13     result = response.json()
14     print(f'Resultado da soma: {result["msg"]}')
15 elif response.status_code == 400:
16     error = response.json()
17     print(f'Erro: {error["erro"]}')
18 else:
19     print(f'Erro na requisição: Código {response.status_code}')
```

Figura 9 - Código de exemplo. Fonte: Autoria própria.

3.3 *Códificação Base 64*

A vantagem de utilizar a comunicação web ao invés de sockets ou serial é a variedade e a quantidade de informações a serem trocadas entre o servidor e cliente. É possível passar imagens completas através dos pacotes JSON para que o cliente mostre em uma página HTML, por exemplo. A imagem pode ser passada como um link ou com uma codificação em base 64.

A codificação base64 para imagens é um método que converte dados binários de uma imagem em uma sequência de caracteres ASCII. Essa codificação é usada para representar imagens como uma sequência de texto, tornando-as fáceis de serem transmitidas em ambientes que suportam apenas caracteres ASCII, como nas APIs REST. Em resumo, a codificação base64 não altera a imagem, apenas a transforma em string.

Para utilizar a codificação em base 64, importe a biblioteca **base64**.



```
import base64
```

3.3.1 Imagem para String Base64

Considere que temos uma variável com uma imagem do OpenCV, seja ela do arquivo, da webcam ou uma imagem processada. Podemos utilizar a função **imencode** para iniciar a codificação. Esta função retorna uma tupla com dois valores: um booleano **retval** que indica se a codificação foi bem-sucedida e o **buffer** que são os bytes da imagem codificada. A função exige dois parâmetros: a extensão do arquivo (‘.png’, ‘.jpg’, ‘.bmp’) e a imagem.

```
retval, buffer = cv.imencode(‘.png’, img)
```

Em seguida, utilizaremos a função **b64encode** da biblioteca base64 para codificar o buffer obtido.

```
buffer64 = base64.b64encode(buffer)
```

Por último, transformamos os bytes em formato base64 para uma string com **decode**, da mesma forma que nas comunicações de sockets e serial. Desta forma, podemos enviar a imagem como uma string.

```
string64 = buffer64.decode(‘ascii’)
```

3.3.2 String Base64 para Imagem

Ao obter uma string que representa uma imagem codificada em base64, podemos transformar essa informação de volta em uma imagem para o OpenCV.

Primeiro devemos transformar a string em um buffer de bytes com a função **b64decode** da biblioteca base64.

```
buffer = base64.b64decode(string64)
```

Nota: caso esteja recebendo uma imagem em formato base64 de uma aplicação Web, pode existir um prefixo na string indicando o formato da imagem.



O prefixo mais comum é “**data:image/png;base64,**”, ou seja, pode ser necessário remover esse prefixo para usar a função **b64decode**. Utilize o método **replace** das strings para remover esse prefixo.

```
buffer = base64.b64decode(string64.replace('data:image/png;base64,', ''))
```

Em seguida vamos gerar um array numpy a partir deste buffer com a função **frombuffer** da biblioteca numpy, passando como parâmetros o buffer e o tipo de dados do array gerado.

```
imgArray = np.frombuffer(buffer, np.int8)
```

Por último, utilizamos a biblioteca do OpenCV para abrir uma imagem a partir de um array numpy com a função **imdecode**, passando como parâmetros o array numpy e uma constante indicando o tipo de imagem da mesma forma que a função **imread**, como abrir a imagem em escala de cinza ou sem alterar a imagem para casos de imagens em fomato png, que será o formato mais comum em aplicações web.

```
img = cv.imdecode(imgArray, cv.IMREAD_UNCHANGED)
```

3.4 Exercícios Imagem da Webcam em Rede

1. Faça um programa em Flask que forneça a imagem da webcam na rota **/img**.
2. Faça um programa que solicite a imagem da webcam do programa anterior e mostre em uma janela. Experimente solicitar a imagem do computador de um colega.



Conclusão

A abordagem desta apostila permite ao leitor uma interação ativa com as ferramentas de processamento de imagem e visão computacional disponíveis na tecnologia apresentada. Ensinar como inspecionar imagens e extrair informações, para dar ao aluno autonomia, estímulo, senso crítico e contribuir para uma aprendizagem mais efetiva.

Começamos com uma introdução a redes e sockets para aprendermos os principais conceitos que seriam necessários para o entendimento de redes em Python. Depois partimos para a comunicação serial que pode ser utilizada com microcontroladores. Em seguida, aborda-se os princípios de comunicação Web com APIs REST, mostrando como a troca de informações em sistemas de software funciona e como podemos tratar imagens através deste tipo de comunicação.

Assim, o conteúdo do curso Comunicabilidade em Python familiariza o leitor com o mundo tecnológico, onde a virtualização ganha, cada vez mais, espaço e agrada as empresas por reduzir custos e aumentar as margens de lucro (adaptar a frase para seu curso). Obrigado por fazer parte do curso e ter realizado a leitura desta apostila. Espero que o interesse pelo Comunicabilidade em Python apresentado neste curso esteja aguçado, para que você pratique e conheça ainda mais as suas maravilhas.



Referências

Barelli, F. (2019). Introdução à Visão Computacional – Uma abordagem prática com Python e OpenCV. Casa do Código. São Paulo.

Documentação da biblioteca OpenCV para Python. (2022). Disponível em> https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.

Documentação Flask. (2023). Disponível em> <https://flask.palletsprojects.com/>

API REST. (2023). Disponível em> <https://restfulapi.net/>

Documentação da biblioteca Base64. (2023). Disponível em> <https://docs.python.org/3/library/base64.html>

Documentação da biblioteca Serial. (2023). Disponível em> <https://pyserial.readthedocs.io/en/latest/>

Protocolo HTTP. (2023). Disponível em> <https://developer.mozilla.org/en-US/docs/Web/HTTP>

Documentação da biblioteca de Sockets. (2023). Disponível em> <https://docs.python.org/3/library/socket.html>

Tutorial sobre sockets em Python. (2023). Disponível em> <https://realpython.com/python-sockets/>



CONTROLE DE REVISÃO DO DOCUMENTO / DOCUMENT REVISION CONTROL

Revisão	Descrição	Razão	Autor	Data
A	Elaboração inicial	Elaboração inicial/Revisão pedagógica	Aurelio Aquino/Luana Forte	11/08/23





BOM CURSO!