**RP2 -L3 22-2-2021**

1. **Namespace, modules and packages**
2. **Oled sturen via I2C en SPI**
   1. **Luma library ( SSD1306 en SH1106)**
   2. **Adafruit ( enkel SSD1306)**

**1/ Namespace, modules, packages en subpackages**

**Namespace** : You can think of a **namespace** as a dictionary in which the keys are the object names and the values are the objects themselves, check with dir()

**Doel van een module :**

Modules worden gebruikt om grote programma’s in korte, **beheersbare** delen op te splitsen. Tevens kun je met modules stukken code **hergebruiken**.

Het is beter om veelgebruikte stukken code als module op te slaan en daar waar nodig te importeren, dan om de stukken code overal waar nodig over te kopiëren.

Nu kun je deze module importeren en gebruiken vanuit een ander Python script. Doe dit als volgt:

**import module\_naam**

Hiermee importeer je alle functies uit de module en gebruik je ze door module\_naam.functie\_naam aan te roepen, in je code refereer je naar functies met module\_naam.fct\_naam()

from module\_naam import fct\_naam\_1, fctnaam\_2 , var1 , var2

Gebruik dit als je slechts specifieke functionaliteiten wilt gebruiken, in je code moet je nu de module\_naam niet gebruiken, fct\_naam1() of var1 is voldoende.

**from module\_naam import \***

Met deze import vorm worden alle functionaliteiten geïmporteerd, behalve zij die beginnen met een underscore (\_), in je code moet je nu de module\_naam niet gebruiken, fct\_naam1() of var1 is voldoende.

**from example\_module import fct\_naam1 as f1**

Wanneer een naam van een te importeren functie uit een module conflicterend is met een al gebruikte naam, dan kun je een te importeren functienaam wijzigen.

**Doel van een package:**

**Een Python package is een verzameling (folder) van modules en eventueel sub-packages**. Door in modules en sub-packages specifieke gerelateerde functionaliteiten afzonderlijk op te slaan en te groeperen in een package ontstaat een gestructureerde verzameling code.

Een package bestaat altijd uit een mappenstructuur en een bestand **\_\_init\_\_.py**. Het \_\_init\_\_.py bestand is nodig zodat Python herkent dat het een package is. Een subpackage moet hierdoor ook altijd een \_\_init\_\_.py bestand bevatten. Het \_\_init\_\_.py bestand mag leeg zijn, maar bevat vaak de code die gebruikt wordt voor het initialiseren van de package.

**Oef : maak en test onderstaande package**

**package\_test**

**|**

**|- \_\_init\_\_.py**

**|**

**|**

**|**

**|- mod1.py**

**| |------leeftijd=50**

**| |------ def hellomod1**

**|**

**|- mod2.py**

**| |------hoogte=26**

**| |------ def hellomod2**

**|**

**|**

**|- -------- sub\_pack1**

**| |**

**| |\_\_init\_\_.py**

**| |**

**| |---- mod3.py**

**| | |**

**| | |**

**| | |------breedte=18**

**| | |------ def hellomod3**

**| |**

**| |----mod4.py**

**| |------aantal=7**

**| |------ def hellomod4**

**|**

**|----------- sub\_pack2**

**|**

**|**

**|\_init\_\_.py**

**|**

**|**

**|**

**|- mod5.py**

**|**

**|-----graad=6**

**|----def hellomod5**

**Oefeningen :**

1. **Maar de bovenstaande package en voeg print functies toe bovenaan elke \_\_init\_\_, elke module en elke functie zodat je kan volgen wat er gebeurd.**
2. **Hoe krijg je toegang tot de variabele graad? Geef enkele verschillende opties.**
3. **Hoe kan je in je main script de functie hellomod5 oproepen? Geef enkele verschillende opties.**
4. **Wat print de built-in functie \_\_name\_\_ in elke module? Is er een verschil waar je je programma start?**

**2/ Oled-display sturen via I2C en SPI**

Driver library = luma.oled

Zie <https://luma-oled.readthedocs.io/en/latest/intro.html>

Python 3 library interfacing OLED matrix displays with the SSD1306, SSD1309, SSD1322, SSD1325, SSD1327, SSD1331, SSD1351, SH1106 or WS0010 driver using I2C/SPI/Parallel on the Raspberry Pi and other linux-based single-board computers - it provides a [Pillow](https://pillow.readthedocs.io/)-compatible drawing canvas, and other functionality to support:

https://luma-oled.readthedocs.io/en/latest/software.html

$ sudo apt-get update

$ sudo apt-get install python3 python3-pip python3-pil libjpeg-dev zlib1g-dev libfreetype6-dev liblcms2-dev libopenjp2-7 libtiff5 -y

$ sudo -H pip3 install luma.oled

**Pillow**

https://pillow.readthedocs.io/en/stable/

Pillow is the friendly PIL fork

What is a fork in coding?

