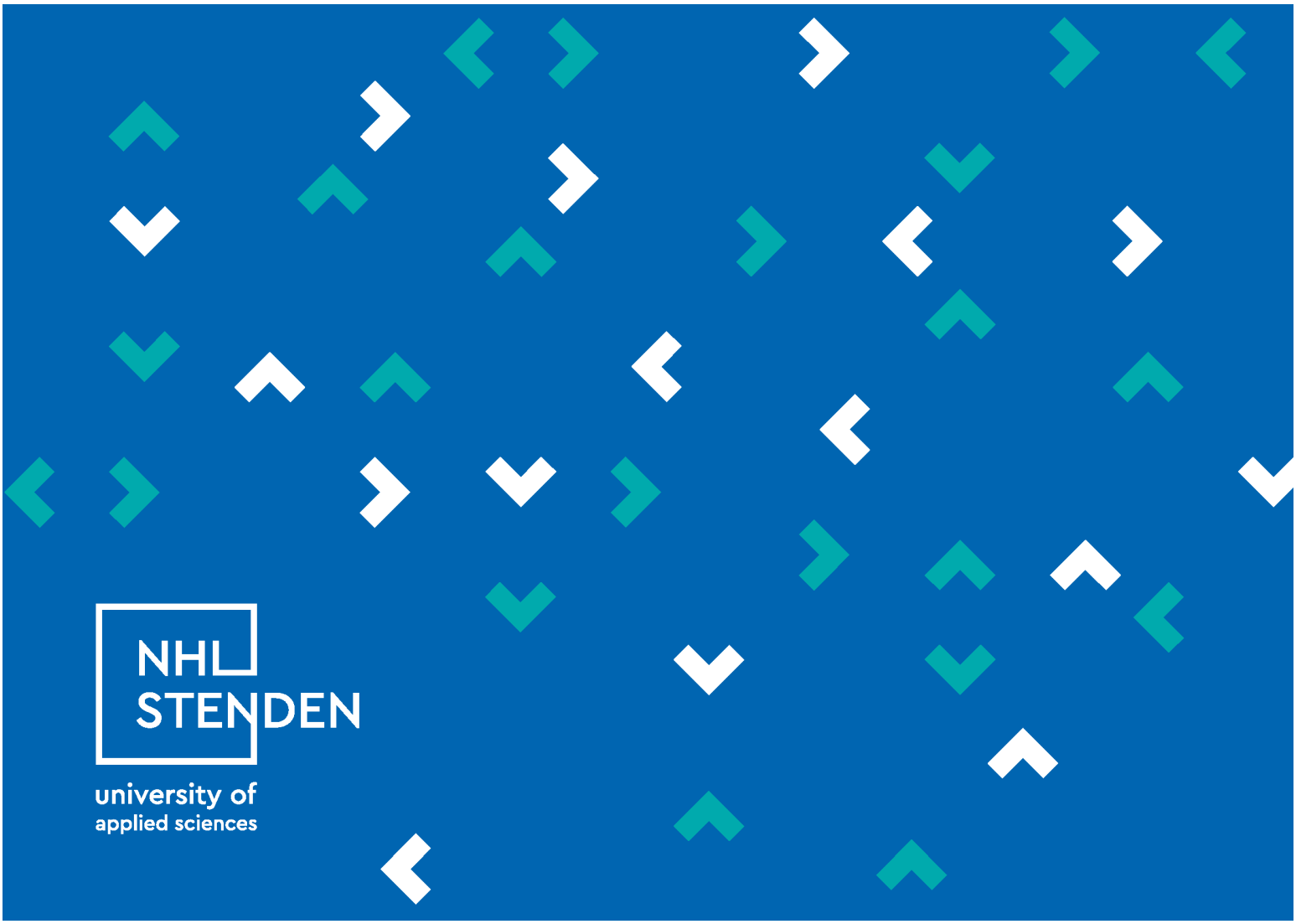
Programmeren 2

Huiswerkopdrachten

Ad IA&R

Rolink, S.



PUBLISHED BY NHL STENDEN

NHLSTENDEN.COM

Eerste editie, November 2021

Inhoudsopgave

Deel 1: Arduino & C/C++

1. Week 1: Webserver

1.1 Ethernet Shield

We beginnen dit hoofdstuk met de laatste opdracht van *Programmeren 1*, de "uitsmijter" van de introductie in embedded systems en embedded programming: de Arduino met IO-webserver.

inleiding

Als onderdeel van de Arduino software worden diverse software-componenten meegeleverd in de vorm van software-bibliotheken (*libraries*) (Je kunt ze onder Windows vinden onder: *MijnDocumenten -> Arduino -> libraries*). Controleer dat. Standaard wordt er, bijvoorbeeld, een servo-library meegeleverd waarmee het werken met servo's sterk wordt vereenvoudigd. Deze libraries kun je in je code gebruiken door ze zogeheten te *includen*, dit doe je door bovenaan in je code de volgende regel toe te voegen:

```
1 #include <NaamVanLibrary.h>
```

Daarna kun je alle handige functies die in de library zitten gebruiken in je eigen code. Op de volgende manier kun je bijv. voor de ethernet-shield een webserver worden geconfigureerd, door gebruik te maken van de *Ethernet*-library:

```
1  #include <Ethernet.h>
2
3  byte mac[] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED};
4  IPAddress ip(192, 168, 1, 3);
5
6  EthernetServer server(80); // start http server on port 80
7
8  void setup()
9  {
10   Ethernet.begin(mac, ip);
```

```
11
12    server.begin();
13 }
```

