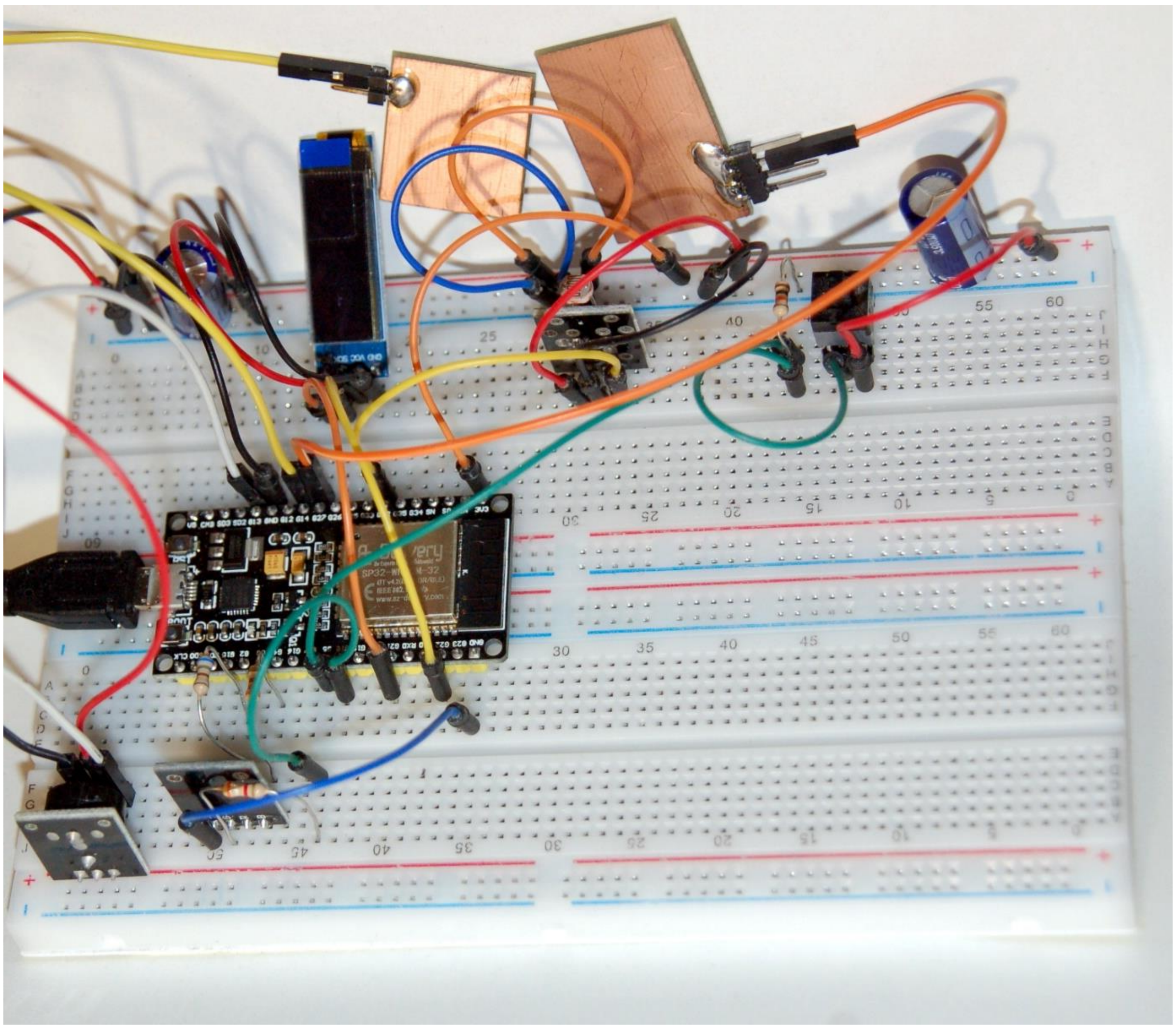


Micropython with ESP32 / ESP8266 – Part4

Modules and Classes

Welcome to the fourth part of MicroPython with the ESP32 / ESP8266. This time we will discuss the use of additional hardware and, above all, take a closer look at the subject of modules and classes, which I think is very interesting. An OLED display is presented and a module for simplified text output and the generation of bar charts is programmed. A hardware timer controls an active buzzer and a LED. Of course, a self-created class is used for easier use in various other programs.

When discussing the homework from part 3, you can program a reaction time measuring device with a touchpad or button and let your imagination run wild.

I moved the chapter about using esptool.py to flash firmware to the next post because this part would have become too extensive.

But before I start with the actual part 4, something completely different.

Raspberry PI Pico - Character and rating

A few days ago the Raspberry Pi Pico came on the market. It is a microcontroller board from the Raspberry.Pi Foundation, which has also developed its first chip for this board, the RP2040 microcontroller. It is a dual-core ARM Cortex M0 + 32bit processor that can be clocked up to 133MHz. The reason why this part appears here in the blog is because MicroPython is already integrated in the firmware. That was to be expected because Python is also the native language of the Raspberry Pi family.

The table compares the properties of the Pico with those of the ESP32 so that you can get an idea of the performance.

Feature	Raspi Pico	ESP32
Flash	2MB	4MB + 448kB ROM
RAM	264kB	520kB + 16kB RTC
Maße	51x21mm	53x28mm
Befestigung	4x2,1mm	4x2,5mm
Vcc	3,3V	3,3V
Spannungsversorgung extern	1,8..5,0V mit zusätzlicher Hardware	3,3..12V
Takt	133MHz	240MHz
GPIO	23+3 analog	28+6 analog
SPI	2	2
I^{2}C	2	2
UART	3	3
ADC	3 x 12bit	3 x 12bit
DAC	-	2 x 8bit
PWM	16	16
I^{2}S	-	ja
USB	1.1	2.0
Touchsensoreingänge	-	10
IR-Controller	-	ja
Pulszähler	-	8 Kanäle, 7 Modi
Bluetooth	-	ja
SD/MMC Controller	ja	ja
RTC	-	ja
Randomnumber-Generator	-	hard- + softwaremäßig
Ethernetcontroller	-	ja mit zusätzlicher Hardware
Hallsensor intern	-	ja
Temperatursensor intern	-	ja
Hardwaretimer	1 mit 4 Alarmeinheiten	2 je 2 Kanäle
Watchdogtimer	-	ja mit extra Takt
WLAN	-	ja Station- und Accesspoint-Mode 802.11 b/g/n
Micropython	in Firmware	ladbar
NodeMCU-LUA	?	ladbar

C/C++	?	Arduino-IDE
AT-Firmware	-	ladbar

At first glance, the table reveals two essential things.

First, the Pico has only half as much memory as the ESP32, in terms of flash and RAM.

Second, the Pico lacks any network connectivity.

Now, of course, you could connect at least an ESP8266-01 via a UART port, but that's not the thing. And an additional ESP32 is out of question for this purpose, because then I can save myself the Pico.

The characteristics of the RP2040 are only half as good as those of the ESP32 in terms of clock frequency. Well, some interfaces of both systems are comparable, but the Pico lacks some features of the ESP32. The most serious deficiency is the lack of network equipment. This refers the Pico to the lower grounds of an Arduino, which also requires additional hardware. This means a huge limitation for the usability of the hardware in practical life.

It is wonderful that the Pico can produce its operating voltage of 3.3V from 1.8V to 5.0V using a buck-boost converter. In the same breath, however, it must also be mentioned that when using an external voltage source, a protective diode or a P-MOSFET transistor must also be installed.

At first glance, you think, oh how nice, the circuit board fits into a 40-pin DIL socket. Far from it, the pins are unfortunately one grid unit too far apart. At least you can solder in standard pin headers and use them to put the board on a breadboard. The semi-perforated edge contacts allow direct assembly as a module on a user board.

Conclusion:
With the huge variety of microcontrollers on the market for every purpose, the introduction of the Pico sounds strange. What is new with this product? The lower price doesn't make up for the lack of performance. Of course it increases the ego of the Rasp.Pi-Foundation - "We can also manufacture microcontrollers."

Let's get back to business now!

The following hardware is used, some of which you already have if you have read the first three blog posts and, above all, actively implemented them.

- 1 ESP32 NodeMCU Module WLAN WiFi Development Board or
- 1 ESP-32 Dev Kit C V4 or
- 1 NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F with CH340
- 1 0.91 inch OLED I2C display 128 32 piel for Arduino and Raspberry
- Pi or
- 1 0.96 inch OLED I2C display 128 64 Piel for Arduino and Raspberry
- pi
- 1 KY-012 buzzer module active
- 2 LED (color does not matter) and
- 2 resistor 330 ohm for LED or
- 1 KY-011 Bi-Color LED module 5mm and
- 2 resistor 560 ohm for LED or
- 1 KY-009 RGB LED SMD module and
- 1 resistor 330 Ohm for blue LED
- 1 resistor 680 Ohm for red LED
- 1 resistor 3.9k Ohm for green LED
- 1 KY-004 button module sensor button or
- 1 keypad-ttp224-14-capacitive
- 1 KY-018 photo LDR resistor
- 2 mini breadboard 400 pin with 4 power rails for Arduino and jumpers
- electric wire
- 1 jumper wire cable 3 40 PCS. 20 cm each M2M / F2M / F2F
- 2 sheet metal approx. 20 20 mm (not aluminum!) Or remnants of circuit boards
- some pins 0.60.612mm

Note on the "touchpads":

Pins for connecting jumper cables must be soldered to the sheet metal or circuit board pieces. Therefore, aluminum should not be used here, because it can only be soldered with a trick that requires the use of chemicals. So use copper, brass or bare tinplate.

Since a focus in this article is to be on the creation and use of modules in addition to the integration of hardware, we will start with a few experiments on this topic.

To repeat a statement:

Modules are collections of thematically related data and functions in a separate file. Data and functions can be combined into classes within the module.

