PIXEL, A VIRTUAL ASSISTANT WITH FACE RECOGNITION

Technical Document

Supervisor: Joseph Kehoe
Institute of Technology Carlow

Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

Tataru Theodora
C00231174

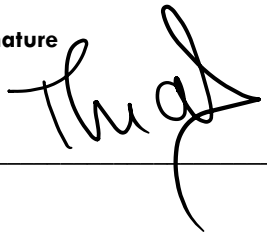
30th of April, 2021

Declaration on plagiarism

Student	Theodora Tataru – C00231174
Tutor	Joseph Kehoe
Institution	Institute of Technology Carlow
Assignment Title	PIXEL, A VIRTUAL ASSISTANT WITH FACE RECOGNITION
Submission Date	30th April 2021

I declare that this research project titled "PIXEL, A VIRTUAL ASSISTANT WITH FACE RECOGNITION" has been written by me under the supervision of Joseph Kehoe.
This document was not presented in any previous research papers for the award of a bachelor's degree to the best of my knowledge. The work is entirely mine, and I accept full responsibility for any errors that might be found in this report. At the same time, the reference to publish materials had been duly acknowledged.
I have provided a complete table of references for all works and sources used in the preparation of this document.
I understand that failure to conform with the Institute's regulations governing plagiarism represents a serious offense.

Signature



Date:

30th April 2021

ABSTRACT

This document aims to provide the technical details necessary for developing Pixel Virtual Assistant.

These technical details include the hardware details and their correct installation, followed by their encapsulation within the wooden frame, the software required for development such as Operating System, Python, Virtual environments, OpenCV and other libraries.

The document also provides an automated installation of Pixel Virtual Assistant, which sets up the system for a smooth performance of Pixel.

The whole code produced for Pixel is detailed and documented in the central part of this document, followed by the code created to test the hardware and several libraries prior to development.

Finally, the document informs how to start the device and provides a link to a video tutorial on how to use Pixel Virtual Assistant.

CONTENTS

Declaration on plagiarism 1
Abstract 2
Table of figures 5
Introduction 7
Hardware 8
Raspberry Pi 8
Camera 9
Speakers 9
Microphone 10
Display 10
GSM/GPRS/GNSS/Bluetooth HAT 11
Cooling fan 11
Two-way mirror 12
Frame 13
SD card 15
Software 16
Operating system 16
Python 19
Virtual environment 20
OpenCV 22
Project dependencies 27
Python scripts 28
Files layout 28
Coordinator 29
Camera 34
Display 38
Virtual_assistant 41
User 47
Skill – Abstract 50
Covid19 51
Thank_you 54
Definition 56
Help 59
Location 65
Weather 69
Good_bye 72
Time 74
Date 76

Register 78
Greeting 85
not_understanding 87
SOS 88
Emergency 93
List 96
Comunication 110
SMS 115
GPS 116
GSMHat 118
Helper 134
Context 134
create_pickle_files 136
delete_pictures 137
folder_exists 138
get_last_word 139
nice_formatted_text 140
substring_after 141
remove_words 142
word_after 143
word_before 144
remove_polite_words 145
get_request 146
Automated setup 147
Video Tutorial 148
Research code 149
Face recognition 1 150
Face recognition 2 155
GSM/GPS hat 159
Pan-Tilt-hat 161
Servos movement 166
And others 171
Conclusion 172
Bibliography 173

TABLE OF FIGURES

Figure 1 "Raspberry Pi 4" 8
Figure 2 "Camera" 9
Figure 3 "Audio module speakers" 9
Figure 4 "USB Microphone" 10
Figure 5 "10.1-inch display" 10
Figure 6 "Back view display" 10
Figure 7 "GPS, GPRS, GNSS and Bluetooth" 11
Figure 8 "Back view GSM hat" 11
Figure 9 "Cooling fan" 11
Figure 10 "Acrylic mirror" 12
Figure 11 "Acrylic mirror installed inside the frame" 12
Figure 12 "Frame" 13
Figure 13 "Tools required for crafting wood" 13
Figure 14 "Before and After case 2" 14
Figure 15 "Before and After case 1" 14
Figure 16 " Back view" 14
Figure 17 " Front view" 14
Figure 18 16
Figure 19 17
Figure 20 17
Figure 21 17
Figure 22 18
Figure 23 18
Figure 24 18
Figure 25 19
Figure 26 20
Figure 27 22
Figure 28 22
Figure 29 23
Figure 30 24
Figure 31 25
Figure 32 25
Figure 33 "Testing code – root folder" 149
Figure 34 “Testing code” 150
Figure 35 "Testing code" 155
Figure 36 "Testing code" 159
Figure 37 "Testing code" 161
Figure 38 "Testing code" 166

Figure 39 "Testing code" 171
Figure 40 "Testing code" 171
Figure 41 "Testing code" 171

INTRODUCTION

Pixel Virtual Assistant is a research project that involves hardware and software in its implementation, and that aims to deliver a smart speaker integrated into a mirror to facilitate people with low computer skills and people with hearing deficiency.

The project is a prototype that gives a developer the essential features to deliver a smart speaker with face recognition.

In this document, the hardware installation is described in detail, and the code for the project is provided with its documentation. The document also contains instructions on how to start the device and how to use the device.

HARDWARE

In this section, all required hardware for Pixel Virtual Assistant is described in detail, along with its proper installation.

All the hardware described in this section is technically detailed in the "Research Document".

RASPBERRY PI

Raspberry Pi 4B+ Module can be identified as the core hardware element of this project. This module allows the possibility of connecting multiple key hardware components necessary for a successful outcome from this project. The Pi is also seen as the brain for this project, hosting the operating system and the necessary code.

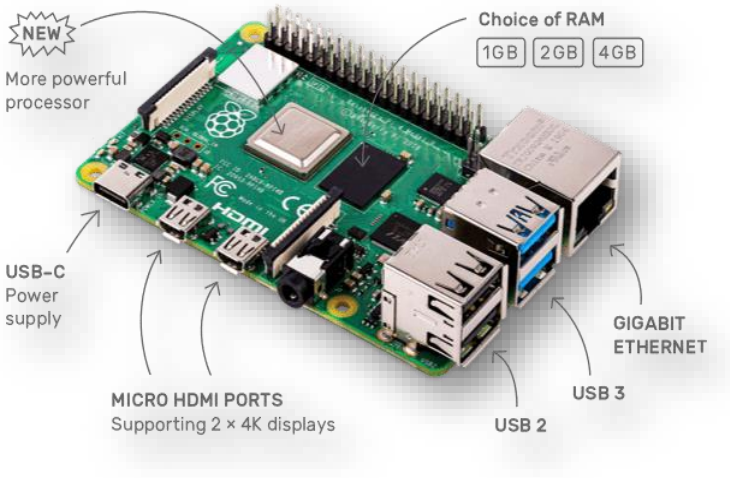


Figure 1 "Raspberry Pi 4"
(Serasinghe, 2020)

The Raspberry Pi is a compact computer about the size of a credit card, making it easy to incorporate inside the frame used for this project. It is a less expensive type of computer that comes without a case.

The fact that the Raspberry Pi 4B+ comes with 4GB of RAM and a 64-bit processor makes it the perfect choice for this project that requires at least 1GB of RAM to enable a better time performance for the OpenCV running process.

The Raspberry Pi can be found on several websites such as Amazon, RS Components, and others.

Raspberry Pi 4B+ model has the following features used in the "Pixel, Virtual Assistant" case (Electronics, 2019):

- 4GB of RAM
- 1.5 GHz quad-core Arm Cortex
- VideoCore VI graphics
- True Gigabit Ethernet
- 2 USBs 3.0
- 2 USBs 2.0
- 2 Micro HDMI
- 1 USB C
- 4kp60 HEVC decode

CAMERA

With a 5 MP resolution and infrared feature, Longruner Raspberry Pi Camera Module switches from day to night vision automatically for a better and accurate view at all times of the day and night, regardless of the light conditions (Amazon, 2021).

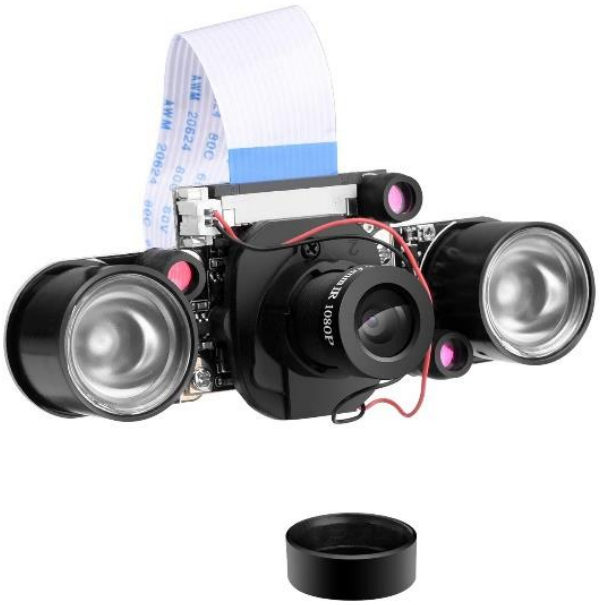


Figure 2 "Camera"
(Amazon, 2021)

This Raspberry Pi Camera Module is a custom-designed, plug-and-play solution. It connects to the Raspberry Pi by plugging into one of the two sockets on the board's upper surface. The camera is connecting with the Raspberry Pi using a ribbon cable which is also used to power de camera. The camera uses a dedicated CSI interface that carries pixel data from the camera back to the processor (Amazon, 2021).

The camera is supported in the latest version of Raspbian and other Raspberry Pi's preferred Operating System (Amazon, 2021).

This type of camera module can be found on the Amazon website alongside other module cameras that can be used, considering the camera's abilities and the impact on the virtual assistant.

Longruner Raspberry Pi Camera Module has the following properties (Amazon, 2021):

- 5 Megapixels
- Focal Length 3.6mm
- Light sensors
- Shooting angle 60 degrees
- 1080p optimum resolution
- Infrared Lens

SPEAKERS

IBest WM8960 HI-FI Hat Audio Module has low power consumption, dual-channel speaker interface, direct drives speakers, and they are small in size (Amazon, 2021).

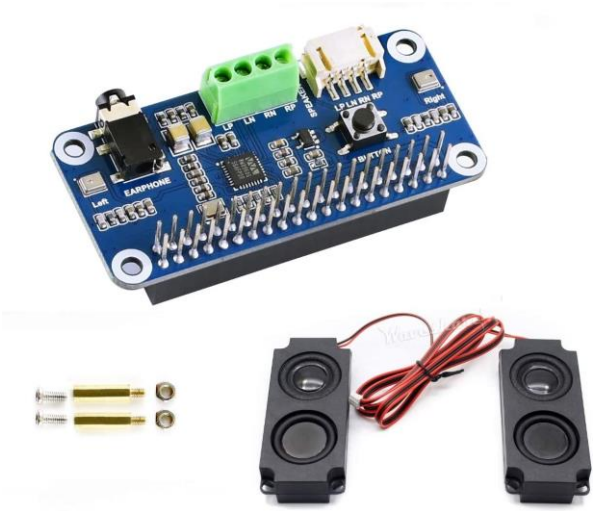


Figure 3 "Audio module speakers"
(Amazon, 2021)

The speaker board connects to the Raspberry Pi via GPIO pins; however, the display used in this project has an I2S port, which allows the speakers to be connected directly to the screen board without using the Hat.

Specifications (Amazon, 2021)

- CODEC: WM8960
- Power supply: 5V
- Logic Voltage: 3.3V
- Control interface: I2C
- Audio interface: I2S
- DAC signal-noise ratio: 98dB
- ADC signal-noise ratio: 94dB
- Earphone driver: 40mW (16 Ω@3.3V)
- Speaker driver: 1W per channel (8 Ω BTL)

This module and similar products can be found on Amazon, Alibaba.com, and official Raspberry Pi websites.

MICROPHONE

Gyvazla USB microphone has a broad frequency response for precise and accurate sound capture. The plug-and-play interface, which uses a USB connector, ensures seamless communication with the Raspberry Pi without installing any driver (Amazon, 2021).

The Raspberry Pi used for this project does not have a JACK port; therefore, this microphone's USB connector feature proves to be the best option.

A long enough cable is needed for the microphone to be mounted in the front of the frame used for this project. If the microphone is placed on the inside of the frame, it could be unable or difficult to capture the user's voice.

This microphone cancels background noise and isolates the primary source of the sound.

Figure 4 "USB Microphone"
(Amazon, 2021)

DISPLAY

The SunFounder 10.1-inch display proves to be the right choice for this project. This display is compatible with Raspberry Pi. It has the option of mounting the Raspberry Pi on its back; thus, it is stable, allows for space savings when designing the project, and produces a neat, finished product.

This project began with a 5-inch touch screen with an HDMI input, but the SunFounder 10.1 inch is the ideal hardware component for achieving the project's target due to its wider screen, increased text size, higher resolution, and the fact that the Raspberry Pi can be connected to the back of the display.

The SunFounder 10.1-inch display and other similar displays can be found on websites like Amazon, AliExpress.com, RS Components, others.

Figure 5 "10.1-inch display"
(Amazon, 2021)

The display board has an I2S port, which allows the project's speakers to be attached directly to the screen board without using their Hat.

Since the Raspberry Pi is connected to the back of the screen and can be powered by it, there is no need to use multiple power cables.

Figure 6 "Back view display"
Source: Theodora Tataru, 2021

GSM/GPRS/GNSS/BLUETOOTH HAT

The GSM/GPRS/GNSS Module includes a SIM slot and a GSM antenna, allowing developers to build features such as "Call location", "Send SOS", and "Get Weather."

Figure 7 "GPS, GPRS, GNSS and Bluetooth"
(Amazon, 2021)

The Hat may be fitted with a speaker/microphone using the 3.5mm earphone/mic jack. This function allows the device to make calls as it was a cell phone, with the microphone as an input and the speakers as an output.

The Raspberry Pi can be wired to the GPS Hat using either the USB or UART interface (Amazon, 2021). The USB interface was the perfect fit for this project since the Hat would cover the GPIO pins, and certain pins are reserved for the cooling fans.

The Hat will retrieve data from all global satellite positioning networks, receivers, and other corresponding antennas using the GNSS (Global Navigation Satellite System). This antenna revives radio signals emitted by satellites on particular frequencies (Ltd, 2020).

The module also has a Bluetooth communication option, which was not used in this project (Amazon, 2021).

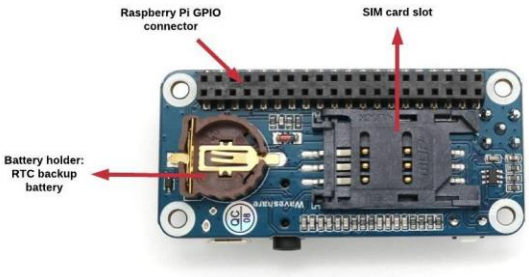


Figure 8 "Back view GSM hat"
Source: Theodora Tataru, 2021

A SIM card must be inserted into the Hat, as many skills for this project specifically benefit from GSM and GPRS functionalities such as SMS and phone calls, which are critical project functions. The GPS module supports and allows the use of GPS, COMPAS, Glonass, LBS base station positioning, and omni-positioning (Amazon, 2021).

A CR1220 battery is used to calculate the passing of time when the board is switched off. For up to three years, this battery will keep the real-time clock (RTC) running (Amazon, 2021).

The module used for this product can be found on the Amazon website. Similar hardware components can be found on the same website or others like RobotShop.com, Waveshare.com, and others.

COOLING FAN

To allow and sustain a safe level of the board temperature, the Makerfire Fan was chosen. The cooling fan lowered the Raspberry Pi's temperature from over 80 degrees Celsius to about 40 degrees Celsius. A second fan was attached to the Raspberry Pi and placed in the back of the camera. Given the proximity of the camera to the wooden frame, this area posed a danger.

Figure 9 "Cooling fan"
(Amazon, 2021)

These fans keep the Raspberry Pi cool and prolong its life by providing a quiet and healthy environment. The Makerfire Raspberry Pi Fan has a one-of-a-kind terminal interface (the "One-to-Two") that allows the creator to select between 3.3V and 5V using the GPIO pins (Amazon, 2021).

Giving their size and the easy way of connecting them with the Raspberry Pi, it proves to be an excellent choice.

The Makerfire Fan, like other similar products, can be found on websites like Amazon, Digi-Key Ireland, RS Components, and others.

TWO-WAY MIRROR

Using the two-way-acrylic, the bright colors of the screen on the mirror's inner side will slip through, giving the viewer the impression that the text is on the mirror's surface. Users of the device can see their reflection in the mirror at the same time.

It is made of acrylic, which has the same characteristics as most acrylic panels, such as being lightweight and shatterproof, making it an ideal choice for use in environments where protection is paramount, such as around children and animals (Amazon, 2021).

It has a ten times greater impact strength, is more flexible, shatter-proof, and stronger than a conventional glass of comparable thickness (Amazon, 2021).

Figure 10 "Acrylic mirror"
(Amazon, 2021)

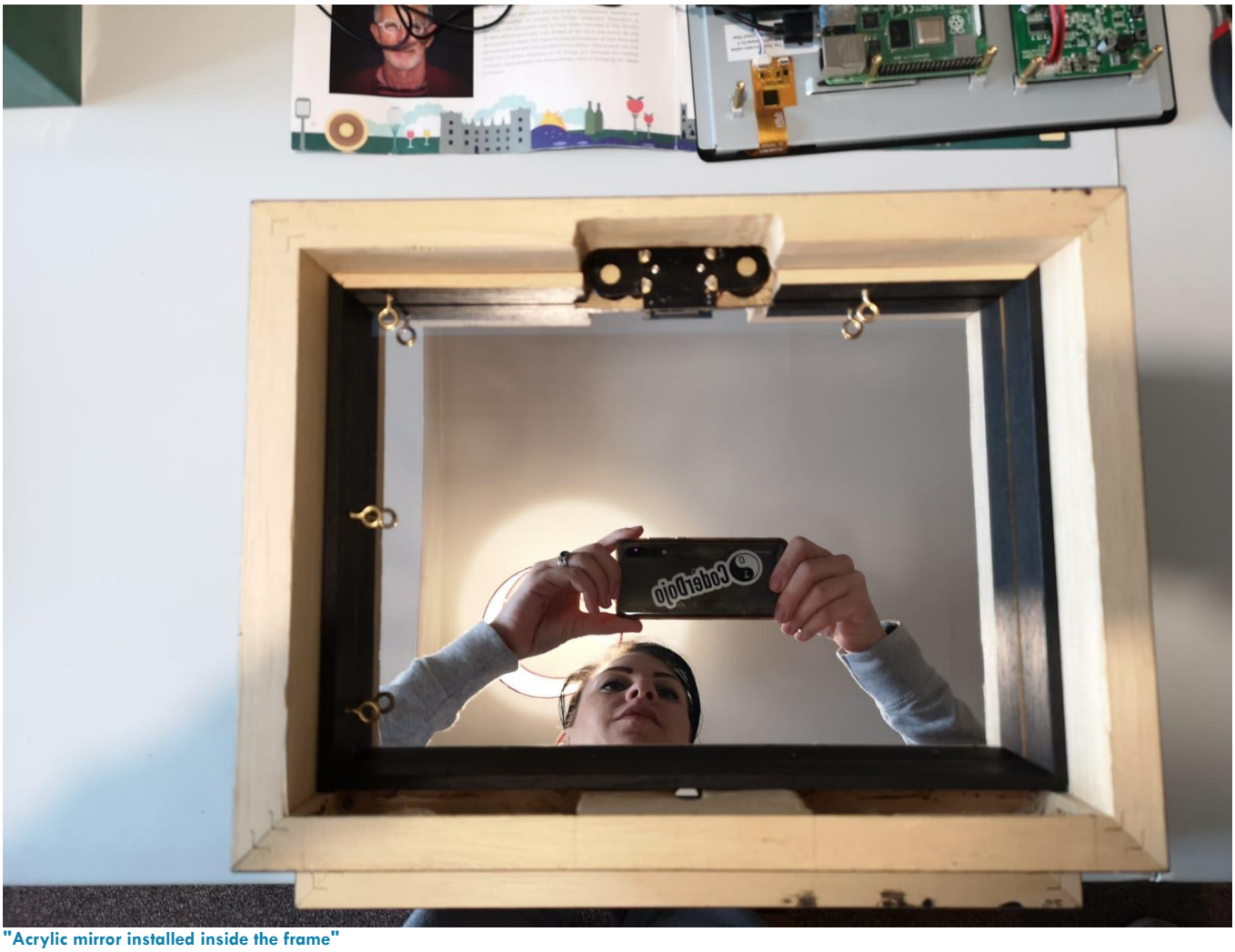


Figure 11
"Acrylic mirror installed inside the frame"
Source: Theodora Tataru, 2021

FRAME

All of the hardware used to execute the "Pixel" Virtual Assistant is encapsulated and hidden within the frame. The two-way acrylic mirror with the dimensions of an A4 (29.7cm/21cm) is a key component in determining the frame size; therefore, the frame used for this project has the following dimensions:

- Length = 34 cm
- Height = 25.5 cm
- Width = 7 cm

Given the challenge of crafting wood for someone who is not a skilled wood crafter, the frame was handcrafted by a local carpenter. Four wood laths were added to the frame to provide stabilization and protection for the two-way acrylic mirror and screen.

Multiple wood carving equipment such as drill machines, various types of saw blades, measuring tape, and other tools were needed for the input and output devices such as camera, microphone, speaker, and temperature sensors.

Figure 12 "Frame"
Source: Theodora Tataru, 2021

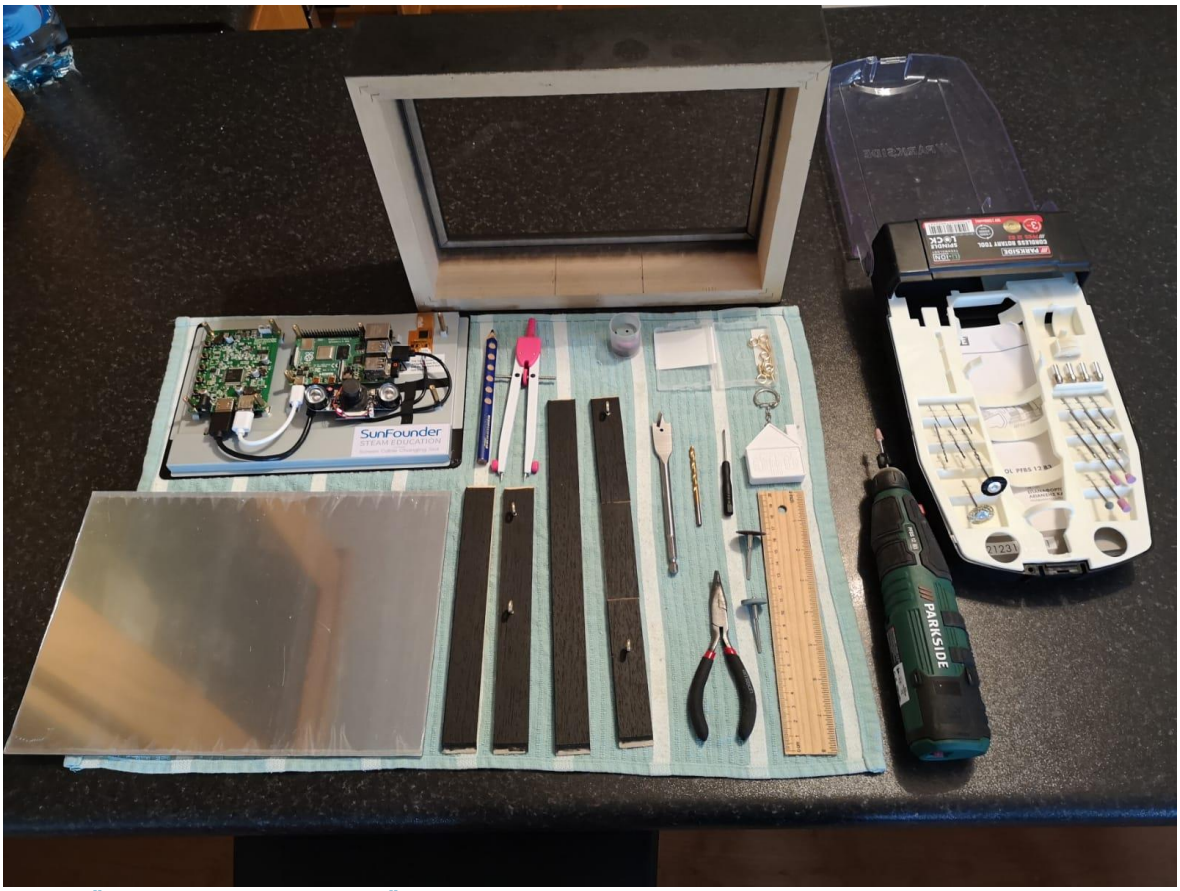


Figure 13 "Tools required for crafting wood"
Source: Theodora Tataru, 2021

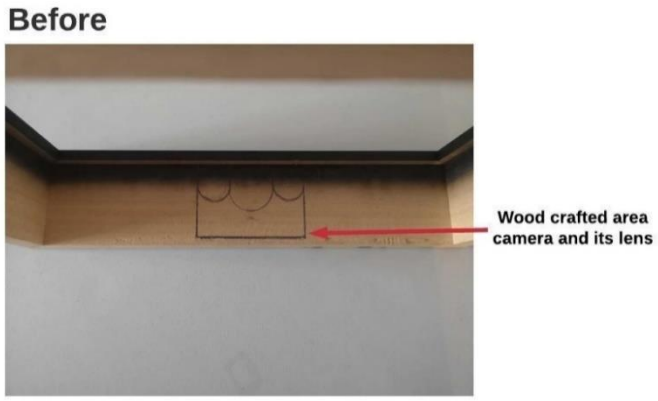


Figure 15 "Before and After case 1"
Source: Theodora Tataru, 2021

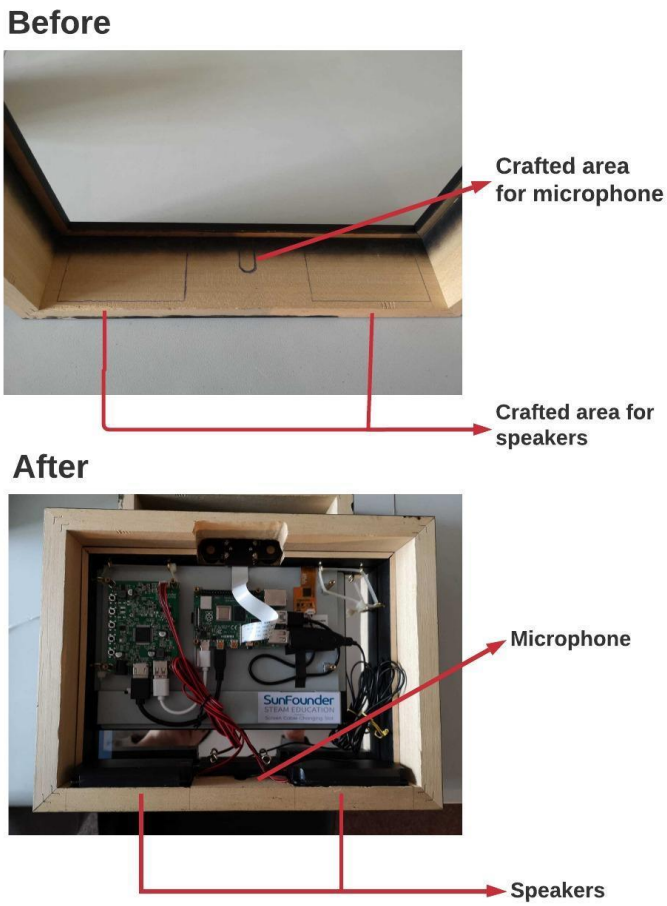


Figure 14 "Before and After case 2"
Source: Theodora Tataru, 2021

Figures 14 and **15** describe the process of inserting the camera with its infrared lens and the speakers into the frame. The "Before" segment in these figures shows the frame in the early stages of construction. However, the camera with the infrared lens and the speakers have been added to the frame in the "After" part of **Figure 14** and **Figure 15**.

Figure 17 " Front view"
Source: Theodora Tataru, 2021

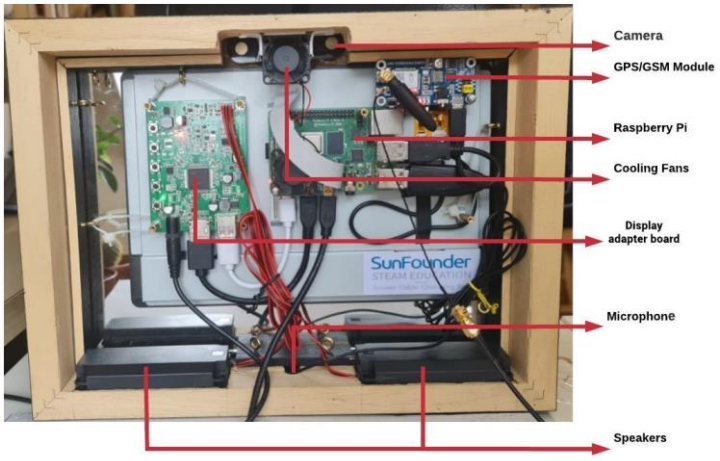


Figure 16 " Back view"
Source: Theodora Tataru, 2021

The figures above show a complete frame with all of the hardware components necessary for maximum functionality of "Pixel, Virtual Assistant". The front view shows the position of the camera, the microphone, and the speakers. The inner layout of the hardware components can be seen in **Figure 16.** Since certain parts have long cables, the excess must be tied to facilitate ventilation and not interfere with the rest of the hardware components.

SD CARD

The Raspberry Pi should work with any SD card, but the following guide ensures a successful installation.

CARD CLASS

The class of the SD card determinates the sustained write speed of the card (Pi, 2021):

- Class 4 is able to write at 4 MB/second
- Class 10 is able to write at 10 MB/second

Keep in mind that a class 10 card will not necessarily outperform a class 4 for general usage, as often the write speed is achieved at the cost of reading speed (Pi, 2021).

CARD PHYSICAL SIZE

Raspberry Pi model 4B+ requires a micro SD card; however, if another Raspberry Pi is chosen for the project, keep in mind that until Pi Model B, a full-size SD card was required, and all the models that preceded the PI Model B, starting with B+ require a micro SD card (Pi, 2021).

When using an SD card, the following steps are recommended to be followed (Pi, 2021):

1. A genuine SD card is required
2. A qualitative power supply is required for the PI
3. Shutting down the Raspberry Pi needs to be done correctly before powring it off
4. Corruption of the SD card had been observer if the PI is overclocking

SOFTWARE

In this section, the code is exhibited along with its documentation. This section is followed by instructions on how to install the necessary files to run the project and a video tutorial that explains how to use the device.

OPERATING SYSTEM

The Operating System used to develop this project resides directly on the Raspberry Pi, and it is called Raspbian OS (**Figure 18**).

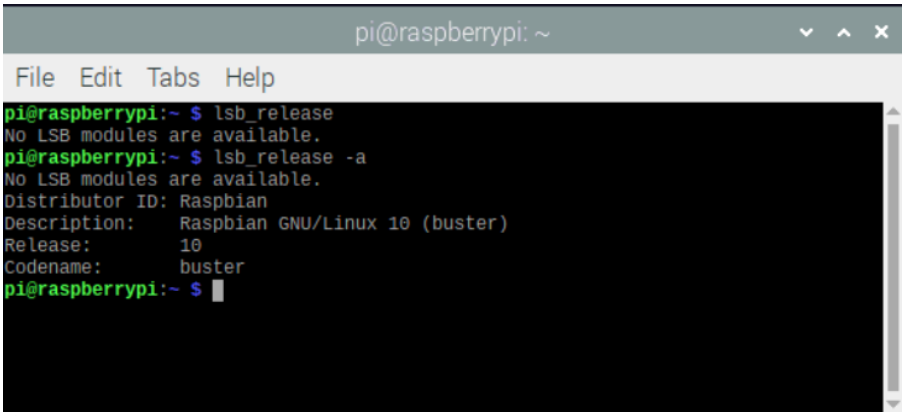


Figure 18
Source: Theodora Tataru, 2021

INSTALLING RASPBIAN OS

In this section, a detailed explanation is provided on how to install the Raspbian OS on the Raspberry Pi.