Om daadwerkelijk met de ethernet-shield een connectie te krijgen, is de library wat informatie van je nodig. Op regel 6, wordt een *EthernetServer* 'object' aangemaakt met de naam *server* en deze krijgt ook het getal 80 mee. Dit is het poort-nummer waarmee je gaat werken, gezien dat we iets met *http* ('het internet') gaan doen, is dit 80 (deze poort is daarvoor gereserveerd). Voor meer informatie over dit *http* protocol *http://betterexplained.com/articles/a-simple-introduction-to-computer-networking/*.

Je router (die alle verkeer in je netwerk beheert) heeft daarnaast informatie nodig van het apparaat zodat deze data daar naartoe kan sturen. Vergelijkbaar met je woonaddress, heeft zo'n apparaat ook een addres, twee zelfs! Een MAC-address (die vaak bestaat uit 6 hexadecimale getallen), die uniek moet zijn voor elk apparaat in het netwerk. Vaak worden de ethernetshields geleverd met een sticker waar een uniek (gereserveerd) MAC-address opstaat, mocht dit voor jou nu niet het geval zijn, vul dan iets willekeurigs in (we sluiten de Arduino enkel aan op het lokale netwerk, dus de kans dat er in dat kleine netwerk een dubbele voorkomt is nagenoeg nihil).
Het IP-adres bestaat uit 4 getallen, vaak in de trant van 192.168.1.X (of bijv. 10.0.0.X), waarbij X het computernummr is, wat een uniek getal moet zijn in je lokale netwerk. Deze addressen worden gebruikt om een connectie te maken tussen apparaten in het lokale netwerk. Beide worden dan ook 'meegegeven' bij het aanroepen van de *begin*() functie van de *Ethernet*-library, deze zorgt dat alle nodige configuratie verder wordt uitgevoerd om met het ethernetshield aan de slag te gaan.

1.1.1 Opdrachten

Opdracht 1.1
Plaats de Arduino ethernetshield op de Arduino UNO en download het bijbehorende programma (Arduino script, iowebserver.ino) via blackboard en laadt het in je Arduino. ■

Let op: als je het ethernetshield op je Arduino aansluit, zijn de pins 10 - 13 niet meer te gebruiken voor andere doeleinden.

Opdracht 1.2
Sluit je ethernetshield aan op je laptop of (beter nog) op een router, waar uiteraard ook je laptop aan hangt (bedraad of draadloos).
Onderzoek wat het IP-adres van je laptop is en noteer dit.
(Merk op dat je laptop naast de bedrade aansluiting ook nog verbonden kan zijn met het internet via de draadloze verbinding. Dat is handig, want dan kun je ondertussen stackoverflow raadplegen.)

Een directe verbinding tussen twee systemen noemt men een ad hoc-netwerk. In beide gevallen, ad hoc of via een router, is het verstandig de netwerkadressen te kiezen in het segment 192.168.1.x, waarbij x het computernummer is. Het voordeel van dit segment is dat het strikt lokaal is. De informatie van computers in dit segment worden niet doorgegeven over het internet.
■

Opdracht 1.3
Bestudeer het webserver-programma en onderzoek of je webserver een IP-adres krijgt en zo ja, welke?
Gebruik de Serial Monitor. Noteer dit adres ook en vergelijk het met dat van je laptop.
Zitten beide systemen in hetzelfde netwerk(segment)?

De toekenning in de Arduino kan statisch of dynamisch, via DHCP. Welke gebruik je?
Zorg er hierbij voor dat je in een gecontroleerde netwerkomgeving werkt, met bijvoorbeeld een eigen router waarvan je de instelling kunt aanpassen.
Op het NHL-netwerk hebt je weinig vat, wat aanleiding kan vormen tot problemen (met firewalls etc.). Bij de statische configuratie bepaal je zelf wat het IP-adres is. Je kunt het (in Windows) aanpassen via het configuratiescherm.
Een nadeel van een statisch IP is dat je, in een netwerk met meerdere computers, conflicten kunt krijgen. Beter is het dan ook je Arduino via DHCP een IP-adres te laten toekennen. Dit wordt ondersteund door de serversoftware. ■

N.B. Voor de rest van deze opgave gaan we uit van de volgende configuratie: de laptop krijgt adres 192.168.1.2 en de Arduino(server) 192.168.1.3. Dat kan bij jou dus, zeker als je DHCP gebruikt, anders zijn.

Probeer, nu je begrijpt hoe eenvoudige netwerkcommunicatie werkt, de Arduino file *webserver.ino* te downloaden van BlackBoard en in de Arduino te ladan. Als dit is gelukt, is de webpagina, via je browser zichtbaar te maken door de volgende URL in de voeren:
http://192.168.1.3
Zie figuur 1.1 voor hoe de site eruit hoort te zien. Mocht dit niet gebeuren en je weet zeker dat alle IP-adressen kloppen, lees dan verder bij de opmerking onderaan deze opdracht.