Additional remark: classes can also be created within a program.

Why use modules?

At the beginning I want to answer a few questions that may have arisen in the course of the last post regarding the statement.

Question 1:
Why should I use modules and classes at all?

Answer:
For some things in your program there will be no getting around the use of modules and classes if you don't reinvent the wheel and B. want to code the access to GPIO pins, timers, the ADC etc. in C / C ++ again. This affects the classes and modules already contained in MicroPython. With the Arduino IDE, you are sure to use the diverse libraries that are offered for all possible sensors. In this sense, we have already made extensive use of classes and modules in the previous sections and even created our own class.

Question2:
Sure, using built-in modules makes sense, but why should I create something like this myself? Isn't that possible without it? Can I do that at all?

Answer:
You will not create your own modules for programs with a few lines. However, as the program grows, you will notice the following things.

- It becomes more and more annoying to jump back and forth between text parts at the beginning and further back
- The longer the program text, the longer it will take to upload to the ESP32
- The longer a program becomes, the more the overview disappears
- The more extensive a program file becomes, the more difficult it becomes to maintain it if you want to change something after a long period of time
- You often copy the same parts of a program that are intended to operate certain hardware into other programs

Perhaps that is why you have already resorted to using your own functions, because it is sufficient to code the same sequences that are used repeatedly, just once, in functions and procedures and to call them up when necessary. When using modules, you first have the option of distributing your project over several files and thus editor windows. Instead of scrolling, switch windows. It is even possible to use any number of text editors in parallel without any problems. This is faster and also has the advantage that you only have to upload part of your project from Thonny or µPyCraft to the ESP if you make changes; that in turn saves time.

Question3:
Isn't it enough if I simply place variables and functions at the beginning of my program?

Answer:
This is a good first approach, but in addition to the growing scope of the program and the disadvantages associated with it (answer to question 2) it has another decisive disadvantage. What if you want to use the same functions in other programs as well. Take the control of an OLED display as an example. Of course, you can then copy

the text from file1 to file2. However, you have not observed another principle, you have created a redundant database.
That means:

- That the same amount of storage space is required on the local medium with each new copy
- That changes to the program text of the functions must be carried out in every existing file
- That you will soon lose track of what has been programmed where and how.

Otherwise it is sufficient to make the changes in a module file. By importing the changes will reach all existing and future programs.

Question4:
What are the advantages of using modules?

Answer:
In essence, I've already answered that for question2 and question3. Let's put it the other way around, formulated positively.

- Modules as a collection of constantly required constants, variables and functions relieve each program file of the size.
- The use of modules makes it easier to maintain the software.
- Modules can be imported into any number of programs.
- The central character of modules avoids redundancy and thus simplifies software maintenance.
- Modules contained in MicroPython facilitate the handling of port-specific hardware and are therefore indispensable

Question5:
What is the difference between a module and a class?

Answer:
In principle, data and functions with completely different meanings and uses can be collected in a module, just like books on all sorts of topics in a library. But in every library, the books are also sorted by subject, which not least promotes findability and an overview of the inventory. In MicroPython, this sorting is done by the classes. A module can contain one or more classes, but does not have to be.

A module can import additional modules or classes of its own accord. A class can take on attributes and methods from another class in its own namespace, this is called inheritance.

Question 6:
OK, if that's that handy, how do you properly create modules and classes?

Answer:
Let’s look at very specific examples. We will examine one or the other highlight of modules and classes more closely. At the end of this blog post, you are module mogul.

Modules and classes experimental

We start with very simple actions, but where it is important to avoid side effects. Such effects arise, for example, from the fact that MicroPython remembers everything that we have entered via REPL, i.e. the command line in interactive mode, or what programs that have already run have left behind. To avoid this behavior, you can either turn off the ESP32 or just press RST.

Now let's take a closer look at one thing that we've just been using so far. It is about functions in the broader sense and variables in the narrower sense. The key question is: How can I address variables where and why, and why are there sometimes problems with them?

The first sequence therefore explains the important topic of namespace. This means the area in which names of constants, variables and functions are valid. The English name for this is scope. This name suggests the meaning of 'visibility', think of the name 'microscope'. The namespace could also be interpreted in such a way that the data and functions are "visible" in it, i.e. can be addressed. From the Arduino IDE, you may be familiar with the error message "... is not declared in this scope!" When you forget to declare a variable before assigning a value to it or referencing it.

In MicroPython, you don't have to declare a variable by type assignment as in the Arduino IDE, but you do have to assign a value to a variable before the variable is queried, for example in a formula term or a statement like print.

MicroPython essentially differentiates between two areas of validity, the global and the local. Everything that is declared in the main program (main ()) or from REPL is regarded as global. The namespace of functions is to be regarded as local. Everything that is newly declared in a function is only visible within this function and cannot be addressed from the main program. We're talking about encapsulation here. This also applies if variables inside and outside the function have the same name.

Now create a new document using New in µPyCraft or Thonny, in the following I will use the term editor as a generic term. You can use this file again and again if you delete, add to or overwrite the content after the test. This is easier and clearer than creating a new file every time.

Enter the program text, save it in the workSpace, transfer the program to the ESP32. Press RST and start the program. Please repeat this procedure for each of the following examples.

Experimets with variables

Variables are characterized by three things in MicroPython, name, value and identity. The name refers to a memory location in which the value is stored. Values can be compared with ==,! =, <,> <= And> =. So that MicroPython can uniquely identify variables, they also have an identity number, which is an integer. Having two variables have the same value doesn't necessarily mean they have to have the same identity, but if they have the same identity, they have to have the same value. Crazy? Yes a little. It is because in the latter case a variable only has two different names, but they both point to the same value.

```
>>>a = 2
>>>b = 2
>>>c = 1+1
```

It is a == b True and of course also a == c is True. It has to be like this, because all three variables have the same identity, which can be checked with the id () function

id (a), id (b) and id (c) always return the same value. In my experiment it was 5. Only if you were to assign the value 3 to the variable b, for example, did its identity change. a, b and c all three originally point to the same memory location in which the value 2 is stored. OK? I don't want to go into any further detail here because this type of memory management can be very strange. Maybe I'll come back to this in another post. Now for the opposite.

```
>>> d = 2.0
>>> d == a True
But
id (a) 5
id (d) 77
```

a and d are of the same value, of course, but they are not identical, because the integer 2 and the floating point number 2.0 are of a different type and therefore cannot be identical. Also OK? For sure!

There are now two variables with the same name a. Outside the function a is global and has the value 10, inside a completely different, local a has the value 7. Think about the meaning of the term encapsulation.

Download: aaa.py

```
def show():
  a=7
  print(a)
  print("id:",id(a))


a=10
show()
print(a)
print("id:",id(a))
```

Ausgabe:

```
7
id: 15
10
id: 21
```

The 7 comes from the function, the 10 has been retained beyond the function call in the a of the main program. If you have the identity of a displayed inside and outside the function by an additional print command, you will see that there are really two different variables.

To pass data to a function for processing, you can use parameter pass. Parameters are listed in round brackets after the function name. The names of the parameters should be different from the names used in the calling program for the sake of uniqueness and clarity of the program. My parameter is called here b and is assigned as argument a when called. The a in the functional body is local again.

Download: aab.py

```
def show(b):
  a=b+7
  print(a,b)

a=10
show(a)
print(a)
```

Ausgabe:

```
17 10
10
```

To get a result back from the function, use return. If you don't specify return, you will get the value None.

Download: aac.py
```
def show(b):
  a=b+7
  print(a,b)
  return a

a=10
a= show(a)
print(a)
```

Ausgabe:
```
17 10
17
```

A global variable can very well be referenced (queried) within a function if a variable with the same name is not created within the function definition, as in the previous examples.

Download: aad.py
```
def show():
  c=a+7
  print(a,c)
  return c

a=10
x=show()
print(a,x)
```

Ausgabe:
```
10 17
10 17
```

However, the following construct leads to an error if you try to change the value of a within the function. The error does not occur if you place line a = 20 before line c = a + 7. But then it still does not have the desired effect, namely that the value of the global variable a is changed.

Download: aae.py
```
def show():
  c=a+7
  print(a,c)
  a=20
  return c

a=10
x=show()
print(a,x)
```

Output:
```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 8, in <module>
  File "<string>", line 2, in show
NameError: local variable referenced before assignment
>>>
```

This message comes about because the interpreter only assigns a memory location to what it sees as a local variable a while translating the text, but does not yet assign a value. If the function is now called during the program run, then a in line 2 does not yet have a value that could be added to 7. The value assignment to a only happens in line 5.

To tell the function that the a from the global namespace should be used in the function, it must look like this. global instructs the interpreter to use the variable from the higher-level namespace. Now the addition in line 3 and also the value change in line 5 work with the a value 10 from the main program or from the command line.

Download: aaf.py

```
def show():
  global a
  c=a+7
  print(a,c)
  a=20
  return c

a=10
x=show()
print(a,x)
```

Ausgabe:
```
10 17
20 17
```

Kaffeepause? OK!

--

Did the coffee taste good? Then let's take a look at some basic things about self-created modules and classes. You will need this, to understand things in the hardware section immediately afterwards.

The four most important versions of module and class import

Similar to the functions is the case with the namespace of modules and classes. Before we create our own modules and classes, let's take a quick look at the import variants. For the following examples, input via the command line is sufficient.

Variant 1

Because I want to use math functions and constants, I have to import the math module. I do this in four different ways and get different results every time I want to get a value for the circle number pi = 3.14 ....

Newstart (RST), first experiment

```
>>>import math
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' isn't defined
>>> math.pi
3.141593
>>>
```

import math imports all structures from the math module in its own scope of the same name. The constant pi is unknown to REPL. I am successful when I use the math namespace. Now the rounded value of the circle number of pi is known.