SD CARD

Before proceeding to the Raspbian OS installation, the Hardware section of this document must be read carefully, including the SD Card requirements section.

INSTALL RASPBIAN OS ON A SD CARD

Raspberry Pi had created a graphical SD card tool for several Operating Systems: Windows, macOS, Linux. Using this tool is the easiest option to download the OS image and install it on the SD card (Pi, 2021).

The OS tool can be found on the Raspberry Pi website at the following link: Raspberry Pi OS – Raspberry Pi.

For a successful installation of the Raspbian OS, the following steps must be followed (Pi, 2021):

1. Download the OS Imager from Raspberry Pi OS – Raspberry Pi
2. Install the Imager
3. Connect the SD card to the computer that contains the Imager tool
4. Open the Imager and select the preferred OS

Figure 19
Source: Theodora Tataru, 2021

5. Select "CHOOSE OS" and select the preferred OS (For this project, the Raspberry Pi OS (32 bit) is essential)

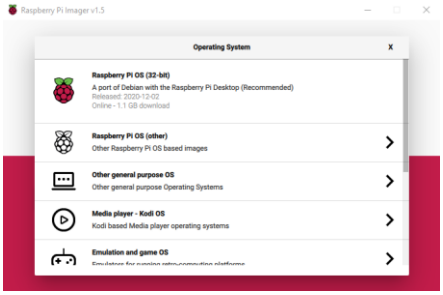


Figure 20
Source: Theodora Tataru, 2021

6. Select "CHOOSE SD CARD" and select the connected SD card

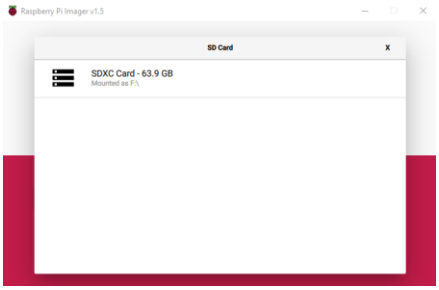


Figure 21
Source: Theodora Tataru, 2021

7. Review your selections and select “WRITE” from the menu and select “YES” as seen in **Figure 23**

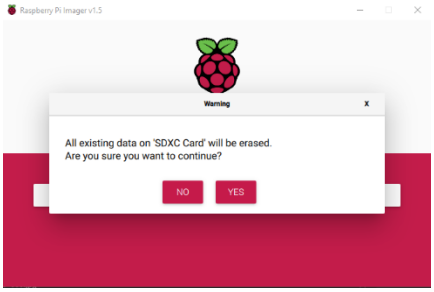


Figure 23
Source: Theodora Tataru, 2021

Figure 22
Source: Theodora Tataru, 2021

8. When the OS had been successfully written on the SD card, the following message is displayed:

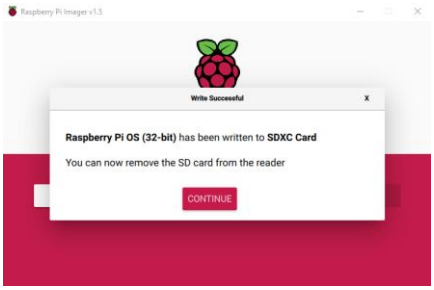


Figure 24
Source: Theodora Tataru, 2021

9. Select “CONTINUE”
10. Remove safety the SD card from the computer
11. Insert the SD card into the Raspberry Pi
12. Power on the Raspberry Pi
13. Follow the steps to set up the OS on the Raspberry Pi
    a. The default username is “pi”
    b. The default password is “raspberry”
    c. The default keyword set on the Pi is UK
    d. The default password should be changed straight away to secure the Pi

PYTHON

The “Pixel” Virtual Assistant project is implemented using Python3.

Most distributions of Linux come with Python3 already installed. To check if the system has Python3 installed , the following command can be used: Python –version. If Python is already installed on the system, its version is displayed following the command, as seen in **Figure 25**.

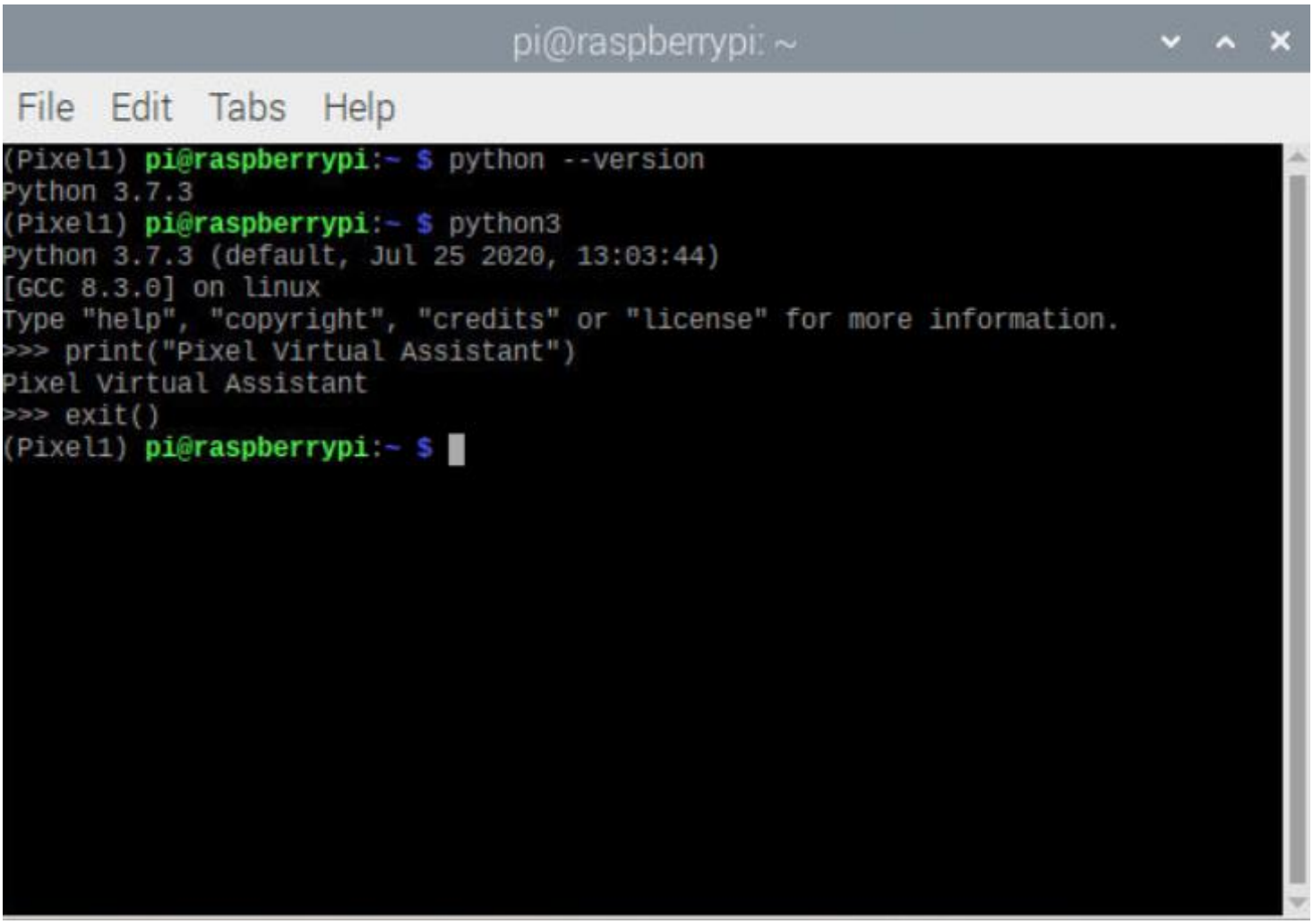


Figure 25
Source: Theodora Tataru, 2021

If Python is not installed, the following command can be used to install it: sudo apt-get install python3.7 . After the installation, check Python’s version using the following command: Python –version .

VIRTUAL ENVIRONMENT

A virtual environment is essential when a Python3 project is developed as it creates an insolated space on the system that contains the executables, packages, and modules used in the project. Python programs often have specific dependencies, and a virtual environment allows the developer to manage each project's distinct dependencies without ever interfering with other projects.

VIRTUALENV

For the "Pixel" Virtual Assistant project, Virtualenv was the virtual environment choice. To install Virtualenv on the system, follow the following steps:

1. Update the system
   sudo apt-get update
   sudo apt-get upgrade

2. Install virtualenv
   sudo apt install virtualenv

3. Create a "python-environments" directory in the user's home directory
   mkdir ~/python-environments

4. Navigate to the newly created directory
   cd ~/python-environments

5. Create a new environment (by default, the environment uses Python 2.5, but this issue will be solved later on). Replace **env** with a proper name for the environment
   virtualenv env

6. As Python3 is required for this project, the following command creates a virtual environment using Python3:
   virtualenv --python=python3 env

   Replace **env** with the name chosen for the virtual environment

7. Checking that the environment created is using Python3
   ls env/lib

   Replace **env** with the name chosen for the virtual environment

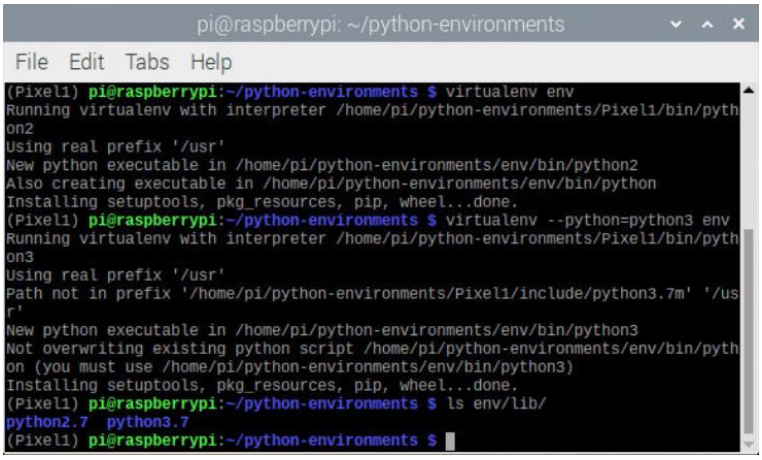


Figure 26
Source: Theodora Tataru, 2021

8. To activate the virtual environment:
   Replace **env** with the name chosen for the virtual environment
   `source env/bin/activate`

9. Deactivate the virtual environment
   `deactivate`

There are other virtual environments available for Python, and others may be used.

OPENCV

This section focuses on installing OpenCV on the Raspberry Pi, adding optimizations, and updating the deep learning module.

The installation of the OpenCV module was achieved by following Adrian Rosebrock's tutorial "Install OpenCV 4 on your Raspberry Pi", which can be found at https://www.pyimagesearch.com/2018/09/26/install-opencv-4-on-your-raspberry-pi/

STEP 1 - EXPANDING THE FILE SYSTEM

Firstly the Raspberry Pi needs to be turned on. The Pi needs a keyboard, a mouse and a display attached to it.

Once that is achieved, the next step is to expand the filesystem to include all available space on the SD card. This can be achieved by the following command: sudo raspi-config. The command, opens a window, as seen in **Figure 27**. Then, the "Advanced Options" menu must be selected.

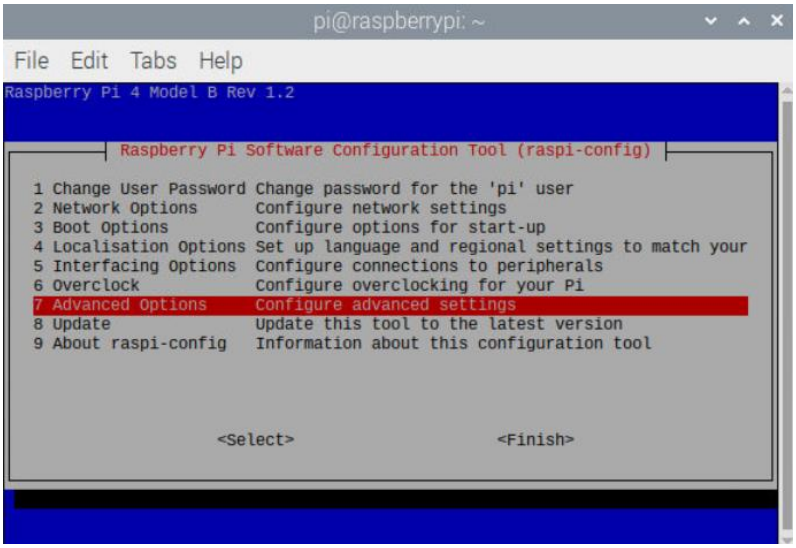


Figure 27
Source: Theodora Tataru, 2021

Followed by selecting "Expand Filesystem"

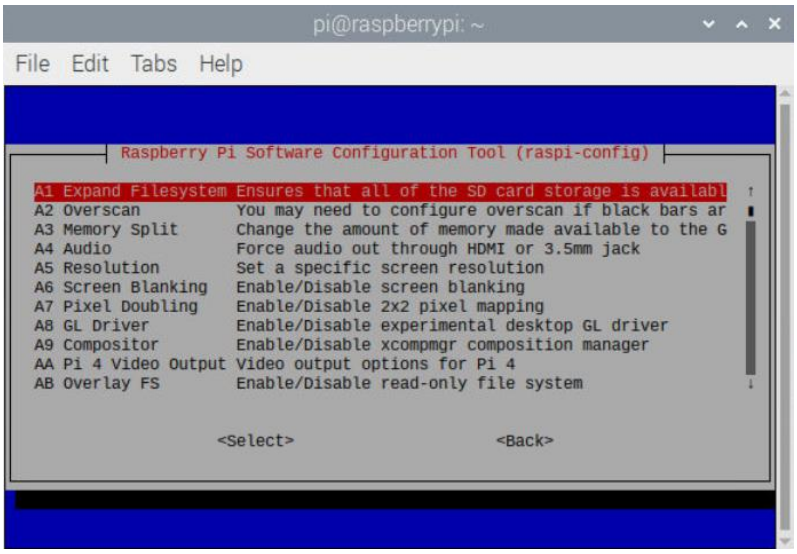
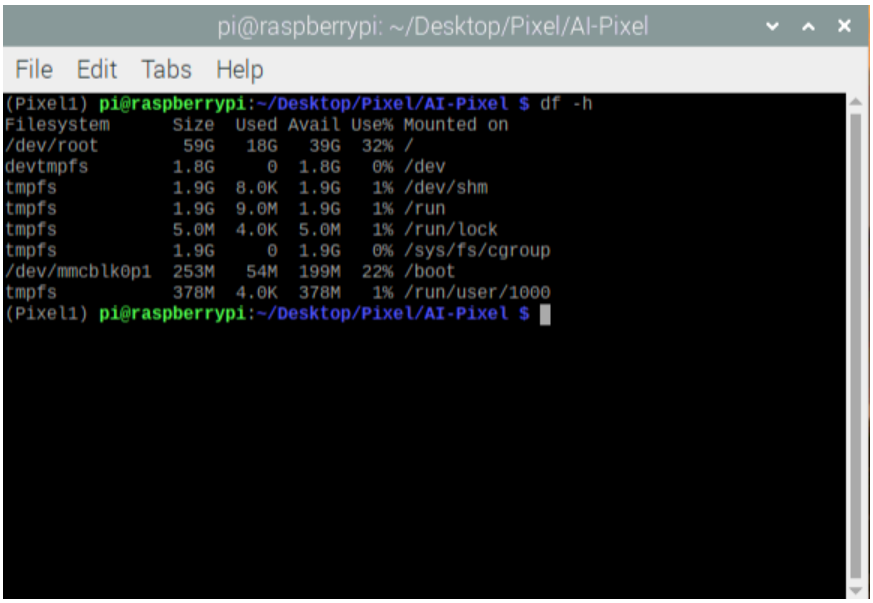


Figure 28
Source: Theodora Tataru, 2021

Once that is achieved, select "<Finish>" and reboot the system with the following command: sudo reboot now.

This cation enables the system to expand the file system to all available space on the SD card. To verify that the file system had expanded, the following command can be used: df -h.



As seen in **Figure 29**, the Raspbian filesystem had been expanded to include almost all 60GB of the SD card.

Figure 29
Source: Theodora Tataru, 2021

To develop "Pixel" Virtual Assistant, some packages that Linux has pre-installed can be uninstalled to free up some space on the SD card. If the card used is over 32GB, this step can be skipped.

To free up some space on the SD card, the following steps must be followed:

sudo apt-get purge wolfram-engine
sudo apt-get purge libreoffice*
sudo apt-get clean
sudo apt-get autoremove

STEP 2 - INSTALL OPENCV DEPENDENCIES

1. Updated the system
   sudo apt-get update
   sudo apt-get upgrade

2. Install the developer tool CMake
   sudo apt-get install build-essential cmake unzip pkg-config

   CMake is an open-source tool, cross-platform designed to build, test, and package software. It is used to control and monitor the compilation process of software (CMake, 2021).

3. Install the image and video libraries that are critical to work with images and video files for this project:
   sudo apt-get install libjpeg-dev libpng-dev libtiff-dev
   sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-dev
   sudo apt-get install libxvidcore-dev libx264-dev

4. Install GTK
   sudo apt-get install libgtk-3-dev

   GTK is a tool used to create GUIs with Python. It is used with OpenCV to create graphical interfaces for the camera.

5. Install a package that reduces GTK warnings
   sudo apt-get install libcanberra-gtk*

6. Installing numerical optimizations for OpenCV
   sudo apt-get install libatlas-base-dev gfortran

7. Install Python3 developer headers
   sudo apt-get install python3-dev

   This library is required as some dependencies require native extensions. In order to build these native extensions, the Python3 developer header must be installed.

STEP 3 - DOWNLOAD OPENCV ON THE RASPBERRY PI

1. Navigate to the home folder
   cd ~

2. Download OpenCV
   wget -O opencv.zip https://github.com/opencv/opencv/archive/4.0.0.zip
   wget -O opencv_contrib.zip https://github.com/opencv/opencv_contrib/archive/4.0.0.zip

3. Unzip the file
   unzip opencv.zip
   unzip opencv_contrib.zip

4. Rename the directories
   mv opencv-4.0.0 opencv
   mv opencv_contrib-4.0.0 opencv_contrib

STEP 4 - CONFIGURE PYTHON3 VIRTUAL ENVIRONMENT

1. Please see: virtual environment section

To verify that the virtual environment was activated, after typing the activation command, the terminal should look like in **Figure 30.** In front of each line, the name of the environment should be present in parenthesis.

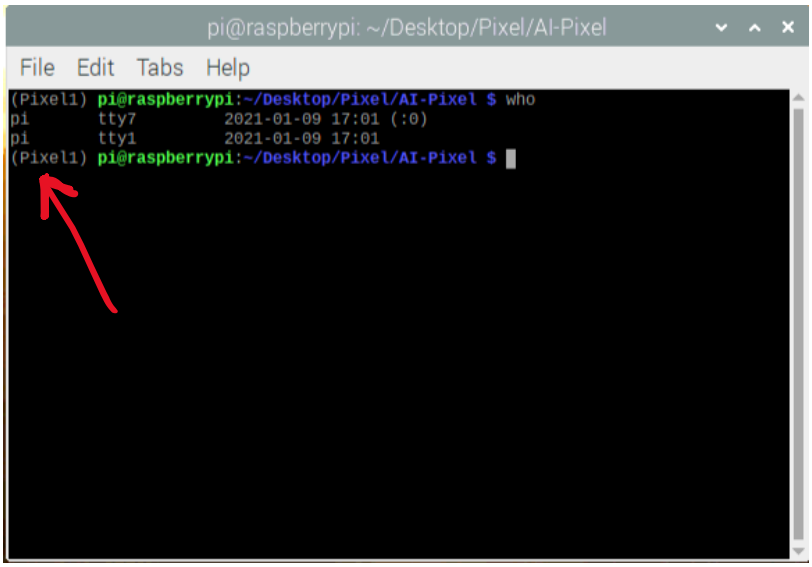


Figure 30
Source: Theodora Tataru, 2021

! Before proceeding, make sure that the Python3 virtual environment is activated.

2. Install NumPy
   python3 -m pip install NumPy

STEP 5 - CMAKE AND COMPILE OPENCV 4

1. Navigate to the OpenCV folder
   cd ~
   cd opencv

2. Create a new directory called "build", and navigate to it
   mkdir build
   cd build

3. Run CMake to configure OpenCV 4 with the following paramaters

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
        -D CMAKE_INSTALL_PREFIX=/usr/local \
        -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
        -D ENABLE_NEON=ON \
        -D ENABLE_VFPV3=ON \
        -D BUILD_TESTS=OFF \
        -D OPENCV_ENABLE_NONFREE=ON \
        -D INSTALL_PYTHON_EXAMPLES=OFF \
        -D BUILD_EXAMPLES=OFF ..
```

"Notice the -D OPENCV_ENABLE_NONFREE=ON flag. Setting this flag with OpenCV 4 ensures that you'll have access to SIFT/SURF and other patented algorithms." - (Rosebrock, 2018)

When the compilation is finished, the output should be similar to **Figure 31 and Figure 32.**

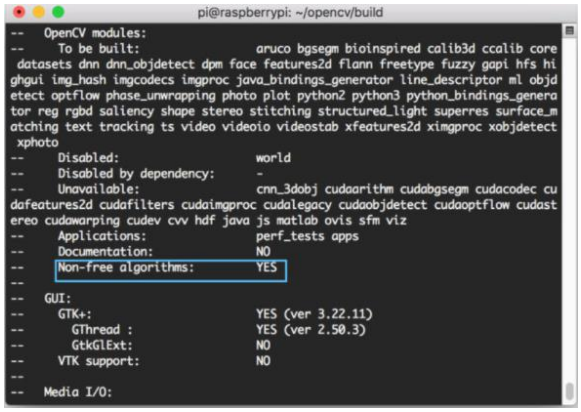


Figure 32
Source: (Rosebrock, 2018)

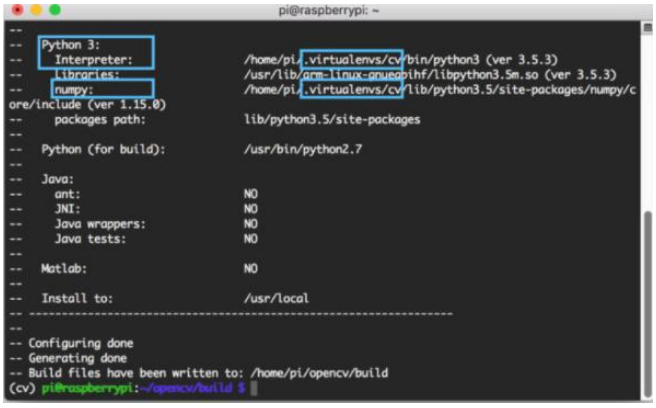


Figure 31
Source: (Rosebrock, 2018)

4. Increase the SWAP of the Raspberry Pi
   This step enables the system to compile OpenCV with all cores of the Raspberry Pi, without the compile hanging due to memory exhausting.
   a. Open the swap file
      sudo nano /etc/dphys-swapfile
   b. Edit the **CONF_SWAPSIZE** variable, from 100MB to 2048MB

```
# set size to absolute value, leaving empty (default) then uses computed value
# you most likely don't want this, unless you have an special disk situation
# CONF_SWAPSIZE=100
```

```
CONF_SWAPSIZE=2048
```

c. Restart the SWAP service

```
$ sudo /etc/init.d/dphys-swapfile stop
$ sudo /etc/init.d/dphys-swapfile start
```

5. Compile OpenCV 4

```
make
```

The compilation takes about 1 hour. Make sure that 100% compilation is achieved without errors.

6. Install OpenCV

```
sudo make install
sudo ldconfig
```

7. Reset CONF_SWAPFILE back to 100MB

a. Open the swap file

```
sudo nano /etc/dphys-swapfile
```

b. Edit the **CONF_SWAPSIZE** variable, back from 100MB to 2048MB

```
# set size to absolute value, leaving empty (default) then uses computed value
# you most likely don't want this, unless you have an special disk situation
CONF_SWAPSIZE=100
```

c. Restart the SWAP service

```
$ sudo /etc/init.d/dphys-swapfile stop
$ sudo /etc/init.d/dphys-swapfile start
```

STEP 6 - LINK OPENCV WITH THE PYTHON3 ENVIRONMENT

This is a critical step, as it links the OpenCV library to the virtual environment. Note, that if the symbolic link is not created; importing OpenCV into the Python scripts won't be available.

Linking the OpenCV to the virtual enfironment:

1. `cd ~/python-environments/env/cv/lib/python3.5/site-packages/`
   where **env** is the name of the virtual environment
2. ln -s /usr/local/python/cv2/python-3.5/cv2.cpython-35m-arm-linux-gnueabihf.so cv2.so
   ln creates a link from the file specified as the first argument (FILE) to the file specified as the second argument (LINK)

STEP 7 – TEST OPENCV INSTALL ON THE RASPBERRY PI

1. Activate the virtual environment

```
source ~/python-environments/env/bin/activate
```

Replace **env** with the name of your environment

2. Open a python shell

```
$ python
>>> import cv2
>>> cv2.__version__
'4.0.0'
>>> exit()
```

If a similar version is printed back in the python shell, after calling the cv2.__version__ is executed, the OpenCV had been successfully installed on the Raspberry P

PROJECT DEPENDENCIES

The project is dependent on several Python3 libraries:

- alabaster==0.7.12
- Babel==2.9.0
- beautifulsoup4==4.9.3
- certifi==2020.6.20
- chardet==3.0.4
- chromedriver==2.24.1
- click==7.1.2
- colorama==0.3.9
- covid==2.4.0
- cycler==0.10.0
- decorator==4.4.2
- dlib==19.21.0
- docutils==0.16
- face-recognition==1.3.0
- face-recognition-models==0.3.0
- fann2==1.1.2
- fasteners==0.15
- Flask==1.1.2
- functiontrace==0.3.4
- future==0.18.2
- geocoder==1.38.1
- geographiclib==1.50
- geolocation-python==0.2.2
- geopy==2.0.0
- gevent==20.9.0
- google==3.0.0
- greenlet==0.4.17
- gTTS==2.2.1
- idna==2.10
- imagesize==1.2.0
- importlib-metadata==3.3.0
- imutils==0.5.3
- inflection==0.5.1
- itsdangerous==1.1.0
- Jinja2==2.11.2
- joblib==1.0.0
- kiwisolver==1.2.0
- lingua-franca==0.2.3
- lxml==4.6.2
- MarkupSafe==1.1.1
- matplotlib==3.3.2
- monotonic==1.5
- newsapi==0.1.1
- newsapi-python==0.2.6
- nltk==3.3
- numpy==1.19.2
- packaging==20.8
- padaos==0.1.10
- padatious==0.4.8
- pantilthat==0.0.7
- picamera==1.13
- Pillow==8.0.1
- pkg-resources==0.0.0
- playsound==1.2.2
- psutil==5.7.3
- PyAudio==0.2.11
- pydantic==1.7.3
- pyee==8.1.0
- Pygments==2.7.3
- PyJWT==1.7.1
- pyparsing==2.4.7
- pyserial==3.5
- python-dateutil==2.6.0
- pyttsx3==2.90
- pytz==2020.5
- pyxdg==0.27
- ratelim==0.1.6
- regex==2020.11.13
- reportlab==3.5.59
- requests==2.24.0
- RPi.GPIO==0.7.0
- rst2pdf==0.98
- ruamel.yaml==0.16.12
- ruamel.yaml.clib==0.2.2
- six==1.15.0
- smartypants==2.0.1
- snakeviz==2.1.0
- snowballstemmer==2.0.0
- soupsieve==2.0.1
- SpeechRecognition==3.8.1
- Sphinx==3.4.2
- sphinx-rtd-theme==0.5.2
- sphinxcontrib-applehelp==1.0.2
- sphinxcontrib-devhelp==1.0.2
- sphinxcontrib-htmlhelp==1.0.3
- sphinxcontrib-jsmath==1.0.1
- sphinxcontrib-qthelp==1.0.3
- sphinxcontrib-serializinghtml==1.1.4
- tornado==6.1
- tqdm==4.56.0
- typer==0.3.2
- typing-extensions==3.7.4.3
- urllib3==1.25.11
- websocket==0.2.1
- websocket-client==0.57.0
- Werkzeug==1.0.1
- wikipedia==1.4.0
- xdg==5.0.0
- xxhash==2.0.0
- yummy-sphinx-theme==0.1.1
- zipp==3.4.0
- zope.event==4.5.0
- zope.interface==5.1.2

All these libraries are installed automatically when the source code is downloaded from GitHub, and the setup script is executed.

PYTHON SCRIPTS

FILES LAYOUT

```
pi@raspberrypi:~/Desktop/Pixel/AI-Pixel $ tree -L 2
        |-- active_user.py
        |-- api_keys.json
        |-- camera.py
        |-- Cascades
        |   |-- encodings.pickle
        |   |-- haarcascade_eye.xml
        |   |-- haarcascade_frontalface_default.xml
        |   `-- haarcascade_smile.xml
        |-- communication.py
        |-- display.py
        |-- Documents
        |   |-- active_user.rst
        |   |-- _build
        |   |-- camera.rst
        |   |-- communication.rst
        |   |-- conf.py
        |   |-- developer.rst
        |   |-- display.rst
        |   |-- gmsHat.log
        |   |-- GSM.rst
        |   |-- helper.rst
        |   |-- index.rst
        |   |-- license.rst
        |   |-- main.rst
        |   |-- make.bat
        |   |-- Makefile
        |   |-- modules.rst
        |   |-- skill.rst
        |   |-- _static
        |   |-- _templates
        |   |-- test.rst
        |   `-- virtual_assistant.rst
        |-- gmsHat.log
        |-- GSM.py
        |-- helper.py
        |-- main.py
        |-- program.prof
        |-- Requirements
        |   `-- requirements.txt
        |-- skill.py
        |-- test.py
        |-- User
        |   `-- Dora
        |-- Setup
        |    -- setup.sh
        `-- virtual_assistant.py
```

SPHINX

Sphinx is a tool, that generates Python, C and C++ documentation. Sphinx uses reStructuredText as a markup language, and it generates HTML, LaTeX, ePub, Texinfo and plain text documentation (Sphinx, 2021).

It is a perfect tool for programmers to document their code; therefore Sphinx was used to document the code and generate the project's website.

COORDINATOR

START()

```
def start(self):
        """
        The following are the three main processes of the system:

            1. camera process

            2. interface process

            3. the virtual assistant process

    Camera:
        The camera starts when the system starts, and its purpose is to detect if a person comes in front of the camera an
d calculate the time between the last moment when a face was detected and the current moment in time.