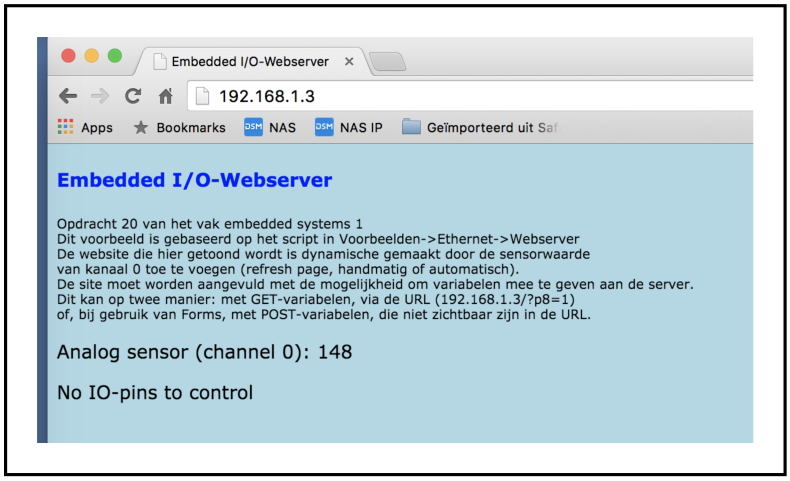


Figure 1.1: Webserver

Opdracht 1.4 Sluit een sensor aan op je Arduino (bijvoorbeeld een lichtsensor of een potmeter op kanaal *A*0) en modificeer je programma zodanig dat je uitsluitend de sensorwaarde, en niet de hele string “Analog sensor (channel 0): ”, bold (vetgedrukt) afdrukt en kleur deze daarnaast rood

indien deze boven de 600 komt. Onderzoek de mogelijkheid om dit automatisch te herhalen elke 3 seconden. *Hint:* Meta Tag.
Gebruik zo nodig:
http://www.w3schools.com/htmlhttp://www.w3schools.com/cssref/css_colornames.asp ■

1.2 Hoofdopdracht: Webserver

Opdracht 1.5 Pas je programma zodanig aan dat je via een zogeheten 'GET-variabele' (als onderdeel van je URL) de server opdracht kunt geven de afzonderlijke LED's aan de pinnen $p8$ en $p9$ aan en uit kunt zetten.

Daarmee bedoelen we dat het aanpassen je je gebruikte url (het IP-adres van de Arduino):
http://192.168.1.3/
naar bijv:
http://192.168.1.3/?p8=1
Waarmee de LED aan pin 8 aan wordt aangezet. Overeenkomstig kan met
http://192.168.1.3/?p9=0
de LED aan pin 9 worden uitgezet (terwijl die aan pin 8 natuurlijk aan blijft).

Zoek onderin je code naar de functie *parseHeader*(), hier moet de URL van hierboven worden verwerkt (*geparsed*).

```
bool parseHeader(String header, int &a, int &v)
{
   // your code goed here
}
```

Vul deze functie in zodat deze de url (na het vraagteken) splits in een argument (in ons geval: $p8$ of $p9$) en een waarde (0 of 1). Sla het argument op in a en de waarde in v.

N.B. De parameters a en v zijn uitvoer-parameters (of reference parameters). Een toekenning, binnen de functie, van een waarde aan één van de beide parameters leidt tot een aanpassing van de argumenten (variabelen) bij aanroep. Kortom: verander je a, dan verandert arg dus mee. Dat is de essentie van *call by reference*.

Meld de gebruiker (via de website) de veranderingen in de status van de beide pinnen, ook als er zich fouten voordoen. ■

1.2.1 Verdieping (optioneel)

Opdracht 1.6 Plaats een SD-card in je ethernetshield en laadt de webpagina van SD-card. ■

Opdracht 1.7 Breng de Arduino ter sprake als de gesprekken, thuis aan tafel of in de kroeg stilvallen. Doe dit met dermate veel enthousiasme dat vriendinnen, jongere broertjes en neefjes ook informatica willen gaan studeren ;) ■

1.3 Optionele opdracht: Webclient

In de vorige opdracht hebben we gezien hoe de Arduino tijdelijk is omgetoverd tot een eenvoudige webserver waarmee, via het http-protocol, sensoren uitgelezen kunnen worden en hardware bestuurd kan worden. In deze opdracht gaan we de rollen omdraaien. We gaan met de Arduino, wederom via http, gegevens van een (externe) website of webserver opvragen. Dit gaat via een http request waarbij de Arduino nu in de rol van client optreedt (vergelijkbaar met wat een browser doet).
We gaan ook onderzoeken hoe snel die gegevensoverdracht eigenlijk is (in termen van Bytes/s).

De basis voor deze opdracht komt van de Arduino-site:
https://www.arduino.cc/en/Tutorial/WebClient
Dit voorbeeld maakt duidelijk hoe je een http request uitvoert met gebruik van een ethernet shield en daarbij gegevens opvraagt van een google server.