Variant 2

Newstart (RST), next experiment

```
>>> import math as m
>>> pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' isn't defined
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>> m.pi
3.141593
```

pi is not known to REPL
but also math.pi no longer knows REPL. This is what m.pi is now known. This is because we have now renamed the math namespace to m. math is no longer accessible. An alias, like m in this example, can be used to save paperwork, or, as in the scind post, to be able to address two different modules with the same name.

Variant 3

Nnewstart (RST), third variant

```
>>>from math import *
>>> pi
3.141593
>>> math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>>
```

With from math import * we import the entire namespace of math into the global namespace. The two areas are effectively merging with one another. This means that every identifier from math is globally available, but the name math itself died with it. In this context, please also note the warning for the fourth variant.

Version 4

Newstart (RST), forth variant

```
>>>from math import sin, pi
>>> pi
3.141593
>>> sin(pi/2)
1.0
>>> math.sin(math.pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' isn't defined
>>>
```

Because I don't need cos, tan, etc., I just import the sin () function and the constant pi. Here too, the use of 'from' has the effect that the imported identifiers are transferred to the global namespace. So you can note

- **import… and import as… imports all members of modules as their own namespace. This is renamed with as.**
- **from ... import ... merges the external with the global or superordinate namespace.**
- **If only parts of a module are imported with from, the remaining members of the module are not available outside the class.**

But, be careful, this may also happen:
Neustart (RST),

```
>>> pi=3.14
>>> from math import sin, pi      oder  >>> from math import *
>>> pi
3.141593
```

The import of attributes, constants and functions into the current namespace overwrites objects with the same name defined there. In the example these are the name defined before the import and the value for pi. The fact that these are different objects again shows the value of identity. You will certainly get other numerical values for id.

newstart (RST)

```
>>> pi=3.14
>>> id(pi)
1073633840
>>> from math import sin, pi
>>> pi
3.141593
>>> id(pi)
1061192072
```

These things are very important. The consideration of functions and the understanding of the import of modules will help you to thunder less crumpled junk notes into the corner if the ESP32 brings nothing but error messages during the import or if completely unexpected calculation results occur.

Own modules and classes - a closer look

With what we know so far, we can start tinkering with a module in which we look a little further behind the scenes than we did at touch.py in Part 3.

Purpose-free and just for fun, I created the following program text, saved it under mathe1.py and transferred it to the ESP32. A few variables are declared and three functions are defined. rezi (n) is the reciprocal of n, sum (n) is the sum of the numbers from 1 to n and fak (n) calculates n !, the product of the numbers from 1 to n. I think the function of the code must be do not explain further. More important is what we're going to do with the code. When you start the program on the ESP32, not much happens visually. In the background, however, MicroPython now knows the strongly rounded value of Euler's number e and a value for Pi and PI. And the ESP32 knows what to do when you issue the rezi (8) instruction via REPL. Try it yourself from the command line.

Download: mathe1.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    print("Fehler: Versuchte Division durch 0!")
    return None

Pi =3.14
PI = 0

def sum(n):
  summe=0
  for i in range(n+1):
    summe+= i
  return summe

def fak(n):
  nfak = 1
  for i in range(2,n+1):
    nfak *= i
  return nfak
```

```
Pi
3.14
>>> e
2.71
>>> rezi(8)
0.125
>>> sum(20)
210
>>> fak(8)
40320
```

Neustart (RST) neue Version

```
>>>import mathe1
>>> Pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Pi' isn't defined
>>> mathe1.Pi
3.14
>>>
```

The module as such is complete and ready for import on the command line and in programs. Our advantage, changes to the text of mathe1 only have to be made in one place, the file mathe1.py and not in every single program. Change the module, save, upload and restart the test program. You can also edit the module and test program in parallel to each other in a separate window. Constant scrolling up and down is no longer necessary. A very simple example:

Download: mathe1a.py

```
import mathe1

print("Kreiszahl:",mathe1.Pi)
n = 100
print("Die Summe von 1 bis",n,"ist",mathe1.sum(n))
```

```
Circle number: 3.14
The sum from 1 to 100 is 5050
```

So far we have created and used the module.

Then we'll use it to make a class that we'll then test in action.

The definition of a class begins with the keyword class, followed by the name of the class and the mandatory colon - we already know. Please note again that, similar to other program structures, everything that should belong to the class body must also be indented. Here is the program text.

Download: mathe2.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    return None

class MH:
  Pi =3.14
  PI = 0

  def sum(n):
    summe=0
    for i in range(n+1):
```

```
    summe+= i
  return summe

def fak(n):
  nfak = 1
  for i in range(2,n+1):
    nfak *= i
  return nfak
```

Save the text as mathe2.py in the workSpace and send the file to the ESP.

Restart (RST)

Now it's getting exciting. I'm sure you've noticed that I didn't include e and rezi in the class. Why? We're going to test everything in a moment.

```
>>> import math2
>>> mathe2.e
2.71
>>> math2.rezi (100)
0.01
```

Very good, everything seems to be working -is it?!

```
>>> math2.sum (1000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'sum'
```

OK, we have set up a new structure within the module with its own namespace, then just like this:

```
>>> math2.MH.sum (1000)
500 500
>>> math2.MH.Pi
3.14
```

Phew, lucky again. But basically nothing has changed in the structure compared to before.

Shouldn't it be possible to derive objects from a class, as we have already done, for example, with machine.Pin?

```
>>>from machine import Pin
>>>taste = Pin(5,Pin.IN)
```

experiment:

```
>>> from mathe2 import MH
>>> m=MH
```

No error message, very good, actually works. Of course, the functions that are now called methods in the class can also be called.

```
>>> m.sum (1000)
500 500
```

```
>>> m.Pi
3.14
```

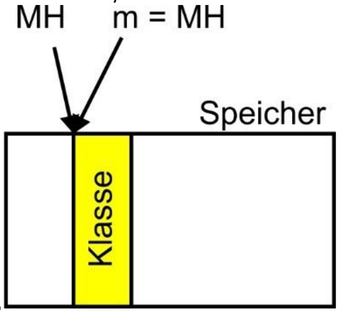
But is m really an instance of class MH now? The id test brings the truth to light.

```
>>> id (m)
1073635408
>>> id (MH)
1073635408
```

And the call of m says the rest.

```
>>> m
<class 'MH'>
```

m and MH are identical or, in other words, m is an alias for MH and therefore m cannot denote an instance of MH.



With a simple command, you can see what identifiers are available for a particular class or module. This also says a lot about how methods are called and how variables are referenced, and in this case shows that both names point to the same structure.

```
>>> dir(m)
['__class__', '__module__', '__name__', '__qualname__', 'sum', '__bases__',
'__dict__', 'Pi', 'PI', 'Rezi', 'fak']

>>> dir(MH)
['__class__', '__module__', '__name__', '__qualname__', 'sum', '__bases__',
'__dict__', 'Pi', 'PI', 'Rezi', 'fak']


>>> dir(mathe2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'mathe2' isn't defined
```

The scopes of m and class MH are identical, while the namespace of math2 was not imported. What is outside of MH cannot be included in the scope of MH. MH is a member of mathe2 and not the other way around.

What I said earlier about variables also applies to other objects, functions and, as here, classes. We shall generalize that.

- **Two objects are identical if they address the same memory area.**
- **Identity thus includes equality of value or the same functionality.**
- **Different names always refer to the same memory area for identical objects.**

The identity of two objects can be determined very simply with the keyword is, apart from the output of the identity value.

```
>>> m is MH
True
```

newstart (RST)

Now that we've built a class, we should be able to derive objects from it too. Fresh to work, one more time.

```
>>> from math2 import MH
>>> m = MH
```

Halt, - that we cannot create any objects with this assignment, only another name for the class MH suggests that something important is still missing.

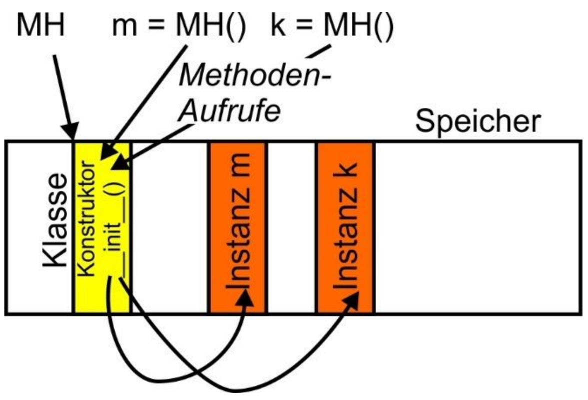
In order to derive objects, our class also needs the so-called constructor, an instruction according to which an object can be created. You already got to know this special method in part 3 of the blog. Here you can now see the purpose that the __init__ method must fulfill. The script mathe3.py represents the summary of the treated cases and also contains a very simple constructor which only creates an instance attribute PI and can report that it has been called. In addition, the two previous class methods sum () and fak () must be made into instance methods. For this reason, a self was added as the first parameter in the parameter list of these methods. That has been missing so far. From the third part we know that this self establishes the relation of the object, which we derive from the class, to itself.

Download: mathe3.py

```
e =  2.71

def rezi(n):
  if n !=0:
    return 1/n
  else:
    return None

class MH:
  Pi =3.14
  PI = 0

  def __init__(self,k=3.1415):
    self.PI = k
    print("PI=",self.PI,"wurde angelegt.")

  def sum(self,n):
    summe=0
    for i in range(n+1):
      summe+= i
    return summe

  def fak(self,n):
    nfak = 1
    for i in range(2,n+1):
      nfak *= i
    return nfak
```

```
>>> from mathe3 import MH
>>> k=MH()
PI= 3.1415 wurde angelegt.
>>> m=MH()
PI= 3.1415 wurde angelegt.
>>> m is k
False
```

The syntactic difference to "instantiation" in the previous example is the pair of round brackets after MH (). Without these brackets, only the class is referenced and the same memory space is assigned to another name. Only by calling the constructor, the method __init__, via the class name MH, an instance, an object is created. So here there is no simple assignment like m = MH but a function call m = MH (), which does more than the assignment. As with any other function, the pair of brackets is necessary for the interpreter to recognize this. The identity comparison with is clearly shows that different instances themselves differ.