            :param user:
                - this variable contains the name of the user, if its registered, or contains the string 'stranger' if the
 user standing in front of the camera is not registered
            :type user: [str]
            :param delay:
                - this variable describes the amount of time since the camera did not detect a face
            :type delay: [float]
            :param timeFaceFound:
                - this variable marks the time when a face was detected the last time
            :type timeFaceFound: [float]
            :param faceFound:
                - True -> a face is detected by the camera
                - False -> there is no face detected by the camera
            :type faceFound: [bool]
            :param runCamera:
                - True -> camera is running from the Camera process
                - False -> the camera is not running from this particular process
            :type runCamera: [bool]
            :param cameraStopped:
                - the variable is used as double check, for the "runCamera" variable, that ensures that the camera is runn
ing or is stopped in the Camera process
            :type cameraStopped: [bool]


    Virtual Assistant:
        The virtual assistant process stops and starts, depending on the idle time calculated by the camera. The idle time
 represents the time difference in seconds since the camera lastly detected a face:
            - if the idle time is greater than 20 seconds, the virtual assistant process is destroyed
            - if the idle time is less than 20 seconds and the virtual assistant is not started, then the process is creat
ed
            - if the virtual assistant process is running, and the idle time is less than 20 seconds, the virtual assistan
t process runs until the idle time is greater than 20 seconds

            :param answer:
                - contains the response provided by the virtual assistant to the user's request
            :type answer: [str]
            :param virtualAssistantStatus:
                - contains the virtual assistant status: processing, listening, answering, calling, etc.
            :type virtualAssistantStatus: [str]
            :param understanding :
                - contains a string that is displayed on the interface that represents the request that was understood by
the system
            :type understanding: [str]
            :param root:
                - GUI object instantiated as the main process
            :type root: [TKinter object]
            :param AIStarted:
                - boolean variable that reflects the virtual_assistant status

                - True -> the virtual assistant status is running

                - False -> the virtual assistant is not running
```

```
                - used like a semaphore, not allowing the instantiation of the virtual_assistant multiple times if the vir
tual assistant runs already
            :type AIStarted: [bool]
        """

        with Manager() as manager:
            ## variables shared between the camera and the display
            user = manager.Value('s', "stranger")
            delay = manager.Value('f', 0)
            timeFaceFound = manager.Value('f', time.time())
            faceFound = manager.Value('b', False)
            runCamera = manager.Value('b', True)
            capture = Process(target=camera.start,args=(user, faceFound, timeFaceFound, delay, runCamera))
            capture.start()
            cameraStopped = manager.Value('b', False)

            ## variables shares between the display and the virtual_assistance
            answer = manager.Value('s','')
            virtualAssistantStatus = manager.Value('s','')
            understanding = manager.Value('s',' ')
            cameraRunning = manager.Value('b', True)
            assistant = Process(target=virtual_assistant.start, args=(user, answer,virtualAssistantStatus, understanding,
cameraRunning))

            ## starting the GUI
            AIStarted = False
            initiatedOnce = True
            GUI_interface = display.Display(user, answer, virtualAssistantStatus, understanding)
            root = GUI_interface.root
            root.attributes("-fullscreen", True)
            root.configure(background='black')

            while True:
                if cameraRunning.value == False and cameraStopped.value == False:
                    runCamera.value = False
                    capture.terminate()
                    capture.join()
                    cameraStopped.value = True
                if cameraRunning.value == True and cameraStopped.value == True:
                    runCamera.value = True
                    delay.value = 1
                    capture = Process(target=camera.start,args=(user, faceFound, timeFaceFound, delay, runCamera))
                    capture.start()
                    cameraStopped.value = False
                if faceFound.value and not AIStarted and initiatedOnce:
                    assistant.start()
                    AIStarted = True
                    initiatedOnce = False
                if faceFound.value and not AIStarted and not initiatedOnce:
                    assistant = Process(target=virtual_assistant.start, args=(user, answer,virtualAssistantStatus, underst
anding, cameraRunning))
                    assistant.start()
                    AIStarted = True
                if delay.value > 20 and AIStarted:
                    assistant.terminate()
                    assistant.join()
                    answer.value = ""
                    virtualAssistantStatus.value = ""
                    understanding.value = ""
                    AIStarted = False
                    #print("Idle time: ", delay.value)
                root.update()
            #root.mainloop()
```

class **main.Coordinator**

Bases: `object`

start()

The following are the three main processes of the system:

1. camera process
2. interface process
3. the virtual assistant process

Camera:

The camera starts when the system starts, and its purpose is to detect if a person comes in front of the camera and calculate the time between the last moment when a face was detected and the current moment in time.

param user: • this variable contains the name of the user, if its registered, or contains the string 'stranger' if the user standing in front of the camera is not registered

type user: [str]

param delay: • this variable describes the amount of time since the camera did not detect a face

type delay: [float]

param timeFaceFound: • this variable marks the time when a face was detected the last time

type timeFaceFound: [float]

param faceFound: • True -> a face is detected by the camera
• False -> there is no face detected by the camera

type faceFound: [bool]

param runCamera: • True -> camera is running from the Camera process
• False -> the camera is not running from this particular process

type runCamera: [bool]

param cameraStopped:
- the variable is used as double check, for the "runCamera" variable, that ensures that the camera is running or is stopped in the Camera process

type cameraStopped: [bool]

Virtual Assistant:

The virtual assistant process stops and starts, depending on the idle time calculated by the camera. The idle time represents the time difference in seconds since the camera lastly detected a face:

- if the idle time is greater than 20 seconds, the virtual assistant process is destroyed
- if the idle time is less than 20 seconds and the virtual assistant is not started, then the process is created
- if the virtual assistant process is running, and the idle time is less than 20 seconds, the virtual assistant process runs until the idle time is greater than 20 seconds

param answer:
- contains the response provided by the virtual assistant to the user's request

type answer: [str]

param virtualAssistantStatus:
- contains the virtual assistant status: processing, listening, answering, calling, etc.

type virtualAssistantStatus: [str]

:param understanding :

- contains a string that is displayed on the interface that represents the request that was understood by the system

type understanding: [str]

param root:
- GUI object instantiated as the main process

type root: [TKinter object]

param AIStarted:
- boolean variable that reflects the virtual_assistant status
- True -> the virtual assistant status is running
- False -> the virtual assistant is not running
- used like a semaphore, not allowing the instantiation of the virtual_assistant multiple times if the virtual assistant runs already

type AIStarted: [bool]

START()

```
def start(user:str, faceFound:bool, timeFaceFound:float, delay:float, runCamera:bool):
    """[This process controls the camera, and informs the main Coordinator process if a face detected, since when the came
ra did not detect a face and if the user in front of the camera is registered or not]

    :param user: [The name of the user in front of the camera (user == 'stranger' if the user is not registered on the dev
ice)]
    :type user: str
    :param faceFound:

        - True -> a face is detected by the camera

        - False -> no face is detected by the camera

    :type faceFound: bool
    :param timeFaceFound: [Last time a face was detected by the camera]
    :type timeFaceFound: float
    :param delay: [How many seconds ago a face was detected by the camera]
    :type delay: float
    :param runCamera:

        - True -> the camera is running from this process

        - False -> the camera runs from another process


    :type runCamera: bool
    """
    ## load the known faces and embeddings along with OpenCV's Haar cascade for face detection
    print("[INFO] loading encodings + face detector...")
    detector = cv2.CascadeClassifier("Cascades/haarcascade_frontalface_default.xml")
    data = pickle.loads(open("Cascades/encodings.pickle", "rb").read())

    ## initialize the video stream and allow the camera sensor to warm up
    print("[INFO] starting video stream...")
    cam = cv2.VideoCapture(0)
    while runCamera:
        ## grab the frame from the threaded video stream and resize it
        ## to 500px (to speedup processing)
        ret, frame = cam.read()
        #  frame = imutils.resize(frame, width=300)
        ## convert the input frame from (1) BGR to grayscale (for face detection) and (2) from BGR to RGB (for face recogn
ition)
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        ## detect faces in the grayscale frame
        rects = detector.detectMultiScale(gray, scaleFactor=1.1,
            minNeighbors=5, minSize=(30, 30),
            flags=cv2.CASCADE_SCALE_IMAGE)
        ## OpenCV returns bounding box coordinates in (x, y, w, h) order
        ## but we need them in (top, right, bottom, left) order, so we
        ## need to do a bit of reordering
        boxes = [(y, x + w, y + h, x) for (x, y, w, h) in rects]

        ## compute the facial embeddings for each face bounding box
        encodings = face_recognition.face_encodings(rgb, boxes)
        #  print("[INFO] encodings: ", encodings)

        ## if a face is dected, update when the face was last seen
        if len(encodings) > 0:
            faceFound.value = True
            timeFaceFound.value = time.time()
        else:
        ## else if no face is found, calculate since when a face was not detected (in seconds)
            faceFound.value = False
            delay.value = idle(timeFaceFound.value)
            print("[INFO] Idle time: ", delay.value)
        names = []
```

```
        ## loop over the facial embeddings
        for encoding in encodings:
            ## attempt to match each face in the input image to our known encodings
            matches = face_recognition.compare_faces(data["encodings"],
                encoding)
            name = "stranger"

            ## check to see if we have found a match
            if True in matches:
                ## find the indexes of all matched faces then initialize a
                ## dictionary to count the total number of times each face was matched
                matchedFaceVectors = [i for (i, b) in enumerate(matches) if b]
                counts = {}

                ## loop over the matched indexes and maintain a count for
                ## each recognized face face
                for i in matchedFaceVectors:
                    name = data["names"][i]
                    counts[name] = counts.get(name, 0) + 1

                ## determine the recognized face with the largest number
                ## of votes (note: in the event of an unlikely tie Python
                ## will select first entry in the dictionary)
                name = max(counts, key=counts.get)
                #  print("[INFO] Counts: ", counts)

            ## update the list of names
            names.append(name)
            user.value = name
            #  print("[INFO] Name: ", name)
        ## loop over the recognized faces
        ## print("[INFO] faceFound: ", faceFound.value)
        for ((top, right, bottom, left), name) in zip(boxes, names):
            ## draw the predicted face name on the image
            cv2.rectangle(frame, (left, top), (right, bottom),
                (0, 255, 0), 2)
            y = top - 15 if top - 15 > 15 else top + 15
            cv2.putText(frame, name, (left, y), cv2.FONT_HERSHEY_SIMPLEX,
                0.75, (0, 255, 0), 2)

        # display the image to our screen
        cv2.imshow("Frame", frame)

    ## if camera is not used by this process, release the video device
    if(runCamera.value == False):
        cam.release()
        cv2.destroyAllWindows()
        print("[INFO] Main camera stopped")
```

camera.start(*user: str, faceFound: bool, timeFaceFound: float, delay: float, runCamera: bool*)

[This process controls the camera, and informs the main Coordinator process if a face detected, since when the camera did not detect a face and if the user in front of the camera is registered or not]

Parameters:

- **user** (*str*) – [The name of the user in front of the camera (user == 'stranger' if the user is not registered on the device)]
- **faceFound** (*bool*) –
  - True -> a face is detected by the camera
  - False -> no face is detected by the camera
- **timeFaceFound** (*float*) – [Last time a face was detected by the camera]
- **delay** (*float*) – [How many seconds ago a face was detected by the camera]
- **runCamera** (*bool*) –
  - True -> the camera is running from this process
  - False -> the camera runs from another process

IDLE()

```
def idle(timeFaceFound):
    """[Method that calculates the time difference in seconds, between this moment in time and last time a face was detect
ed.]

    :param timeFaceFound: [Last time a face a detected by the camera]
    :type timeFaceFound: [float]
    :return: [Time difference in seconds, since now and the last time a face was detected by the camera]
    :rtype: [float]
    """
    idleTime = time.time() - timeFaceFound
    return idleTime
```

camera.idle(*timeFaceFound*)

[Method that calculates the time difference in seconds, between this moment in time and last time a face was detected.]

Parameters: **timeFaceFound** (*[float]*) – [Last time a face a detected by the camera]

Returns: [Time difference in seconds, since now and the last time a face was detected by the camera]

Return type: [float]

DISPLAY

```
class Display:
    """[This class is the interface of the system and the main process of the system]
    """
    root = Tk()
    ## Dividing the screen in 1 column and 5 rows
    root.grid_rowconfigure(5, weight=1)
    root.grid_columnconfigure(1, weight=1)

    def __init__(self, user:str, response:str, virtualAssistantStatus:str, understanding:str):
        """[Constructor of the Display class]

        :param user: [The name of the user in front of the mirror (if user is not registered on the device, user == 'stran
ger')]
        :type user: str
        :param response: [The answer that the virtual assistant has for the user's request, in text format]
        :type response: str
        :param virtualAssistantStatus: [Virtual Assistant's status (listening, processing, answering, calling, etc)]
        :type virtualAssistantStatus: str
        :param understanding: [What the virtual assistant understood from the user's request]
        :type understanding: str
        """
        self.response = response
        self.virtualAssistantStatus = virtualAssistantStatus
        self.understanding = understanding
        #self.video_source = 0

        ## Time Displayed
        self.time1 = ''
        self.time2 = datetime.datetime.now().strftime("%I:%M:%S %p")
        self.clock_label = Label(self.root, text=self.time2, font=('Helvica', 45), fg='white')
        self.clock_label.configure(background='black')
        self.clock_label.grid(row=0, column=0, padx=1, pady=30, sticky=W)
        self.updateTime()

        ## AI Processing Message
        self.AI_processing_message = "Pixel started"
        self.AI_processing_label = Label(self.root, justify=LEFT ,text = self.AI_processing_message, font=("Helvica", 20),
 bg="black", fg="white")
        self.AI_processing_label.grid(row=0, column=3, padx=1, sticky=E)
        self.displayVirtualAssistantStatus()

        ## Virtual assistant understanding
        self.understanding_text = " "
        self.understanding_label = Label(self.root, justify=LEFT, text = self.understanding_text, font=("Helvica", 17), bg
="black", fg="yellow" )
        self.understanding_label.grid(row=1, column=0, padx=1,pady=10, sticky=W)
        self.displayVirtualAssistantUnderstanding()

        ## Virtual Assistant Response Displayed
        self.greeting_text = ""
        self.AI_Message_label = Label(self.root, justify=LEFT, text=self.greeting_text, font=('System', 20), bg='black', f
g='white')
        self.AI_Message_label.grid(row=4, column=0, padx=1, pady=30, sticky=W)
        self.displayResponse()

        ## Camera
        #self.video = camera.Camera(self.video_source)
        #self.video_frame = Canvas(self.root, width=self.video.width, height=self.video.height)
        #self.video_frame.grid(row=2, column=0, padx=45, pady=30, sticky=W)
        #self.updatedCameraFrame()
```

class **display.Display**(*user: str, response: str, virtualAssistantStatus: str, understanding: str*)

Bases: `object`

[This class is the interface of the system and the main process of the system]

UPDATETIME()

```
## update time on GUI
    def updateTime(self):
        """[The method updates the current time on the display]
        """
        self.time2 = datetime.datetime.now().strftime("%I:%M:%S %p")
        self.clock_label.configure(text=self.time2)
        self.clock_label.after(200, self.updateTime)
```

updateTime()

[The method updates the current time on the display]

DISPLAYRESPONSE()

```
## update virtual assistant's response on GUI
    def displayResponse(self):
        """[The method displays on the screen the virtual assistant's response to the user's request]
        """
        self.answer = self.response.value
        self.AI_Message_label.configure(text=self.answer)
        self.AI_Message_label.after(200, self.displayResponse)
```

displayResponse()

[The method displays on the screen the virtual assistant's response to the user's request]

DISPLAYVIRTUALASSISTANTSTATUS()

```
## update virtual assistant's status on GUI
    def displayVirtualAssistantStatus(self):
        """[The method displays on the screen the virtual assistant's status: listening, answering, etc.]
        """
        self.virtualAssistantStatus_message = self.virtualAssistantStatus.value
        self.AI_processing_label.configure(text=self.virtualAssistantStatus_message)
        self.AI_processing_label.after(200, self.displayVirtualAssistantStatus)
```

displayVirtualAssistantStatus()

[The method displays on the screen the virtual assistant's status: listening, answering, etc.]

DISPLAYVITRUALASSISTANTUNDERSTANDING()

```
## update virtual assistant's understanding
    def displayVirtualAssistantUnderstanding(self):
        """[The method displays on the screen what the virtual assistant understood from the user's request]
        """
        self.understanding_text = self.understanding.value
        self.understanding_label.configure(text=self.understanding_text)
        self.understanding_label.after(200, self.displayVirtualAssistantUnderstanding)
```

displayVirtualAssistantUnderstanding()

[The method displays on the screen what the virtual assistant understood from the user's request]

CLOSE()

```
def close(self):
        """[The method closes the interface]
        """
        self.root.destroy()
    '''
    def updatedCameraFrame(self):
        ret, frame = self.video.getFrame()
        if ret:
            self.photo = ImageTk.PhotoImage(image = Image.fromarray(frame))
            self.video_frame.create_image(0,0, image = self.photo)
        self.video_frame.after(15, self.updatedCameraFrame) '''
```

close()

[The method closes the interface]

START()

```
def start(user:str, response:str, AIstatus:str, understanding:str, cameraRunning:bool):
    """[This method starts the virtual assistant, that will process the user's request and delivers a meaningful answer]

    :param user: [The user in front of the camera]
    :type user: str
    :param response: [The answer from the virtual's assistant in text format that is displayed to the screen]
    :type response: str
    :param AIstatus: [The virtual assistant's status displated to the screen]
    :type AIstatus: str
    :param understanding: [The user's voice input, in text format]
    :type understanding: str
    :param cameraRunning: Camera module running from the main Camera process:

        - True -> Camera is running from the main Camera process

        - False -> Camera module is not running from the main Camera process

    :type cameraRunning: bool
    """
    AIstatus.value = status["start"]
    GPS = communication.Communication()
    GPS.start_GPS()
    greet = skill.Greeting()
    greeting = greet.prepare(user)
    greet.run(greeting, response)
    #clean_object(greet)


    while True:
        request = listening(AIstatus)
        request = request.lower()
        understanding.value = "Responding to: " + request

        if "pixel" in request:
            if not "help" in request and not "how do i" in request:
                if "register" in request or "sign up" in request:
                    register = skill.Register()
                    exists = register.user_exists(user, response)
                    if exists == False:
                        name = register.get_name(AIstatus, response, understanding, user)
                        confirm = register.prepare(name)
                        if confirm == True:
                            register.run(AIstatus, cameraRunning, name, response, status)
                    #clean_object(register)

                elif "open" in request or "opening hour" in request or "address"  in request or "where" in request or "cal
l" in request:
                    location_details = skill.Location()
                    request = location_details.prepare(request)
                    if "call" in request:
                        location_details.run(AIstatus, request, response, status, GPS, call=True)
                    else:
                        location_details.run(AIstatus, request, response, status, GPS, call=False)
                    ##clean_object(location_details)

                elif "time" in request:
                    time_now = skill.Time()
                    now = time_now.prepare(AIstatus, status)
                    time_now.run(now, AIstatus, response, status)
                    #clean_object(time_now)

                elif "date" in request:
                    date_today = skill.Date()
                    date = date_today.prepare(AIstatus, status)
                    date_today.run(date, AIstatus, response, status)
                    #clean_object(date_today)
```

```
        elif "weather" in request or "temperature" in request:
            #TODO: put forecast and temperature here
            #TODO: play more with weather, has issues!
            weather = skill.Weather()
            time.sleep(3)
            city = weather.prepare(AIstatus, status, request, GPS)
            try:
                weather.run(city, AIstatus, response, status)
            except:
                weather.error(user, response, AIstatus, status, city)
            #clean_object(weather)

        elif "covid" in request:
            covid = skill.Covid19()
            country = covid.prepare(request, AIstatus, status)
            try:
                covid.run(country, response, status, AIstatus)
            except:
                covid.error(user, country, response, AIstatus, status)
            #clean_object(covid)

        elif "thank you" in request:
            thank_you_message = skill.Thank_you()
            answer = thank_you_message.prepare(user, AIstatus, status)
            thank_you_message.run(response, AIstatus, status, answer)
            #clean_object(thank_you_message)

        elif "define" in request or "definition" in request or "tell me about" in request:
            definition = skill.Definition()
            word = definition.prepare(AIstatus, request, status)
            try:
                definition.run(response, AIstatus, word, status)
            except:
                definition.error(user, word, response, AIstatus, status)
            clean_object(definition)

        elif "bye" in request or "see you" in request:
            bye = skill.Good_bye()
            answer = bye.prepare()
            bye.run(AIstatus, response, answer, status)
            #clean_object(bye)

        elif "send" in request and "sos" in request or "emergency" in request:
            this_user = active_user.User(user)
            name = this_user.get_user(user)
            emergency = skill.Emergency(name)
            user_is_registered = this_user.is_user_registered()
            user_has_SOS_contact = this_user.user_has_SOS_contact()
            emergency.run(response, AIstatus, GPS, status, user_is_registered, user_has_SOS_contact)
            #clean_object(emergency)

        elif "sos" in request or "emergency" in request and not "send" in request:
            #TODO: play more with sos, has problems
            this_user = active_user.User(user)
            name = this_user.get_user(user)
            user_is_registered = this_user.is_user_registered()
            SOS = skill.SOS(name)
            if "add" in request or "create" in request:
                SOS.run_add_contact(response, AIstatus, GPS, understanding, status, user_is_registered)
            #clean_object(SOS)

        elif "list" in request:
            this_user = active_user.User(user)
            name = this_user.get_user(user)
            lists = skill.List(name)
            if this_user.is_user_registered():
                if "create" in request:
                    lists.run_create_list(response, AIstatus, understanding, status)
                elif "lists" in request or "all" in request:
                    lists.see_all_lists(AIstatus, status, response)
                elif "add" in request or "put" in request:
                    lists.run_add_item_to_list(AIstatus, response, status, understanding, request)
```

```
                    elif "my" in request or "show" in request or "see" in request:
                        lists.run_see_list(AIstatus, response, status, request, understanding)
                    elif "remove" in request or "delete" in request and "from" in request:
                        lists.run_remove_item_from_list(AIstatus, status, understanding, response, request)
                    elif "remove" in request or "delete" in request and not "from" in request:
                        lists.run_delete_list(AIstatus, response, status, understanding)
                else:
                    lists.error(response, AIstatus, status)
            else:
                no_skill_request = skill.not_understanding()
                no_skill_request.run(response)
                #clean_object(no_skill_request)

        elif "help" in request or "how do i" in request:
            help = skill.Help()
            help_for = help.prepare(AIstatus, status, request)
            try:
                help.run(help_for, AIstatus, status, response, user)
            except:
                help.error(AIstatus, status, response, user)
```

virtual_assistant.start(*user: str, response: str, AIstatus: str, understanding: str, cameraRunning: bool*)

[This method starts the virtual assistant, that will process the user's request and delivers a meaningful answer]

Parameters:

- **user** (*str*) – [The user in front of the camera]
- **response** (*str*) – [The answer from the virtual's assistant in text format that is displayed to the screen]
- **AIstatus** (*str*) – [The virtual assistant's status displated to the screen]
- **understanding** (*str*) – [The user's voice input, in text format]
- **cameraRunning** (*bool*) –

  Camera module running from the main Camera process:
  - True -> Camera is running from the main Camera process
  - False -> Camera module is not running from the main Camera process

CLEAN_OBJECT()

```
def clean_object(obj:object):
    """[Method that cleans the objects created]

    :param obj: [Skill object]
    :type obj: Skill
    """
    print("[INFO] cleaning created objects... Deleted: " + str(obj))
    del obj
```

virtual_assistant.clean_object(*obj: object*)

[Method that cleans the objects created]

Parameters: **obj** (*Skill*) – [Skill object]

LISTENING()

```
def listening(AIStatus:str):
    """[Method that listens to the user's voice input (using voice recognition) and transforms it into text]

    :param AIStatus: [The virtual assistant's status: e.g.: processing, listeninging, etc]
    :type AIStatus: str
    :return: [The virtual assistant's response to the user's request]
    :rtype: [str]
    """
    req = speech.Recognizer()
    request = ""
    with speech.Microphone(device_index=3,sample_rate=48000) as microphoneSource:
        ## gathering the voice input
        AIStatus.value = "Pixel is listening..."
        req.pause_threshold = 0.5
        req.adjust_for_ambient_noise(microphoneSource)
        audio = req.listen(microphoneSource, phrase_time_limit=10)
        try:
            ## translate the voice input into text
            AIStatus.value = "Pixel heard you..."
            request = req.recognize_google(audio, language='en')
            print("[INFO] request: " + request)
        except:
            ## if there was no voice input
            AIStatus.value = "Pixel cannot hear you..."
            #speak("Say that again please")
            request = ""
            return request
    return request
```

virtual_assistant.listening(*AIStatus: str*)

[Method that listens to the user's voice input (using voice recognition) and transforms it into text]

Parameters: **AIStatus** (*str*) – [The virtual assistant's status: e.g.: processing, listeninging, etc]

Returns: [The virtual assistant's response to the user's request]

Return type: [str]

LISTEN_DECISION()

```
def listen_decision(AIStatus:str):
    """[Method that listens to the user's decision in audio format (using voice recognition) and transforms it into text]

    :param AIStatus: [The virtual assistant's status: e.g.: processing, listeninging, etc]
    :type AIStatus: str
    :return: [The virtual assistant's response to the user's request]
    :rtype: [str]
    """
    req = speech.Recognizer()
    request = ""
    with speech.Microphone(device_index=3,sample_rate=48000) as microphoneSource:
        ## gathering the voice input
        AIStatus.value = "Pixel is listening..."
        req.adjust_for_ambient_noise(microphoneSource)
        req.pause_threshold = 0.5
        # TODO: ! phrase_time_limit needs to be removed!!!
        audio = req.listen(microphoneSource)#, phrase_time_limit=10)
        try:
            ## translate the voice input into text
            AIStatus.value = "Pixel heard you..."
            request = req.recognize_google(audio, language='en')
            print(request)
        except:
            ## if there was no voice input
            AIStatus.value = "Pixel cannot hear you..."
            #speak("Say that again please")
            request = ""
            return request
    return request
```

virtual_assistant.listen_decision(*AIStatus: str*)

[Method that listens to the user's decision in audio format (using voice recognition) and transforms it into text]

Parameters:	**AIStatus** (*str*) – [The virtual assistant's status: e.g.: processing, listeninging, etc]
Returns:	[The virtual assistant's response to the user's request]
Return type:	[str]

PLAYAUDIOFILE()

```
def playAudiofile():
    """[Playing the virtual assistant's response back to the user in audio format]
    """
    ## playing the audio file using vlc
    os.system('cvlc text.mp3 --play-and-exit')
```

virtual_assistant.playAudiofile()

[Playing the virtual assistant's response back to the user in audio format]

SPEAK()

```
def speak(text:str):
    """[This method transforms the virtual assistant's response from text to voice, using gTTS (Google's voice)]

    :param text: [String that needs to be transformed into an audio output]
    :type text: str
    """
    speakText = gTTS(text)
    ## saving the audio file
    speakText.save("text.mp3")
    ## playing the video file
    playAudiofile()
    ##removing the video file
    os.remove("text.mp3")
```

virtual_assistant.speak(*text: str*)

[This method transforms the virtual assistant's response from text to voice, using gTTS (Google's voice)]

Parameters: **text** (*str*) – [String that needs to be transformed into an audio output]

USER

```
class User():
    """[This class is dedicated to the user standing in from of the camera]
    """
    def __init__(self, user:str):
        """[Constructor of the class]

        :param user: [The user standing in front of the camera. user == 'stranger' if the user is not registered on the de
vice]
        :type user: str
        """
        self.name = self.get_user(user)
        print("[INFO constructor] " + str(self.name))
```

class **active_user.User**(*user: str*)

Bases: `object`

[This class is dedicated to the user standing in from of the camera]

GET_USER()

```
def get_user(self, user:str):
        """[Method that returns the name of the user in front of the camera

        - user == stranger if the user is not registered

        - user == name the user registered with]

        :param user: [The user standing in front of the camera]
        :type user: str
        :return: [the name of the user --> 'stranger' or the name the user registered with]
        :rtype: str
        """
        name = str(user.value)
        if "stranger" not in name:
            print("[INFO] " + name)
            return name
        else:
            return "stranger"
```

get_user(*user: str*)

[Method that returns the name of the user in front of the camera

- user == stranger if the user is not registered
- user == name the user registered with]

Parameters:	**user** (*str*) – [The user standing in front of the camera]
Returns:	[the name of the user –> ‘stranger’ or the name the user registered with]
Return type:	str

IS_USER_REGISTERED()

```
def is_user_registered(self):
        """[Method that returns TRUE if the user in front of the camera is registered on the device, or FALSE if the user
is not registered on the device]

        :return:

            - True -> the user is registered on the device

            - False -> the user is not registered on the device
        :rtype: [bool]
        """
        name = str(self.name)
        if "stranger" in name:
            print("[INFO] is_user_registered(): False")
            return False
        else:
            print("[INFO] is_user_registered(): True")
            return True
```

is_user_registered()

[Method that returns TRUE if the user in front of the camera is registered on the device, or FALSE if the user is not registered on the device]

Returns:

- True -> the user is registered on the device
- False -> the user is not registered on the device

Return type: [bool]

USER_HAS_SOS_CONTACT()

```
def user_has_SOS_contact(self):
        """[The method returns TRUE if the user in question has created an SOS contact]

        :return:

            - True -> an SOS contact exists for the user in front of the camera

            - False -> an SOS contact does not exist for the user in front of the camera

        :rtype: [bool]
        """
        if self.is_user_registered():
            name = str(self.name)
            path = "User/" + name + "/" + "SOS_contact"
            with open(path, 'rb') as f:
                x = pickle.load(f)
                if "name" in x and "number" in x:
                    return True
                else:
                    return False
        else:
            return False
```

user_has_SOS_contact()

[The method returns TRUE if the user in question has created an SOS contact]

Returns:

- True -> an SOS contact exists for the user in front of the camera
- False -> an SOS contact does not exist for the user in front of the camera

Return type: [bool]

SKILL – ABSTRACT

```
class Skill(ABC):
    """[This class is an abstract class for Skill derivated classes]

    """
    def prepare(*argv):
        """[Method that prepares all the necessary resources to run the skill]
        """
        pass
    def run(*argv):
        """[Method that runs the skill and provides the information back to the user]
        """
        pass
    def error(*argv):
        """[Method that runs, if the run() method fails]
        """
        pass
```

class **skill.Skill**

Bases: `abc.ABC`

[This class is an abstract class for Skill derivated classes]

error()

[Method that runs, if the run() method fails]

prepare()

[Method that prepares all the necessary resources to run the skill]

run()

[Method that runs the skill and provides the information back to the user]

COVID19

```
class Covid19(Skill):
    """[This skill provides the user, with covid-19 statistics from www.worldometers.info]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Covid19**

Bases: `skill.Skill`

[This skill provides the user, with covid-19 statistics from www.worldometers.info]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, request:str, AIstatus:str, status:dict):
        """[Method that prepares the environment to retrieve Covid-19 statistics]

        :param request: [User's request in raw format]
        :type request: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :return: [Country for which Covid-19 statistics are required]
        :rtype: [str]
        """
        AIstatus.value = status["process"]
        operator = helper.Context(helper.word_after())
        country = operator.run(request, "in")
        print("[INFO] city: ", country)
        return country
```

prepare(*request: str, AIstatus: str, status: dict*)

[Method that prepares the environment to retrieve Covid-19 statistics]

Parameters:
- **request** (*str*) – [User’s request in raw format]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

Returns: [Country for which Covid-19 statistics are required]

Return type: [str]

RUN()

```
def run(self, country:str, response:str, status:dict, AIstatus:str):
        """[Method that retrieves the Covid-19 statistics, and provides the information back to the user]

        :param country: [Country for which the Covid-19 statistics are required]
        :type country: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        """
        covid19 = Covid(source="worldometers")
        covidResults = covid19.get_status_by_country_name(country)
        country = covidResults["country"]
        activeCases = covidResults["active"]
        totalDeaths = covidResults["deaths"]
        newDeaths = covidResults["new_deaths"]
        newCases = covidResults["new_cases"]
        recovered = covidResults["recovered"]
        answer = {"answer" : "In {} there are {} new cases and {} new deaths".format(country, newCases, newDeaths)}
        answer["country"] = "Country: " + country + "\n"
        answer["newCases"] = "New Cases: " + str(newCases) + "\n"
        answer["newDeaths"] = "New deaths: " + str(newDeaths) + "\n"
        answer["activeCases"] = "Total active cases: " + str(activeCases) + "\n"
        answer["recovered"] = "Total recovered cases: " + str(recovered) + "\n"
        answer["totalDeaths"] = "Total deaths: " + str(totalDeaths) + "\n"
        response.value = answer["country"] + answer["newCases"] + answer["newDeaths"] + answer["activeCases"] + answer["re
covered"] + answer["totalDeaths"]
        print(answer["answer"])
        AIstatus.value = status["answer"]
        virtual_assistant.speak(answer["answer"])
```

run(*country: str, response: str, status: dict, AIstatus: str*)

[Method that retrieves the Covid-19 statistics, and provides the information back to the user]

Parameters:

- **country** (*str*) – [Country for which the Covid-19 statistics are required]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]

ERROR()