Opdracht 1.8 Download de code van deze website en sla deze op onder de naam *WebClient.ino*. Bestudeer de code. Enkele opmerkingen daarover:

```
// Set the static IP address to use if the DHCP fails to
   assign
IPAddress ip(192, 168, 0, 177);
IPAddress myDns(192, 168, 0, 1);
```

Pas deze gegevens aan aan je eigen (thuis)situatie. ■

N.B. *Let op*: in de code wordt in eerste instantie geprobeerd een IP-adres te verkrijgen via DHCP. Lukt dat niet dan wordt teruggevallen op een statisch IP-adres. Dat geldt ook voor de optie om met DNS te werken zodat je niet via het IP-adres een google-site benadert, maar via de URL. Je kunt de DNS van je router gebruiken, dat is (doorgaans) de gateway naar je provider. Bij het verkrijgen van een IP-adres kan het programma nog onderscheid maken tussen diverse foutsituaties, zoals die waarbij er geen shield of geen ethernet kabel is aangesloten. Handig! Helaas wordt dat niet door elke ethernet-chip ondersteunt, wat bij correcte aansluiting verder geen problemen oplevert.

```
EthernetClient client;
```

De variabele *client*, via welke het verdere interface verloopt is van het type *EthernetClient*. Deze is bekend, en daardoor oranje in de code, via het include statement, bovenin het programma.

```
Ethernet.begin(mac, ip, myDns);
```

N.B. *Let op*: *Ethernet* is feitelijk de abstractie van de shield. Via dit object hebben we toegang tot de shield. Uiteraard heeft dit object methoden, zoals *begin*(), waarmee de shield wordt geïnitialiseerd. *begin*() heeft meerdere verschijningsvormen of interfaces. Eén met alleen een mac-adres of één met een mac-adres en een IP-adres. De hier gekozen variant gaat uit van een DNS-adres zodat de server via een URL kan worden benaderd. Voor meer informatie, zie bij:
https://www.arduino.cc/en/Reference/EthernetBegin

Eenmaal (goed) geïnitialiseerd kan het lokale IP-adres worden afgedrukt met:

```
Serial.println(Ethernet.localIP());
```

Daarna wordt er gepoogd een verbinding met een externe server te maken, met:

```
if (client.connect(server, 80))
```

Merk op dat connect een methode is van het client object, dat server een argument is dat hieraan meegegeven wordt en dat server , i.v.m. de DNS-optie, hier een URL is. Dat is hier een (statische) string. Zie bovenin het programma. Je kunt, via out-commenting, ook kiezen voor een 'hard' IP-adres, als de DNS-optie niet werkt.
Eenmaal verbonden met de server, kun je het IP-adres, nu van de server, opvragen en printen met

```
Serial.println(client.remoteIP());
```

Daarna voert het programma zijn taak uit. Dat begint met:

```
client.println("GET␣/search?q=arduino␣HTTP/1.1");
```

waarin, vanuit de setup code, de feitelijke http request wordt gedaan, waarna de bijbehorende http response, in de loop code, wordt verwerkt. In dit geval betekent dat, dat zolang de *client.available*() is, het lezen van de reactie van de server, met een mogelijkheid tot printen en het tellen van het aantal gelezen bytes, zodat, op grond daarvan en van een tijdcriterium, kan worden vastgesteld wat de overdrachtssnelheid is. Merk op dat voor het meten van de leestijd de functie *micros*() wordt gebruikt. Deze snelheid wordt doorgaans uitgedrukt in MB/s, maar vanwege de relatieve traagheid van de Arduino hier in kB/s. Let op (alweer), dat het printen van de response een grote invloed heeft op deze snelheid. Beredeneer waarom dat zo is? Voor een betrouwbaarder meting is het dan ook raadzaam het printen uit te zetten. Oh ja, misschien is dit een goed moment om de baudrate van de Arduino (*Serial.begin*) aan te passen van 9600 bps (bits/s) naar 115200 bps, zowel in de code als in het serial monitor. 115200 bps komt neer op ongeveer 11000 tekens per seconde. Dat kan de Arduino wel aan, maar jij, wat betreft leessnelheid, waarschijnlijk niet ;) Voer de code uit en stelt vast wat de google-server terugstuurt (bij zoeken op de term "Arduino") en wat de overdrachtsnelheid hierbij is.

Het programma is in deze vorm slechts geschikt voor één taak. Vaak is het realistisch dat er meerdere requests gedaan moeten worden, bij meerdere servers. Daarover straks meer. Om dat voor te bereiden is het nodig om het request en de response in één functie uit te voeren.

Opdracht 1.9 Modificeer het programma zodanig dat deze functionaliteit wordt uitgevoerd door de functie *speedTest*, waarvan deze functie-header wordt gebruikt:

```
void speedTest(bool doPrint = false);
```

Een mogelijke aanroep van deze functie is speedTest()waarbij het printen standaard uitstaat. Hier wordt gebruik gemaakt van de mogelijkheid een parameter in de heading een initiële waarde te geven. De koppeling van de waarde aan de variabele vindt plaats tijdens de aanrop. Een andere mogelijkheid is deze:

```
#define PRINT true // let op de conventie van hoofdletters
speedTest(PRINT);
```

waarmee bij aanroep bepaald kan worden of de speedtest-functie wel of niet de response laat zien. Defines worden meestal helemaal boven in het programma opgenomen, in hoofdletters.