You can now also set the constant PI of m to a different value. k is not affected by this.

```
>>> m.PI = 3.14
>>> k.PI
3.1415
>>> m.PI
3.14
```

Just like attributes, functions can also be overwritten in an instance. It works similarly with the inheritance of classes. But more about that another time.

```
>>> def reziSum(n):
        summe=0
        for i in range(1,n+1):
            summe += 1/i
            print(1/i)
        return summe

>>> reziSum(4)
1.0
0.5
0.3333333
0.25
2.083333
>>> from mathe3 import MH
>>> m=MH()
PI= 3.1415 wurde angelegt.
>>> m.sum(4)
10
>>> m.sum = reziSum
>>> m.sum(4)
1.0
0.5
0.3333333
0.25
2.083333
```

By the way, even without the definition of an __init__ method, the statement m = MH () creates an instance from the class MH of the module math2 with the methods sum

() and fak (). Then there are no instance attributes available, but the class attributes Pi and PI are. Try it!
Kaffeepause!

Back? Then let's get down dealing with the new hardware.

Hardware

The timer and tweeting – The buzer

There are three ways to wait in MicroPython.
• sleep and do nothing else
• wait and do the same thing over and over again
• Set the alarm clock, do any other thing and when the alarm clock rings, do something else in between.

We know the first variant from the blinker program from Part 1. We used the second variant to measure the speed of the ADC in Part 3. And we are now exploring the third type.

For my project, I need the option of making a piezo buzzer sound for a certain (short) time. Because I don't want to worry about generating the signal frequency as well, I choose an active piezo buzzer that only needs to be controlled by a digital 1. He creates the whistle himself. While the buzzer is active, an LED should flash at the same time. I activate the switch-on myself; the switch-off should happen automatically, regardless of what my program is currently doing. In the meantime, the main program can do any other important thing.

Such cases are often dealt with using so-called interrupts, literally translated as interruptions. The program is then sent on a coffee break, the interrupt service routine takes over control, and when it is finished, the interrupted program gets the rudder back. Interruptions can be triggered by external sources or internal processes of the ESP32 / 8622. The controller can react to the change in the logic level on a pin or, as in my case, to the expiry of a timer.

The methods and values for this functionality are provided by the Timer class from the machine module. Interesting that Timer does not live in the time module. The fact is, however, that time is implemented in software, while the timer affects the hardware of the ESP32 and therefore logically has to be located in machine. Timers are hardware modules whose counter registers are incremented by their own clock without the involvement of the processor and can trigger an interruption when a specified counter reading is reached. The ESP32 / 8266 has 4 of them.

A timer must first be instantiated for use. Then it is important how long it should take for the alarm clock to ring and it must be known whether the alarm clock should rattle only once or repeatedly. In this case you have to be able to switch off the alarm clock. The init () method does the initialization. The syntax for this is very straightforward.

Now create a timer object t, up to 4 timers are possible. The number is passed to the class constructor.

```
>>>from machine import Timer
```

t = **Timer**(0)

The timer object t must be initialized, same in the example for a one-time action.

callback is a function that is called when the alarm goes off.

callback = lambda f: action

The keyword lambda creates an anonymous function that executes the action. This can be a print statement or an arithmetic expression or a function call like here. period receives the duration in milliseconds. ONE_SHOT (once) and PERIODIC (repeated) are constants of the Timer class. It all adds up to:

```
t.init (period = 5000, mode = Timer.ONE_SHOT, callback = lambda f: action)


t1 = timer (1)
t1.init (period = 2000, mode = Timer.PERIODIC, callback = lambda f: action)

t1.deinit ()
```

deinit () is required to stop the periodically initialized timer t1. With ONE_SHOT this is not necessary because it happens automatically when the timer expires.

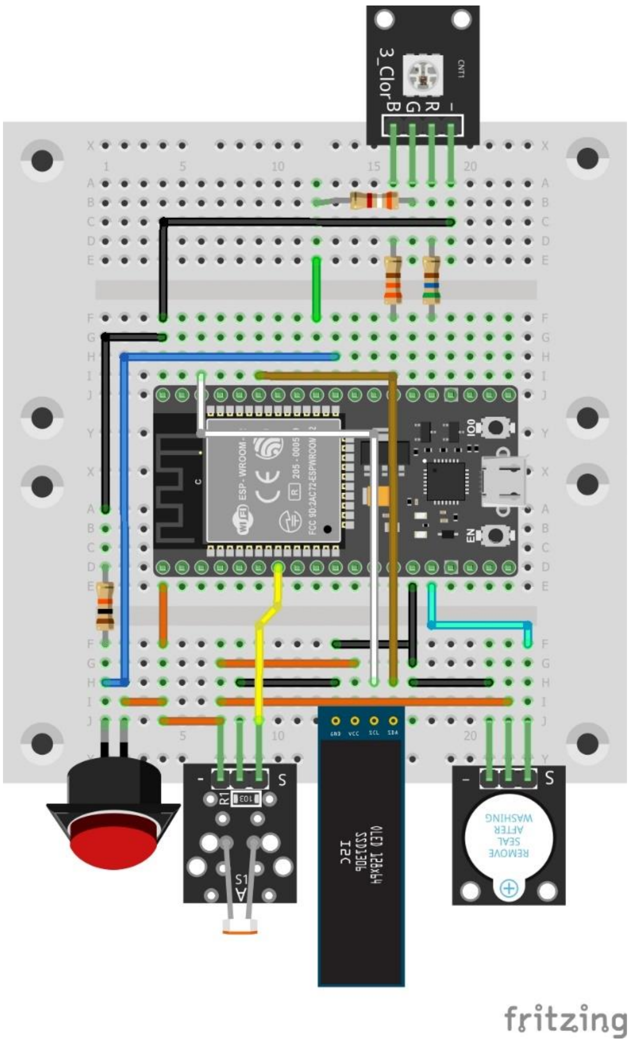
Because we want to do it right now, we are also making a beep module with the BEEP class to demonstrate and deepen the knowledge we have acquired.

Download: beep.py

```
from machine import Pin, Timer
import os

DAUER   =    const(15)

class BEEP:
  dauer   =    DAUER

  def __init__(self, led, buzz, duration=dauer):
    self.ledPin=Pin(led, Pin.OUT)
    self.buzzPin=Pin(buzz, Pin.OUT)
    self.beepOff()
    self.tim=Timer(0)
    self.dauer = duration
    print("Konstruktor von BEEP")
    print("LED:{}, Buzz:{}, dauer={}".format(led, buzz, self.dauer))

  def beepOff(self):
    self.ledPin.value(0)
    self.buzzPin.value(0)
```

```
def beep(self, puls=None):
  if puls == None:
    tick = self.dauer
  else:
    tick = puls
  self.buzzPin.value(1)
  self.ledPin.value(1)
  self.tim.init(mode=Timer.ONE_SHOT,period=tick,callback=lambda t: self.beepOff())

def setDuration(self, duration=dauer):
  self.dauer=duration

def getDuration(self):
  return self.dauer
```

The connection for the red LED on GPIO2 and the buzzer on GPIO13 looks like this.



In the module area, outside of the class, we import from machine pin and timer as well as os. Then the constant DURATION is defined. Then the BEEP class definition begins.

We assign the value of the static constant DURATION to the class attribute duration - these are different objects, MicroPython is case sensitive! Now comes the most interesting part of the module, the constructor function.

Because the introductory phrases are already known, I will first concentrate on the parameter list (self, led, buzz, duration).

To repeat: we are in the process of defining a class and therefore every instance method has self as the first parameter. This refers to the object that is later derived from the class, not to the content of the function body or even the class. The specification of this parameter is mandatory, i.e. necessary. The same applies to the prefix self for instance attributes.

Two so-called position parameters follow, led and buzz. These are assigned by the interpreter with the arguments specified in this position when the function is called. Position parameters must be specified when they are called, in exactly the same order as they are defined in the parameter list.

duration = duration is an optional parameter. When called, it can simply be specified by a value or set specifically by a name = value pair. However, it can also be omitted, in which case the default value from the function definition is used. The examples show this. But position parameters must always be listed before the optional parameters!

```
>>> from beep import BEEP

>>> b = BEEP (2,13,700)
BEEP constructor
LED: 2, Buzz: 13, duration = 700

>>> c = BEEP (2, 13, duration = 1000)
BEEP constructor
LED: 2, Buzz: 13, duration = 1000
```

In the following example the order was reversed; that does not work.

```
>>> a = BEEP (2, duration = 1000, 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: non-keyword arg after keyword arg
```

Position parameters are always at the beginning of the parameter list. Optional parameters can follow, but do not necessarily have to, and their order is also free, provided the name = value pairs are specified. Calling the timer initialization is as the example shown here.

```
self.tim.init(mode=Timer.ONE_SHOT,period=puls,callback=lambda t: self.beepOff())
```

What applies to normal functions, of course, also applies to the BEEP constructor. The GPIO number of the LED and the buzzer pin must be specified, the beep duration is optional. If it is not specified in any of the permitted ways when calling BEEP (), the __init__ function sets it to 15. So that you can see whether the constructor has run and what it has done, I have included the two print statements for it.

What else does the constructor do? It defines the two pin instances ledPin and buzzPin, the timer instance tim0 and sets the beep duration of the generated BEEP object to the default setting or the transferred argument. In addition, it switches off the beepOff buzzer and LED using the instance method and thus creates a defined initial state.