```
def error(self, user:str, country:str, response:str, AIstatus:str, status:dict):
        """[Method that runs if the run() method fails to retrieve information]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :param country: [Country for which Covid-19 statistics were requested]
        :type country: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        answer = {"answer" : "Hmmm {}..., I cannot find any Covid19 statistics for {}.".format(user.value, country)}
        answer["help"] = "\n\n{} you can always ask me for help if you need.\nTry 'Pixel I need help!'".format(user.value)
        response.value = answer["answer"] + answer["help"]
        print(response.value)
        AIstatus.value = status["answer"]
        virtual_assistant.speak(answer["answer"])
```

error(*user: str, country: str, response: str, AIstatus: str, status: dict*)

[Method that runs if the run() method fails to retrieve information]

Parameters:

- **user** (*str*) – [The name of the user that stands in front of the camera]
- **country** (*str*) – [Country for which Covid-19 statistics were requested]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

THANK_YOU

```
class Thank_you(Skill):
    """[This skill provides the user with a response for 'Thank you']

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Thank_you**

Bases: `skill.Skill`

[This skill provides the user with a response for 'Thank you']

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, user:str, AIstatus:str, status:dict):
        """[Method that chooses randomly a thank you message from a list]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :param AIstatus: [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :return: [Random thank you message]
        :rtype: [str]
        """
        AIstatus.value = status["process"]
        thankYouAnswers = []
        thankYouAnswers.append("{} is always a pleasure to help you!".format(user.value))
        thankYouAnswers.append("{}, no problem, any time!".format(user.value))
        thankYouAnswers.append("{}, is always my pleasure!".format(user.value))
        thankYouAnswers.append("No problem {}, it was the least I could do.".format(user.value))
        thankYouAnswers.append("Glad to help you {}!".format(user.value))
        thankYouAnswers.append("Thank YOU, {}!".format(user.value))
        return random.choice(thankYouAnswers)
```

prepare(*user: str, AIstatus: str, status: dict*)

[Method that chooses randomly a thank you message from a list]

Parameters:
- **user** (*str*) – [The name of the user that stands in front of the camera]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]

Returns: [Random thank you message]

Return type: [str]

RUN()

```
def run(self, response:str, AIstatus:str, status:list, answer):
        """[Method that answers back to a thank you request]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: list
        :param answer: [The thank you answer returned by the prepare() method]
        :type answer: [str]
        """
        response.value = answer
        AIstatus.value = status["answer"]
        virtual_assistant.speak(response.value)
```

run(*response: str, AIstatus: str, status: list, answer*)

[Method that answers back to a thank you request]

Parameters:

- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*list*) – [Virtual assistant’s statuses available]
- **answer** (*[str]*) – [The thank you answer returned by the prepare() method]

DEFINITION

```
class Definition(Skill):
    """[This skill provides the user a definition for a certain word from wikipedia]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Definition**

Bases: `skill.Skill`

[This skill provides the user a definition for a certain word from wikipedia]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, AIstatus:str, request:str, status:dict):
        """[Method that prepares the environment to retrieve a definition]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param request: [User's request in raw format]
        :type request: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :return: [The word for which the definition is request for]
        :rtype: [str]
        """
        AIstatus.value = status["process"]
        operator = helper.Context(helper.remove_polite_words())
        request = operator.run(request)
        operator = helper.Context(helper.word_after())
        if "define" in request:
            word = operator.run(request, "define")
        if "definition" in request:
            word = operator.run(request, "definition")
        if "tell me about" in request:
            word = operator.run(request, "about")
        print("[INFO] Definition: ", word)
        return word
```

prepare(*AIstatus: str, request: str, status: dict*)

[Method that prepares the environment to retrieve a definition]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **request** (*str*) – [User’s request in raw format]
- **status** (*dict*) – [Virtual assistant’s statuses available]

Returns: [The word for which the definition is request for]

Return type: [str]

RUN()

```
 def run(self, response:str, AIstatus:str, word:str, status:dict):
        """ [Method that retrieves the definition for the word requested, and provides the information back to the user]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param word: [The word for which the definition was requested for]
        :type word: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        operator = helper.Context(helper.nice_formatted_long_text())
        answer = []
        answer.append(wikipedia.summary(word, sentences=1, auto_suggest=False))
        answer.append(operator.run(answer[0]))
        print(answer[0])
        response.value = answer[1]
        AIstatus.value = status["answer"]
        virtual_assistant.speak(answer[0])
```

run(*response: str, AIstatus: str, word: str, status: dict*)

[Method that retrieves the definition for the word requested, and provides the information back to the user]

Parameters:

- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **word** (*str*) – [The word for which the definition was requested for]
- **status** (*dict*) – [Virtual assistant's statuses available]

ERROR()

```
def error(self, user:str, word:str, response:str, AIstatus:str, status:dict):
        """[Method that runs if the run() method fails to retrieve information]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :param word: [The word for which the definition is requested for]
        :type word: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        answer = {"answer" : "Hmmm {}..., I am not sure I can provide a definition for {}.".format(user.value, word)}
        answer["help"] = "\n\n{} you can always ask me for help if you need.\nTry 'Pixel I need help!'".format(user.value)
        response.value = answer["answer"] + answer["help"]
        print(response.value)
        AIstatus.value = status["answer"]
        virtual_assistant.speak(answer["answer"])
```

error(*user: str, word: str, response: str, AIstatus: str, status: dict*)

[Method that runs if the run() method fails to retrieve information]

Parameters:

- **user** (*str*) – [The name of the user that stands in front of the camera]
- **word** (*str*) – [The word for which the definition is requested for]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

HELP

```
class Help(Skill):
    """[This skill provides the user some guidance on how to use the device]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Help**

Bases: `skill.skill`

[This skill provides the user some guidance on how to use the device]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, AIstatus:str, status:dict, request:str):
        """[Method that prepares the environment to retrieve the help requested]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param request: [User's request in raw format]
        :type request: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :return: [The subject on which the user needs help with]
        :rtype: str
        """
        AIstatus.value = status["process"]
        operator = helper.Context(helper.remove_polite_words())
        request = operator.run(request)
        operator = helper.Context(helper.word_after())
        if "with" in request:
            help_for = operator.run(request, "with")
            print("[info help_for] ", help_for)
        else:
            help_for = ""
        return help_for
```

prepare(*AIstatus: str*, *status: dict*, *request: str*)

[Method that prepares the environment to retrieve the help requested]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **request** (*str*) – [User’s request in raw format]
- **status** (*dict*) – [Virtual assistant’s statuses available]

Returns: [The subject on which the user needs help with]

Return type: str

RUN()

```
def run(self, subject:str, AIstatus:str, status:dict, response:str, user:str):
        """[Method that retrieves the requested guidance about the device, and provides the information back to the user]

        :param subject: [The subject on which the user needs help with]
        :type subject: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        """
        help = ""
        if subject == "":
            help = self.general_help(user)
        elif "weather" in subject:
            help = self.weather_help(user)
        elif "covid" in subject:
            help = self.covid_help(user)
        elif "register" in subject or "sign up" in subject:
            help = self.register_help(user)
        elif "location" in subject or "address" in subject:
            help = self.location_help(user)
        elif "sos" in subject or "emergency" in subject:
            help = self.SOS_help(user)
        elif "list" in subject:
            help = self.list_help(user)

        response.value = help
        AIstatus.value = status["answer"]
        virtual_assistant.speak(response.value)
```

run(*subject: str*, *AIstatus: str*, *status: dict*, *response: str*, *user: str*)

[Method that retrieves the requested guidance about the device, and provides the information back to the user]

Parameters:

- **subject** (*str*) – [The subject on which the user needs help with]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **user** (*str*) – [The name of the user that stands in front of the camera]

GENERAL_HELP()

```
def general_help(self, user:str):
        """[General help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [General guidance on how to use the device]
        :rtype: [str]
        """
        help = "{}, I see you need some help! \nHere is how to interract with me:".format(user.value)
        help = help + "\n\n   1. Pixel what is the time/date"
        help = help + "\n   2. Pixel covid-19 stats in Ireland"
        help = help + "\n   3. To define a word: Pixel define dinosaur"
        help = help + "\n   4. Pixel tell me the weather in Dublin"
        help = help + "\n   5. Pixel where is the cinema in Dublin"
        help = help + "\n   6. Pixel call the cinema in Dublin"
        help = help + "\n   7. Pixel create an emergency contact"
        help = help + "\n   8. Pixel create a grocery list"
        help = help + "\n   9. Pixel add butter to the grocery list"
        help = help + "\n  10. Pixel show me the grocery list"
        help = help + "\n  11. To stop the conversation, step away from the mirror"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

general_help(*user: str*)

[General help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [General guidance on how to use the device]

Return type: [str]

REGISTER_HELP()

```
def register_help(self, user:str):
        """[Register help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [Register guidance]
        :rtype: [str]
        """
        help = "{}, I see you need some help! \nHere is how to ask to register:".format(user.value)
        help = help + "\n\n   1. Pixel I want to register"
        help = help + "\n   2. Pixel I want to sign up"
        help = help + "\n   3. I will provide further guidance when we begin the registration process"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

register_help(*user: str*)

[Register help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [Register guidance]

Return type: [str]

LOCATION_HELP()

```
def location_help(self, user:str):
        """[Location help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [Location guidance]
        :rtype: [str]
        """
        help = "{}, I see you need some help! \nHere is how to ask for location details:".format(user.value)
        help = help + "\n\n   1. Pixel where is IT Carlow"
        help = help + "\n   2. Pixel where is the cinema in Kilkenny?"
        help = help + "\n   3. Pixel si Supervalue open?"
        help = help + "\n   4. Pixel give me the address for Lidl Kilkenny"
        help = help + "\n   5. Pixel call Lidl Kilkenny"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

location_help(*user: str*)

[Location help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [Location guidance]

Return type: [str]

WEATHER_HELP()

```
def weather_help(self, user:str):
        """[Weather help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [Weather guidance]
        :rtype: [str]
        """
        help = "{}, I see you need some help! \nHere is how to ask for weather forecast:".format(user.value)
        help = help + "\n\n    1. Pixel tell me the weather in Dublin"
        help = help + "\n    2. Pixel how is the weather in Dublin?"
        help = help + "\n    3. Pixel what is the temperature outside?"
        help = help + "\n    3. Pixel what is the temperature in Dublin?"
        help = help + "\n    4. Keep in mind you can use any city"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

weather_help(*user: str*)

[Weather help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [Weather guidance]

Return type: [str]

COVID_HELP()

```
def covid_help(self, user:str):
        """[Covid-19 help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [Covid-19 guidance]
        :rtype: [str]
        """
        help = "{}, I see you need some help! \nHere is how to ask for Covid-19 statistics:".format(user.value)
        help = help + "\n\n    1. Pixel tell me covid stats in Ireland"
        help = help + "\n    2. Pixel how is covid situation in Ireland?"
        help = help + "\n    3. Keep in mind you can use any country"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

covid_help(*user: str*)

[Covid-19 help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [Covid-19 guidance]

Return type: [str]

SOS_HELP()

```
def SOS_help(self, user:str):
        """[SOS help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [SOS guidance]
        :rtype: [str]
        """
        help = "{}, you can send an SOS message to a contact if you ever need immediate help!\nHere is how to create an em
ergency contact".format(user.value)
        help = help + "\n\n    1. Pixel I want to create an SOS contact"
        help = help + "\n    2. Pixel I want to create an emergency contact"
        help = help + "\n    3. Keep in mind you need a the phone number of your next of kin!"
        help = help + "\nIf you already have an SOS contact, you can send an SOS when needed like this:!"
        help = help + "\n\n  1. Pixel send an SOS!"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

SOS_help(*user: str*)

[SOS help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [SOS guidance]

Return type: [str]

LIST_HELP()

```
def list_help(self, user:str):
        """[Lists help]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [Lists guidance]
        :rtype: [str]
        """
        help = "{}, you can create your own personal lists: ".format(user.value)
        help = help + "\n\n    1. Pixel I want to create a list"
        help = help + "\n    2. Pixel I want to see all my lists"
        help = help + "\n    3. Pixel add butter to the grocery list"
        help = help + "\n    4. Pixel delete butter from the grocery list"
        help = help + "\n    5. Pixel delete the grocery list"
        help = help + "\n\n! Remember that every command needs to start with PIXEL !"
        return help
```

list_help(*user: str*)

[Lists help]

Parameters: **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [Lists guidance]

Return type: [str]

ERROR()

```
        def error(self, AIstatus:str, status:dict, response:str, user:str):
        """[Method that runs if the run() method fails to retrieve information]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        """
        subject = ""
        self.run(subject, AIstatus, status, response, user)
```

error(*AIstatus: str, status: dict, response: str, user: str*)

[Method that runs if the run() method fails to retrieve information]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **user** (*str*) – [The name of the user that stands in front of the camera]

LOCATION

```
class Location(Skill):
    """[This skill provides the user with details about a specific location, using Google Location API]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Location**

Bases: `skill.skill`

[This skill provides the user with details about a specific location, using Google Location API]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, request:str):
        """[Method that prepares the environment to retrieve location details]

        :param request: [User's request in raw format]
        :type request: str
        :return: [User's request after being cleaned from unnecessary words]
        :rtype: str
        """
        operator = helper.Context(helper.remove_polite_words())
        request = operator.run(request)
        request_words = ["open", "opening hour", "address", "where"]
        operator = helper.Context(helper.remove_words())
        request = operator.run(request, request_words)
        return request
```

prepare(*request: str*)

[Method that prepares the environment to retrieve location details]

Parameters: **request** (*str*) – [User's request in raw format]

Returns: [User's request after being cleaned from unnecessary words]

Return type: str

RUN()

```
def run(self, AIstatus:str, location:str, response:str, status:dict, conn:object, call:bool):
        """[Method that retrieves location details and provides the information back to the user]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param location: [The location the user requested details about]
        :type location: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        with open('api_keys.json') as API:
            API = json.load(API)
            google_API = API['google']
        operator = helper.Context(helper.get_request())

        base_url_get_place_id = "https://maps.googleapis.com/maps/api/place/findplacefromtext/json?input={}&inputtype=text
query&fields=place_id&key={}".format(location, google_API)
        response_ID = operator.run(base_url_get_place_id)
        try:
            place_ID = str(response_ID['candidates'][0]['place_id'])
            base_url_get_location_details = "https://maps.googleapis.com/maps/api/place/details/json?place_id={}&fields=op
ening_hours,name,rating,formatted_address,formatted_phone_number&key={}".format(place_ID, google_API)
            print(base_url_get_location_details)
        except:
            return

        response_details = operator.run(base_url_get_location_details)
        isOpen = ""
        try:
            if response_details['result']['opening_hours']['open_now'] == True:
                isOpen = "Open at the moment. "
            else:
                isOpen = "Not open at the moment."
            location_details = {"answer" : "{} is {} {} phone number is {}".format(response_details['result']['name'], isO
pen, response_details['result']['name'], response_details['result']['formatted_phone_number'])}
        except:
            location_details = {"answer" : "{} phone number is {}".format(response_details['result']['name'], response_det
ails['result']['formatted_phone_number'])}
        location_details['name'] = response_details['result']['name'] + " is "
        try:
            location_details['open_now'] = isOpen.lower() + "\n\n"
        except:
            location_details['open_now'] = ""
        operator = helper.Context(helper.nice_formatted_long_text())
        location_details['address'] = operator.run("Address: " + response_details['result']['formatted_address'] + "\n\n")
        location_details['phone'] = "Phone number: " + response_details['result']['formatted_phone_number'] + "\n\n"

        try:
            opening_hours = str(response_details['result']['opening_hours']['weekday_text']).replace('\u2013', '-')
            opening_hours = opening_hours.replace(',','\n')
            opening_hours = opening_hours.replace("'", "\t")
            opening_hours = opening_hours.replace("[", "\n")
            opening_hours = opening_hours.replace("]", "")
            location_details['opening_hours'] = "Opening hours: " + opening_hours
        except:
            location_details['opening_hours'] = "Opening hours: " + "-"
        if call:
            phone_number = response_details['result']['formatted_phone_number']
            location_name = response_details['result']['name']
            confirmation_message = '\nAre you sure you want to call ' + response_details['result']['name'] +'?\nAnswer YES
 CALL LOCATION if you wish to proceed.'
            response.value = location_details["name"] + location_details["open_now"] + location_details["phone"] + locatio
n_details["address"] + confirmation_message
            print(response.value)
            AIstatus.value = status["answer"]
            virtual_assistant.speak(location_details["answer"] + confirmation_message)
            answer = virtual_assistant.listen_decision(AIstatus)
```

```
            if "call" in answer or "yes" in answer:
                self.call(phone_number, location_name, conn, response, AIstatus, status)
        else:
            response.value = location_details["name"] + location_details["open_now"] + location_details["phone"] + locatio
n_details["address"] + location_details["opening_hours"]
            print(response.value)
            AIstatus.value = status["answer"]
            virtual_assistant.speak(location_details["answer"])
```

run(*AIstatus: str, location: str, response: str, status: dict, conn: object, call: bool*)

[Method that retrieves location details and provides the information back to the user]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **location** (*str*) – [The location the user requested details about]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

CALL()

```
def call(self, phone_number:str, location:str,  conn:object, response:str, AIstatus:str, status:dict):
        """[Method that performs a call to a specific location]

        :param phone_number: [Location's phone number]
        :type phone_number: str
        :param location: [Location's details]
        :type location: str
        :param conn: [GPS object]
        :type conn: Communication
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        import re
        print("[INFO] Parameter: " , phone_number)
        phone = re.sub('\D', "", phone_number)
        print("[INFO] phone number to call: " + phone)
        print("[INFO] phone number to display: " + str(phone_number))
        response.value = "Calling now: " + str(location) + "\nPhone number: " + str(phone_number) + '\n\nCalling...'
        virtual_assistant.speak("Calling now: " + str(location))
        print(response.value)
        AIstatus.value = status['call']
        conn.call(phone)
        hang_up = False
        while hang_up == False:
            hang_up = conn.call_ongoing()
        response.value = "Your call ended. \n\nCan Pixel assist you with anything else?"
        virtual_assistant.speak(response.value)
        print(response.value)
```

call(*phone_number: str*, *location: str*, *conn: object*, *response: str*, *AIstatus: str*, *status: dict*)

[Method that performs a call to a specific location]

Parameters:

- **phone_number** (*str*) – [Location’s phone number]
- **location** (*str*) – [Location’s details]
- **conn** (*Communication*) – [GPS object]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

WEATHER

```
class Weather(Skill):
    """[This skill provides weather details to particular locations]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Weather**

Bases: `skill.skill`

[This skill provides weather details to particular locations]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, AIstatus:str, status:list, request:str, conn:object):
        """[Method that prepares the environment to retrieve weather details]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: list
        :param request: [User's request in raw format]
        :type request: str
        :return: [The location requested for weather forecast]
        :rtype: str
        """
        AIstatus.value = status["process"]
        operator = helper.Context(helper.remove_polite_words())
        city = operator.run(request)
        operator = helper.Context(helper.word_after())
        city = operator.run(request, "in")
        if city !=  "":
            #city = helper.substring_after(request, "for")
            city = city.strip()
            return city
        else:
            current_location = conn.get_location()
            county = current_location['address']['county'].split(" ")
            city = county[-1]
            return city
```

prepare(*AIstatus: str, status: list, request: str, conn: object*)

[Method that prepares the environment to retrieve weather details]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*list*) – [Virtual assistant’s statuses available]
- **request** (*str*) – [User’s request in raw format]

Returns: [The location requested for weather forecast]

Return type: str

RUN()

```
def run(self, city:str, AIstatus:str, response:str, status:dict):
        """[Method that retrieves weather details and provides the information back to the user]

        :param city: [The city for which weather forecast is provided]
        :type city: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        with open('api_keys.json') as API:
            API = json.load(API)
            key = API['weather']
        base_url = "http://api.openweathermap.org/data/2.5/weather?q="
        complete_url = base_url + city + "&appid=" + key + "&units=metric"
        operator = helper.Context(helper.get_request())
        details = operator.run(complete_url)
        desc = str(details['weather'][0]['description'])
        temp = str(details['main']['temp'])
        #print(response)
        weather_details = {"answer" : "The weather in {} is\n {} and with the temperature of {} Celsius".format(city, desc
, temp)}
        city = city.strip()
        city = city.capitalize()
        weather_details["location"] = "Location:\t" + city + "\n"
        weather_details["descrition"] = "Weather:\t" + desc + "\n"
        weather_details["temperature"] =  "Temperature: " + temp + "C \n"
        response.value = weather_details["location"]+ weather_details["descrition"]+ weather_details["temperature"]
        print(weather_details["answer"])
        AIstatus.value = status["answer"]
        virtual_assistant.speak(weather_details["answer"])
```

run(*city: str, AIstatus: str, response: str, status: dict*)

[Method that retrieves weather details and provides the information back to the user]

Parameters:

- **city** (*str*) – [The city for which weather forecast is provided]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

ERROR()

```
def error(self, user:str, response:str, AIstatus:str, status:list, city:str):
        """[Method that runs if the run() method fails to retrieve information]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: list
        :param city: [The city for which the weather forecast is requested]
        :type city: str
        """
        answer = {"answer" : "Hmmm {}..., I am not sure I can provide the forecast for {}.".format(user.value, city)}
        answer["help"] = "\n\n{} you can always ask me for help if you need.\nTry 'Pixel I need help!'".format(user.value)
        response.value = answer["answer"] + answer["help"]
        print(response.value)
        AIstatus.value = status["answer"]
        virtual_assistant.speak(answer["answer"])
```

error(*user: str, response: str, AIstatus: str, status: list, city: str*)

[Method that runs if the run() method fails to retrieve information]

Parameters:

- **user** (*str*) – [The name of the user that stands in front of the camera]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*list*) – [Virtual assistant’s statuses available]
- **city** (*str*) – [The city for which the weather forecast is requested]

GOOD_BYE

```
class Good_bye(Skill):
    """[This skill provides a good-bye message for the user]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Good_bye**

Bases: `skill.skill`

[This skill provides a good-bye message for the user]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self):
        """[Method that chooses a random good-bye message from a list]

        :return: [Good-bye message]
        :rtype: [str]
        """
        byeAnswers = []
        byeAnswers.append("Good bye!")
        byeAnswers.append("See you soon")
        byeAnswers.append("Was a pleasure, see you soon")
        byeAnswers.append("See you later, aligator")
        byeAnswers.append("Bye bye")
        byeAnswers.append("Can't wait to meet again")
        return random.choice(byeAnswers)
```

prepare()

[Method that chooses a random good-bye message from a list]

Returns: [Good-bye message]

Return type: [str]

RUN()

```
def run(self, AIstatus:str, response:str, message:str, status:dict):
        """[Method that retrieves a good-bye message, and provides the information back to the user]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param message: [Good-bye message]
        :type message: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        AIstatus.value = status["process"]
        response.value = message
        AIstatus.value = status["answer"]
        virtual_assistant.speak(response.value)
```

run(*AIstatus: str, response: str, message: str, status: dict*)

[Method that retrieves a good-bye message, and provides the information back to the user]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **message** (*str*) – [Good-bye message]
- **status** (*dict*) – [Virtual assistant’s statuses available]

TIME

```
class Time(Skill):
    """[This skill provides current time]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Time**

Bases: `skill.skill`

[This skill provides current time]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, AIstatus:str, status:dict):
        """[Method that retrieves current time]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :return: [Current time]
        :rtype: [str]
        """
        AIstatus.value = status["process"]
        timeNow = "The time is "
        timeNow = timeNow +  datetime.datetime.now().strftime("%I:%M %p")
        return timeNow
```

prepare(*AIstatus: str*, *status: dict*)

[Method that retrieves current time]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

Returns: [Current time]

Return type: [str]

RUN()

```
def run(self, now:str, AIstatus:str, response:str, status:dict):
        """[Method that provides the current time back to the user]

        :param now: [Current time]
        :type now: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        AIstatus.value = status["answer"]
        response.value = now
        print(response.value)
        virtual_assistant.speak(response.value)
```

run(*now: str, AIstatus: str, response: str, status: dict*)

[Method that provides the current time back to the user]

Parameters:

- **now** (*str*) – [Current time]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

DATE

```
class Date(Skill):
    """[This skill provides the current date]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Date**

Bases: `skill.skill`

[This skill provides the current date]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, AIstatus:str, status:list):
        """[Method that retrieves the current date]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: list
        :return: [Current date]
        :rtype: [str]
        """
        AIstatus.value = status["process"]
        dateNow = "Today's date is "
        dateNow = dateNow + datetime.datetime.now().strftime("%A %d %B %Y")
        return dateNow
```

prepare(*AIstatus: str*, *status: list*)

[Method that retrieves the current date]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*list*) – [Virtual assistant’s statuses available]

Returns: [Current date]

Return type: [str]

RUN()

```
def run(self, date_today:str, AIstatus:str, response:str, status:dict):
        """[Method that returns the current date, and provides the information back to the user]

        :param date_today: [Current date]
        :type date_today: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        response.value = date_today
        AIstatus.value = status["answer"]
        print(response.value)
        virtual_assistant.speak(response.value)
```

run(*date_today: str, AIstatus: str, response: str, status: dict*)

[Method that returns the current date, and provides the information back to the user]

Parameters:

- **date_today** (*str*) – [Current date]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

REGISTER

```
class Register(Skill):
    """[This skill registers a new user on the device]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Register**

Bases: `skill.skill`

[This skill registers a new user on the device]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

TAKE_PICTURE()

```
def take_pictures(self, name:str, response:str):
        """[Method takes approximately 80 pictures with the user that is in front of the camera]

        :param name: [Name of the user that registers]
        :type name: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        response.value = "For the next 5 seconds please look into the mirror " + name + ".\nI will start taking some pictu
res with you.\nDo not worry I will delete them afterwards."
        virtual_assistant.speak("For the next 5 seconds please look into the mirror " + name + ".\nI will start taking som
e pictures with you.\nDo not worry I will delete them afterwards.")
        path = "User/"+name
        os.mkdir(path)
        print("[INFO] Folder created")
        capture = cv2.VideoCapture(0);
        print("[INFO] Camera started to take pictures")

        print("[INFO] Taking pictures")
        now = t.time() + 5
        i = 0
        while(t.time() < now):
            ret, image = capture.read()
            i+=1
            cv2.imwrite('User/'+name+'/'+str(i)+'.png', image)
        del(capture)
```

take_pictures(*name: str, response: str*)

[Method takes approximately 80 pictures with the user that is in front of the camera]

Parameters:

- **name** (*str*) – [Name of the user that registers]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

TRAIN()

```
def train(self, name:str, response:str):
        """[Method that trains the device to recognize a certain user. This method converts images to embedded 128-
d vectors and saves the vector to a pickle file]

        :param name: [Name of the user that requested to register]
        :type name: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        print("[INFO] Starting training...")
        response.value = "Now I will learn your face features " + name +", this will take a few minutes..."
        virtual_assistant.speak("Now I will learn your face features " + name +", this will take a few minutes...")
        path = "User/"+name
        encodings = pickle.loads(open("Cascades/encodings.pickle", "rb").read())

        imagePaths = list(paths.list_images(path))
        knownFaces = []
        knownNames = []

        for (i, imagePath) in enumerate(imagePaths):
            print("[INFO] Processing images: " + imagePath)
            self.registration_diaglog(i, name, len(imagePaths), response)
            image = cv2.imread(imagePath)
            rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            boxes = face_recognition.face_locations(rgb, model="hog")
            encodings = face_recognition.face_encodings(rgb, boxes)

            for encoding in encodings:
                knownFaces.append(encoding)
                knownNames.append(name)
        print("[INFO] Save encodings...")
        data = {"encodings": knownFaces, "names": knownNames}
        file = open("Cascades/encodings.pickle", "wb")
        file.write(pickle.dumps(data))
        file.close()
        response.value = "Done! Now we are friends " + name + "!\n\nNow you have access to all my skills!\nTry to ask me f
or weather details, Covid19 stats and others!\nI can always guide you how to communicate with me, by asking:\n'PIXEL I NEE
D HELP'"
        virtual_assistant.speak("Done! Now we are friends " + name + "!\n\nNow you have access to all my skills!\nTry to a
sk me for weather details, Covid19 stats and others!\nI can always guide you how to communicate with me, by asking:\n'PIXE
L I NEED HELP'")
```

train(*name: str, response: str*)

[Method that trains the device to recognize a certain user. This method converts images to embedded 128-d vectors and saves the vector to a pickle file]

Parameters:
- **name** (*str*) – [Name of the user that requested to register]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

REGISTRATION_DIALOG()

```
def registration_diaglog(self, picture_number:int, name:str, len_images:int, response):
        """[Method that makes the registration interactive, by communicating with the user during the process]

        :param picture_number: [The picture number that the process of training is at]
        :type picture_number: int
        :param name: [The name of the user and the name of the folder where the pictures are]
        :type name: str
        :param len_images: [The total number of pictures used for the registration process]
        :type len_images: int
        :param response:  [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        first_quarter = len_images//4
        second_quarter = first_quarter + first_quarter
        third_quarter = second_quarter + first_quarter
        if picture_number == first_quarter:
            response.value = "Looking good "  + name + "!\nI am starting to know you!"
            virtual_assistant.speak("Looking good " + name + " ! I am starting to know you!")
        elif picture_number == second_quarter:
            response.value = "You have very interesting features "  + name + "!\nWe are half way done."
            virtual_assistant.speak("You have very interesting features"  + name + "! We are half way done.")
        elif picture_number == third_quarter:
            response.value = name + " we are almost done! \nDo not go anywhere!"
            virtual_assistant.speak(name + " ! we are almost done! \nDo not go anywhere!")
```

registration_diaglog(*picture_number: int, name: str, len_images: int, response*)

[Method that makes the registration interactive, by communicating with the user during the process]

Parameters:

- **picture_number** (*int*) – [The picture number that the process of training is at]
- **name** (*str*) – [The name of the user and the name of the folder where the pictures are]
- **len_images** (*int*) – [The total number of pictures used for the registration process]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

USER_EXISTS()

```
def user_exists(self, user:str, response:str):
        """[Method that checks if the user requesting to register is already registered or not]

        :param user: Name of the user requesting to register:

            - stranger -> unregistered user

            - other name -> if the user is already registered
        :type user: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :return:
            - True - > the user is already registered
            - False -> the user is not registered
        :rtype: [bool]
        """
        if user.value == "stranger" or user.value == "":
            return False
        else:
            response.value = "Ummm... " + user.value + " you are already registered.\nAre you trying to confuse me?"
            virtual_assistant.speak("Ummm... " + user.value + " you are already registered.\nAre you trying to confuse me?
" )
            return True
```

user_exists(*user: str, response: str*)

[Method that checks if the user requesting to register is already registered or not]

Parameters:

- **user** (*str*) –
  Name of the user requesting to register:
  - stranger -> unregistered user
  - other name -> if the user is already registered
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

Returns:

- True - > the user is already registered
- False -> the user is not registered

Return type: [bool]

GET_NAME()

```
def get_name(self, AIstatus:str, response:str, understanding:str, user:str):
        """[Method that retrieves the name the user wants to use to register on the device]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        :return: [The name the user wants to be registered with]
        :rtype: [str]
        """
        print("[INFO] user asking to register " + user.value)
        username_exists = True
        name = "stranger"
        confirm = "no"
        while username_exists or "yes" not in confirm or "yea" not in confirm or "right" not in confirm and "cancel" not i
n confirm:
            response.value = "To register I need your name. Please tell me your name."
            virtual_assistant.speak("To register I need your name. Please tell me your name")
            name = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + name
            operator = helper.Context(helper.get_last_word())
            name = operator.run(name)
            operator = helper.Context(helper.folder_exists())
            username_exists = operator.run(name)
            if(username_exists):
                response.value = "The name " + name + " is already used. Please try again"
                virtual_assistant.speak("The name " + name + " is already used. Please try again")
                name = "stranger"
            else:
                response.value = "You said " + name + ", right?\nPlease answer YES THIS IS RIGHT or NO THIS IS NOT CORRECT
. \nIf you do not want to register anymore say CANCEL REGISTRATION."
                virtual_assistant.speak("You said " + name + " right? Please answer YES THIS IS RIGHT or NO THIS IS NOT CO
RRECT. If you do not want to register anymore say CANCEL REGISTRATION.")
                confirm = virtual_assistant.listen_decision(AIstatus)
                understanding.value = "Responding to: " + confirm
                if "yes" in confirm or "yea" in confirm or "right" in confirm and "not" not in confirm:
                    return name
                if "cancel" in confirm:
                    response.value = "You cancelled the registration.\nWell, maybe next time!\n\nI am ready to answer othe
r questions now."
                    virtual_assistant.speak("You cancelled the registration.\nWell, maybe next time!\n\nI am ready to answ
er other questions now.")
                    return "stranger"
        return name
```

get_name(*AIstatus: str, response: str, understanding: str, user: str*)

[Method that retrieves the name the user wants to use to register on the device]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]
- **user** (*str*) – [The name of the user that stands in front of the camera]

Returns: [The name the user wants to be registered with]

Return type: [str]

PREPARE()

```
def prepare(self, name:str):
        """[Method that prepares the environment to run the registration]

        :param name: [Name of the user standing in front of the mirror ("stranger" if the user is not registered)]
        :type name: str
        :return:

            - True -> The user is not registered
            - False -> The user is already registered
        :rtype: [bool]
        """
        if name != "stranger" and name != "":
            return True
        else:
            return False
```

prepare(*name: str*)

[Method that prepares the environment to run the registration]

Parameters: **name** (*str*) – [Name of the user standing in front of the mirror ("stranger" if the user is not registered)]

Returns:
- True -> The user is not registered
- False -> The user is already registered

Return type: [bool]

RUN()

```
def run(self, AIstatus:str, cameraRunning:bool, name:str, response:str, status:dict):
        """ Method that controls the registration process.

            - This method takes control of the camera module from the main Camera process until pictures with the user are
 taken
            - When the process of taking pictures with the user is finished, the control of the camera module is given bac
k to the main Camera process


        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param cameraRunning: [Variable that controls the camera module for the main Camera process]
        :type cameraRunning: bool
        :param name: [Name of the user that registers]
        :type name: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        AIstatus.value = status["process"]
        cameraRunning.value = False
        self.take_pictures(name, response)
        self.train(name, response)
        cameraRunning.value = True
        operator = helper.Context(helper.delete_pictures())
        operator.run(name)
        operator = helper.Context(helper.create_pickle_files())
        operator.run(name)
```

run(*AIstatus: str, cameraRunning: bool, name: str, response: str, status: dict*)

Method that controls the registration process.