Wat moet de functie precies doen? Vier deelfuncties:

1 verbinding met de server maken
2 uitvoeren van de http request (GET)
3 het verwerken van de respeonse van de server
4 het afhandelen van de timing en de speed-berekeningen

■

Opmerkingen:

- De eenmalige handelingen voor het initialiseren van het ethernet shield blijven,logischerwijs, in de setup code.
- Alle variabelen gerelateerd aan deze functies worden uiteraard lokaal gehouden, dus binnen de functie gedefinieerd. Dat geldt voor: *beginMicros*, *endMicros*, *byteCount* en ook de $buffer$ en len.
- Dit klinkt als een eenvoudige opdracht (en dat klopt, het is ook niet zo moeilijk), maar goed programmeren begint met een strikte scheiding in functies, het parameteriseren daarvan (zodat bij de aanroep het gedrag van de functie beïnvloed kan worden) en met met het vermijden van globale variabelen. Er zijn nog wel meer criteria te geven, maar voor nu is dit wel even voldoende.
- Nog een belangrijk punt van aandacht: het bestaande programma maakt gebruik van het herhalende karakter van de functie loop. Dat raken we (bewust) kwijt door deze aanpassing. Loop blijft hier voorlopig leeg. Dat betekent dat je zelf een oplossing moet bedenken de noodzakelijke herhaling bij het lezen van de response. Dat gaat namelijk in regel (van max. 80 tekens).
  Een ander punt daarbij is: wanneer is het lezen klaar? Wanneer is de gehele response van de server verwerkt? Daarbij doemen ook de vragen op, als: wanneer wordt de verbinding verbroken en wie is daarvoor verantwoordelijk? Hint:
  - kijk naar $client.connected()$
  - overweeg dit te gebruiken: $while(buflen = client.available() > 0)$ Dat mag in C: het toekennen van het functieresultaat aan een lokale variabelen en het (bijna) tegelijkertijd testen van deze variabelen, in dit geval op groter dan 0, waarmee wordt aangegeven dat er nog steeds iets te lezen valt.
  - zet om de read-actie even een digitalWrite, zodat je een visuele feedback krijgt in de vorm van een knipperend ledje.

Opdracht 1.10 Test je programma op juiste werking. Welke speed haal je? ■

1.3.1 Verdieping (optioneel)

Opdracht 1.11 parameterisering, zeker bij klassiek, gestructureerd programmeren in een functionele taal, een sterk middel om je programma overzichtelijk en flexibel te houden. Ook het streven naar zo generiek mogelijke functies en oplossingen hoort daarbij. Ontwerp en functie, bijvoorbeeld httpRequest, die generieker is en waarbij je niet alleen print-criterium meegeeft, maar ook de url en de server en het antwoord, bijvoorbeeld als JSON-string, laat opleveren als functieresultaat. ■

Opdracht 1.12 http request zijn erg handig om data of informatie van verschillende servers te halen en te verwerken in je programma. Onderzoek de mogelijkheid om dat te doen voor bijvoorbeeld weer-informatie (bij *openweathermap.org*) of nieuws (bij *newsapi.org*). Je kunt deze gegevens, in een embedded systeem, tonen op bijvoorbeeld een displaytje. ■

Deel 2: Raspberry Pi & Python

2. Week 2: Introductie Python

> *And now for something completely different.*

John Cleese, *Monty Python*, 1975

We stappen nu een compleet nieuwe programmeer-wereld in, namelijk die van *Python*. *Python* is een programmeertaal van hogere orde ten opzichte van *C*. Hoe hoger de orde van een programmeertaal, des te verder de taal weg staat van daadwerkelijke machine-instructies.
Python is een favoriete taal voor menig programmeur, omdat het redelijk makkelijk te leren is, de code compact en leesbaar is, voor een enorm scala aan projecten in te zetten is (van websites en kunstmatige intiligentie tot scripts en Raspberry Pi's), en de enorme gemeenschap die de taal gebruikt en ook verder uitbreidt d.m.v. softwarebibliotheken (hierover later meer).

Zoals altijd komen al deze voordelen ook met enige nadelen (waar omheen gewerkt kan worden). *Python* is bijv. een stuk langzamer dan *C* (afhankelijk van de toepassing zo'n 2 tot 100 keer), vandaar dat we met dit vak afstappen van de inmiddels bekende *Arduino* en overstappen naar de *Raspberry Pi* en je eigen laptop/PC.

2.1 Installatie

Opdracht 2.1 Installeer de laatste versie van Python op je eigen laptop via:
https://www.python.org/downloads/
Let er bij het installeren op dat je alle vinkjes aanzet (vooral *pip* is een belangrijke). ■

2.2 Interpreter

Nu dat de installatie van *Python* is voltooid is het tijd om te checken of alles goed is gegaan. Open een *shell prompt* (onder windows: startmenu -> zoeken naar *cmd* of *powershell*, onder linux en

mac: open de *terminal*). Dit is een interactieve scherm waarmee je via tekst commando's kunt uitvoeren. De *Python* interpreter is nu één van die commandos, typ *python3* en druk dan op enter. Als het goed is zie je iets vergelijkbaars als in Figuur 2.1.