Here you can see exactly which attribute and which function is linked to an instance of the BEEP class. They are all identified by the prefix self. This works great with calls to methods and references to variables within the definition of the class. The beep () and beepOff () methods make use of this. A transfer of instance attributes as parameters of a function is basically not possible and also not necessary, since an object can of course also use its in-house variables within in-house functions. I present a special case after the first experiments.

In use it looks like this, we already know such calls from various applications.

The duration is omitted when the constructor is called, so the default of the static variable DURATION is used automatically.

```
>>> from beep import BEEP
>>> c = BEEP (2.13)
BEEP constructor
LED: 2, Buzz: 13, duration = 15
```

The following instances are assigned different values.

```
>>> b = BEEP (2,13, 400)
BEEP constructor
LED: 2, Buzz: 13, duration = 400

>>> d = BEEP (2.13, duration = 1000)
BEEP constructor
LED: 2, Buzz: 13, duration = 1000

>>> c. duration
15th
```

There is also a homework assignment for the following test at the end of the article.

```
>>> from beep import BEEP
>>> b = BEEP (2,13, 400)
BEEP constructor
LED: 2, Buzz: 13, duration = 400
```

400ms signal

```
>>> b.beep ()

>>> b.setDuration (800)

800ms signal
>>> b.beep ()

>>> b.getDuration ()
800

Overwrite 800ms
>>> b.beep (6000)

Before the 6 seconds have elapsed
>>> b.beepOff ()

>>> b.getDuration ()
800

Set class default value 15ms
>>> b.setDuration ()
>>> b.beep ()
>>>
```

The module is equally suitable for ESP32 and ESP8266, since no resources were used that are only available on the ESP32. The pins on the ESP8266 only need to be chosen so that they do not interfere with any other functions.

Let's take another look at the beep () method

```
 def beep(self, puls=None):
   if puls == None:
     tick = self.dauer
   else:
     tick = puls
   self.buzzPin.value(1)
   self.ledPin.value(1)
   self.tim.init(mode=Timer.ONE_SHOT,period=tick,callback=lambda t: self.beepOff())
```

Without an argument, the value of the instance stored in self. Duration should be used. However, it cannot be included in the parameter list, not even as an optional parameter. Alternatively, a given argument should serve as the pulse duration. The trick now is that the parameter list is presented as the default None (0 would also be possible). This is tested in the if construct. Instead of None, the actually desired default value is set and otherwise the passed argument.

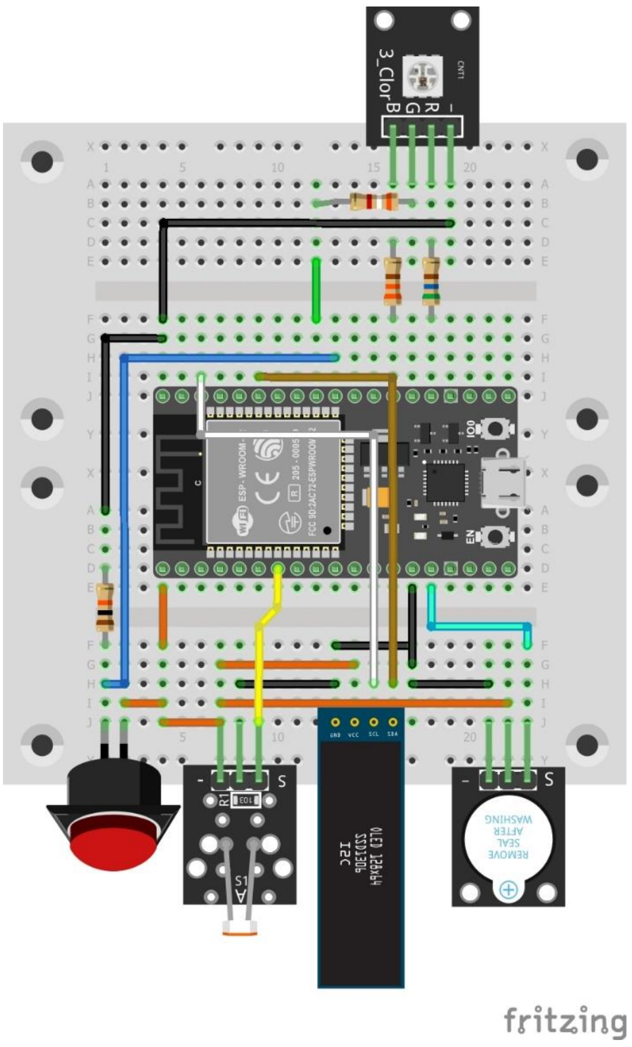
Because you shouldn't mess around with variables and the like, the value of self.duration is set and queried here using methods, as it should be.

OLED at ESP32/ESP8266

- So that the ESP slowly becomes independent, we are now giving it an OLED display. With one like this, we are no longer necessarily dependent on the output of messages via the terminal. Two inexpensive displays are these:0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder
- 0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi

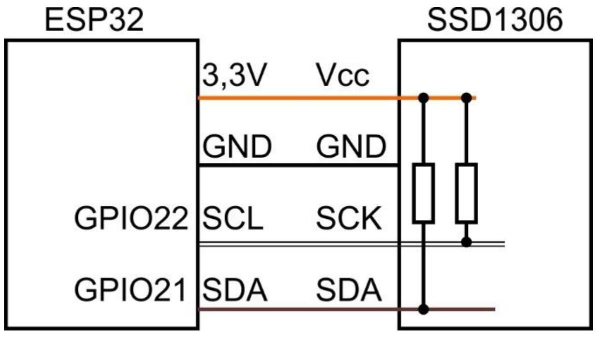
textboxThe reason why I opted for an OLED display and not a 2x16 LCD is quite simply that I can also display simple graphics in a space-saving manner with an OLED display. This is not possible with a purely alphanumeric LCD. There is also a module for the 1306 series OLEDs that I will use.

Here is the wiring on the breadboard. Also included are the LDR module, the RBG LED module and the active buzzer as well as the button from episode 1. To protect the eyes and ESP pins, I increased the resistances on the RGB LED to lower the brightness.



The displays mentioned are controlled via the I2C bus. It consists of two lines, the clock line SCL (white) and the data line SDA (brown). Both lines must be wired with a 4.7 kOhm to 10 kOhm resistor to Vcc. With the SSD1306, this has already been done on the board by the manufacturer. The supply voltage can range between 3.3V and 5V. We connect Vcc of the display to the 3.3V output of the ESP32. This prevents more than 3.3V from being applied to the inputs of the ESP32 via the pull-up resistors of the SCL and SDA lines. Of course, GND goes to GND of the ESP32. The power requirement of the display is very low, so that the controller on the ESP32 board can still cope with it.

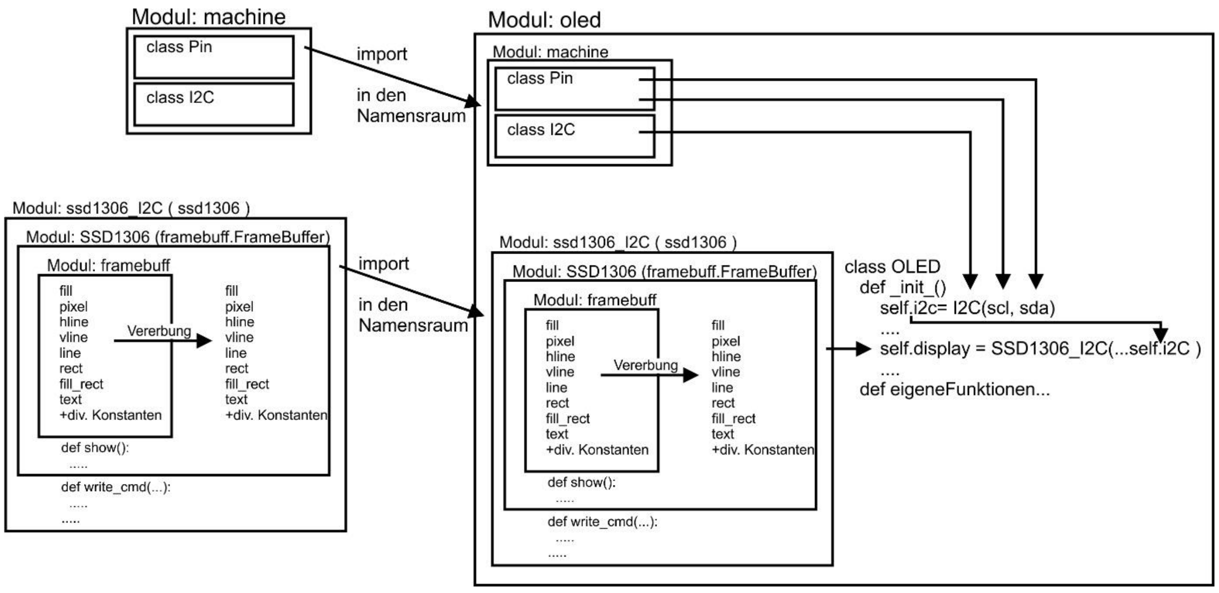
Depending on the controller, SCL and SDA are connected to different GPIO pins. Please note the pin assignment of our OLED module, which may differ from that in the circuit diagram.



Controller	SDA	SCL
ESP32	21	22
ESP8266	4 (D2)	5 (D1)

A ssd1306 module is already included in µPyCraft. It is located in the uPy_lib folder. But I used a different source file for my module oled.py. In principle, both variants use the same original FrameBuffer class from the framebuf module and ultimately also provide the same API for the user, but the way from A to B is different. Variant 1 imports framebuffer and uses it to create cumbersome new methods with the same name.

The ssd1306.py I used already has the FrameBuffer namespace through inheritance and does not have to define any new methods. I tried to show that on a diagram.