- This method takes control of the camera module from the main Camera process until pictures with the user are taken
- When the process of taking pictures with the user is finished, the control of the camera module is given back to the main Camera process

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **cameraRunning** (*bool*) – [Variable that controls the camera module for the main Camera process]
- **name** (*str*) – [Name of the user that registers]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

GREETING

```
class Greeting(Skill):
    """[This skill greets the user when the virtual assistant starts]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.Greeting**

Bases: `skill.skill`

[This skill greets the user when the virtual assistant starts]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

PREPARE()

```
def prepare(self, user:str):
        """[Method that prepares the greeting according to the current time of the day]

        :param user: [The user standing in front of the camera]
        :type user: str
        :return: [Greeting]
        :rtype: [str]
        """
        hour = int(datetime.datetime.now().strftime("%H"))
        timeOfDay = ""
        if hour < 2:
            timeOfDay = timeOfDay + "Good night"
        elif hour < 12:
            timeOfDay = timeOfDay + "Good morning"
        elif hour < 17:
            timeOfDay = timeOfDay + "Good afternoon"
        elif hour < 22:
            timeOfDay = timeOfDay + "Good evening"
        else:
            timeOfDay = timeOfDay + "Good night"
        timeOfDay = timeOfDay + " " + user.value +  "! How Pixel can assist you?"
        return timeOfDay
```

prepare(*user: str*)

[Method that prepares the greeting according to the current time of the day]

Parameters: **user** (*str*) – [The user standing in front of the camera]

Returns: [Greeting]

Return type: [str]

RUN()

```
def run(self, greeting:str, response:str):
        """[Method that greets the user that stands in front of the camera]

        :param greeting: [Greeting]
        :type greeting: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        response.value = greeting
        virtual_assistant.speak(greeting)
```

run(*greeting: str*, *response: str*)

[Method that greets the user that stands in front of the camera]

Parameters:

- **greeting** (*str*) – [Greeting]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

NOT_UNDERSTANDING

```
class not_understanding(Skill):
    """[This skill informs the user that the request does not match any skill]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
```

class **skill.not_understanding**

Bases: `skill.skill`

[This skill informs the user that the request does not match any skill]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

RUN()

```
def run(self, response:str):
        """[Method that informs the user that no skill matched its request]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        response.value = "Hmmm, I do not know this one..."
        virtual_assistant.speak(response.value)
```

run(*response: str*)

[Method that informs the user that no skill matched its request]

Parameters: **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

SOS

```
class SOS(Skill):
    """[This skill manages the SOS contact]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
    def __init__(self, user:str):
        """[Constructor of the SOS skill]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        """
        self.name = user
```

class **skill.SOS**(*user: str*)

Bases: `skill.Skill`

[This skill manages the SOS contact]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

RUN_ADD_CONTACT()

```
def run_add_contact(self, response:str, AIstatus:str, conn:object, understanding:str, status:dict, user_is_registered:bool
):
        """[Mehtod that controls the creation of the SOS contact]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param conn: [GPS object]
        :type conn: Communication
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param user_is_registered:
            - True -> the user is registered on the device
            - False -> the user is not registered on the device
        :type user_is_registered: bool
        """
        if user_is_registered:
            user_name = str(self.name)
            name = ""
            number = ""
            name = self.get_name(response, AIstatus, understanding)
            if name != "":
                number = self.get_phone_number(response, AIstatus, name, understanding)
            if number != "":
                AIstatus.value = status['process']
                self.send_sms_to_new_contact(response, AIstatus, conn, name, number)
                self.save_contact(name, number, user_name)
        else:
            AIstatus.value = status['answer']
            response.value = "I am really sorry, you need to be registered to add an emergency contact."
            virtual_assistant.speak(response.value)
```

run_add_contact(*response: str, AIstatus: str, conn: object, understanding: str, status: dict, user_is_registered: bool*)

[Mehtod that controls the creation of the SOS contact]

Parameters:

- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **conn** (*Communication*) – [GPS object]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **user_is_registered** (*bool*) –
  - True -> the user is registered on the device
  - False -> the user is not registered on the device

GET_NAME()

```
def get_name(self, response:str, AIstatus:str, understanding:str):
        """[Method that retrieves the name of the emergency contact from the user]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :return: [Name of the emergency contact]
        :rtype: [str]
        """
        answer = ""
        while "yes" not in answer or "right" not in answer or "cancel" not in answer:
            # get name for contact
            response.value = "I need a name for the emergency contact.\n\nPlease tell me the name of the person who will \
nreceive an SOS in case of emergency."
            virtual_assistant.speak(response.value)
            name = virtual_assistant.listen_decision(AIstatus)
            operator = helper.Context(helper.get_last_word())
            name = operator.run(name)
            understanding.value = "Responding to: " + name
            #confirm name
            response.value = "You said " + name + ", right?\n\nPlease answer YES THIS IS RIGHT or NO THIS IS NOT CORRECT.
\n\nIf you do not want to create an SOS contact anymore \nsay I WANT TO CANCEL CONTACT."
            virtual_assistant.speak("You said " + name + ", right?\n\nPlease answer YES THIS IS RIGHT or NO THIS IS NOT CO
RRECT.")
            answer = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + answer
            if "cancel" in answer:
                return ""
            elif "yes" in answer or "right" in answer:
                return name
```

get_name(*response: str, AIstatus: str, understanding: str*)

[Method that retrieves the name of the emergency contact from the user]

Parameters:
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]

Returns: [Name of the emergency contact]

Return type: [str]

GET_PHONE_NUMBER()

```
def get_phone_number(self, response:str, AIstatus:str, name:str, understanding:str):
        """[Method that retrieves the phone number of the emergency contact from the user]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param name: [Name of the emergency contact]
        :type name: str
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :return: [Phone number of the emergency contact]
        :rtype: [str]
        """
        answer = ""
        while "yes" not in answer or "right" not in answer or "cancel" not in answer:
            # get number for contact
            response.value = "To create the new SOS contact, I need " + name + "'s phone number."
            virtual_assistant.speak(response.value)
            number = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + number
            #confirm number
            response.value = "You said " + number + ", right?\n\nPlease answer YES THIS IS RIGHT or NO THIS IS NOT CORRECT
. \n\nIf you do not want to create an SOS contact anymore \nsay I WANT TO CANCEL CONTACT."
            virtual_assistant.speak("You said " + number + ", right?\n\nPlease answer YES THIS IS RIGHT or NO THIS IS NOT
CORRECT.")
            answer = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + answer
            if "cancel" in answer:
                return ""
            elif "yes" in answer or "right" in answer:
                number = number.replace(" ", "")
                print("[INFO] " + number)
                return number
```

get_phone_number(*response: str, AIstatus: str, name: str, understanding: str*)

[Method that retrieves the phone number of the emergency contact from the user]

Parameters:
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **name** (*str*) – [Name of the emergency contact]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]

Returns: [Phone number of the emergency contact]

Return type: [str]

SEND_SMS_TO_NEW_CONTACT()

```
def send_sms_to_new_contact(self, response:str, AIstatus:str, conn:object, name:str, number:str):
        """[Method that sends a text message to the emergency contact, informing that particular person that it had been a
ssigned as an emergency contact by the user]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param conn: [GPS object]
        :type conn: Communication
        :param name: [Name of the emergency contact]
        :type name: str
        :param number: [Phone number of the emergency contact]
        :type number: str
        """
        name_user = str(self.name)
        conn.send_SMS(name_user, number, "You had been assigned as an emergency contact of " + name_user + ". In case of e
mergency, Pixel Virtual Assistant will message you with details.")
        response.value = "The account is now created. I will inform now " + name + "\nthat it is assigned as an emergency
contact."
        virtual_assistant.speak(response.value)
```

send_sms_to_new_contact(*response: str, AIstatus: str, conn: object, name: str, number: str*)

[Method that sends a text message to the emergency contact, informing that particular person that it had been assigned as an emergency contact by the user]

Parameters:

- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **conn** (*Communication*) – [GPS object]
- **name** (*str*) – [Name of the emergency contact]
- **number** (*str*) – [Phone number of the emergency contact]

SAVE_CONTACT()

```
def save_contact(self, name:str, number:str, name_user:str):
        """[Saving the new emergency contact in a pickle file]

        :param name: [Name of the emergency contact]
        :type name: str
        :param number: [Phone number of the emergency contact]
        :type number: str
        :param name_user: [Name of the user in front of the camera]
        :type name_user: str
        """
        import os
        name_user = str(self.name)
        path = "User/" + name_user + "/" + "SOS_contact"
        SOS = {}
        SOS['number'] = number
        SOS['name'] = name
        with open(path, 'wb') as f:
            pickle.dump(SOS, f)
```

save_contact(*name: str, number: str, name_user: str*)

[Saving the new emergency contact in a pickle file]

Parameters:

- **name** (*str*) – [Name of the emergency contact]
- **number** (*str*) – [Phone number of the emergency contact]
- **name_user** (*str*) – [Name of the user in front of the camera]

EMERGENCY

```
class Emergency(Skill):
    """[This skill sends an emergency message, and calls the emergencies]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
    def __init__(self, user_device:str):
        """[Constructor of the SOS skill]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        """
        self.name = user_device
```

class **skill.Emergency**(*user_device: str*)

Bases: `skill.Skill`

[This skill sends an emergency message, and calls the emergencies]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

RUN()

```
def run(self, response:str, AIstatus:str, conn:object, status:dict, user_is_registered:bool, user_has_SOS_contact:bool):
        """[Method that sends an emergency text message to the SOS contact and calls the emergency]
            :Note:
                - Performing the emergency call, I used the phone number of the SOS contact, but it can be replaced by 112
 or 911

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param conn: [GPS object]
        :type conn: Communication
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param user_is_registered:

            - True -> the user is registered on the deice

            - False -> the user is not registered on the device

        :type user_is_registered: bool
        :param user_has_SOS_contact:

            - True -> the user has an SOS contact designated

            - False -> the user has no SOS contact designated

        :type user_has_SOS_contact: bool
        """
        print("[INFO] send_SOS() method called")
        answer = self.confirm_send_SOS(response, AIstatus, user_is_registered, user_has_SOS_contact)
        AIstatus.value = status['process']
        name = str(self.name)
        if "cancel" not in answer:
            path = "User/" + name + "/" + "SOS_contact"
            with open(path, 'rb') as f:
                x = pickle.load(f)
```

```
                try:
                    full_address = conn.get_location()
                    address = full_address['display_name']
                    message = name + " sent you an SOS!\nPlease get in touch as soon as possible.\n\nHere is the full addr
ess: " + address
                    conn.send_SMS(x['name'], x['number'], message)
                    AIstatus.value = status['answer']
                    response.value = name + " I sent an SOS to " + x['name'] + "! \nI will now call the emergencies!\n\nCa
lling now..."
                    virtual_assistant.speak(response.value)
                    AIstatus.value = status['call']
                    conn.call(x['number'])
                    hang_up = False
                    while hang_up == False:
                        hang_up = conn.call_ongoing()
                    response.value = "Your call ended. \n\nCan Pixel assist you with anything else?"
                    virtual_assistant.speak(response.value)
                    print(response.value)
                except:
                    print("Error")
```

run(*response: str, AIstatus: str, conn: object, status: dict, user_is_registered: bool, user_has_SOS_contact: bool*)

[Method that sends an emergency text message to the SOS contact and calls the emergency]

Note:
- Performing the emergency call, I used the phone number of the SOS contact, but it can be replaced by 112 or 911

Parameters:
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **conn** (*Communication*) – [GPS object]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **user_is_registered** (*bool*) –
  - True -> the user is registered on the deice
  - False -> the user is not registered on the device
- **user_has_SOS_contact** (*bool*) –
  - True -> the user has an SOS contact designated
  - False -> the user has no SOS contact designated

CONFIRM_SEND_SOS

```
def confirm_send_SOS(self, response:str, AIstatus:str, user_is_registered:bool, user_has_SOS_contact:bool):
        """[Method that allows the user to cancel the action of sending an emergency message and calling the emergencies]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param user_is_registered:

            - True -> the user is registered on the device

            - False -> the user is not registered on the device

        :type user_is_registered: bool
        :param user_has_SOS_contact:

            - True -> the user has an SOS contact designated

            - False -> the user has no SOS contact designated

        :type user_has_SOS_contact: bool

        :return: ["cancel" if the user cancels the action]
        :rtype: [str]
        """
        name = str(self.name)
        print("[INFO] confirm_send_SOS() " + name)
        if user_is_registered and user_has_SOS_contact:
            response.value = name + " are you sure you want to send an SOS?\n\nAnswer YES SEND SOS or NO CANCEL"
            virtual_assistant.speak(response.value)
            confirmation = virtual_assistant.listen_decision(AIstatus)
            return confirmation
        elif not user_has_SOS_contact:
            response.value = name + " you do not have an SOS contact saved.\nI cannot send an SOS!\n\nIf you want to creat
e an SOS contact\nsay PIXEL, CREATE AN SOS CONTACT."
            virtual_assistant.speak(response.value)
            return "cancel"
        else:
            response.value = name + "You are not registered, I cannot send an SOS!\n\nIf you want to register,\nsay PIXEL,
 I WANT TO REGISTER."
            virtual_assistant.speak(response.value)
            return "cancel"
```

LIST

```
class List(Skill):
    """[This skill allows the user to manipulate personal lists]

    :param Skill: [Skill - is the parent abstract class]
    :type Skill: [abstract class]
    """
    def __init__(self, user_device:str):
        """[Constructor of the List skill]

        :param user: [The name of the user that stands in front of the camera]
        :type user: str
        """
        self.name = user_device
```

class **skill.List**(*user_device: str*)

Bases: `skill.Skill`

[This skill allows the user to manipulate personal lists]

Parameters: **Skill** (*[abstract class]*) – [Skill - is the parent abstract class]

RUN_CREATE_LIST()

```
def run_create_list(self, response:str, AIstatus:str, understanding:str, status:dict):
        """[Method that prepares the creation of a list]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        AIstatus.value = status['process']
        name = str(self.name)
        confirm = ""
        while "cancel" not in confirm and "list" not in confirm and "yes" not in confirm and "right" not in confirm:
            AIstatus.value = status['answer']
            response.value = "What is the name of the list you want to create?"
            virtual_assistant.speak(response.value)
            name_list = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + name_list
            words_to_remove = []
            words_to_remove.append("list")
            operator = helper.Context(helper.remove_words())
            name_list = operator.run(name_list, words_to_remove)
            name_list = name_list.strip()
            AIstatus.value = status['answer']
            response.value = "You said " + name_list + ", right?\n\nAnswer YES THIS IS RIGHT or NO THIS IS NOT CORRECT.\nO
r say CANCEL THIS LIST if you do not want to proceed anymore."
            virtual_assistant.speak(response.value)
            confirm = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + confirm
        if "yes" in confirm or "right" in confirm:
                self.create_list(name, AIstatus, status, response, name_list)
```

run_create_list(*response: str, AIstatus: str, understanding: str, status: dict*)

[Method that prepares the creation of a list]

Parameters:

- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]

CREATE_LIST()

```
def create_list(self, name:str, AIstatus:str, status:dict, response:str, name_list:str):
        """[Method that creates a list and adds it to the correct pickle file]

        :param name: [Name of the user standing in front of the camera]
        :type name: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param name_list: [Name of the list that needs to be created]
        :type name_list: str
        """
        AIstatus.value = status['process']
        path = "User/" + name + "/" + "user_lists"
        list_exists = False
        if os.path.getsize(path) == 0:
            list_exists = False
        else:
            with open(path, 'rb') as f:
                x = pickle.load(f)
                list_exists = self.list_exists(x, name_list)
        if list_exists:
            AIstatus.value = status['answer']
            response.value = "The " + name_list + " list already exists."
            virtual_assistant.speak(response.value)
        else:
            with open(path, 'rb') as f:
                if os.path.getsize(path) == 0:
                    lists = {}
                else:
                    old_lists = pickle.load(f)
                    lists = old_lists
            with open(path, 'wb') as f:
                lists[name_list] = []
                lists[name_list].append("nothing")
                pickle.dump(lists, f)
            AIstatus.value = status['answer']
            response.value = "The " + name_list + " list was created."
            virtual_assistant.speak(response.value)
```

create_list(*name: str, AIstatus: str, status: dict, response: str, name_list: str*)

[Method that creates a list and adds it to the correct pickle file]

Parameters:
- **name** (*str*) – [Name of the user standing in front of the camera]
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **name_list** (*str*) – [Name of the list that needs to be created]

LIST_EXIST()

```
def list_exists(self, lists:dict, name_of_list:str):
        """[Method that checks if a specific list already exists]

        :param lists: [All saved lists]
        :type lists: dict
        :param name_of_list: [List that needs to created]
        :type name_of_list: str
        :return:
            - True -> the list already exists

            - False -> the list does not exist

        :rtype: [type]
        """
        try:
            for key in lists:
                if key == name_of_list:
                    return True
            return False
        except:
            return False
```

list_exists(*lists: dict, name_of_list: str*)

[Method that checks if a specific list already exists]

Parameters:
- **lists** (*dict*) – [All saved lists]
- **name_of_list** (*str*) – [List that needs to created]

Returns:
- True -> the list already exists
- False -> the list does not exist

Return type: [type]

RUN_DELETE_LIST()

```
def run_delete_list(self, AIstatus:str, response:str, status:dict, understanding:str):
        """[Method that prepares the environment to delete of a certain list]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        """
        AIstatus.value = status['process']
        name = str(self.name)
        confirm = ""
        while "cancel" not in confirm and "list" not in confirm and "yes" not in confirm and "right" not in confirm:
            AIstatus.value = status['answer']
            response.value = "What is the name of the list you want to delete?"
            virtual_assistant.speak(response.value)
            name_list = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + name_list
            words_to_remove = []
            words_to_remove.append("list")
            operator = helper.Context(helper.remove_words())
            name_list = operator.run(name_list, words_to_remove)
            name_list = name_list.strip()
            AIstatus.value = status['answer']
            response.value = "You said " + name_list + ", right?\n\nAnswer YES THIS IS RIGHT or NO THIS IS NOT CORRECT.\nO
r say CANCEL THIS LIST if you do not want to proceed anymore."
            virtual_assistant.speak(response.value)
            confirm = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + confirm
        if "yes" in confirm or "right" in confirm:
                self.delete_list(AIstatus, response, name_list, status, name)
```

run_delete_list(*AIstatus: str, response: str, status: dict, understanding: str*)

[Method that prepares the environment to delete of a certain list]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **understanding** (*str*) – [What the virtual assistant understood from the user’s request (in text format). This text is displayed on the mirror surface]

DELETE_LIST()

```
def delete_list(self, AIstatus:str, response:str, name_list:str, status:dict, name:str):
        """[Method that deletes a list from the pickle file]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param name_list: [Name of the list to be deleted]
        :type name_list: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param name: [Name of the user in front of the camera]
        :type name: str
        """
        AIstatus.value = status['process']
        path = "User/" + name + "/" + "user_lists"
        list_exists = False
        if os.path.getsize(path) == 0:
            list_exists = False
        else:
            with open(path, 'rb') as f:
                x = pickle.load(f)
                list_exists = self.list_exists(x, name_list)
        if not list_exists:
            AIstatus.value = status['answer']
            response.value = "The " + name_list + " list, does not exist."
            virtual_assistant.speak(response.value)
        else:
            with open(path, 'rb') as f:
                if os.path.getsize(path) == 0:
                    lists = {}
                else:
                    old_lists = pickle.load(f)
                    lists = old_lists
            with open(path, 'wb') as f:
                lists.pop(name_list)
                pickle.dump(lists, f)
                AIstatus.value = status['answer']
                response.value = "The " + name_list + " list was deleted."
                virtual_assistant.speak(response.value)
```

delete_list(*AIstatus: str, response: str, name_list: str, status: dict, name: str*)

[Method that deletes a list from the pickle file]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **name_list** (*str*) – [Name of the list to be deleted]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **name** (*str*) – [Name of the user in front of the camera]

RUN_REMOVE_ITEM_FROM_LIST()

```
def run_remove_item_from_list(self, AIstatus:str, status:dict, understanding:str, response:str, request:str):
        """[Method that prepares the environment to remove an item from a list]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param request: [User's request in raw format]
        :type request: str
        """
        AIstatus.value = status['process']
        name = str(self.name)
        confirm = ""
        AIstatus.value = status['process']
        operator = helper.Context(helper.word_after())
        if "remove" in request:
            item_to_add = operator.run(request, "remove")
        else:
            item_to_add = operator.run(request, "delete")
        operator = helper.Context(helper.word_before())
        the_list = operator.run(request, "list")
        AIstatus.value = status["answer"]
        response.value = name + " you want to delete " + item_to_add.upper() + " from the " + the_list.upper() + " list, r
ight?\n\nAnswer YES THIS IS RIGHT or NO, THIS IS NOT CORRECT."
        virtual_assistant.speak(response.value)
        confirm = virtual_assistant.listen_decision(AIstatus)
        understanding.value = "Responding to: " + confirm
        if "right" in confirm or "yes" in confirm:
            removed = self.delete_item_from_list(AIstatus, status,  name, the_list, item_to_add, response)
            if removed:
                AIstatus.value = status['answer']
                response.value = "The " + item_to_add + " was deleted from the " + the_list + " list."
                virtual_assistant.speak(response.value)
                self.see_list(AIstatus, response, the_list, status, name)
            else:
                AIstatus.value = status["answer"]
                response.value = name + " I cannot find the " + the_list + " list."
                virtual_assistant.speak(response.value)
```

run_remove_item_from_list(*AIstatus: str, status: dict, understanding: str, response: str, request: str*)

[Method that prepares the environment to remove an item from a list]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **understanding** (*str*) – [What the virtual assistant understood from the user’s request (in text format). This text is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **request** (*str*) – [User’s request in raw format]

DELETE_ITEM_FROM_LIST()

```
def delete_item_from_list(self, AIstatus:str, status:str, name:str, the_list:str, item_to_delete:str, response:str):
        """[Methods that deletes an item from a certain list and updates the pickle file containing the lists]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: str
        :param name: [Name of the user in front of the camera]
        :type name: str
        :param the_list: [Name of the list, from which an item will be deleted]
        :type the_list: str
        :param item_to_delete: [Item to be deleted from the list]
        :type item_to_delete: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :return:

            - True -> the item was removed from the list

            - False -> the item was not removed from the list
        :rtype: [bool]
        """
        AIstatus.value = status['process']
        path = "User/" + name + "/" + "user_lists"
        list_exists = False
        removed = False
        if os.path.getsize(path) == 0:
            list_exists = False
        else:
            with open(path, 'rb') as f:
                x = pickle.load(f)
                list_exists = self.list_exists(x, the_list)
        try:
            if list_exists:
                with open(path, 'rb') as f:
                    old_lists = pickle.load(f)
                    lists = old_lists
                lists[the_list].remove(item_to_delete)
                with open(path, 'wb') as f:
                    pickle.dump(lists, f)
                    removed = True
        except:
            AIstatus.value = status["answer"]
            response.value = name + " I cannot find the " + item_to_delete + " on the " + the_list
            virtual_assistant.speak(response.value)
        return removed
```

delete_item_from_list(*AIstatus: str, status: str, name: str, the_list: str, item_to_delete: str, response: str*)

[Methods that deletes an item from a certain list and updates the pickle file containing the lists]

Parameters:
- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*str*) – [Virtual assistant's statuses available]
- **name** (*str*) – [Name of the user in front of the camera]
- **the_list** (*str*) – [Name of the list, from which an item will be deleted]
- **item_to_delete** (*str*) – [Item to be deleted from the list]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]

Returns:
- True -> the item was removed from the list
- False -> the item was not removed from the list

Return type: [bool]

RUN_ADD_ITEM_TO_LIST()

```
def run_add_item_to_list(self, AIstatus:str, response:str, status:dict, understanding:str, request:str):
        """[Method that prepares the environment to add an item to a list]

        :param AIstatus: [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        :param request: [User's request in raw format]
        :type request: str
        """
        AIstatus.value = status['process']
        name = str(self.name)
        confirm = ""
        AIstatus.value = status['process']
        operator = helper.Context(helper.word_after())
        if "add" in request:
            item_to_add = operator.run(request, "add")
        else:
            item_to_add = operator.run(request, "put")
        operator = helper.Context(helper.word_before())
        the_list = operator.run(request, "list")
        AIstatus.value = status["answer"]
        response.value = name + " you want to add " + item_to_add.upper() + " in the " + the_list.upper() + " list, right?
\n\nAnswer YES THIS IS RIGHT or NO, THIS IS NOT CORRECT."
        virtual_assistant.speak(response.value)
        confirm = virtual_assistant.listen_decision(AIstatus)
        understanding.value = "Responding to: " + confirm
        if "right" in confirm or "yes" in confirm:
            added = self.add_item_to_list(AIstatus, status,  name, the_list, item_to_add)
            if added:
                AIstatus.value = status['answer']
                response.value = "The " + item_to_add + " was added to the " + the_list + " list."
```

```
                virtual_assistant.speak(response.value)
                self.see_list(AIstatus, response, the_list, status, name)
            else:
                response.value = "The " + the_list + " list does not exist. Would you like to create it?\n\nAnswer YES I W
ANT TO CREATE or NO CANCEL THIS LIST."
                virtual_assistant.speak(response.value)
                confirm = virtual_assistant.listen_decision(AIstatus)
                if "yes" in confirm or "create" in confirm:
                    self.create_list(name, AIstatus, status, response, the_list)
                    self.add_item_to_list(AIstatus, status, name, the_list, item_to_add)
                    AIstatus.value = status['answer']
                    response.value = "The " + item_to_add + " was added to the " + the_list + " list."
                    virtual_assistant.speak(response.value)
```

run_add_item_to_list(*AIstatus: str, response: str, status: dict, understanding: str, request: str*)

[Method that prepares the environment to add an item to a list]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]
- **request** (*str*) – [User's request in raw format]

ADD_ITEM_TO_LIST()

```
def add_item_to_list(self, AIstatus:str, status:dict,  name:str, the_list:str, item_to_add:str):
        """[Method that adds an item to a certain list and updates the pickle file containing the lists]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param name: [The name of the user standing in front of the camera]
        :type name: str
        :param the_list: [Name of the list where the item will be added]
        :type the_list: str
        :param item_to_add: [Name of the item to be added]
        :type item_to_add: str
        :return:

            - True -> the item was added to the list

            - False -> the item was not added to the list

        :rtype: [type]
        """
        AIstatus.value = status['process']
        path = "User/" + name + "/" + "user_lists"
        list_exists = False
        added = False
        if os.path.getsize(path) == 0:
            list_exists = False
        else:
            with open(path, 'rb') as f:
                x = pickle.load(f)
                list_exists = self.list_exists(x, the_list)
        if list_exists:
            with open(path, 'rb') as f:
                old_lists = pickle.load(f)
                lists = old_lists
            lists[the_list].append(item_to_add)
            with open(path, 'wb') as f:
                pickle.dump(lists, f)
                added = True
        return added
```

add_item_to_list(*AIstatus: str, status: dict, name: str, the_list: str, item_to_add: str*)

[Method that adds an item to a certain list and updates the pickle file containing the lists]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **name** (*str*) – [The name of the user standing in front of the camera]
- **the_list** (*str*) – [Name of the list where the item will be added]
- **item_to_add** (*str*) – [Name of the item to be added]

Returns:

- True -> the item was added to the list
- False -> the item was not added to the list

Return type: [type]

SEE_ALL_LISTS()

```
def see_all_lists(self, AIstatus:str, status:dict, response:str):
        """[Method that retrieves all the lists that the user has already]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        """
        AIstatus.value = status['process']
        name = str(self.name)
        AIstatus.value = status['process']
        path = "User/" + name + "/" + "user_lists"
        if os.path.getsize(path) == 0:
            response.value = name + " you do not have any lists at the moment.\nIf you want to create one say PIXEL I WANT
 TO CREATE A LIST."
            virtual_assistant.speak(response.value)
        else:
            answer = name +  " you have the following lists: \n\n"
            i = 1
            with open(path, 'rb') as f:
                x = pickle.load(f)
                for key in x:
                    answer = answer + "\t" + str(i) + ". " + key
                    if i%2 == 0:
                        answer = answer + "\n"
                    i = i + 1
            AIstatus.value = status['answer']
            response.value = answer
            virtual_assistant.speak(response.value)
```

see_all_lists(*AIstatus: str, status: dict, response: str*)

[Method that retrieves all the lists that the user has already]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]

RUN_SEE_LIST()

```
def run_see_list(self, AIstatus:str, response:str, status:dict, request:str, understanding:str):
        """[Method that prepares the environment to retrieve a certain list and its items]

        :param AIstatus: [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param request: [User's request in raw format]
        :type request: str
        :param understanding: [What the virtual assistant understood from the user's request (in text format). This text i
s displayed on the mirror surface]
        :type understanding: str
        """
        name = str(self.name)
        confirm = ""
        operator = helper.Context(helper.word_before())
        the_list = operator.run(request, "list")
        while "cancel" not in confirm and "yes" not in confirm and "right" not in confirm:
            AIstatus.value = status["answer"]
            response.value = name + " you want to see the " + the_list.upper() + " list, right?\n\nAnswer YES THIS IS RIGH
T or NO THIS IS NOT CORRECT."
            virtual_assistant.speak(response.value)
            confirm = virtual_assistant.listen_decision(AIstatus)
            understanding.value = "Responding to: " + confirm
            AIstatus.value = status["process"]
            if "no" in confirm or "correct" in confirm:
                AIstatus.value = status["answer"]
                response.value = name + ", tell me the name of the list again!"
                virtual_assistant.speak(response.value)
                the_list = virtual_assistant.listen_decision(AIstatus)
                understanding.value = "Responding to: " + the_list
                if "list" in the_list:
                    the_list = operator.run(the_list, "list")
            elif "right" in confirm or "yes" in confirm:
                self.see_list(AIstatus, response, the_list, status, name)
```

run_see_list(*AIstatus: str, response: str, status: dict, request: str, understanding: str*)

[Method that prepares the environment to retrieve a certain list and its items]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant's current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant's statuses available]
- **request** (*str*) – [User's request in raw format]
- **understanding** (*str*) – [What the virtual assistant understood from the user's request (in text format). This text is displayed on the mirror surface]

SEE_LIST()