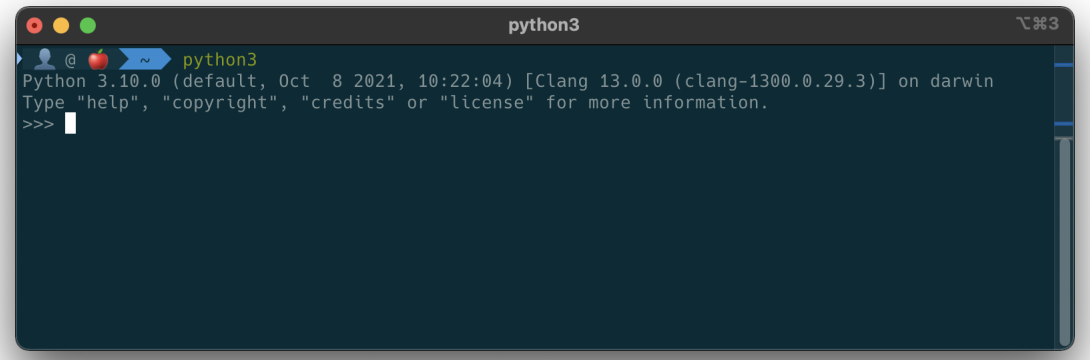


Figure 2.1: De *Python* interpreter, met versie *3.10.0*

Opdracht 2.2 Check of de versie van *Python* die je geïnstalleerd hebt overeen komt met de versie die de interpreter aangeeft. ■

De interpreter is een super handig stukje software, waarin je *Python* code, regel voor regel in kunt typen, zodat je kunt checken of de code doet wat je verwacht.

Opdracht 2.3 Traditiegetrouw begint het leren van een nieuwe programmeertaal altijd met uitvoeren van *Hello world!*. Typ in het scherm het volgende in, en zie wat er gebeurt:

```
1 print('Hello World!')
```

Hoeveel regels code zou je hier nodig voor zijn bij de Arduino? ■

N.B. Als er in dit document een stukje voorbeeld code, de regel begint met '>>>', dan wordt deze code ingetypt in de interpreter. De opvolgende regel zonder '>>>' is dan de uitvoer van de interpreter. Bijvoorbeeld:

```
1 >>> print('Hello World!')
2 Hello World!
```

Elk ander stukje voorbeeld code zullen (gedeeltes van) scripts zijn.

2.3 Editor

Elke regel op deze manier intypen is natuurlijk voor grotere programma's niet ideaal, deze slaan we dan ook op in *Python* scripts (bestanden met als extensie *.py*). Deze kun je in principe gewoon maken in een tekst-editor, maar net als bij de *Arduino* werkt het een stuk fijner in een ontwikkelomgeving. Voor *Python* zijn er een flink scala aan zulke ontwikkelomgevingen (ook wel: IDE's). En je bent uiteraard helemaal vrij om zelf je favoriet daarin te kiezen om de opdrachten gedurende dit vak uit te werken.

Voor de beginners kan ik de IDE *Thonny* aanraden: *https://thonny.org/*.
Deze heeft naast een gebruikersvriendelijke interface, ook enkele handige tools aan boord die later van pas gaan komen. Bijkomend voordeel is dat dit programma ook standaard wordt meegeleverd met de *Raspberry Pi*. Tijdens de les zullen de voorbeelden ook worden gegeven aan de hand van *Thonny*.

Opdracht 2.4 Installeer een *Python* ontwikkelomgeving. ■

2.4 Data Types

Net als in *Arduino C*, zijn in *Python* verschillende datatypes beschikbaar, die de basis vormen van de taal. Zo zijn er gehele getallen (vergelijkbaar met *int*s in *C*):

```
>>> x = 3
>>> y = 4
>>> x + y
7
```

Wat direct opvalt, is dat je in *Python* dus niet hoeft aan te geven welk data type je wilt gebruiken. De taal 'snapt' dit automatisch. De puntkomma aan het eind, die bij *C* elke regel code afsluit, wordt hier niet gebruikt. Verder kun je (op vergelijkbare manier als in *C*) de getallen ook schrijven in binair en hexadecimaal:

```
>>> x = 0b101
>>> x
3
>>> y = 0x1A
>>> y
26
```

Naast gehele getallen zijn er ook uiteraard kommagetallen (vergelijkbaar met *float*s in *C*):

```
>>> x = 3.0
>>> y = 4.5
>>> x / y
0.6666666666666666
```

Ook kent de taal Booleans, die enkel `True` of `False` kunnen zijn. Waarmee digitale operaties kunnen worden uitgevoerd (vergelijkbaar met de *bool*s uit *C*):

```
>>> a = True
>>> b = False
>>> a and not b
True
```

en voor nu als laatste: `str`, de *string* waarmee tekst kan worden toegekend aan een variabele. (Vergelijkbaar met de *String*s uit *C*):

```
>>> a = 'Hello'
>>> b = "World!"
>>> a + ' ' + b  # Plak beide aan elkaar, met een spatie ertussen.
'Hello World!'
```

Het maakt bij `str`s dus ook niet uit of je nu enkele (`'`) of dubble (`"`) aanhalingstekens gebruikt. Verder zie je hier ook voor het eerst een regel commentaar, die je kunt toevoegen na een hekje (`#`).