Now download the file ssd1306.py from github and copy it into your workspace in Windows Explorer. Open the file there in the editor and remove this class definition at the end of the file

class SSD1306_SPI (SSD1306):

including all lines up to the end of the file. What you have removed is the SPI interface, which we do not need because the display is addressed via I2C. This is how we save storage space. Because we only use the I2C bus indirectly via the ssd1306 module and do not address it directly, I will not go into any more detail about the functions of this type of data transfer via I2C. That alone would be a topic for your own blog post.

So that you can integrate ssd1306.py as a module, it must be uploaded to the device directory of the ESP32 after it has been saved in the workspace.

Now let's take a look at the oled.py module. I will discuss individual parts in detail, the rest otherwise follows exactly the same pattern.

My goal was primarily a line and column oriented text output like with an alphanumeric LCD. In addition, there was the desire to be able to specifically delete text line areas and to have variable columns available from the baseline for a diagram, optionally x- and y-axis. The built-in methods of ssd1306 were helpful, but not sufficient in themselves. I have defined the following methods according to my wishes.

OLED ([sclw = SCLPin], [sdaw = SDAPin], [widthw = [64 | 128]], [heightw = [32 | 64]]) writeAt (string, xpos, ypos) clearFT (xv, yv [, xb = column] [, yb = line]) clearAll () pillar (xpos, width, height) setContrast (value) xAxis (), yAxis () switchOn (), switchOff ()

Some modules or classes are imported.
from machine import pin, I2C # ssd1306.py must be in the device directory from ssd1306 import SSD1306_I2C

By the way, it's worth taking a look at the ssd1306.py file. This makes it easy to see how inheritance and the import of namespaces work. It is helpful to make your own experiments on the command line.

The class definition begins and a few constants are specified. The different assignments for ESP32 and ESP8266 must be taken into account, ideally through an automatic in the form of the if structure. By the way, you cannot assign constants to SD and SC here as with WIDTH and HEIGHT, because otherwise a "strange" error message appears about the reasons for this here https://forum.micropython.org/viewtopic.php?t=7989.

```
class OLED:
  device = sys.platform
  if device == 'esp32':
    SD =    21
    SC =    22
  elif device == 'esp8266':
    SD =    4
    SC =    5
  else:
    print("Unbekannter Controller!")
    sys.exit()
  WIDTH = const(128)       # Pixelbreite des Displays
  HEIGHT = const(32)       # Pixelhöhe
```

Das Wesentlichste der Konstruktormethode

```
def __init__(self, sclw=SC, sdaw=SD, widthw=WIDTH, heightw=HEIGHT):
  #ESP32 Pin assignment
  self.columns = widthw // 8
  self.rows = heightw // 10
  self.sd = sdaw
  self.sc = sclw
  self.i2c = I2C(-1, scl=Pin(sclw), sda=Pin(sdaw))
  self.width = widthw
  self.height = heightw
  self.display = SSD1306_I2C(widthw, heightw, self.i2c)
  self.display.contrast(0x3f) # Maximum ist 0xff
  self.display.fill(0)
  print("Konstruktor von OLED")
  print("SDA:{}, SCL:{}, Size:{}x{}".format(self.sd, self.sc, self.width, self.height))
```

The constructor takes 4 optional parameters. This allows us to change the order of the parameters when they are called if they are passed as name = value pairs. You can also omit some of the parameters or even all of them, then the defaults in the header are used. A call like the following is possible. He sets up the ESP32 with the default values for the I2C pins. The same applies to the ESP8266.

```
>>> from oled import OLED
>>> d = OLED ()
OLED constructor
SDA: 21, SCL: 22, Size: 128x32
```

An I2C object is created and the number of lines and columns of text is calculated. We create a display object for the SSD1306, set the contrast or brightness and delete the display content.

The clearFT () method clears the area from text column x / line y to text column xb / line yb.

```
def clearFT(self,x,y,xb=maxcol,yb=maxrow):
  xv = x * 8
  yv = y * 10
  if xb >= self.columns:
    xb = self.columns*8
  else:
    xb = (xb+1) *8
  if yb >= self.rows:
    yb = self.rows*10
  else:
    yb = (yb + 1)*10
  self.display.fill_rect(xv,yv,xb-xv,yb-yv,0)
  self.display.show()
```

There are two positional arguments that must be specified, that is the upper left corner in the column-row space. No values need to be specified for the remaining two parameters; they are optional. You can also simply specify numbers (variables) for the two, then the sequence xb, yb remains. If you want to change the order, you have to specify name = value pairs. If you omit the last two arguments, the default is deleted up to the lower right corner of the display.

A plausibility query is initially carried out so that no inadmissible values are passed on to the display. Then the column-row information is converted into pixel data and finally the corresponding rectangle is prepared for deletion with the fill_rect () method, which ultimately comes from the FrameBuffer class. The show () method displays the deletion. show must always be called when one or more changes have been made to the content of the frame buffer. Keep this phrase in mind if the display stays black or the display doesn't change. Usually the display is not defective, you just forgot to call show (). This cannot happen with any of the commands in the OLED class, because the show () command is already integrated everywhere. However, this is associated with an extension of the runtime of the writing OLED methods. Homework 3rd and 4th deal with this.

While the methods from FrameBuffer work across displays, show () and a few other methods are responsible for very special families of displays and are therefore defined in class SSD1306. The way in which commands and data are sent to the display does not only depend on the bus used. This is the I2C bus here. The class SSD1306_I2C, which inherits from SSD1306, is available for this. We will look at the inheritance of classes in one of the next articles.

The most extensive method of OLED is the following. This is put into perspective again if you omit the six comment lines.

```
 def writeAt(self,s,x,y):
   if x >= self.columns or y >= self.rows: return None
   text = s
   length = len(s)
   xp = x * 8
   yp = y * 10
   if x+length < self.columns:
     b = length * 8
   else:
     b = (self.columns - x) * 8
     text = text[0:self.columns-x]
   self.display.fill_rect(xp,yp,b,9,0)
   # loesche length Zeichen von xp bis xp+length*8-1 incl
   # Das 1. Zeichen steht an Position xp, xp * 8
   # Zeile ist an Position yp und ist 9 Pixel hoch
   # zu loeschende Pixelpositionen: b
   # Loeschen = Farbe 0
   self.display.text(text,xp,yp)
   self.display.show()
   #print("textuebergabe: {0}, at {1},{2}".format(text,x,y))
   return text
```

All three parameters are positional parameters and must be specified when calling up. s contains the string to be output. Numbers or byte objects must be converted into strings before being transferred or integrated into the transferred string using format commands.

If x or y are not in the permitted column or row frame, the output is aborted and None is returned, which is nothing. Otherwise the text actually output will be returned.

We determine the length of the string and the pixel positions for deleting the area in which the string should land. Then it is checked whether the string fits into the remaining columns from column x and we calculate the width b of the deletion rectangle = number of valid character positions times 8 pixels character width. If necessary, the string is deprived of characters that would go beyond the edge of the display (slicing). Then we delete the line area, send the moderated text to the framebuffer with display.text (), display the framebuffer with show (), output everything to the terminal for control purposes and return the moderated text. By comparing it with the argument when calling it, the calling program can see whether the text was output completely or only garbled.

The last method I'll discuss is pillar (). Pillar comes from the English language area and means column. That is also the task of this method, it is to draw a column on a diagram.

```
 def pillar(self,x,b,h):
   if x+b > self.width or b<=2 or x < 0: return None
   if h > self.height: h=self.height
   y=self.height-h
   self.display.fill_rect(x,y,b,h,1)
   self.display.show()
   return h
```

All three positional parameters that... must be specified correct. x is the position of the lower left corner, b the width and h the desired height of the column or, better, the rectangle that the column should represent.

Position x and width b are checked for plausibility and applicability. If the position is too far to the right or left or the width is too poor, nothing is drawn and None is returned.

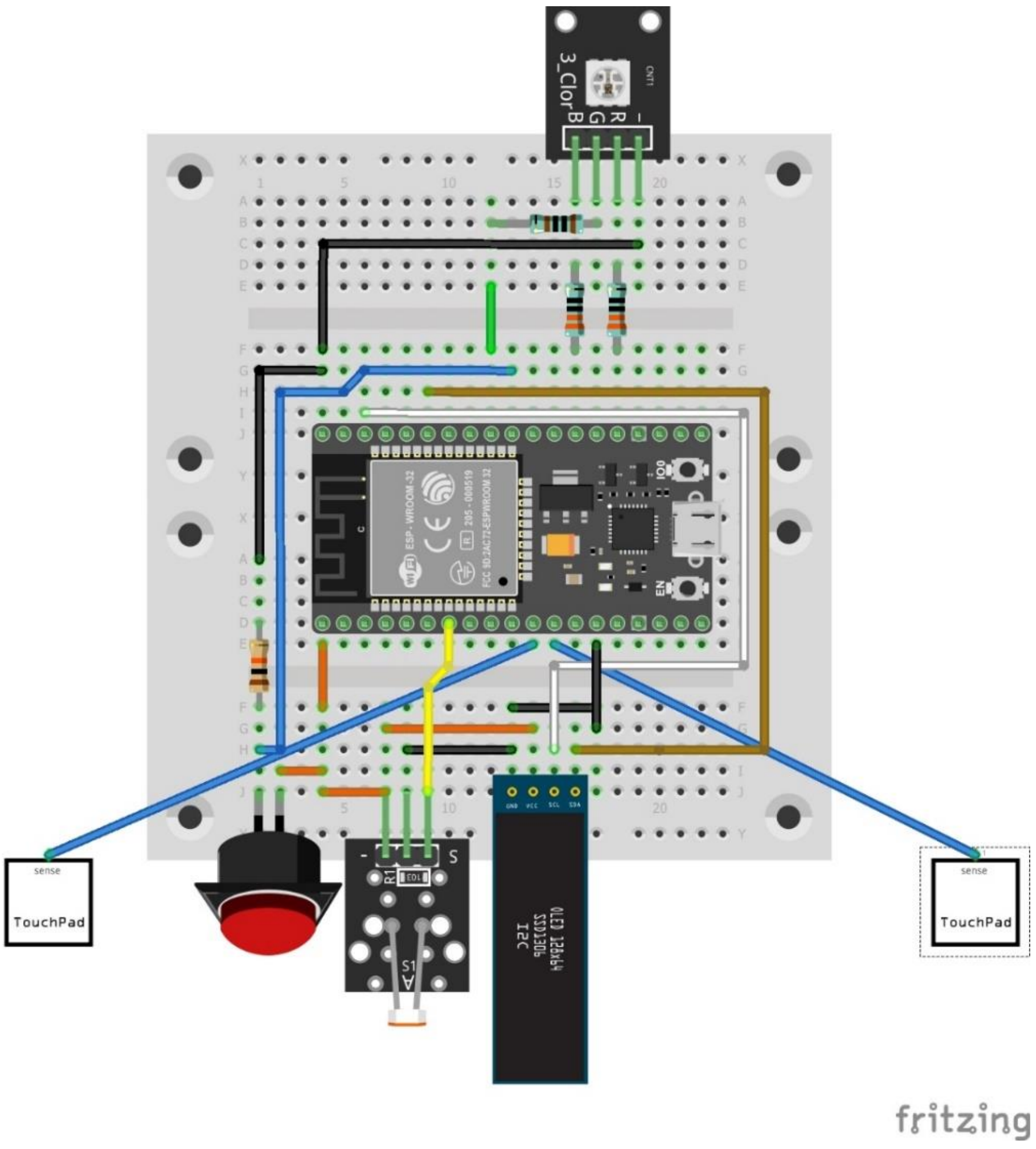
If necessary, the amount will be trimmed to the maximum amount. Then we have the correct bits set in the frame buffer, fill_rect () does that again and the show () method sends the data to the display.