```
def see_list(self, AIstatus:str, response:str, the_list:str, status:dict, name:str):
        """[Method that retrieves a certain list from the pickle file]

        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param the_list: [List to be displayed]
        :type the_list: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        :param name: [Name of the user in front of the camera]
        :type name: str
        """
        AIstatus.value = status["process"]
        path = "User/" + name + "/" + "user_lists"
        with open(path, 'rb') as f:
            x = pickle.load(f)
            if self.list_exists(x, the_list):
                i = 1
                answer = ""
                for key in x:
                    if key == the_list:
                        answer = key + ":\n\n"
                        for item in x[key]:
                            if "nothing" not in item:
                                answer = answer + "\t\t" + str(i) +". " + item
                                if i%2 == 0:
                                    answer = answer + "\n"
                                i = i + 1
                AIstatus.value = status["answer"]
                response.value = answer
                virtual_assistant.speak(response.value)
            else:
                AIstatus.value = status["answer"]
                response.value = name + " I am really sorry, I could not find the " + the_list + " list."
                virtual_assistant.speak(response.value)
```

see_list(*AIstatus: str, response: str, the_list: str, status: dict, name: str*)

[Method that retrieves a certain list from the pickle file]

Parameters:

- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **the_list** (*str*) – [List to be displayed]
- **status** (*dict*) – [Virtual assistant’s statuses available]
- **name** (*str*) – [Name of the user in front of the camera]

ERROR()

```
def error(self, response:str, AIstatus:str, status:dict):
        """[Method that runs if the user requesting to manipulate a list is not registered on the device]

        :param response: [The Virtual Assistant's answer in text format. This text is displayed on the mirror surface]
        :type response: str
        :param AIstatus: [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirr
or surface]
        :type AIstatus: str
        :param status: [Virtual assistant's statuses available]
        :type status: dict
        """
        AIstatus.value = status["answer"]
        response.value = "Ummm... stranger, you are not registered, so you cannot create any list.\nPlease say PIXEL I WAN
T TO REGISTER, to unlock all my skills!"
        virtual_assistant.speak(response.value)
```

error(*response: str, AIstatus: str, status: dict*)

[Method that runs if the user requesting to manipulate a list is not registered on the device]

Parameters:

- **response** (*str*) – [The Virtual Assistant’s answer in text format. This text is displayed on the mirror surface]
- **AIstatus** (*str*) – [Virtual assistant’s current status: listening, answer, etc. This status is displayed on the mirror surface]
- **status** (*dict*) – [Virtual assistant’s statuses available]

COMUNICATION

```
class Communication:
    """[This class is an API for the GSM class]
    """
    def __init__(self):
        """[Constructor which creates a GSMHat object]
        """
        self.gsm = GSM.GSMHat('/dev/ttyUSB0', 115200)
        self.hang_up = True
```

class **communication.Communication**

Bases: `object`

[This class is an API for the GSM class]

SEND_SMS()

```
def send_SMS(self, sender:str, receiver:str, message:str):
        """[Method that sends and a text message, from the GSM hat module to a certain phone number]

        :param sender: [Name of the sender]
        :type sender: str
        :param receiver: [Phone number of the receiver]
        :type receiver: str
        :param message: [Text message]
        :type message: str
        """
        Number = receiver
        Message = sender + " sent you the following message:\n\n" + message
        self.gsm.SMS_write(Number, Message)
        time.sleep(5)
        print("-----------------------------------------")
        print("Sent to:  " + receiver)
        print("Message:  " + message)
        print("[STATUS]: SENT!")
        print("-----------------------------------------")
```

send_SMS(*sender: str, receiver: str, message: str*)

[Method that sends and a text message, from the GSM hat module to a certain phone number]

Parameters:

- **sender** (*str*) – [Name of the sender]
- **receiver** (*str*) – [Phone number of the receiver]
- **message** (*str*) – [Text message]

CHECK_RECEIVED_SMS()

```
def check_received_SMS(self):
        """[Method that checks if the SIM inserted in the GPS module had received any text messages. If any text messages
were received, they are printed to the console]
        """
        time.sleep(1)
        print(self.gsm.SMS_available())
        if self.gsm.SMS_available() > 0:
            number_messages = self.gsm.SMS_available()
            while number_messages != 0:
                new_sms = self.gsm.SMS_read()
                print("-----------------------------------------")
                print('Got new SMS from number %s' % new_sms.Sender)
                print('It was received at %s' % new_sms.Date)
                print('The message is: %s' % new_sms.Message)
                print("-----------------------------------------")
                number_messages -= 1
        time.sleep(3)
```

check_received_SMS()

[Method that checks if the SIM inserted in the GPS module had received any text messages. If any text messages were received, they are printed to the console]

GET_LOCATION()

```
def get_location(self):
        """[Method that retrieves  location details such as:

        - [Longitute]

        - [Latitude]

        - [Altitude]

        - [Course]

        - [GPS Satellites]

        - [GNSS Satellites]

        - [Movement Speed]

        - [Others]

        The method transforms the longitude and latitude into the corresponding address
        ]

        :return: [Address according to the longitude and latitude]
        :rtype: [str]
        """
        from geopy.geocoders import Nominatim
        # Lets print some values
        time.sleep(1)
        GPS_Data = self.gsm.GPS_Data_List()
        for key in GPS_Data:
            print(key + ": " + GPS_Data[key])

        lat = GPS_Data['Latitude']
        lon = GPS_Data['Longitude']
        point = lat + "," + lon
        try:
            geolocator = Nominatim(user_agent="Pixel")
            location = geolocator.reverse(point)
            address = location.address
            raw = location.raw
            return raw
        except:
            print("GPS not working")
```

get_location()

[Method that retrieves location details such as:

- [Longitute]
- [Latitude]
- [Altitude]
- [Course]
- [GPS Satellites]
- [GNSS Satellites]
- [Movement Speed]
- [Others]

The method transforms the longitude and latitude into the corresponding address]

Returns: [Address according to the longitude and latitude]

Return type: [str]

CALL()

```
def call(self, number):
        """[Method that performs a call to another number]

        :param number: [Phone number to call]
        :type number: [str]
        """
        self.hang_up = False
        print("Dialing...")
        print("Calling...")
        self.gsm.Call(number, 60)
        time.sleep(60)
```

call(*number*)

[Method that performs a call to another number]

Parameters: **number** (*[str]*) – [Phone number to call]

START_GPS()

```
def start_GPS(self):
        """[Method that activates the GPS]
        """
        self.gsm.start_GPS("ATI+CGPSPWR=1\r\n")
        self.gsm.start_GPS("ATI+CGPSPWR=?\r\n")
```

start_GPS()

[Method that activates the GPS]

CALL_ONGOING()

```
def call_ongoing(self):
        """[Method that checks if a phone call is ongoing]

        :return:

            - True -> an active call is ongoing

            - False -> there is not active call ongoing

        :rtype: [bool]
        """
        return self.gsm.Call_OnGoing()
```

call_ongoing()

[Method that checks if a phone call is ongoing]

Returns:

- True -> an active call is ongoing
- False -> there is not active call ongoing

Return type: [bool]

SMS

```
class SMS:
    """[Class than handles text messages]
    """
    def __init__(self):
        self.Message = ''
        self.Sender = ''
        self.Receiver = ''
        self.Date = ''
```

class **GSM.SMS**

Bases: `object`

[Class than handles text messages]

GPS

```
class GPS:
    """[Class that handles GPS details]
    """
    EarthRadius = 6371e3         # meters

    @staticmethod
    def CalculateDeltaP(Position1, Position2):
        """[Method that calculates the distance between two points on Earth]

        :param Position1: [Point one on Earth]
        :type Position1: [float]
        :param Position2: [Point two on Earth]
        :type Position2: [float]
        :return: [Distance between point one and point two]
        :rtype: [float]
        """
        phi1 = Position1.Latitude * math.pi / 180.0
        phi2 = Position2.Latitude * math.pi / 180.0
        deltaPhi = (Position2.Latitude - Position1.Latitude) * math.pi / 180.0
        deltaLambda = (Position2.Longitude - Position1.Longitude) * math.pi / 180.0

        a = math.sin(deltaPhi / 2) * math.sin(deltaPhi / 2) + math.cos(phi1) * math.cos(phi2) * math.sin(deltaLambda / 2)
* math.sin(deltaLambda / 2)
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
        d = GPS.EarthRadius * c    # in meters

        return d

    def __init__(self):
        """[Constructor of the GPS class]
        """
        self.GNSS_status = ""
        self.Fix_status = ""
        self.UTC = ''               # yyyyMMddhhmmss.sss
        self.Latitude = ""          # ±dd.dddddd            [-90.000000,90.000000]
        self.Longitude = ""         # ±ddd.dddddd           [-180.000000,180.000000]
        self.Altitude = ""          # in meters
        self.Speed = ""             # km/h [0,999.99]
        self.Course = ""            # degrees [0,360.00]
        self.HDOP = ""              # [0,99.9]
        self.PDOP = ""              # [0,99.9]
        self.VDOP = ""              # [0,99.9]
        self.GPS_satellites = ""    # [0,99]
        self.GNSS_satellites = ""   # [0,99]
        self.Signal = ""            # %      max = 55 dBHz
```

class **GSM.GPS**

Bases: `object`

[Class that handles GPS details]

static **CalculateDeltaP**(*Position1*, *Position2*)

[Method that calculates the distance between two points on Earth]

Parameters:
- **Position1** (*[float]*) – [Point one on Earth]
- **Position2** (*[float]*) – [Point two on Earth]

Returns: [Distance between point one and point two]

Return type: [float]

EarthRadius= *6371000.0*

```
class GSMHat:
    """[Class that handles the GSM hat]
    """

    regexGetSingleValue = r'([+][a-zA-Z\ ]+(:\ ))([\d]+)'
    regexGetAllValues = r'([+][a-zA-Z:\s]+)([\w\",\s+-\/:.]+)'
    timeoutSerial = 5
    timeoutGPSActive = 1
    timeoutGPSInactive = 2000
    cSMSwaittime = 2500             # milliseconds
    cGPRSstatusWaittime = 5000      # milliseconds

    def __init__(self, SerialPort, Baudrate, Logpath='gmsHat.log'):
        """[Constructor of the GSMHat]

        :param SerialPort: [Serial port]
        :type SerialPort: [str]
        :param Baudrate: [Baundrate]
        :type Baudrate: [int]
        :param Logpath: [path of the log file], defaults to 'gmsHat.log'
        :type Logpath: str, optional
        """
        self.__baudrate = Baudrate
        self.__port = SerialPort

        self.__logger = logging.getLogger(__name__)
        self.__logger.setLevel(logging.DEBUG)
        self.__loggerFileHandle = logging.FileHandler(Logpath)
        formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
        self.__loggerFileHandle.setFormatter(formatter)
        self.__loggerFileHandle.setLevel(logging.DEBUG)
        self.__logger.addHandler(self.__loggerFileHandle)

        self.__connect()
        self.__startWorking()
        self.GPS_Data = {}

    def start_GPS(self, command:str):
        """[Method that starts the GPS on the hat]

        :param command: [command that starts the hat]
        :type command: str
        """
        #!/usr/bin/python
        import serial
        import time
        ser = serial.Serial(
        port='/dev/ttyUSB0',\
        baudrate=115200,\
        parity=serial.PARITY_NONE,\
        stopbits=serial.STOPBITS_ONE,\
        bytesize=serial.EIGHTBITS,\
        timeout=0)
        print("Connected to: " + ser.portstr)
        command = command.encode()
        ser.write(command)
        time.sleep(3)
        command_no = 0
        while True:
            command_no = command_no +1
            line = ser.readline()
            if line:
                print(command_no,  line.decode(encoding='UTF-8'))
            else:
                break
        ser.close()
```

```
    def __connect(self):
        self.__ser = serial.Serial(self.__port, self.__baudrate)
        self.__ser.flushInput()
        self.__serData = ''
        self.__writeLock = False
        self.__logger.info('Serial connection to '+self.__port+' established')

    def __disconnect(self):
        self.__ser.close()

    def __startWorking(self):
        self.__working = True
        self.__state = 1
        self.__nextState = 0
        self.__smsToRead = 0
        self.__retryAfterTimeout = False
        self.__retryAfterTimeoutCount = 0
        self.__init = False
        self.__lastCommandSentString = ''
        self.__readRAW = 0
        self.__smsToBuild = None
        self.__smsList = []
        self.__smsSendList = []
        self.__SMSwaittime = 0
        self.__numberToCall = ''
        self.__sendHangUp = False
        self.__startGPS = False
        self.__GPRSwaittimeStatus = 0
        self.__GPRSIPaddress = ''
        self.__GPRSready = False
        self.__GPRSuserAPN = None
        self.__GPRSuserUSER = None
        self.__GPRSuserPWD = None
        self.__GPRScallUrlList = []
        self.__GPRSdataReceived = []
        self.__GPRSwaitForData = False
        self.__GPSstarted = False
        self.__GPSstartSending = False
        self.__GPSstopSending = False
        self.__GPScollectData = False
        self.__GPSactualData = GPS()
        self.__GPStimeout = self.timeoutGPSInactive
        self.__GPSwaittime = 0
        self.__workerThread = threading.Thread(target=self.__workerThread, daemon=True)
        self.__workerThread.start()

    def __stopWorking(self):
        self.__working = False
        self.__workerThread.join(10.0)  # Timeout = 10.0 Seconds

    def __sendToHat(self, string):
        if self.__writeLock == False:
            self.__lastCommandSentString = string
            string = string + '\n'
            self.__ser.write(string.encode('iso-8859-1'))
            self.__writeLock = True
            self.__sentTimeout = int(round(time.time())) + self.timeoutSerial
            self.__logger.debug('Sent to hat: %s' % string)
            return True
        else:
            self.__logger.debug('Wait for Lock...   state: ' + str(self.__state) + ' senddata: ' + string)
            time.sleep(1)
            return False

    def __pressPowerKey(self):
        """[Power on the GPS hat (if the hat is connected via the GPIO pins)]
        """
        GPIO.setmode(GPIO.BOARD)
        GPIO.setup(7, GPIO.OUT)
        while True:
            GPIO.output(7, GPIO.LOW)
            time.sleep(4)
            GPIO.output(7, GPIO.HIGH)
```

```
            break
        GPIO.cleanup()
        time.sleep(10)

    def SMS_available(self):
        """[Method that checks how many received messages are]

        :return: [Number of text messages received]
        :rtype: [int]
        """
        return len(self.__smsList)

    def SMS_read(self):
        """[Method that reads received text message]

        :return: [Text message]
        :rtype: [str]
        """
        if self.SMS_available() > 0:
            retSMS = self.__smsList[0]
            del self.__smsList[0]
            return retSMS
        return None

    def SMS_write(self, NumberReceiver, Message):
        """[Method that sends a text message]

        :param NumberReceiver: [Receiver message]
        :type NumberReceiver: [str]
        :param Message: [Text message]
        :type Message: [str]
        """
        newSMS = SMS()
        newSMS.Receiver = NumberReceiver
        newSMS.Message = Message
        self.__smsSendList.append(newSMS)

    def Call(self, Number, Timeout = 15):
        """[Method that starts a call]

        :param Number: [Receiver phone number]
        :type Number: [str]
        :param Timeout: [Hang up after this number of seconds], defaults to 15
        :type Timeout: int, optional
        :return: [Call performed]
        :rtype: [bool]
        """
        if self.__numberToCall == '':
            self.__numberToCall = str(Number)
            self.__callTimeout = Timeout
            return True
        return False

    def Call_OnGoing(self):
        """[Method that checks if there is an active call ingoing]

        :return:

            - True -> a call is ongoing

            - False -> there is not active call

        :rtype: [type]
        """
        #!/usr/bin/python
        import serial
        import time
        ser = serial.Serial(
        port='/dev/ttyUSB0',\
        baudrate=115200,\
        parity=serial.PARITY_NONE,\
        stopbits=serial.STOPBITS_ONE,\
        bytesize=serial.EIGHTBITS,\
```

```
        timeout=0)
        print("Connected to: " + ser.portstr)
        command = "ATI+CPAS\r\n"
        command = command.encode()
        ser.write(command)
        time.sleep(3)
        command_no = 0
        while True:
            command_no = command_no +1
            line = ser.readline()
            if line:
                answer = line.decode(encoding='UTF-8')
                print(command_no,  answer)
                if "+CPAS: 0":
                    return False
                elif "+CPAS: 4":
                    return True
            else:
                break
        ser.close()


    def HangUp(self):
        """[Method that hangs up a call]
        """
        self.__sendHangUp = True

    def GetActualGPS(self):
        return self.__GPSactualData

    def UrlResponse_available(self):
        return len(self.__GPRSdataReceived)

    def UrlResponse_read(self):
        if self.UrlResponse_available() > 0:
            retResponse = self.__GPRSdataReceived[0]
            del self.__GPRSdataReceived[0]
            return retResponse

        return None

    def CallUrl(self, url):
        self.__GPRScallUrlList.append(url)
        self.__logger.debug('Got new URL call')

    def PendingUrlCalls(self):
        return len(self.__GPRScallUrlList)

    def SetGPRSconnection(self, APN, Username, Password):
        self.__GPRSuserAPN = APN
        self.__GPRSuserUSER = Username
        self.__GPRSuserPWD = Password

    def __startGPSUnit(self):
        self.__startGPS = True

    def __startGPSsending(self):
        self.__GPSstartSending = True

    def __stopGPSsending(self):
        self.__GPSstopSending = True

    def __collectGPSData(self):
        self.__GPScollectData = True

    def ColData(self):
        self.__collectGPSData()

    def close(self):
        self.__disconnect()
        self.__logger.info('Serial connection to '+self.__port+' closed')
        self.__stopWorking()
```

```
    def __processData(self):
        """[Method that processes GPS data such as:

        - [Longitute]

        - [Latitude]

        - [Altitude]

        - [Course]

        - [GPS Satellites]

        - [GNSS Satellites]

        - [Movement Speed]

        - [Others]]

        """
        if self.__serData != '':
            if self.__readRAW > 0:
                self.__logger.debug('Received Raw Data: %s' % self.__serData)
                if self.__readRAW == 1:
                    # Handle SMS
                    if self.__serData == 'OK\r\n':
                        self.__smsToBuild.Message = self.__smsToBuild.Message.rstrip('\r\n')
                        self.__smsList.append(self.__smsToBuild)
                        self.__readRAW = 0
                        self.__writeLock = False
                    else:
                        self.__smsToBuild.Message = self.__smsToBuild.Message + self.__serData
                elif self.__readRAW == 2:
                    # Handle HTTP Response
                    if self.__serData == 'OK\r\n':
                        self.__readRAW = 0
                        self.__writeLock = False
                        self.__GPRSdataToBuild = self.__GPRSdataToBuild.rstrip('\r\n')
                        self.__GPRSdataReceived.append(self.__GPRSdataToBuild)
                    else:
                        self.__GPRSdataToBuild = self.__GPRSdataToBuild + self.__serData
            else:
                self.__logger.debug('Received Data: %s' % self.__serData)

                if 'OK' in self.__serData:
                    self.__writeLock = False
                    self.__logger.debug('Lock Off')
                if 'ERROR' in self.__serData:
                    # ERROR Handling here
                    if self.__state == 71:
                        # Error after sending AT+HTTPINIT
                        # Lets terminate request before starting new one
                        self.__logger.info('Error after starting new HTTP Request.')
                        self.__writeLock = False
                        self.__state = 75
                elif '+CME ERROR:' in self.__serData:
                    self.__writeLock = False

                    match = re.findall(self.regexGetSingleValue, self.__serData)
                    self.__cmeErr = int(match[0][1])

                    self.__logger.info('Got CME ERROR: %s' % match[0][1])
                elif '+CMS ERROR:' in self.__serData:
                    self.__writeLock = False

                    match = re.findall(self.regexGetSingleValue, self.__serData)
                    self.__cmsErr = int(match[0][1])

                    self.__logger.info('Got CMS ERROR: %s' % match[0][1])
                elif '+CPMS:' in self.__serData:
                    match = re.findall(self.regexGetAllValues, self.__serData)
                    rawData = match[0][1].split(',')
                    self.__masSMSSpace = int(rawData[1])
```

```
            numSMS = int(rawData[0])
            if numSMS > 0:
                self.__smsToRead = 1

        elif '+CMGR:' in self.__serData:
            # read SMS content
            match = re.findall(self.regexGetAllValues, self.__serData)
            rawData = match[0][1].split('","')
            self.__readRAW = 1
            self.__smsToBuild = SMS()
            #self.__smsToBuild.Sender = bytearray.fromhex(rawData[1]).decode()
            self.__smsToBuild.Sender = rawData[1]
            self.__smsToBuild.Date = rawData[3].replace('"', '')
            self.__smsToBuild.Date = datetime.strptime(rawData[3].replace('"', '')[:-3], '%y/%m/%d,%H:%M:%S')
            self.__smsToBuild.Message = ''

        elif '+SAPBR:' in self.__serData:
            # check if IP is valid
            # Return value looks like: +SAPBR: 1,3,"0.0.0.0"
            match = re.findall(self.regexGetAllValues, self.__serData)
            rawData = match[0][1].split(',')
            self.__GPRSIPaddress = rawData[2].replace('"', '')

            if self.__GPRSIPaddress != '0.0.0.0':
                self.__GPRSready = True
            else:
                self.__GPRSready = False

        elif '+HTTPREAD:' in self.__serData:
            # read HTTP content
            self.__GPRSdataToBuild = ''
            self.__readRAW = 2

        elif '+HTTPACTION:' in self.__serData:
            # Return value looks like: +HTTPACTION: 0,200,0
            match = re.findall(self.regexGetAllValues, self.__serData)
            rawData = match[0][1].split(',')
            self.__GPRSgotHttpResponse = True
            if len(rawData) == 3:
                requestMethod = int(rawData[0])
                httpStatus = int(rawData[1])
                recvDataLength = int(rawData[2])
                if httpStatus == 200:  # Successful request
                    self.__GPRSnewDataReceived = True
                elif httpStatus == 601:  # Successful request
                    self.__logger.info('HTTPACTION Network Error ' + str(httpStatus))
                else:
                    self.__logger.info('HTTPACTION Unhandled Error ' + str(httpStatus))

            else:
                self.__logger.info('HTTPACTION return value is not expected: ' + match[0][1])

        # unannounced data reception below (e.g. new SMS oder phone call)
        elif '+CMTI:' in self.__serData:
            self.__logger.info('Received new SMS')
            match = re.findall(self.regexGetAllValues, self.__serData)
            rawData = match[0][1].split(',')
            storage = rawData[0]
            numSMS = int(rawData[1])
            self.__logger.debug('New SMS in memory ' + storage + ' at position ' + str(numSMS))
            self.__smsToRead = numSMS

        # GPS Data coming here
        elif '+CGNSINF:' in self.__serData:
            self.__logger.debug('New GPS Data:')
            match = re.findall(self.regexGetAllValues, self.__serData)
            rawData = match[0][1].split(',')
            if len(rawData) == 21:
                newGPS = GPS()

                newGPS.GNSS_status = rawData[0]
                #print("GNSS Status: " + rawData[0])
                self.GPS_Data["GNSS Status"] = rawData[0]
```

```
                    newGPS.Fix_status = str(rawData[1])
                    #print("Fix Status: " + rawData[1])
                    self.GPS_Data["Fix Status"] = rawData[1]

                    #newGPS.UTC = datetime.strptime(rawData[2][:-4], '%Y%m%d%H%M%S')
                    #print("UTC:" + datetime.strptime(rawData[2][:-4], '%Y%m%d%H%M%S'))

                    newGPS.Latitude = str(rawData[3])
                    #print("Latitude: " + str(rawData[3]))
                    self.GPS_Data["Latitude"] = rawData[3]

                    newGPS.Longitude = str(rawData[4])
                    #print("Longitude: " + str(rawData[4]))
                    self.GPS_Data["Longitude"] = rawData[4]

                    newGPS.Altitude = str(rawData[5])
                    #print("Altitude: " + str(rawData[5]))
                    self.GPS_Data["Altitude"] = rawData[5]

                    newGPS.Speed = str(rawData[6])
                    #print("Speed: " + str(rawData[6]))
                    self.GPS_Data["Speed"] = rawData[6]

                    newGPS.Course = str(rawData[7])
                    #print("Course: " + str(rawData[7]))
                    self.GPS_Data["Course"] = rawData[7]

                    newGPS.HDOP = str(rawData[10])
                    #print("HDOP" + str(rawData[10]))
                    self.GPS_Data["HDOP"] = rawData[10]

                    newGPS.PDOP = str(rawData[11])
                    #print("PDOP: " + str(rawData[11]))
                    self.GPS_Data["PDOP"] = rawData[11]

                    newGPS.VDOP = str(rawData[12])
                    #print("VDOP: " + str(rawData[12]))
                    self.GPS_Data["VDOP"] = rawData[12]

                    newGPS.GPS_satellites = str(rawData[14])
                    #print("GPS Satellites: " + str(rawData[14]))
                    self.GPS_Data["GPS Satellites"] = rawData[14]

                    newGPS.GNSS_satellites = str(rawData[15])
                    #print("GNSS Satelites: " + str(rawData[15]))
                    self.GPS_Data["GNSS Satelites"] = rawData[15]

                    newGPS.Signal = str(rawData[18])
                    #print("Signal: " + str(rawData[18]))
                    self.GPS_Data["Signal"] = rawData[18]

                    self.__GPSactualData = newGPS

            self.__serData = ''

    def GPS_Data_List(self):
        """[Method that returns GPS details:

        - [Longitute]

        - [Latitude]

        - [Altitude]

        - [Course]

        - [GPS Satellites]

        - [GNSS Satellites]

        - [Movement Speed]
```

```
        - [Others]]

        :return: [GPS details]
        :rtype: [dict]
        """
        self.__processData()
        return self.GPS_Data

    def __restartProcedure(self):
        self.__logger.error('Try to restart gsm module')
        self.__pressPowerKey()
        self.__state = 1
        self.__writeLock = False
        self.__retryAfterTimeout = False
        self.__sentTimeout = 0

    def __waitForUnlock(self):
        actTime = int(round(time.time()))
        if self.__sentTimeout > 0 and actTime > self.__sentTimeout:
            # Timeout
            self.__logger.error('Timeout during data reception')
            self.__logger.info('Command sent: ' + self.__lastCommandSentString)
            self.__logger.info('Actual state of programme: ' + str(self.__state))

            if self.__state == 2 or self.__state == 97:
                # It might be that the gsm module is not powered on
                # So let's try to restart
                self.__restartProcedure()
                return False
            elif  self.__state == 3:
                # Tried to check for new SMS
                # Retry 3 times
                if self.__retryAfterTimeout:
                    if self.__retryAfterTimeoutCount > 0:
                        self.__retryAfterTimeoutCount -= 1
                    else:
                        self.__restartProcedure()
                        return False
                else:
                    self.__retryAfterTimeout = True
                    self.__retryAfterTimeoutCount = 3

                self.__state = 2
                self.__writeLock = False
                self.__sentTimeout = 0
                return False
            else:
                self.__logger.critical('Exception: Unhandled timeout during data reception')
                raise 'Unhandled timeout during data reception'

        if self.__writeLock:
            return False
        else:
            self.__sentTimeout = 0
            return True

    def __workerThread(self):
        self.__logger.info('Worker started')
        self.__waitTime = 0

        while self.__working:
            # Check for incoming chars
            while self.__ser.inWaiting() > 0:
                newChar = self.__ser.read().decode('iso-8859-1')

                if newChar == '\n':
                    if self.__readRAW > 0:
                        self.__serData += newChar
                    self.__processData()
                else:
                    if newChar == '\r':
                        if self.__readRAW > 0:
                            self.__serData += newChar
```

```
                else:
                    self.__serData += newChar

            # Statemachine
            actTime = int(round(time.time() * 1000))
            if self.__state == 1:
                if self.__sendToHat('AT+CMGF=1'):
                    self.__startGPSUnit()
                    self.__stopGPSsending()
                    self.__state = 2
            elif self.__state == 2:
                if self.__waitForUnlock():
                    if self.__sendToHat('AT+CPMS="SM"'):
                        self.__state = 3
            elif self.__state == 3:
                if self.__waitForUnlock():
                    self.__state = 97
            elif self.__state == 20:
                # Read SMS
                if self.__sendToHat('AT+CMGR='+str(self.__smsToRead)):
                    self.__state = 21
            elif self.__state == 21:
                if self.__waitForUnlock():
                    if self.__smsToBuild == None:
                        # An der Stelle self.__smsToRead gab es keine SMS zu lesen
                        pass
                    else:
                        # Es gab eine neue SMS
                        self.__logger.info('New Message from ' + self.__smsToBuild.Sender + ' was received')
                        self.__smsToBuild = None

                    # Lösche die behandelte SMS an der Stelle
                    if self.__sendToHat('AT+CMGD='+str(self.__smsToRead)):
                        self.__state = 22

            elif self.__state == 22:
                if self.__waitForUnlock():
                    if(self.__smsToRead == 20):
                        self.__smsToRead = 0
                    else:
                        self.__smsToRead = self.__smsToRead + 1

                    self.__state = 97

            elif self.__state == 30:
                # SMS versenden
                retSMS = self.__smsSendList[0]
                messageString = 'AT+CMGS="' + retSMS.Receiver + '"\n' + retSMS.Message + '\x1A'
                self.timeoutSerial = 30
                if self.__sendToHat(messageString):
                    self.__state = 31

            elif self.__state == 31:
                if self.__waitForUnlock():
                    retSMS = self.__smsSendList[0]
                    self.__logger.info('Message to ' + retSMS.Receiver + ' successfully sent')
                    del self.__smsSendList[0]
                    self.timeoutSerial = 5

                    self.__state = 97

            elif self.__state == 40:
                if self.__sendToHat('ATD' + self.__numberToCall + ';'):
                    self.__state = 41

            elif self.__state == 41:
                if self.__waitForUnlock():
                    self.__waitTime = actTime + self.__callTimeout * 1000
                    self.__state = 42

            elif self.__state == 42:
                    # Wait x Seconds
                if actTime > self.__waitTime or self.__sendHangUp == True:
```

```
            self.__numberToCall = ''
            self.__sendHangUp = True
            self.__state = 97

        elif self.__state == 43:
            if self.__sendToHat('AT+CHUP'):
                self.__state = 44

        elif self.__state == 44:
            if self.__waitForUnlock():
                self.__sendHangUp = False
                self.__state = 97

        elif self.__state == 50:
            if self.__sendToHat('AT+CGNSPWR=1'):
                self.__state = 51

        elif self.__state == 51:
            if self.__waitForUnlock():
                self.__logger.debug('GPS powered on')
                self.__startGPS = False
                self.__state = 97

        elif self.__state == 52:
            if self.__sendToHat('AT+CGNSTST=1'):
                self.__state = 55
                self.__logger.debug('GPS start sending')
                self.__GPSstartSending = False

        elif self.__state == 53:
            if self.__sendToHat('AT+CGNSTST=0'):
                self.__state = 55
                self.__GPSstopSending = False

        elif self.__state == 54:
            if self.__sendToHat('AT+CGNSINF'):
                self.__state = 55
                self.__GPScollectData = False

        elif self.__state == 55:
            if self.__waitForUnlock():
                self.__state = 97

        elif self.__state == 60:
            if self.__sendToHat('AT+SAPBR=2,1'):
                self.__state = 61

        elif self.__state == 61:
            if self.__waitForUnlock():
                tempState = 97
                if self.__GPRSready == True:
                    pass
                else:
                    if self.__GPRSuserAPN != None and self.__GPRSuserUSER != None and self.__GPRSuserPWD != None:
                        # try to connect
                        tempState = 62

                self.__state = tempState

        elif self.__state == 62:
            if self.__sendToHat('AT+SAPBR=3,1,"Contype","GPRS"'):
                self.__state = self.__state + 1

        elif self.__state == 63:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+SAPBR=3,1,"APN","' + self.__GPRSuserAPN + '"'):
                    self.__state = self.__state + 1

        elif self.__state == 64:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+SAPBR=3,1,"USER","' + self.__GPRSuserUSER + '"'):
                    self.__state = self.__state + 1
```