Opdracht 2.5 *Python* heeft een aantal operatoren voor getallen die bekend voor zullen komen: `+`, `-`, `*` om resp. getallen op tellen, af te trekken en te vermenigvuldigen. Maar kent ook een aantal die wat minder voor de hand liggen. Probeer er achter te komen wat `%` en `**` doen. ■

Naast dat het bij *Python* dus niet nodig is om van te voren te specificeren welk datatype je gaat gebruiken voor een variabele, kun je ook variabelen van type laten veranderen gedurende het programma. Dit is voor de leesbaarheid niet aan te bevelen, maar het laat wel zien dat *Python* veel losser om gaat met de verschillende types dan dat we gewend waren met *C*:

```
1 >>> x = 12  # x is nu een int
2 >>> x
3 12
4 >>> x = 'abc'  # En vanaf hier een str
5 >>> x
6 'abc'
```

2.5 If/Else statements

Een van de meest voorkomende en belangrijke dingen die er gebeurt in een programma, is keuzes maken op basis van condities. In z'n meest eenvoudige manier, doen we dat met een `if`-statement:

```
1 a = 200
2 if a > 100:
3   print('conditie is True!')
4   print('a is groter dan 100')
```

De `if`-statement is opgebouwd uit 3 delen: allereerst het woordje `if`. Dan gevolgd door een conditie (ook wel expressie genoemd), en uiteindlijk een dubbele punt `:`. Die conditie is het meest interessant, als daar namelijk `True` uitkomt worden de volgende regels code uitgevoerd, komt er `False`, gebeurt er in dit geval niets.

N.B. **Let op:** Ook hier is de schrijfwijze van *Python* weer een stuk compacter dan in *C*: waar je bij die laatste vaak gekrulde haken {, } ziet, geruikt *Python* juist inspringing om duidelijk te maken wat bijv. bij een statement hoort. Voor het inspringen gebruik je 4 spaties of een tab. Dit is dus **essentieel** voor de werking van je programma! Zie bijv. het volgende voorbeeld:

```
1 a = 200
2 if a > 100:
3   print('conditie is True!')
4 print('a is groter dan 100')
```

Hier wordt dus regel 3 enkel uitgevoerd als `a>100` is, maar regel 4 altijd. Let dus goed op bij het inspringen van code!

De `if`-stament kan uitgebreid worden met een `else`-clausule, die uitgevoerd wordt als de conditie `False` is.

```
1 a = 200
2 if a > 100:
3   print('conditie is True!')
4   print('a is groter dan 100')
5 else:
6   print('conditie is False!')
7   print('a is kleiner of gelijk dan 100')
```

Tip: Als je niet zeker van jezelf bent wat er precies uit de conditie komt van je `if`-statement, kun je hier dus handig gebruik maken van de interpreter:

```
1 >>> a = 200
2 >>> a > 100
3 True
```

Soms wil je meerdere condities checken, dit kan met `elif` (dit staat voor een samentrekking van `else if`). Je kan daarvan zoveel als je wil in je `if`-statement stoppen:

```
1 a = 200
2 if a > 100:
3   print('a is groter dan 100')
4 elif a == 100:
5   print('a is precies 100!')
6 else:
7   print('a is kleiner dan 100')
```

Opdracht 2.6 Typ het bovenstaande over in je editor, sla het op als een *.py* bestand. Probeer het programma te starten (In *Thonny*: Zoek naar de groene play knop, genaamd 'run'). Waar kun je de uitvoer van het programma zien? ■

2.6 Input/Output

Zo'n programma als hierboven, die afhankelijk van de waarde van een vast getal iets uitvoert is natuurlijk niet superspannend. Het wordt al interessanter als je input van de gebruiker van het programma kunt vragen. Dat doe je met de functie `input()`:

```
1 print('Typ iets: ')
2 a = input()
3 print('Je typte: ' + a)
```

N.B. **Tip:** Regel 1 en 2 zijn ook te combineren tot `a = input('Typ iets: ')`.

Opdracht 2.7 Voer het bovenstaande stuk code uit. Merk op dat halverwege het programma wordt gepauzeerd. Wat moet je doen om het programma weer verder te laten gaan? ■

N.B. **Tip:** In plaats van regel 3, kun je ook gebruik maken van de zogeheten *f-strings*. Dit is een wat efficiëntere manier van schrijven, maar doet precies hetzelfde:

```
1 print(f'Je typte: {a}')
```

Vooral bij langere wat langere strings, kan dit de leesbaarheid van je code bevorderen.

2.7 Huiswerkopdrachten

Opdracht 2.8 Maak een programma dat de gebruiker om zijn naam vraagt. Groet hierna de gebruiker in de vorm van *"Hallo, <naam>!"*. ■

Opdracht 2.9 Breid het programma van opdracht 2.8 uit, zodat het programma je schreeuwend begroet: *"HALLO, <NAAM>!"*.
Tip: om een string naar hoofdletters om te zetten, gebruik `naam.upper()`-functie. ■

Opdracht 2.10 Schrijf een programma dat de gebruiker vraagt om de straal (r) van een cirkel in te typen. Geef daarna de omtrek ($= 2 * r * \pi$) en de oppervlakte ($= r^2 * \pi$) op basis van de straal.

Tip 1: Alles wat de gebruiker intypt, en je afvangt met `input()` is van het type string. Om van een string met cijfers, een daadwerkelijk getal te maken, gebruik je de functie `int()`. Bijv. `x = int('123')`.