The rest of the methods are so short and clear that an explanation is no longer necessary. You can download the oled.py file. A description of the graphics commands in FrameBuffer can be found in the MicroPython documentation.

Finally, we set the contrast of the OLED display with the help of two touchpads. Alternatively, we can record the ambient brightness with the LDR and use this value, adjusted accordingly, as a manipulated variable for automatic contrast control of the display, as known from smartphones. "By the way" you will learn how touchpads can be used very well as a replacement for mechanical buttons.

The touchtest.py program I use for this can be downloaded as a whole. Please make sure that the following files are in the device directory of the ESP.
touch.py or touch8266.py
oled.py
ssd1306.py
Now transfer touchtest.py there as well.

Now comes the hardware. The OLED display still has to be connected to the ESP8266 / ESP32, the two touchpads / buttons and the LDR are probably still connected to the ESP. However, the pictures show the parts and connections for this experiment.



Let's go the other way around and first look at what the program does. After that, let's examine how it workst.

Board selection
So that the program works for both ESP32 and ESP8266, the automatic selection of the board with the initialization of the pads or buttons right at the beginning.

Touchpad function test
The first part is a test of whether the touchpads are working properly. You can touch the pads or press buttons for ten seconds. The values read in are output on the terminal.

Program continuation after touch
The program waits up to 10 seconds for an action on the "Up" unit in order to continue the program.

Adjust the contrast using the pads
The "Up" unit increases the contrast value by 10 when pressed, the other, "Down", lowers it by 10. If no action is taken for 5 seconds, the program continues.

Finally, you can make the display darker or brighter by darkening or illuminating the LDR.

So start the touchtest.py program and try it out.

Now about how it works.

```
import sys
from machine import Pin, ADC
device = sys.platform
if device == 'esp32':
  from touch import TP
  up = TP(27)
  down = TP(14)
  ldr = ADC(Pin(32))
  ldr.atten(ADC.ATTN_11DB) #Full range: 3.3v
  ldr.width(ADC.WIDTH_10BIT)
elif device == 'esp8266':
  from touch8266 import TP
  up = TP(14)    # D5
  down = TP(12)  # D6
  ldr = ADC(0)
else:
  print("Unbekannter Controller")
  sys.exit()
from time import sleep, time
from oled import OLED

d=OLED()
```

We import various classes for operating the input units, for OLED, ADC and time control. The touchpad instances up and down on GPIO27 and 14 on the ESP32 or the button inputs on GPIO14 and GPIO12 on the ESP8266 as well as a display object d are generated. The analog connection is assigned and the same resolution is set for the ESP32 as for the ESP8266.

```
print ("touchpad function test")
i = 0
while i <10:
  print (up.getTouch (), down.getTouch (), "up - down", 10-i)
  sleep (1)
  i + = 1
```

Every second we read in the input values and send them to the terminal together with a countdown.

```
print ("Waiting for touch")
value = up.waitForTouch (10)
print (value)
```

Class TP facilitates program guidance by touching touchpads. Switches or buttons can also do this, but are subject to mechanical wear and tear and bounce. We still have to fall back on it with the ESP8266. Alternatively, you can use touchpad modules. In the material list I have listed one, the keypad-ttp224-14-capacitive.

But due to the careful programming of the touch.py and touch8266.py modules, only one test program is required for both controller families.

The most extensive part of the program is the manual brightness control.

```
k=32
d.setKontrast(k)
delay=5
schritt=10
start=time()
end=start+delay
current=start
while current < end:
  plus=up.getTouch()
  minus=down.getTouch()
  if up.threshold:
    plus = (plus < up.threshold)
  if down.threshold:
    minus = (minus < down.threshold)
  if plus:
    k=(k+schritt if k<255-schritt else 255)
    end=time()+delay
  elif minus:
    k=(k-schritt if k>schritt else 0)
    end=time()+delay
  d.setKontrast(k)
  print(k)
  sleep(0.2)
  current=time()
```

After setting the initial values, it goes into a time loop. The input units are queried. While for the ESP8266 the keys already deliver a Boolean result, 0 = False or 1 = True, in this case this must first be established for the fuzzy logic of the touchpad. So if the threshold value of the inputs is not None (= False), namely for the touchpads, the read-in value is converted to True if it was initially below the limit value, i.e. the pad was touched.

Then, depending on the input, the brightness is increased or decreased in steps of 10. The conditional expressions limit the range to 0 .. 255. If an entry has been made, the end time of the loop runs is also reset so that the setting can be made without stress.

Finally, the brightness is set to the last value, this is output and the current time is updated after the throughput speed is slowed down. If there is no more input for 5 seconds, the loop breaks off and the last part of the program with the query of the ADC value and its conversion to the range of the contrast value starts.

Let's summarize what you learned from this episode.
• You know more about the namespaces of variables, functions and classes.
• You can use timers in your programs
• You can use an OLED display like a conventional LCD.
• You have defined your own modules and classes and learned something about the memory usage of MicroPython
• You can estimate when it is better to import entire modules and when you should selectively import classes or individual members.
• You can compare objects and check their identity
• You have seen how touchpads or buttons can be used in conjunction with a module to control processes

In the next episode we will build the hardware for the final project. OLEDs, buttons, touch pads, LEDs and tweeters will be used again. And the system can be set up autonomously. This means that the ESP starts up automatically, allows some preparatory actions and then waits for the start of the measurement loop. What is being measured? - Stay tuned!

You can download this episode as PDF .

If you missed to read one of the older episodes, here are the links to them.
Folge 1
Folge 2
Folge 3

New homework

1. Let us assume that you make the following entries on the command line:

```
>>> from beep import BEEP

>>> BEEP. Duration = 5000


>>> BEEP. Duration

5000


>>> b = BEEP (2.13)

???

???

>>> b.beep ()

???
```

How does the ESP react to the last command? Can you justify the behavior?

2. Display the output in the touchtest.py file on the OLED display instead of on the terminal. When does that make sense, when not?

3. Measure the speed that can be achieved with the output on the terminal and on the OLED display.

4. Does it make a difference to the speed how many characters are output on the display and on the terminal per pass?

5. Use the pillar () command and the random number generator from the homework solution to create a bar chart made up of 10 bars that uses the full surface of a display as much as possible. Remember that there are displays of different widths.

6. Prepare two different outputs for the OLED display, for example a text screen and a bar chart. Have both displayed alternately for 3 seconds, while (at the same time!) The numbers from 1 to 30 are output continuously in the terminal according to the following scheme.

```
1

12

123

1234

12345

...
```

7. Modify the reaction tester from the homework solution in such a way that a different LED is sharp with each round. Of course you have to tell the player via the OLED which color it is.

8. Can you set it up so that the ESP32 and ESP8266 variants are kept in a new touch module and, depending on the controller type, the correct one is automatically used during import?

Lösungen der Hausaufgaben aus Teil 3

1. Write a sequence that checks the limit value parameter for valid information in the __init __ () method. The value must be an integer, positive, and less than 256.

A first dry run could look like this. You can create a program file or just use REPL.

Download: hausi1a.py

```
# hausi1a.py                                                  #1
grenzwert = 345                                               #2
grenzwert = int(grenzwert)                                    #3
grenzwert = (grenzwert if grenzwert >0 and grenzwert <256 else 64) #4
print("Als Grenzwert wird {} verwendet.".format(grenzwert))       #5

#1 Dateiname[, Version, Autor, Zweck]
#2 Grenzwert zum Test angeben
#3 Ganzzahligen Wert erzwingen
#4 Bereich abchecken
#5 Ergebnis mitteilen
```

Any specified floating point number is truncated by line 3 to form an integer.
The conditional expression in line 4 reassigns its value to the variable limit value if it is in the correct range, otherwise 64 is assigned.
Line 5 shows the result.