```
        elif self.__state == 65:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+SAPBR=3,1,"PWD","' + self.__GPRSuserPWD + '"'):
                    self.__state = self.__state + 1

        elif self.__state == 66:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+SAPBR=1,1'):
                    self.__state = self.__state + 1

        elif self.__state == 67:
            if self.__waitForUnlock():
                self.__state = 97

        elif self.__state == 70:
            # Call the first URL in List
            if self.__sendToHat('AT+HTTPINIT'):
                self.__state = self.__state + 1

        elif self.__state == 71:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+HTTPPARA="CID",1'):
                    self.__state = self.__state + 1

        elif self.__state == 72:
            if self.__waitForUnlock():
                getUrl = self.__GPRScallUrlList[0]
                if self.__sendToHat('AT+HTTPPARA="URL","' + getUrl + '"'):
                    del self.__GPRScallUrlList[0]
                    self.__GPRSwaitForData = True
                    self.__GPRSnewDataReceived = False
                    self.__GPRSgotHttpResponse = False
                    self.__state = self.__state + 1

        elif self.__state == 73:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+HTTPACTION=0'):
                    self.__state = 97
                    #self.__state = self.__state + 1

        elif self.__state == 74:
            if self.__waitForUnlock():
                if self.__sendToHat('AT+HTTPREAD'):
                    self.__state = self.__state + 1

        elif self.__state == 75:
            # Close HTTP Request
            if self.__waitForUnlock():
                if self.__sendToHat('AT+HTTPTERM'):
                    self.__state = 97
                    self.__GPRSwaitForData = False
                    self.__GPRSnewDataReceived = False

        elif self.__state == 97:
            # Check if new SMS to send is there
            if len(self.__smsSendList) > 0:
                self.__state = 30

            # Check if we have to Call somebody
            elif self.__numberToCall != '':
                self.__state = 40

            # Should I Hang Up ?
            elif self.__sendHangUp:
                self.__state = 43

            # Check if new SMS is there
            elif self.__smsToRead > 0:
                self.__state = 20

            # Check if we should call some Urls
            elif len(self.__GPRScallUrlList) > 0 and self.__GPRSready and self.__GPRSwaitForData == False:
                self.__state = 70
```

```
            elif self.__GPRSwaitForData and self.__GPRSgotHttpResponse:
                if self.__GPRSnewDataReceived:
                    self.__state = 74
                else:
                    self.__state = 75

            # Check if GPS Unit should start
            elif self.__startGPS:
                self.__state = 50

            # Check if GPS Unit should start send
            elif self.__GPSstartSending:
                self.__state = 52

            # Check if GPS Unit should stop send
            elif self.__GPSstopSending:
                self.__state = 53

            # Check if Single GPS Data should be collected
            elif self.__GPScollectData:
                self.__state = 54

            elif actTime > self.__GPSwaittime:
                self.__GPScollectData = True
                self.__GPSwaittime = actTime + self.__GPStimeout

            elif actTime > self.__SMSwaittime:
                self.__state = 2
                self.__SMSwaittime = actTime + self.cSMSwaittime

            elif actTime > self.__GPRSwaittimeStatus:
                self.__state = 60
                self.__GPRSwaittimeStatus = actTime + self.cGPRSstatusWaittime

            # Wait x Seconds
            elif actTime > self.__waitTime:
                if self.__nextState > 0:
                    self.__state = self.__nextState
                    self.__nextState = 0
        elif self.__state == 98:
            #Check if alive
            self.__logger.debug('Check if alive 98')
            if self.__sendToHat('AT'):
                self.__state = 99
        elif self.__state == 99:
            #Check if alive
            if self.__waitForUnlock():
                self.__state = 97
                self.__nextState = 98
                self.__waitTime = actTime + 5000

        # Let other Threads also do their job
        time.sleep(0.1)
    self.__logger.info('Worker ended')
```

class **GSM.GSMHat**(*SerialPort, Baudrate, Logpath='gmsHat.log'*)

Bases: `object`

[Class that handles the GSM hat]

Call(*Number, Timeout=15*)

[Method that starts a call]

Parameters:
- **Number** (*[str]*) – [Receiver phone number]
- **Timeout** (*int, optional*) – [Hang up after this number of seconds], defaults to 15

Returns: [Call performed]

Return type: [bool]

CallUrl(*url*)

Call_OnGoing()

[Method that checks if there is an active call ingoing]

Returns:
- True -> a call is ongoing
- False -> there is not active call

Return type: [type]

ColData()

GPS_Data_List()

[Method that returns GPS details:

- [Longitute]
- [Latitude]
- [Altitude]
- [Course]
- [GPS Satellites]
- [GNSS Satellites]
- [Movement Speed]
- [Others]]

Returns: [GPS details]

Return type: [dict]

GetActualGPS()

HangUp()

[Method that hangs up a call]

PendingUrlCalls()

SMS_available()

[Method that checks how many received messages are]

Returns: [Number of text messages received]

Return type: [int]

SMS_read()

[Method that reads received text message]

Returns: [Text message]

Return type: [str]

SMS_write(*NumberReceiver*, *Message*)

[Method that sends a text message]

Parameters:
- **NumberReceiver** (*[str]*) – [Receiver message]
- **Message** (*[str]*) – [Text message]

SetGPRSconnection(*APN*, *Username*, *Password*)

UrlResponse_available()

UrlResponse_read()

cGPRSstatusWaittime= *5000*

cSMSwaittime= *2500*

close()

regexGetAllValues= *'([+][a-zA-Z:\\s]+)([\\w\\",\\s+-\\/:.]+)'*

regexGetSingleValue= *'([+][a-zA-Z\\ ]+(:\\ ))([\\d]+)'*

start_GPS(*command: str*)

[Method that starts the GPS on the hat]

Parameters: **command** (*str*) – [command that starts the hat]

timeoutGPSActive= *1*

timeoutGPSInactive= *2000*

timeoutSerial= *5*

HELPER

```
class Helper(ABC):
    """[Interface for Helper classes. These classes contain simple methods that support skill classes.

    The helper applies the Strategy Pattern]
    """
    def run(self, *argv):
        pass
```

class **helper.Helper**

Bases: `abc.ABC`

[Interface for Helper classes. These classes contain simple methods that support skill classes.

The helper applies the Strategy Pattern]

run(**argv*)

CONTEXT

```
class Context():
    """[Context class for the Strategy Pattern, for helper]
    """
    def __init__(self, operator:Helper):
        """[Constructor]

        :param operator: [The class that is activated]
        :type operator: Helper
        """
        self._strategy = operator
```

class **helper.Context**(*operator: helper.Helper*)

Bases: `object`

[Context class for the Strategy Pattern, for helper]

STRATEGY()

```
def strategy(self):
        """[getter]

        :return: [returns the Helper object]
        :rtype: [type]
        """
        return self._strategy

    def strategy(self, operator:Helper):
        """[setter]

        :param operator: [sets the helper object]
        :type operator: Helper
        """
```

```
        self._strategy = operator
```

strategy(*operator: helper.Helper*)

[setter]

Parameters: **operator** (*Helper*) – [sets the helper object]

RUN()

```
def run(self, *argv):
        """[runs the helper object]

        :return: [returns the value returned by the helper object]
        :rtype:
            - bool
            - str
            - int
            - float
        """
        return self._strategy.run(*argv)
```

run(**argv*)

[runs the helper object]

Returns: [returns the value returned by the helper object]

Return type:

- bool
- str
- int
- float

```
class create_pickle_files(Helper):
    def run(self, name:str):
        """[After registration, this method creates three pickle files where user's details are stored:
            - [SOS contact details]

            - [Personal lists]

            - [Personal settings]]

        :param name: [The name of the user in front of the camera and the name of the folder where the pickle files are cr
eated]
        :type name: str
        """
        os.chdir("User/"+name)
        dictionaries = {}
        file_name = "user_lists"
        outfile = open(file_name, 'wb')
        pickle.dump(dictionaries, outfile)
        outfile.close()
        file_name = "user_settings"
        outfile = open(file_name, 'wb')
        pickle.dump(dictionaries, outfile)
        outfile.close()
        file_name = "SOS_contact"
        outfile = open(file_name, 'wb')
        pickle.dump(dictionaries, outfile)
        outfile.close()
        os.chdir("../..")
```

class **helper.create_pickle_files**

Bases: `helper.Helper`

run(*name: str*)

[After registration, this method creates three pickle files where user's details are stored:

- [SOS contact details]
- [Personal lists]
- [Personal settings]]

Parameters: **name** (*str*) – [The name of the user in front of the camera and the name of the folder where the pickle files are created]

DELETE_PICTURES

```
class delete_pictures(Helper):
    def run(self, name:str):
        """[Method that removes the pictures after registration, but keeps the folder for the user's personal settings and
 lists]

        :param name: [The name of the folder and the name of the user]
        :type name: str
        """
        shutil.rmtree("User/"+name)
        os.mkdir("User/"+name)
```

class **helper.delete_pictures**

Bases: `helper.Helper`

run(*name: str*)

[Method that removes the pictures after registration, but keeps the folder for the user's personal settings and lists]

Parameters: **name** (*str*) – [The name of the folder and the name of the user]

```
class folder_exists(Helper):
    def run(self, folder:str):
        """[The method checks if a certain path exists or not]

        :param folder: [The path that needs to be checked if it exists]
        :type folder: str
        :return:

            - True -> the path exists

            - False -> the path does not exist

        :rtype: str
        """
        if path.exists("User/"+folder):
            return True
        else:
            return False
```

class **helper.folder_exists**

Bases: `helper.Helper`

run(*folder: str*)

[The method checks if a certain path exists or not]

Parameters: **folder** (*str*) – [The path that needs to be checked if it exists]

Returns:

- True -> the path exists
- False -> the path does not exist

Return type: str

GET_LAST_WORD

```
class get_last_word(Helper):
    def run(self, string:str):
        """[The method returns the last word from a sentence]

        :param string: [The sentence that needs to be parsed]
        :type string: str
        :return: [The last word of initial sentence]
        :rtype: str
        """
        words = string.split(" ")
        last_word = words[-1]
        return last_word
```

class **helper.get_last_word**

Bases: `helper.Helper`

run(*string: str*)

[The method returns the last word from a sentence]

Parameters: **string** (*str*) – [The sentence that needs to be parsed]

Returns: [The last word of initial sentence]

Return type: str

```
class nice_formatted_long_text(Helper):
    def run(self, longString:str):
        """[Method that formats a long string to multiple rows]

        :param longString: [String received, that needs to be formatatted]
        :type longString: str
        """
        prettyString = ""
        for piece in self.splitter(5, longString):
            prettyString = prettyString + piece + "\n"
        prettyString = prettyString + "\n"
        return prettyString

    def splitter(self, numberOfWords:int,longString:str):
        """[Method created as a helper for the "nice_formatted_long_text(longString:str)" ]

        :param numberOfWords: [Denotes how many words are allowed on a single row]
        :type numberOfWords: int
        :param longString: [The initial string that needs to be parsed]
        :type longString: str
        """
        pieces = longString.split()
        answer = (" ".join(pieces[i:i+numberOfWords]) for i in range(0, len(pieces), numberOfWords))
        return answer
```

class **helper.nice_formatted_long_text**

Bases: `helper.Helper`

run(*longString: str*)

[Method that formats a long string to multiple rows]

Parameters: **longString** (*str*) – [String received, that needs to be formatatted]

splitter(*numberOfWords: int, longString: str*)

[Method created as a helper for the “nice_formatted_long_text(longString:str)”]

Parameters:

- **numberOfWords** (*int*) – [Denotes how many words are allowed on a single row]
- **longString** (*str*) – [The initial string that needs to be parsed]

```
class substring_after(Helper):
    def run(self, string:str, delimiter:str):
        """[Method returning the next two words after a keyword ]

        :param string: [Initial string]
        :type string: str
        :param delimiter: [The key word searched in the string]
        :type delimiter: str
        """
        answer = string.partition(delimiter)[2]
        return answer
```

class **helper.substring_after**

Bases: `helper.Helper`

run(*string: str, delimiter: str*)

[Method returning the next two words after a keyword]

Parameters:
- **string** (*str*) – [Initial string]
- **delimiter** (*str*) – [The key word searched in the string]

```
class remove_words(Helper):
    def run(self, string:str, words:list):
        """[Method that hat removes certain substrings from a string]

        :param string: [String that neeeds to be cleaned]
        :type string: str
        :param words: [The words that the string needs to be cleaned from]
        :type words: list
        :return: [The cleaned string]
        :rtype: str
        """
        for word in words:
            print("[INFO] removed " + word)
            print("[INFO] from " + string)
            string = string.replace(word, "")
            print("[INFO] string " + string)
        return string
```

class **helper.remove_words**

Bases: `helper.Helper`

run(*string: str*, *words: list*)

[Method that hat removes certain substrings from a string]

Parameters:
- **string** (*str*) – [String that neeeds to be cleaned]
- **words** (*list*) – [The words that the string needs to be cleaned from]

Returns: [The cleaned string]

Return type: str

```
class word_after(Helper):
    def run(self, string:str, delimiter:str):
        """[method that returns a certain word, following a particular string]

        :param string: [Initial string]
        :type string: str
        :param delimiter: [The word after which the string is required]
        :type delimiter: str
        :return: [A certain string]
        :rtype: [str]
        """
        print("[INFO] string to split: " + str(string))
        words = str(string).split()
        word = ""
        if delimiter in words:
            word = str(words[words.index(delimiter)+1])
        print("[INFO] word_after: " + word)
        return word
```

class **helper.word_after**

Bases: `helper.Helper`

run(*string: str*, *delimiter: str*)

[method that returns a certain word, following a particular string]

Parameters:
- **string** (*str*) – [Initial string]
- **delimiter** (*str*) – [The word after which the string is required]

Returns: [A certain string]

Return type: [str]

```
class word_before(Helper):
    def run(self, string:str, delimiter:str):
        """[Method that returns a string found before after a certain word]

        :param string: [Initial string]
        :type string: str
        :param delimiter: [The word before the string is required]
        :type delimiter: str
        :return: [A certain string]
        :rtype: [str]
        """
        print("[INFO] string to split: " + str(string))
        words = str(string).split()
        if delimiter in words:
            word = str(words[words.index(delimiter)-1])
            return word
```

class **helper.word_before**

Bases: `helper.Helper`

run(*string: str*, *delimiter: str*)

[Method that returns a string found before after a certain word]

Parameters:
- **string** (*str*) – [Initial string]
- **delimiter** (*str*) – [The word before the string is required]

Returns: [A certain string]

Return type: [str]

```
class remove_polite_words(Helper):
    def run(self, string:str):
        """[Method removing the polite words from a string, as they are not used by the virtual_assistant to process a req
uest]

        :param string: [String that needs to be cleaned from certain words]
        :type string: str
        """
        string = string.replace("please", "")
        string = string.replace("thank you", "")
        string = string.replace("hello", "")
        string = string.replace("the ", "")
        #string = string.replace("for", "")
        string = string.replace("of", "")
        string = string.replace("pixel", "")
        return string
```

class **helper.remove_polite_words**

Bases: `helper.Helper`

run(*string: str*)

[Method removing the polite words from a string, as they are not used by the virtual_assistant to process a request]

Parameters: **string** (*str*) – [String that needs to be cleaned from certain words]

GET_REQUEST

```
class get_request(Helper):
    def run(self, link:str):
        """[Method that performs a request and returns the response of the request in a JSON format]

        :param link: [Link to get the response from]
        :type link: str
        :return: [Returns the response of the server in a json format]
        :rtype: [JSON format]
        """
        response = requests.get(link)
        response = response.json()
        return response
```

class **helper.get_request**

Bases: `helper.Helper`

run(*link: str*)

[Method that performs a request and returns the response of the request in a JSON format]

Parameters: **link** (*str*) – [Link to get the response from]

Returns: [Returns the response of the server in a json format]

Return type: [JSON format]

AUTOMATED SETUP

INITIAL SETUP

To install Pixel Virtual Assistant on a Raspberry Pi, the following steps are required:

1. Assembly the hardware
2. Install the Raspbian OS
3. Clone the project from DoraTheodora/Pixel: "Pixel" is an Echo Dot with face recognition (github.com) with the following command:
    - git clone https://github.com/DoraTheodora/Pixel.git
4. Open a terminal and go to the cloned repository
5. Once in the repository's root directory, go to the "Setup" directory
    - cd AI-Pixel/Setup
6. Type the following command to install the virtual environment and the necessary libraries to run the project:
    - source ./setup.sh
7. Once the script finishes its execution, the virtual environment will be activated, and the project can be started with the following command:
    - python3 main.py

Note: The device will start only if the hardware is correctly assembled and installed.

STARTING PIXEL

1. Go to the root directory of the repository
2. Go to AI-Pixel directory
    - cd AI-Pixel
3. To activate the environment
    - source ~/python-env/Pixel/bin/activate
4. Starting Pixel (in the root/AI-Pixel folder of the repository):
    - python3 main.py
5. To deactivate the environment
    - deactivate

VIDEO TUTORIAL

A video tutorial on how to interact with Pixel can be found at Pixel Virtual Assistant - YouTube.

RESEARCH CODE

Some code was produced in the research phase to test the hardware, face recognition, text-to-speech, speech-to-text, servos movement, and others.

Some of the testing code was produced following some open-source projects, and some was created from scratch. In both cases, a valuable amount of time was invested.

These tests were necessary to find the best approach, technologies, test the hardware capabilities and get familiar with Raspberry Pi, and Python.

All the files containing testing code can be found at Pixel/Test_RasberryPi at main · DoraTheodora/Pixel (github.com).

The code produced in this phase is not well documented, as it was produced just to experiment with different software libraries and test the capabilities of the hardware.

Figure 33 "Testing code – root folder"

FACE RECOGNITION 1

This folder includes code testing face recognition using Dlib.

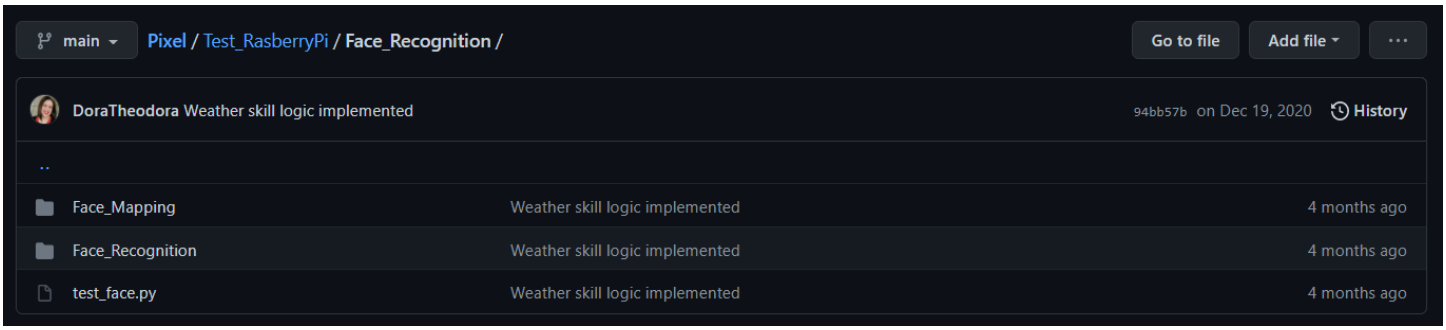


Figure 34 "Testing code"

FACE MAPPING

```
# import the necessary packages
from imutils import face_utils
import dlib
import cv2
# initialize dlib's face detector (HOG-based) and then create
# the facial landmark predictor
p = "shape_predictor_68_face_landmarks.dat"
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(p)
# load the input image and convert it to grayscale
image = cv2.imread("000.png")
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# detect faces in the grayscale image
rects = detector(gray, 0)
# loop over the face detections
for (i, rect) in enumerate(rects):
    # determine the facial landmarks for the face region, then
    # convert the facial landmark (x, y)-coordinates to a NumPy
    # array
    shape = predictor(gray, rect)
    shape = face_utils.shape_to_np(shape)
    # loop over the (x, y)-coordinates for the facial landmarks
    # and draw them on the image
    for (x, y) in shape:
        cv2.circle(image, (x, y), 2, (0, 255, 0), -1)
# show the output image with the face detections + facial landmarks
cv2.imshow("Output", image)
cv2.waitKey(0)
```

ENCODE FACES

```
#Theodora Tataru
#Test from pyserachimages
#16 October 2020

# USAGE
# When encoding on laptop, desktop, or GPU (slower, more accurate):
# python encode_faces.py --dataset dataset --encodings encodings.pickle --detection-method cnn
# When encoding on Raspberry Pi (faster, more accurate):
# python encode_faces.py --dataset dataset --encodings encodings.pickle --detection-method hog

# import the necessary packages
from imutils import paths
import face_recognition
import argparse
import pickle
import cv2
import os

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--dataset", required=True,
    help="path to input directory of faces + images")
ap.add_argument("-e", "--encodings", required=True,
    help="path to serialized db of facial encodings")
ap.add_argument("-d", "--detection-method", type=str, default="cnn",
    help="face detection model to use: either `hog` or `cnn`")
args = vars(ap.parse_args())

# grab the paths to the input images in our dataset
print("[INFO] quantifying faces...")
imagePaths = list(paths.list_images(args["dataset"]))

# initialize the list of known encodings and known names
knownEncodings = []
knownNames = []

# loop over the image paths
for (i, imagePath) in enumerate(imagePaths):
    # extract the person name from the image path
    print("[INFO] processing image {}/{}".format(i + 1,
        len(imagePaths)))
    name = imagePath.split(os.path.sep)[-2]

    # load the input image and convert it from RGB (OpenCV ordering)
    # to dlib ordering (RGB)
    image = cv2.imread(imagePath)
    rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # detect the (x, y)-coordinates of the bounding boxes
    # corresponding to each face in the input image
    boxes = face_recognition.face_locations(rgb,
        model=args["detection_method"])

    # compute the facial embedding for the face
    encodings = face_recognition.face_encodings(rgb, boxes)

    # loop over the encodings
    for encoding in encodings:
        # add each encoding + name to our set of known names and
        # encodings
        knownEncodings.append(encoding)
        knownNames.append(name)

# dump the facial encodings + names to disk
print("[INFO] serializing encodings...")
data = {"encodings": knownEncodings, "names": knownNames}
f = open(args["encodings"], "wb")
f.write(pickle.dumps(data))
f.close()
```

```
#Theodora Tataru
#Test from pyserachimages
#16 October 2020

# USAGE
# python pi_face_recognition.py --cascade haarcascade_frontalface_default.xml --encodings encodings.pickle

# import the necessary packages
from imutils.video import VideoStream
from imutils.video import FPS
import face_recognition
import argparse
import imutils
import pickle
import time
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-c", "--cascade", required=True,
    help = "path to where the face cascade resides")
ap.add_argument("-e", "--encodings", required=True,
    help="path to serialized db of facial encodings")
args = vars(ap.parse_args())

# load the known faces and embeddings along with OpenCV's Haar
# cascade for face detection
print("[INFO] loading encodings + face detector...")
data = pickle.loads(open(args["encodings"], "rb").read())
detector = cv2.CascadeClassifier(args["cascade"])

# initialize the video stream and allow the camera sensor to warm up
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
# vs = VideoStream(usePiCamera=True).start()
time.sleep(2.0)

# start the FPS counter
fps = FPS().start()

# loop over frames from the video file stream
while True:
    # grab the frame from the threaded video stream and resize it
    # to 500px (to speedup processing)
    frame = vs.read()
    frame = imutils.resize(frame, width=500)

    # convert the input frame from (1) BGR to grayscale (for face
    # detection) and (2) from BGR to RGB (for face recognition)
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

    # detect faces in the grayscale frame
    rects = detector.detectMultiScale(gray, scaleFactor=1.1,
        minNeighbors=5, minSize=(30, 30),
        flags=cv2.CASCADE_SCALE_IMAGE)

    # OpenCV returns bounding box coordinates in (x, y, w, h) order
    # but we need them in (top, right, bottom, left) order, so we
    # need to do a bit of reordering
    boxes = [(y, x + w, y + h, x) for (x, y, w, h) in rects]

    # compute the facial embeddings for each face bounding box
    encodings = face_recognition.face_encodings(rgb, boxes)
    names = []

    # loop over the facial embeddings
    for encoding in encodings:
        # attempt to match each face in the input image to our known
        # encodings
```

```
        matches = face_recognition.compare_faces(data["encodings"],
            encoding)
        name = "Unknown"

        # check to see if we have found a match
        if True in matches:
            # find the indexes of all matched faces then initialize a
            # dictionary to count the total number of times each face
            # was matched
            matchedIdxs = [i for (i, b) in enumerate(matches) if b]
            counts = {}

            # loop over the matched indexes and maintain a count for
            # each recognized face face
            for i in matchedIdxs:
                name = data["names"][i]
                counts[name] = counts.get(name, 0) + 1

            # determine the recognized face with the largest number
            # of votes (note: in the event of an unlikely tie Python
            # will select first entry in the dictionary)
            name = max(counts, key=counts.get)

        # update the list of names
        names.append(name)

    # loop over the recognized faces
    for ((top, right, bottom, left), name) in zip(boxes, names):
        # draw the predicted face name on the image
        cv2.rectangle(frame, (left, top), (right, bottom),
            (0, 255, 0), 2)
        y = top - 15 if top - 15 > 15 else top + 15
        cv2.putText(frame, name, (left, y), cv2.FONT_HERSHEY_SIMPLEX,
            0.75, (0, 255, 0), 2)

    # display the image to our screen
    cv2.imshow("Frame", frame)
    key = cv2.waitKey(1) & 0xFF

    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        break

    # update the FPS counter
    fps.update()

# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```

```
# import the necessary packages
from imutils import face_utils
import dlib
import cv2
# initialize dlib's face detector (HOG-based) and then create
# the facial landmark predictor
p = "shape_predictor_68_face_landmarks.dat"
detector = dlib.get_frontal_face_detector()
predictor = dlib.shape_predictor(p)
# load the input image and convert it to grayscale
image = cv2.imread("000.png")
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# detect faces in the grayscale image
rects = detector(gray, 0)
# loop over the face detections
for (i, rect) in enumerate(rects):
    # determine the facial landmarks for the face region, then
    # convert the facial landmark (x, y)-coordinates to a NumPy
    # array
    shape = predictor(gray, rect)
    shape = face_utils.shape_to_np(shape)
    # loop over the (x, y)-coordinates for the facial landmarks
    # and draw them on the image
    for (x, y) in shape:
        cv2.circle(image, (x, y), 2, (0, 255, 0), -1)
# show the output image with the face detections + facial landmarks
cv2.imshow("Output", image)
cv2.waitKey(0)
```

FACE RECOGNITION 2

This folder includes code testing face recognition using LBPH.

Figure 35 "Testing code"

DATASET

```
import cv2
import os
cam = cv2.VideoCapture(0)
cam.set(3, 640) # set video width
cam.set(4, 480) # set video height
face_detector = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')
# For each person, enter one numeric face id
face_id = input('\n enter user id end press <return> ==>  ')
print("\n [INFO] Initializing face capture. Look the camera and wait ...")
# Initialize individual sampling face count
count = 0
while(True):
    ret, img = cam.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_detector.detectMultiScale(gray, 1.3, 5)
    for (x,y,w,h) in faces:
        cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 2)
        count += 1
        # Save the captured image into the datasets folder
        cv2.imwrite("Dataset/User." + str(face_id) + '.' +
                    str(count) + ".jpg", gray[y:y+h,x:x+w])
        cv2.imshow('image', img)
    k = cv2.waitKey(100) & 0xff # Press 'ESC' for exiting video
    if k == 27:
        break
    elif count >= 500: # Take 30 face sample and stop video
         break
# Do a bit of cleanup
print("\n [INFO] Exiting Program and cleanup stuff")
cam.release()
cv2.destroyAllWindows()
```

TRAINING

```
import cv2
import numpy as np
from PIL import Image
import os
# Path for face image database
path = '../Dataset/Dataset'
recognizer = cv2.face.LBPHFaceRecognizer_create()
detector = cv2.CascadeClassifier("haarcascade_frontalface_default.xml");
# function to get the images and label data
def getImagesAndLabels(path):
    imagePaths = [os.path.join(path,f) for f in os.listdir(path)]
    faceSamples=[]
    ids = []
    for imagePath in imagePaths:
        PIL_img = Image.open(imagePath).convert('L') # grayscale
        img_numpy = np.array(PIL_img,'uint8')
        id = int(os.path.split(imagePath)[-1].split(".")[1])
        faces = detector.detectMultiScale(img_numpy)
        for (x,y,w,h) in faces:
            faceSamples.append(img_numpy[y:y+h,x:x+w])
            ids.append(id)
    return faceSamples,ids
print ("\n [INFO] Training faces. It will take a few seconds. Wait ...")
faces,ids = getImagesAndLabels(path)
recognizer.train(faces, np.array(ids))
# Save the model into trainer/trainer.yml
recognizer.write('trainer.yml')
# Print the numer of faces trained and end program
print("\n [INFO] {0} faces trained. Exiting Program".format(len(np.unique(ids))))
```

CAMERA

```
import numpy as np
import cv2
cap = cv2.VideoCapture(0)
while(True):
    ret, frame = cap.read()
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    cv2.imshow('frame', frame)
    cv2.imshow('gray', gray)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

FACE DETECTION

```
## Theodora Tataru
## Pixel the Echo Dot
## inspired by Marcelo Rovai from https://towardsdatascience.com/real-time-face-recognition-an-end-to-end-project-b738bb0f7348

import numpy as np
import cv2
faceCascade = cv2.CascadeClassifier('Cascades/haarcascade_frontalface_default.xml')
cap = cv2.VideoCapture(0)
cap.set(3,640) # set Width
cap.set(4,480) # set Height
while True:
    ret, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.2,
        minNeighbors=5,
        minSize=(20, 20)
    )
    for (x,y,w,h) in faces:
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img[y:y+h, x:x+w]
    cv2.imshow('video',img)
    k = cv2.waitKey(30) & 0xff
    if k == 27: # press 'ESC' to quit
        break
cap.release()
cv2.destroyAllWindows()
```

```
import cv2

import numpy as np
import os
recognizer = cv2.face.LBPHFaceRecognizer_create()
recognizer.read('Trainer/trainer.yml')
cascadePath = "Dataset/haarcascade_frontalface_default.xml"
faceCascade = cv2.CascadeClassifier(cascadePath);
font = cv2.FONT_HERSHEY_SIMPLEX
#iniciate id counter
id = 0
# names related to ids: example ==> Marcelo: id=1,  etc
names = ['None', 'Dora', 'Mike']
# Initialize and start realtime video capture
cam = cv2.VideoCapture(0)
cam.set(3, 640) # set video widht
cam.set(4, 480) # set video height
# Define min window size to be recognized as a face
minW = 0.1*cam.get(3)
minH = 0.1*cam.get(4)
while True:
    ret, img =cam.read()
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor = 1.2,
        minNeighbors = 5,
        minSize = (int(minW), int(minH)),
       )
    for(x,y,w,h) in faces:
        cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 2)
        id, confidence = recognizer.predict(gray[y:y+h,x:x+w])