In computer **programming**, a **fork** is when developers take the source code for an existing project and use it to create new software based on the original code

The Python Imaging Library adds image processing capabilities to your Python interpreter.This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities.The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

Pillow installeren op RP4:

$ sudo apt-get install python3-pil

Of $ sudo apt-get install python3-imaging

Indien hardware problemen probeer :

$ sudo usermod -a -G spi,gpio,i2c pi

Opm: Op Raspberry Pi problemen om pillow image open te bekijken… vergelijk PC en RP.

In linux default viewer xv ontbreekt, install xli en link xv naar xli indien nodig

**sudo apt-get install xli**

**cd /usr/local/bin**

**sudo ln -s /usr/bin/xli xv**

**Eenvoudige tests met pillow :**

**Test1:**

**from PIL import Image #, ImageDraw # PIL manage packages pillow-pil**

**#Create image object**

**im = Image.open("pianoles.jpg")**

**# show image object**

**im.show()**

**Test2, gebruik label tkinter om de foto te tonen:**

**import tkinter as tk**

**from PIL import ImageTk, Image**

**import time**

**#This creates the main window of an application**

**window = tk.Tk()**

**window.title("Join")**

**window.geometry("1220x800")**

**window.configure(background='yellow')**

**path = "pianoles.jpg"**

**#Creates a Tkinter-compatible photo image, which can be used everywhere Tkinter expects an image object.**

**image =Image.open(path)**

**img = ImageTk.PhotoImage(image)**

**#The Label widget is a standard Tkinter widget used to display a text or image on the screen.**

**my\_foto\_lbl = tk.Label(window, image = img)**

**my\_foto\_lbl.place(x=5,y=5)**

**Test 3 gebruik canvas widget om foto te tonen**

**from tkinter import \***

**from PIL import ImageTk,Image**

**window = Tk()**

**window.title("Join")**

**window.geometry("1240x825")**

**window.configure(background='yellow')**

**canvas = Canvas(window, width = 1220, height = 805)**

**canvas.pack()**

**img\_pillow=Image.open("pianoles.jpg")**

**print(img\_pillow.size)**

**img\_tk = ImageTk.PhotoImage(img\_pillow)**

**canvas.create\_image(10, 10, anchor=NW, image=img\_tk )**

**window.mainloop()**

**Oefeningen (zoek op https://pillow.readthedocs.io/en/latest/handbook/tutorial.html)**

**1/ Hoe verklein je een foto met pillow?**

**2/ Hoe crop je een foto met pillow?**

**3/ Hoe spiegel je een foto met pillow?**

**4/ Hoe draai je een foto 45 en/of 90 graden?**

**Test OLED met RP:**

**from luma.core.interface.serial import i2c, spi, pcf8574**

**from luma.core.interface.parallel import bitbang\_6800**

**from luma.core.render import canvas**

**from luma.oled.device import ssd1306, ssd1309, ssd1325, ssd1331, sh1106, ws0010**

**import RPi.GPIO as GPIO**

**GPIO.setwarnings(False)**

**serial\_i2c = i2c(port=1, address=0x3C)**

**serial\_spi = spi(device=0, port=0)**

**# substitute ssd1331(...) or sh1106(...) below if using that device**

**device\_i2c = sh1106(serial\_i2c)**

**device\_spi = ssd1306(serial\_spi)**

**with canvas(device\_spi) as draw:**

**draw.rectangle(device\_spi.bounding\_box, outline="white", fill="black")**

**draw.text((10, 40), ">Hello World SPI", fill="white")**

**with canvas(device\_i2c) as draw:**

**draw.rectangle(device\_i2c.bounding\_box, outline="white", fill="black")**

**draw.text((10, 40), ">Hello World I2C", fill="white")**

**What are modules in Python?**

Modules refer to a file containing Python statements and definitions.

A file containing Python code, for example: example.py, is called a module, and its module name would be example.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Let us create a module. Type the following and save it as example.py.

# Python Module example

def add(a, b):

"""This program adds two

numbers and return the result"""

result = a + b

return result

Here, we have defined a [function](https://www.programiz.com/python-programming/function) add() inside a module named example. The function takes in two numbers and returns their sum.

**How to import modules in Python?**

We can import the definitions inside a module to another module or the interactive interpreter in Python.

We use the import keyword to do this. To import our previously defined module example, we type the following in the Python prompt.

>>> import example

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name example there.

Using the module name we can access the function using the dot . operator. For example:

>>> example.add(4,5.5)

9.5

Python has tons of standard modules. You can check out the full list of [Python standard modules](http://docs.python.org/3/py-modindex.html) and their use cases. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules.

There are various ways to import modules. They are listed below..

**Python import statement**

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

# import statement example

# to import standard module math

import math

print("The value of pi is", math.pi)

When you run the program, the output will be:

The value of pi is 3.141592653589793

**Import with renaming**

We can import a module by renaming it as follows:

# import module by renaming it

import math as m

print("The value of pi is", m.pi)

We have renamed the math module as m. This can save us typing time in some cases.

Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

**Python from...import statement**

We can import specific names from a module without importing the module as a whole. Here is an example.

# import only pi from math module

from math import pi

print("The value of pi is", pi)

Here, we imported only the pi attribute from the math module.

In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

>>> from math import pi, e

>>> pi

3.141592653589793

>>> e

2.718281828459045

**Import all names**

We can import all names(definitions) from a module using the following construct:

# import all names from the standard module math

from math import \*

print("The value of pi is", pi)

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).

Importing everything with the asterisk (\*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

**Python Module Search Path**

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if built-in module not found), Python looks into a list of directories defined in sys.path. The search is in this order.

* The current directory.
* PYTHONPATH (an environment variable with a list of directories).
* The installation-dependent default directory.

>>> import sys

>>> sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']

We can add and modify this list to add our own path.

**Reloading a module**

The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named my\_module.

# This module shows the effect of

# multiple imports and reload

print("This code got executed")

Now we see the effect of multiple imports.

>>> import my\_module

This code got executed

>>> import my\_module

>>> import my\_module

We can see that our code got executed only once. This goes to say that our module was imported only once.

Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.

Python provides a more efficient way of doing this. We can use the reload() function inside the imp module to reload a module. We can do it in the following ways:

>>> import imp

>>> import my\_module

This code got executed

>>> import my\_module

>>> imp.reload(my\_module)

This code got executed

<module 'my\_module' from '.\\my\_module.py'>

**The dir() built-in function**

We can use the dir() function to find out names that are defined inside a module.

For example, we have defined a function add() in the module example that we had in the beginning.

We can use dir in example module in the following way:

>>> dir(example)

['\_\_builtins\_\_',

'\_\_cached\_\_',

'\_\_doc\_\_',

'\_\_file\_\_',

'\_\_initializing\_\_',

'\_\_loader\_\_',

'\_\_name\_\_',

'\_\_package\_\_',

'add']

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).

For example, the \_\_name\_\_ attribute contains the name of the module.

>>> import example

>>> example.\_\_name\_\_

'example'

All the names defined in our current namespace can be found out using the dir() function without any arguments.

>>> a = 1

>>> b = "hello"

>>> import math

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'b', 'math', 'pyscripter']

**modules in python,**

**how to divide code in modules with python?**

https://docs.python-guide.org/writing/structure/

Once modu.py is found, the Python interpreter will execute the module in an isolated scope.

Any top-level statement in modu.py will be executed, including other imports if any.

Function and class definitions are stored in the module’s dictionary.

Then, the module’s variables, functions, and classes will be available to the caller through the module’s namespace,

a central concept in programming that is particularly helpful and powerful in Python.

In many languages, an include file directive is used by the preprocessor to take all code found in the file

and ‘copy’ it into the caller’s code.

It is different in Python: the included code is isolated in a module namespace,

which means that you generally don’t have to worry that the included code could have unwanted effects,

e.g. override an existing function with the same name.

It is possible to simulate the more standard behavior by using a special syntax of the import statement:

from modu import \*. This is generally considered bad practice.

Using import \* makes code harder to read and makes dependencies less compartmentalized.

Very bad

[...]

from modu import \*

[...]

x = sqrt(4) # Is sqrt part of modu? A builtin? Defined above?

Better

from modu import sqrt

[...]

x = sqrt(4) # sqrt may be part of modu, if not redefined in between

Best

import modu

[...]

x = modu.sqrt(4) # sqrt is visibly part of modu's namespace

## What are packages?

We don't usually store all of our files on our computer in the same location. We use a well-organized hierarchy of directories for easier access.

Similar files are kept in the same directory, for example, we may keep all the songs in the "**music**" directory. Analogous to this, Python has packages for directories and [modules](https://www.programiz.com/python-programming/modules) for files.

As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.

Similarly, as a directory can contain subdirectories and files, a Python package can have sub-packages and modules.

A directory must contain a file named \_\_init\_\_.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.

Here is an example. Suppose we are developing a game. One possible organization of packages and modules could be as shown in the figure below.

Package Module Structure in Python Programming

## Importing module from a package

We can import modules from packages using the dot (.) operator.

For example, if we want to import the start module in the above example, it can be done as follows:

import Game.Level.start

Now, if this module contains a [function](https://www.programiz.com/python-programming/function) named select\_difficulty(), we must use the full name to reference it.

Game.Level.start.select\_difficulty(2)

If this construct seems lengthy, we can import the module without the package prefix as follows:

from Game.Level import start

We can now call the function simply as follows:

start.select\_difficulty(2)

Another way of importing just the required function (or class or variable) from a module within a package would be as follows:

from Game.Level.start import select\_difficulty

Now we can directly call this function.

select\_difficulty(2)

Although easier, this method is not recommended. Using the full [namespace](https://www.programiz.com/python-programming/namespace) avoids confusion and prevents two same identifier names from colliding.

While importing packages, Python looks in the list of directories defined in sys.path, similar as for [module search path](https://www.programiz.com/python-programming/modules#search).