Download: hausi1b.py

```
# hausi1b.py                                                  #1
import sys                                                    #2
grenzwert = "abcd"                                            #3
try:                                                          #4
  grenzwert = int(grenzwert)                                  #5
except ValueError:                                            #6
  print("Falsche Angabe: {}\nBitte eine Ganzzahl angeben".format(grenzwert)) #7
  sys.exit()                                                  #8
grenzwert = (grenzwert if grenzwert >0 and grenzwert <256 else 64)  #9
print("Als Grenzwert wird {} verwendet.".format(grenzwert))         #10

#1 Dateiname[, Version, Autor, Zweck]
#2 Modul sys importieren fuer exit()-Funktion
#3 Grenzwert zum Test angeben
#4 Versuch der Umwandlung in einen ganzzahligen Wert
#5 Umwandlung
#6 Ausnahmebehandlung, falls der Versuch misslingt
#7 Fehlermeldung
#8 Programmabbruch
#9 Falls die Umwandlung erfolgreich war, wird der Exceptblock Ã¼bersprungen
#10 Meldung des Ergebnisses
```

The second variant catches an error if an attempt is made to convert a specification into an integer for which this is not possible, e.g. B. in the case of a string. An error message is then output and the program is aborted. You should run this example as a file so you can see how it goes.

Variant 3 also reports an error when the range is exceeded and stops.

Download: hausi1c.py

```
# hausi1c.py                                         #1
import sys                                           #2
grenzwert = 45.9                                     #3
try:                                                 #4
  grenzwert = int(grenzwert)                         #5
except ValueError:                                   #6
  print("Falsche Angabe: {}\nBitte eine Ganzzahl angeben".format(grenzwert)) #7
  sys.exit()                                         #8
if grenzwert <0 or grenzwert >255:                   #9
  print("Falsche Angabe: {} liegt nicht im Bereich zwischen 0(incl.) und
256(excl.)".format(grenzwert))
  sys.exit()                                         #11
print("Als Grenzwert wird {} verwendet.".format(grenzwert))        #12

#1 Dateiname[, Version, Autor, Zweck]
#2 Modul sys importieren fuer exit()-Funktion
#3 Grenzwert zum Test angeben
#4 Versuch der Umwandlung in einen ganzzahligen Wert
#5 Umwandlung
#6 Ausnahmebehandlung, falls der Versuch misslingt
#7 Fehlermeldung
#8 Programmabbruch
#9 Bereichspruefung
#10 Fehlermeldung
#11 Abbruch
#12 Meldung des Ergebnisses
```

Abschließend die Lösung für touch.__init__()

Download: hausi1.py

```
# hausi1.py
# Ausschnitt aus touch.TOUCH.__init__()
# Ueberpruefung des Grenzwerts
  # touch related methods
  # ************************************************
  def __init__(self, pinNbr, grenzwert=Grenze):
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)                              #3
    gw = (gw if gw >0 and gw <256 else 64) #4
    print("Als Grenzwert wird {} verwendet.".format(gw))        #5
    self.threshold = gw
```

2. Create a plausibility check in __init__, which checks the validity of the GPIO number for the touchpin before the object is created. To do this,

define a list of valid pin numbers. Use the keyword in to check whether the entry corresponds to a value in the list. Example for the command line:

```
>>> eingabe = 4
>>> touchliste = [15,2,0,4,13,12,14,27,33,32]
>>> eingabe in touchliste
True
```

Here is an example as a stand-alone test program:

Download: hausi2a.py

```
from machine import Pin,TouchPad
import sys
pinNbr=7
touchliste = [15,2,0,4,13,12,14,27,33,32]                    #7
if not pinNbr in touchliste:                                  #8
  print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
  sys.exit()                                                  #10
tpin = TouchPad(Pin(pinNbr))
print("Pin({}) als Touchpin eingerichtet".format(pinNbr))
```

The constructor method in touch.TP then looks like this:

Download: hausi2.py

```
# hausi2.py
# Ausschnitt aus touch.TOUCH.__init__()
# Ueberpruefung des Grenzwerts
```

```
  # touch related methods
  # ************************************************
import sys
  def __init__(self, pinNbr, grenzwert=Grenze):
    touchliste = [15,2,0,4,13,12,14,27,33,32]                  #7
    if not pinNbr in touchliste:                                #8
      print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
      sys.exit()                                                #10
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)
    gw = (gw if gw >0 and gw <256 else 64)
    print("Als Grenzwert wird {} verwendet.".format(gw))
    self.threshold = gw
```

```
# 7 List of valid touchpins
# 8 check if pinNbr does not appear in the list,
# 9 error message and
# 10 demolition
```

3. Can you change waitForTouch so that if you touch it within the runtime, instead of the value of the touchpad, the delay from the start of the method to the time of touch is returned?

For my solution, I took a step back and replaced the time measurement in seconds with one in milliseconds for task 4. The sample program also shows the use of a class directly in the program. Therefore the import of the class TP is not necessary here. The solutions to exercise 1 and 2 are already included in the class definition.

Download: hausi3a.py

```
from machine import Pin, TouchPad
from time import time, ticks_ms

class TP:
  # touch related values
  # Default-Grenzwert fuer Beruehrungsdetermination
  # **************************************************
  Grenze = const(150)

  # touch related methods
  # **************************************************
  def __init__(self, pinNbr, grenzwert=Grenze):
    touchliste = [15,2,0,4,13,12,14,27,33,32]                    #7
    if not pinNbr in touchliste:                                 #8
      print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr)) #9
      sys.exit()                                                 #10
    self.number = pinNbr
    self.tpin = TouchPad(Pin(pinNbr))
    gw = int(grenzwert)
    gw = (gw if gw >0 and gw <256 else 64)
    print("Als Grenzwert wird {} verwendet.".format(gw))
    self.threshold = gw

  # Liest den Touchwert ein und gibt ihn zurueck. Im Fehlerfall wird
  # None zurueckgegeben.
  def getTouch(self):
    # try to read touch pin
    try:
      tvalue = self.tpin.read()
    except ValueError:
      print("ValueError while reading touch_pin")
      tvalue = None
    return tvalue

  # delay = 0 wartet ewig und gibt gegf. einen Wert < threshold zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Touch bemerkt,
  # wird None zurueckgegeben, sonst ein Integer, der der Beruehrung enspricht
  def waitForTouch(self, delay):
    laufzeit = delay*1000
    start = ticks_ms()
    end = (start + laufzeit if delay >0 else start+10000)
    current = start
    while current < end:
      val = self.getTouch()
      if (not val is None) and val < self.threshold:
        return current-start
      current = ticks_ms()
      if delay==0:
        end=current+10000
    return None

t=TP(27)
zeit=t.waitForTouch(5)
print("{} ms".format(zeit))
```

The lines in bold must be changed or supplemented. The touchpad is connected to GPIO27.

4. Expand exercise 3 into a reaction test device by lighting up an LED at the beginning of the runtime, which is switched off again after touching it.

The last three lines from the hausi3a.py example are replaced by the following lines.

Download: hausi4a.py

```
ledG=Pin(18,Pin.OUT)
ledG.off()
t=TP(27)
sleep(4)
ledG.on()
zeit=t.waitForTouch(5)
ledG.off()
print("{} ms".format(zeit))
```

After a lead time of 4 seconds the green LED lights up (construction from part 3 with RGB LED). From then on the time runs until the touch.

Form of increase:
Let one of three LEDs light up randomly, whereby only one may be touched. If you type on the wrong LED, you will be degraded. The right touches are counted.

Replace the last three program lines from hausi3a.py as follows:
Download: hausi4b.py

```
import os
runden = 0
ledG=Pin(18,Pin.OUT)
ledG.off()
ledR=Pin(2,Pin.OUT)
ledR.off()
ledB=Pin(4,Pin.OUT)
ledB.off()
ledList=[ledG,ledR,ledB]          #60
```

```
t=TP(27)                              #61
testLed=2                             #62
for i in range(11):                   #63
  led = os.urandom(1)[0] & 0x03       #64
  led = (led if led != 3 else 2)      #65
  ledx =ledList[led]                  #66
  sleep(4)                            #67
  ledx.on()
  #print(led)
  if led==testLed:                    #70
    zeit=t.waitForTouch(3)
    if zeit and zeit < 500:
      runden += 1
      print("{} ms".format(zeit))
    else:
      print("Leider zu langsam")
  else:                               #77
    zeit=t.waitForTouch(2)
    if zeit:
      runden -= 1
      print("Falsch getouched, Punktabzug")
    else:
      runden += 1
      pass
  ledx.off()                          #84
print("Treffer: {}".format(runden))
```

To the function of the program:
# 60 The three LEDs are defined, I add the objects to the ledList list.
# 61, # 62 define touchpin and determine the "sharp" LED
Repeat # 63 10 times
# 64, # 65 Roll a random number 0 <= led <= 2.
# 66 Get the LED object from the list
# 67 Wait and LED on
# 70ff Release touchpad, measure time, possibly increase or comfort
# 77ff touched incorrectly, point deduction, otherwise increase score
# 84 Result of the test series

5. What happens using the ESP32 and the ESP8266 if you uncomment the last line in touchtest8266.py?

The recognized chip is reported in the terminal window, with each definition of the touchpad the limit value is output and that's it when you use an ESP32.

In the case of an ESP8266, an error is reported that the object t has no threshold attribute.**6. Finden Sie eine Möglichkeit, den Fehler aufzufangen, der beim Abfragen des Attributs t.threshold auftritt, wenn man das Modul touch8266 benutzt.**

Add the following line to the touch.TP .__ init __ ():

```
self.threshold = None
```

So that even if you

```
print (t.threshold)
```

use, None, nothing at all, will be output. But there is also no error message and this solution has the side effect that you can use the threshold attribute in the program to query at any time whether buttons or touchpads are initialized. Incidentally, this is not just a matter of the controller type, because touchpads can of course also be replaced by buttons on the ESP32.