        # If confidence is less them 100 ==> "0" : perfect match
        if (confidence < 100):
            id = names[id]
            confidence = "  {0}%".format(round(100 - confidence))
        else:
            id = "unknown"
            confidence = "  {0}%".format(round(100 - confidence))

        cv2.putText(
                    img,
                    str(id),
                    (x+5,y-5),
                    font,
                    1,
                    (255,255,255),
                    2
                   )
        cv2.putText(
                    img,
                    str(confidence),
                    (x+5,y+h-5),
                    font,
                    1,
                    (255,255,0),
                    1
                   )

    cv2.imshow('camera',img)
    k = cv2.waitKey(10) & 0xff # Press 'ESC' for exiting video
    if k == 27:
        break
# Do a bit of cleanup
print("\n [INFO] Exiting Program and cleanup stuff")
cam.release()
cv2.destroyAllWindows()
```

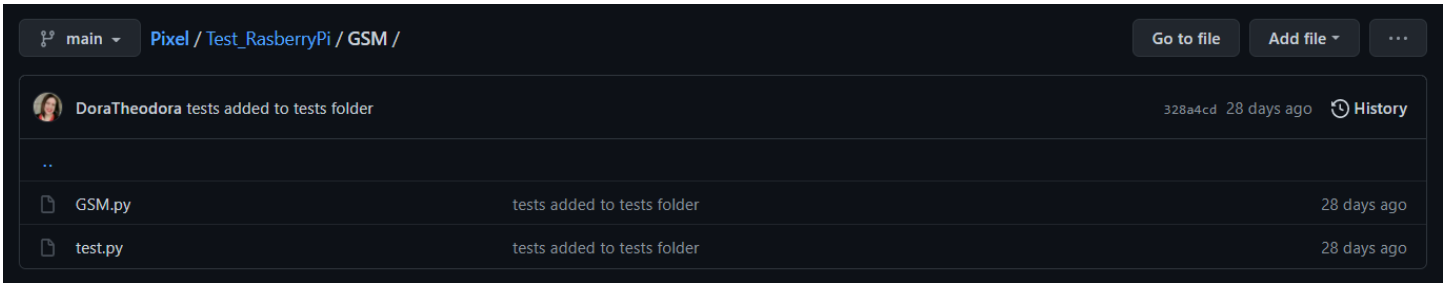


Figure 36 "Testing code"

GSM/GPS

```
#import gsmHat as GSM
import GSM
import time



def send_SMS(sender:str, receiver:str, message:str):
    # Send SMS
    gsm = GSM.GSMHat('/dev/ttyUSB0', 115200)
    Number = receiver
    Message = sender + " sent you the following message:\n\n" + message
    gsm.SMS_write(Number, Message)
    time.sleep(1)

def check_received_SMS():
    gsm = GSM.GSMHat('/dev/ttyUSB0', 115200)
    time.sleep(1)
    print(gsm.SMS_available())
    if gsm.SMS_available() > 0:
        number_messages = gsm.SMS_available()
        while number_messages != 0:
            new_sms = gsm.SMS_read()
            print("-----------------------------------------")
            print('Got new SMS from number %s' % new_sms.Sender)
            print('It was received at %s' % new_sms.Date)
            print('The message is: %s' % new_sms.Message)
            print("-----------------------------------------")
            number_messages -= 1
    time.sleep(1)

def get_location():
    from geopy.geocoders import Nominatim
    # Lets print some values
    gsm = GSM.GSMHat('/dev/ttyUSB0', 115200)
    time.sleep(1)
    GPSObj = gsm.GetActualGPS()
    time.sleep(1)
    GPS_Data = gsm.GPS_Data_List()
    for key in GPS_Data:
        print(key + ": " + GPS_Data[key])

    lat = GPS_Data['Latitude']
    lon = GPS_Data['Longitude']
    point = lat + "," + lon
    geolocator = Nominatim(user_agent="Pixel")
    location = geolocator.reverse(point)
    address = location.address
    print("The address: " + address)
    raw = location.raw
    print("Raw location: " + str(raw))
```

```
def call(number):
    gsm = GSM.GSMHat('/dev/ttyUSB0', 115200)
    print("Dialing...")
    print("Calling...")
    gsm.Call(number, 20)    # Or lets change the timeout to 20 seconds. This call hangs up automatically after 60 seconds
    time.sleep(10)          # Wait 10 seconds ...
    gsm.HangUp()



#send_SMS("Dora", "+353861645641", "Sending third text message!")
#check_received_SMS()
get_location()
#call("+353861645641")
```

PAN-TILT-HAT

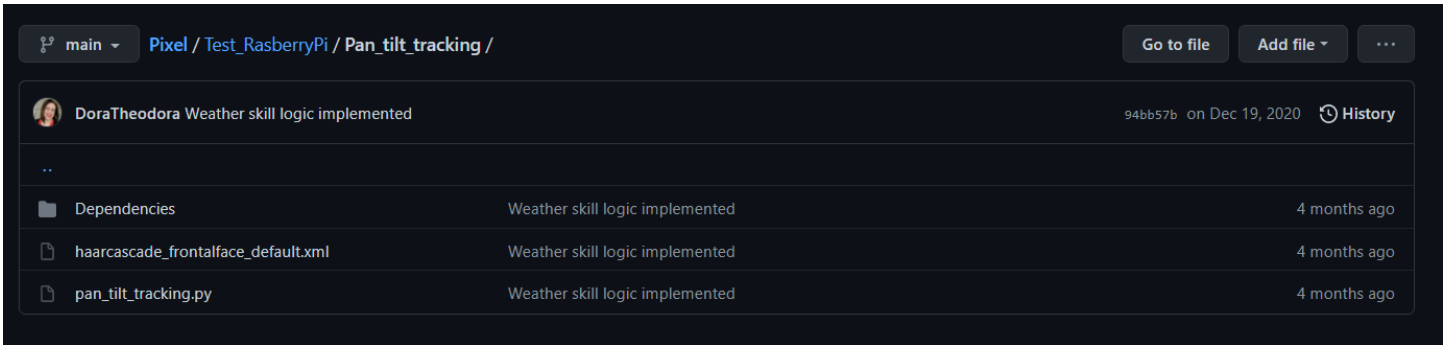


Figure 37 "Testing code"

OBJECT_CENTER

```
## Theodora Tataru
## Testing face tracking from https://www.pyimagesearch.com/2019/04/01/pan-tilt-face-tracking-with-a-raspberry-pi-and-opencv/
## By Adrian Rosebrock
## 19 October 2020

# import necessary packages
import imutils
import cv2

class ObjCenter:
    def __init__(self, haarPath):
        # load OpenCV's Haar cascade face detector
        self.detector = cv2.CascadeClassifier(haarPath)

    def update(self, frame, frameCenter):
        # convert the frame to grayscale
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # detect all faces in the input frame
        rects = self.detector.detectMultiScale(gray, scaleFactor=1.05,
            minNeighbors=9, minSize=(30, 30),
            flags=cv2.CASCADE_SCALE_IMAGE)

        # check to see if a face was found
        if len(rects) > 0:
            # extract the bounding box coordinates of the face and
            # use the coordinates to determine the center of the
            # face
            (x, y, w, h) = rects[0]
            faceX = int(x + (w / 2.0))
            faceY = int(y + (h / 2.0))

            # return the center (x, y)-coordinates of the face
            return ((faceX, faceY), rects[0])

        # otherwise no faces were found, so return the center of the
        # frame
        return (frameCenter, None)
```

PID CONTROLLER

```
## Theodora Tataru
## Testing face tracking from https://www.pyimagesearch.com/2019/04/01/pan-tilt-face-tracking-with-a-raspberry-pi-and-opencv/
## By Adrian Rosebrock
## 19 October 2020

# import necessary packages
import time

class PID:
    def __init__(self, kP=1, kI=0, kD=0):
        # initialize gains
        self.kP = kP
        self.kI = kI
        self.kD = kD

    def initialize(self):
        # intialize the current and previous time
        self.currTime = time.time()
        self.prevTime = self.currTime

        # initialize the previous error
        self.prevError = 0

        # initialize the term result variables
        self.cP = 0
        self.cI = 0
        self.cD = 0

    def update(self, error, sleep=0.1):
        # pause for a bit
        time.sleep(sleep)

        # grab the current time and calculate delta time
        self.currTime = time.time()
        deltaTime = self.currTime - self.prevTime

        # delta error
        deltaError = error - self.prevError

        # proportional term
        self.cP = error

        # integral term
        self.cI += error * deltaTime

        # derivative term and prevent divide by zero
        self.cD = (deltaError / deltaTime) if deltaTime > 0 else 0

        # save previous time and error for the next update
        self.prevTime = self.currTime
        self.prevError = error

        # sum the terms and return
        return sum([
            self.kP * self.cP,
            self.kI * self.cI,
            self.kD * self.cD])
```

```
## Theodora Tataru
## Testing face tracking from https://www.pyimagesearch.com/2019/04/01/pan-tilt-face-tracking-with-a-raspberry-pi-and-opencv/
## By Adrian Rosebrock
## 19 October 2020

# USAGE
# python pan_tilt_tracking.py --cascade haarcascade_frontalface_default.xml

# import necessary packages
from multiprocessing import Manager
from multiprocessing import Process
from imutils.video import VideoStream
from Dependencies.objcenter import ObjCenter
from Dependencies.pid import PID
import pantilthat as pth
import argparse
import signal
import time
import sys
import cv2

# define the range for the motors
servoRange = (-90, 90)

# function to handle keyboard interrupt
def signal_handler(sig, frame):
    # print a status message
    print("[INFO] You pressed `ctrl + c`! Exiting...")

    # disable the servos
    pth.servo_enable(1, False)
    pth.servo_enable(2, False)

    # exit
    sys.exit()

def obj_center(args, objX, objY, centerX, centerY):
    # signal trap to handle keyboard interrupt
    signal.signal(signal.SIGINT, signal_handler)

    # start the video stream and wait for the camera to warm up
    vs = VideoStream(usePiCamera=True).start()
    time.sleep(2.0)

    # initialize the object center finder
    obj = ObjCenter(args["cascade"])

    # loop indefinitely
    while True:
        # grab the frame from the threaded video stream
        frame = vs.read()

        # calculate the center of the frame as this is where we will
        # try to keep the object
        (H, W) = frame.shape[:2]
        centerX.value = W // 2
        centerY.value = H // 2

        # find the object's location
        objectLoc = obj.update(frame, (centerX.value, centerY.value))
        ((objX.value, objY.value), rect) = objectLoc

        # extract the bounding box and draw it
        if rect is not None:
            (x, y, w, h) = rect
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0),
                2)

        # display the frame to the screen
```

```
        cv2.imshow("Pan-Tilt Face Tracking", frame)
        cv2.waitKey(1)

def pid_process(output, p, i, d, objCoord, centerCoord):
    # signal trap to handle keyboard interrupt
    signal.signal(signal.SIGINT, signal_handler)

    # create a PID and initialize it
    p = PID(p.value, i.value, d.value)
    p.initialize()

    # loop indefinitely
    while True:
        # calculate the error
        error = centerCoord.value - objCoord.value

        # update the value
        output.value = p.update(error)

def in_range(val, start, end):
    # determine the input vale is in the supplied range
    return (val >= start and val <= end)

def set_servos(pan, tlt):
    # signal trap to handle keyboard interrupt
    signal.signal(signal.SIGINT, signal_handler)

    # loop indefinitely
    while True:
        panAngle = tlt.value
        tltAngle = pan.value

        # if the pan angle is within the range, pan
        if in_range(panAngle, servoRange[0], servoRange[1]):
            pth.pan(panAngle)

        # if the tilt angle is within the range, tilt
        if in_range(tltAngle, servoRange[0], servoRange[1]):
            pth.tilt(tltAngle)

# check to see if this is the main body of execution
if __name__ == "__main__":
    # construct the argument parser and parse the arguments
    ap = argparse.ArgumentParser()
    ap.add_argument("-c", "--cascade", type=str, required=True,
        help="path to input Haar cascade for face detection")
    args = vars(ap.parse_args())

    # start a manager for managing process-safe variables
    with Manager() as manager:
        # enable the servos
        pth.servo_enable(1, True)
        pth.servo_enable(2, True)

        # set integer values for the object center (x, y)-coordinates
        centerX = manager.Value("i", 0)
        centerY = manager.Value("i", 0)

        # set integer values for the object's (x, y)-coordinates
        objX = manager.Value("i", 0)
        objY = manager.Value("i", 0)

        # pan and tilt values will be managed by independed PIDs
        pan = manager.Value("i", 0)
        tlt = manager.Value("i", 0)

        # set PID values for panning
        panP = manager.Value("f", 0.09)
        panI = manager.Value("f", 0.08)
        panD = manager.Value("f", 0.002)

        # set PID values for tilting
        tiltP = manager.Value("f", 0.11)
```

```
        tiltI = manager.Value("f", 0.10)
        tiltD = manager.Value("f", 0.002)

        # we have 4 independent processes
        # 1. objectCenter  - finds/localizes the object
        # 2. panning       - PID control loop determines panning angle
        # 3. tilting       - PID control loop determines tilting angle
        # 4. setServos     - drives the servos to proper angles based
        #                    on PID feedback to keep object in center
        processObjectCenter = Process(target=obj_center,
            args=(args, objX, objY, centerX, centerY))
        processPanning = Process(target=pid_process,
            args=(pan, panP, panI, panD, objX, centerX))
        processTilting = Process(target=pid_process,
            args=(tlt, tiltP, tiltI, tiltD, objY, centerY))
        processSetServos = Process(target=set_servos, args=(pan, tlt))

        # start all 4 processes
        processObjectCenter.start()
        processPanning.start()
        processTilting.start()
        processSetServos.start()

        # join all 4 processes
        processObjectCenter.join()
        processPanning.join()
        processTilting.join()
        processSetServos.join()

        # disable the servos
        pth.servo_enable(1, False)
        pth.servo_enable(2, False)
```

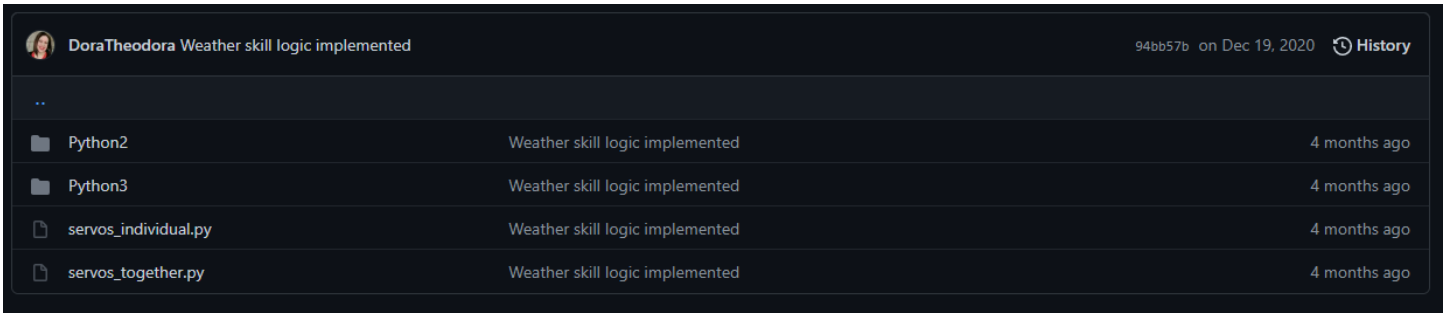


Figure 38 "Testing code"

PCA9685

```
#!/usr/bin/python

import time
import math
import smbus

# ============================================================================
# Raspi PCA9685 16-Channel PWM Servo Driver
# ============================================================================

class PCA9685:

  # Registers/etc.
  __SUBADR1            = 0x02
  __SUBADR2            = 0x03
  __SUBADR3            = 0x04
  __MODE1              = 0x00
  __MODE2              = 0x01
  __PRESCALE           = 0xFE
  __LED0_ON_L          = 0x06
  __LED0_ON_H          = 0x07
  __LED0_OFF_L         = 0x08
  __LED0_OFF_H         = 0x09
  __ALLLED_ON_L        = 0xFA
  __ALLLED_ON_H        = 0xFB
  __ALLLED_OFF_L       = 0xFC
  __ALLLED_OFF_H       = 0xFD


  def __init__(self, address=0x40, debug=False):
    self.bus = smbus.SMBus(1)
    self.address = address
    self.debug = debug
    if (self.debug):
      print("Reseting PCA9685")
    self.write(self.__MODE1, 0x00)

  def write(self, reg, value):
    "Writes an 8-bit value to the specified register/address"
    self.bus.write_byte_data(self.address, reg, value)
    if (self.debug):
      print(("I2C: Write 0x%02X to register 0x%02X" % (value, reg)))

  def read(self, reg):
    "Read an unsigned byte from the I2C device"
    result = self.bus.read_byte_data(self.address, reg)
    if (self.debug):
      print(("I2C: Device 0x%02X returned 0x%02X from reg 0x%02X" % (self.address, result & 0xFF, reg)))
    return result

  def setPWMFreq(self, freq):
```

```
    "Sets the PWM frequency"
    prescaleval = 25000000.0    # 25MHz
    prescaleval /= 4096.0       # 12-bit
    prescaleval /= float(freq)
    prescaleval -= 1.0
    if (self.debug):
      print(("Setting PWM frequency to %d Hz" % freq))
      print(("Estimated pre-scale: %d" % prescaleval))
    prescale = math.floor(prescaleval + 0.5)
    if (self.debug):
      print(("Final pre-scale: %d" % prescale))

    oldmode = self.read(self.__MODE1);
    newmode = (oldmode & 0x7F) | 0x10        # sleep
    self.write(self.__MODE1, newmode)        # go to sleep
    self.write(self.__PRESCALE, int(math.floor(prescale)))
    self.write(self.__MODE1, oldmode)
    time.sleep(0.005)
    self.write(self.__MODE1, oldmode | 0x80)
    self.write(self.__MODE2, 0x04)

  def setPWM(self, channel, on, off):
    "Sets a single PWM channel"
    self.write(self.__LED0_ON_L+4*channel, on & 0xFF)
    self.write(self.__LED0_ON_H+4*channel, on >> 8)
    self.write(self.__LED0_OFF_L+4*channel, off & 0xFF)
    self.write(self.__LED0_OFF_H+4*channel, off >> 8)
    if (self.debug):
      print(("channel: %d  LED_ON: %d LED_OFF: %d" % (channel,on,off)))

  def setServoPulse(self, channel, pulse):
    "Sets the Servo Pulse,The PWM frequency must be 50HZ"
    pulse = pulse*4096/20000        #PWM frequency is 50HZ,the period is 20000us
    self.setPWM(channel, 0, int(pulse))

  def setRotationAngle(self, channel, Angle):
    if(Angle >= 0 and Angle <= 180):
        temp = Angle * (2000 / 180) + 501
        self.setServoPulse(channel, temp)
    else:
        print("Angle out of range")

  def start_PCA9685(self):
    self.write(self.__MODE2, 0x04)
    #Just restore the stopped state that should be set for exit_PCA9685

  def exit_PCA9685(self):
    self.write(self.__MODE2, 0x00)#Please use initialization or __MODE2 =0x04
```

```
#Theodora Tataru
#C00231174

#This class implements the communication between the chip set

import time
import math
import RPi.GPIO as GPIO
from PCA9685 import PCA9685


class Directions(object):
    def __init__(self):
        print("starting...")
        self.pwm = PCA9685() #communication with the chipset
        self.PWM_FREQUENCY = 50 #max 1500
        self.pwm.setPWMFreq(self.PWM_FREQUENCY)

        self.INIT_HORIZONTAL_ANGLE = 70
        self.INIT_VERTICAL_ANGLE = 70
        self.move(horizontal = self.INIT_HORIZONTAL_ANGLE, vertical=self.INIT_VERTICAL_ANGLE)

    def __del__(self):
        """ Class that will run when the object is destroyed """
        print("done...")
        self.pwm.exit_PCA9685()
        GPIO.cleanup()

    def move(self, horizontal, vertical):
        self.move_horizontal(horizontal_angle = horizontal)
        self.move_vertical(vertical_angle= vertical)

    def move_horizontal(self, horizontal_angle):
        self.pwm.setRotationAngle(0, horizontal_angle)

    def move_vertical(self, vertical_angle):
        self.pwm.setRotationAngle(1, vertical_angle)

    ## Range tests
    def horizontal_range_test(self):
        """ Testing the X axis range """
        for angle in range(10,170,1):
            print(("Moving Horizontally...." + str(angle)))
            self.move_horizontal(horizontal_angle=angle)
            time.sleep(0.1)
        for angle in range(170,10,-1):
            print(("Moving Horizontally...." + str(angle)))
            self.move_horizontal(horizontal_angle=angle)
            time.sleep(0.1)

    def vertical_range_test(self):
        """ Testing the Y axis range """
        for angle in range(10,170,1):
            print(("Moving Vetically..." + str(angle)))
            self.move_vertical(vertical_angle=angle)
            time.sleep(0.1)
        for angle in range(170,10,-1):
            print(("Moving Vetically..." + str(angle)))
            self.move_vertical(vertical_angle=angle)
            time.sleep(0.1)

    ## Tests
    def input_horizontal_test(self):
        while True:
            x_axis = input("Horizontal Angle: ")
            angle = int(x_axis)
            self.move_horizontal(horizontal_angle=angle)

    def input_vertical_test(self):
        while True:
```

```
            y_axis = input("Vertical Angle: ")
            angle = int(y_axis)
            self.move_vertical(vertical_angle=angle)


    def circle_test(self, repetitions=1):

        # These values are base on Hardware observations where they are stable.
        max_range = 100
        min_range = 30
        period = 0.1
        increments = 10

        for num in range(repetitions):
            for angle in range(0,359,increments):
                h_angle = int((((math.sin(math.radians(angle)) + 1.0) / 2.0) * (max_range - min_range)) + min_range)
                v_angle = int((((math.cos(math.radians(angle)) + 1.0) / 2.0) * (max_range - min_range)) + min_range)

                print(("Moving Yaw="+str(v_angle)))
                print(("Moving Pitch=" + str(h_angle)))

                self.move_vertical(h_angle)
                self.move_horizontal(v_angle)
                time.sleep(period)

            for angle in range(0,359,-increments):
                h_angle = int((((math.sin(math.radians(angle)) + 1.0) / 2.0) * (max_range - min_range)) + min_range)
                v_angle = int((((math.cos(math.radians(angle)) + 1.0) / 2.0) * (max_range - min_range)) + min_range)

                print(("Moving Yaw="+str(v_angle)))
                print(("Moving Pitch=" + str(h_angle)))

                self.move_vertical(v_angle)
                self.move_horizontal(h_angle)
                time.sleep(period)

def horizontal_test():
    this_object = Directions()
    this_object.horizontal_range_test()

def vertical_test():
    this_object = Directions()
    this_object.vertical_range_test()

def input_horizontal_test():
    this_object = Directions()
    this_object.input_horizontal_test()

def input_vertical_test():
    this_object = Directions()
    this_object.input_vertical_test()

def input_test():
    this_object = Directions()
    while True:
        x = input("x angle: ")
        y = input("y angle: ")
        this_object.move(int(x), int(y))



if __name__ == "__main__":
    #horizontal_test()
    #vertical_test()
    #input_horizontal_test()
    #input_vertical_test()
    input_test()
```

INDIVIDUAL SERVOS

```
## Theodora Tataru
## Pixel the echodot
## Servos Script

import pantilthat
from time import sleep
import RPi.GPIO as GPIO

def setServoAngle(servo, angle):
    ## vertical
    if servo == 'pan':
        pantilthat.servo_two(angle)
    ## horizontal
    if servo == 'tilt':
        pantilthat.servo_one(angle)
    sleep(0.3)


## this script will move one servo at the time
if __name__ == '__main__':
    import sys
    servo = str(sys.argv[1])
    angle = int(sys.argv[2])
    setServoAngle(servo, angle)
    #GPIO.cleanup()
```

SERVOS TOGETHER

```
## Theodora Tataru
## Pixel the echodot
## Servos Script

import pantilthat
from time import sleep
#import RPi.GPIO as GPIO

def setServoAngle(servo, angle):
    ## vertical
    if servo == 'pan':
        pantilthat.servo_two(angle)
    ## horizontal
    if servo == 'tilt':
        pantilthat.servo_one(angle)
    sleep(0.3)


## this script will move both servos
if __name__ == '__main__':
    import sys
    if len(sys.argv) == 1:
        setServoAngle('pan', 0)
        setServoAngle('tilt', 0)
    else:
        setServoAngle('pan', int(sys.argv[1]))
        setServoAngle('tilt', int(sys.argv[2]))
```

Figure 39 "Testing code"

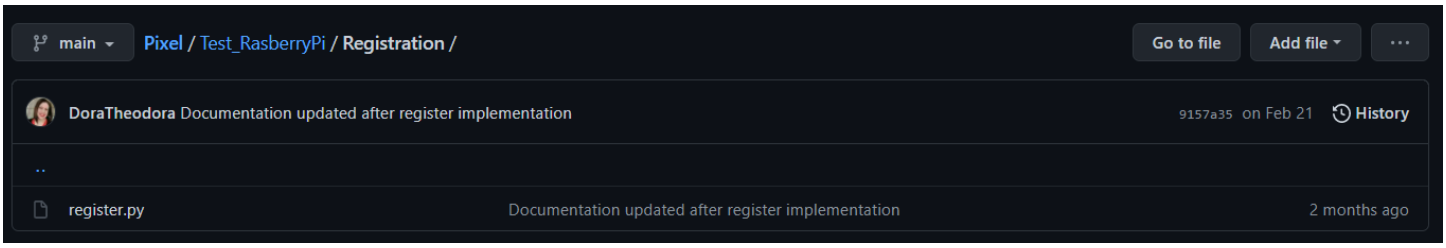


Figure 40 "Testing code"

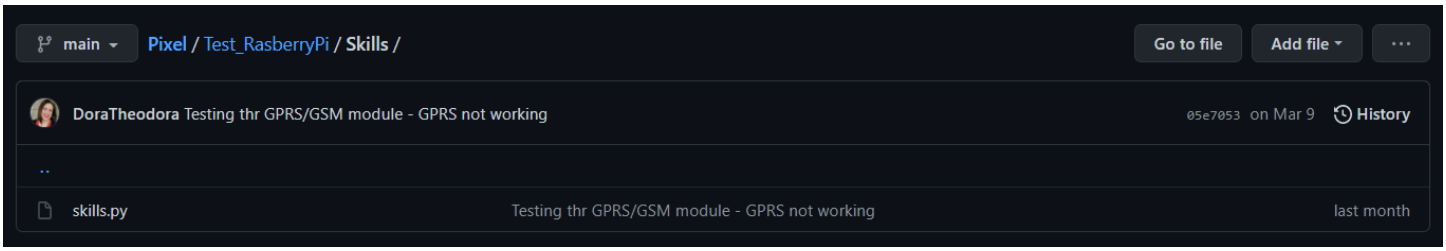


Figure 41 "Testing code"

CONCLUSION

This document provides the installation guidelines and describes the code developed for Pixel Virtual Assistant, a smart speaker with face recognition.

The report focuses on the assembly of the hardware requested for Pixel Virtual Assistant and the software produced for the project.

The document details the produced code and its documentation and provides setup instructions and a video tutorial on how to use the device once it is installed.

BIBLIOGRAPHY

Amazon, 2021. *IBest WM8960 Hi-Fi Sound Card HAT Audio Module for Raspberry Pi Supports Stereo Encoding/Decoding Hi-Fi Playing/Recording directly Drive Speakers to Play Music,I2S I2C Interface.* [Online]
Available at: https://www.amazon.co.uk/gp/product/B07R8M3XFQ/ref=ppx_yo_dt_b_asin_title_o09_s00?ie=UTF8&psc=1
[Accessed 29 March 2021].

Amazon, 2021. *IBest WM8960 Hi-Fi Sound Card HAT Audio Module for Raspberry Pi Supports Stereo Encoding/Decoding Hi-Fi Playing/Recording directly Drive Speakers to Play Music,I2S I2C Interface.* [Online]
Available at:
https://www.amazon.co.uk/gp/product/B07R8M3XFQ/ref=ppx_yo_dt_b_asin_title_o09_s00?ie=UTF8&psc=1https://www.amazon.co.uk/gp/product/B07R8M3XFQ/ref=ppx_yo_dt_b_asin_title_o09_s00?ie=UTF8&psc=1
[Accessed 14 April 2021].

Amazon, 2021. *Longruner Raspberry Pi 4 Model B+ Camera Module Automatic IR-Cut Switching Day/Night Vision Video Module Adjustable Focus 5MP OV5647 Sensor 1080p HD Webcam for Raspberry Pi B/B+/A+/4/3/2/1 LC26.* [Online]
Available at: https://www.amazon.co.uk/gp/product/B07R4JH2ZV/ref=ppx_yo_dt_b_asin_image_o05_s00?ie=UTF8&psc=1
[Accessed 14 April 2021].

Amazon, 2021. *Makerfire Raspberry Pi Fan DC Brushless Cooling Fan 3.3V 5V Heatsink Cooler Separating Terminal for Raspberry Pi 4B / 3B+ / 3B/ Zero/Zero W or Other Robot Project (2 Pack).* [Online]
Available at: https://www.amazon.co.uk/gp/product/B071JN6HKM/ref=ppx_yo_dt_b_asin_title_o02_s00?ie=UTF8&psc=1
[Accessed 22 March 2021].

Amazon, 2021. *Raspberry Pi 10 Inch Touch Screen - SunFounder 10.1" HDMI 1280x800 IPS LCD Touchscreen for RPi 400 4 Model B 3 Model B+ 3B 2B LattePanda Beagle Bone.* [Online]
Available at: https://www.amazon.com/Raspberry-Inch-Touch-Screen-Touchscreen/dp/B0776VNW9C
[Accessed 15 April 2021].

Amazon, 2021. *Sign Materials Direct 3mm Two Way/See-Thru Mirror Acrylic Sheet 15 SIZES TO CHOOSE (297mm x 210mm / A4).* [Online]
Available at: https://www.amazon.co.uk/gp/product/B01N8TY2TO/ref=ppx_yo_dt_b_asin_title_o06_s00?ie=UTF8&psc=1
[Accessed 15 April 2021].

Amazon, 2021. *USB Microphone, Gyvazla Omnidirectional Condenser Lavalier Lapel Clip on Mic for Computer, Laptop, Podcast, Interviews, Network singing, Skype, MSN, Audio Video Recording [Plug and Play].* [Online]
Available at: https://www.amazon.co.uk/gp/product/B071171DBP/ref=ppx_yo_dt_b_asin_title_o01_s00?ie=UTF8&psc=1
[Accessed 18 March 2021].

Amazon, 2021. *Waveshare Raspberry Pi GSM/GPRS/GNSS Bluetooth HAT Expansion Board GPS Module with Low-Power Consumption Based on SIM868 Compatible With Raspberry Pi 2B 3B 4B Zero Zero W.* [Online]
Available at: https://www.amazon.co.uk/gp/product/B076CPX4NN/ref=ppx_yo_dt_b_asin_title_o01_s00?ie=UTF8&psc=1
[Accessed 18 March 2021].

CMake, 2021. *Build with CMake. Build with Confidence..* [Online]
Available at: https://cmake.org
[Accessed 20 October 2020].

Electronics, D.-K., 2019. *Raspberry Pi 4 Model B.* [Online]
Available at: https://www.digikey.ie/en/product-highlight/r/raspberry-pi/raspberry-pi-4-model-b?utm_adgroup=General&utm_source=google&utm_medium=cpc&utm_campaign=Dynamic%20Search_EN_RLSA_Product_Purchaser&utm_term=&productid=&gclid=CjwKCAjw5p_8BRBUEiwAPpJO6_R0E1j9VUEsm
[Accessed 14 April 2021].

Ltd, T. C., 2020. *FREQUENTY ASKED QUESTIONS WHEN DESIGNING-IN A GNSS ANTENNA.* [Online]
Available at: https://www.te.com/content/dam/te-com/documents/about-te/marketing/global/select-campaign/gnss-antennas-

faqs.pdf
[Accessed 17 March 2021].

Pi, R., 2021. *Installing operating system images.* [Online]
Available at: https://www.raspberrypi.org/documentation/installation/installing-images/
[Accessed 08 January 2021].

Pi, R., 2021. *SD cards.* [Online]
Available at: https://www.raspberrypi.org/documentation/installation/sd-cards.md
[Accessed 21 January 2021].

Rosebrock, A., 2018. *Install OpenCV 4 on your Raspberry Pi.* [Online]
Available at: https://www.pyimagesearch.com/2018/09/26/install-opencv-4-on-your-raspberry-pi/
[Accessed 20 October 2020].

Serasinghe, S., 2020. *Setting up your Raspberry PI 4 - Headless mode.* [Online]
Available at: https://sahansera.dev/setting-up-raspberry-pi-4-headless-mode/
[Accessed 14 April 2021].

Sphinx, 2021. *Installing Sphinx*. [Online]
Available at: http://www.sphinx-doc.org
[Accessed 20 April 2021].