Tip 2: Zet bovenin je programma `import math`, je hebt daarna toegang tot allerlei wiskundige functies. Zo ook `math.pi` en `math.pow()`. ■

N.B. **Tip:** Als je niet zeker van jezelf bent wat voor type een variabele krijgt, kun je ook hier handig gebruik maken van de interpreter en de functie `type()`:

```
1 >>> a = 'een string'
2 >>> b = 4
3 >>> type(a)
4 <class 'str'>
5 >>> type(b)
6 <class 'int'>
```

Opdracht 2.11 Breid het programma van opdracht 2.8 uit, zodat het programma de gebruiker vraagt om achtereenvolgens z'n voornaam, achternaam, geslacht en leeftijd.
Groet hierna de gebruiker in de vorm *"Hallo, <voornaam>!"*.
Is de persoon in kwestie echter ouder dan 50, gebruik dan een formelere begroeting op basis van zijn of haar geslacht: *"Gegroet, Mr. <achternaam>!"* of *"Gegroet, Mevr. <achternaam>!"* ■

Opdracht 2.12 Maak een programma dat de gebruiker vraagt om een getal, geef daarna aan of dat getal even of oneven is. ■

Opdracht 2.13 Schrijf een programma dat de gebruiker vraagt om 2 getallen in te voeren, print daarna de grootste van de twee op het scherm. ■

Opdracht 2.14 Schrijf een programma dat de gebruiker vraagt om een getal. Dit getal correspondeert met een dag in de week (1 = maandag, 2 = dinsdag, etc.). print de correspondeerde dag op het scherm. Als de gebruiker een getal groter dan 7 of kleiner dan 1 invoert, geef dan een 'error'. ■

3. Week 3: Python op de Pi

Nu we enige basis van *Python* bezitten, en wat ervaring hebben met de taal in combinatie op je laptop/pc, gaan we deze week aan de slag met de *Raspberry Pi*. We gaan er deze week voor zorgen dat je je een beetje wegwijs maakt op dit kleine machientje, en dan gaan we aan de slag met de *GPIO*-poorten van de *Pi* in *Python*. Maar voordat we dat doen, gaan we eerst kijken naar een ander belangrijk concept: *loops*.

3.1 Loops

Als we bepaalde onderdelen van onze code vaker uit willen voeren, kunnen we dat doen aan de hand van *loops*. In *C* hadden we al kennisgemaakt met de `for`-loop. Deze zit gelukkig ook in *Python*, maar werkt wel een beetje anders. We gebruiken deze vaak in combinatie met de funtie `range()`:

```
1 for x in range(0, 10, 1):
2   print(x)
```

In het bovenstaande voorbeeld roepen we de functie `range()` aan met 3 argumenten(0, 10 en 1). De eerste geeft het startgetal voor *x* aan, de tweede bij welke waarde van *x* de loop moet stoppen, en de derde en laatste met welke hoeveel we *x* we moeten verhogen bij elke nieuwe iteratie. Dit voorbeeld print dus de getallen 0 t/m 9 op het scherm:

```
1  0
2  1
3  2
4  3
5  4
6  5
7  6
8  7
9  8
10 9
```

N.B. De functie `range()` kun je op 3 verschillende manieren aanroepen:
- `range(stop)`: Met enkel 1 argument, de stop waarde. $start = 0, step = 1$.
- `range(start, stop)`: Met 2 argumenten, voor start en stop. $step = 1$.
- `range(start, stop, stap)`: En 3, zoals in het voorbeeld.

In het bovenstaande stukje voorbeeldcode kan dus `range(0, 10, 1)` worden vervangen door `range(10)`, want die levert dezelfde functionaliteit.

Als tweede voorbeeld een loop die van 2 t/m 25 loopt, met stapjes van 3:

```
1 for x in range(2, 25, 3):
2   print(x)
```

Dit print het volgende uit: $2, 5, 8, 11, 14, 17, 20, 23$. De `for`-loop stopt na 23, omdat $23 + 3 = 26$, wat hoger is dan 25, de stop waarde.

Naast de `for`-loop, bestaat er ook de `while`-loop. Deze zit ook in *C*, maar hebben we destijds niet gebruikt. Wellicht is het handig om te weten dat hij bestaat, en te snappen hoe je 'm gebruikt:

```
1 i = 1
2 while i < 6:
3   print(i)
4   i += 1
```

In het bovenstaand voorbeeld wordt de code in de `while`-loop uitgevoerd *zolang* de conditie `i < 6` gelijk is aan `True`. We beginnen met `i = 1`, en tellen er elke keer `1` bij op, waardoor de uitvoer van het programma 1 t/m 5 is.

Met een `while`-loop is ook vrij makkelijk een oneindige loop te maken, vergelijkbaar met de *loop()*-functie bij de *Arduino*:

```
1 while True:
2   print('en door..')
```

N.B. Het is bij zo'n loop wel handig om te weten hoe je 'm weer stopt. Want in de meeste gevallen maak je 'm per ongeluk. Bij een *IDE* heb je naast een run knopt (groen driehoekje), vaak ook een stop knop (rood vierkantje). In de terminal kun je de toetscombinatie Ctrl+C gebruiken.

3.2 Editor

Nu het concept van loops ook bekend is in *Python* gaan we ons richtin op de *Raspberry Pi*. Allereerst, is net als bij je laptop ook op de *Pi* handiger om een *IDE* te gebruiken. Dus daar gaan we eerst voor zorgen. Als het goed is staat *Thonny* standaard geïnstalleerd, maar ook hier mag je alles gebruiken wat je zelf prettig vindt. Tijdens de lessen wordt zoals eerder vermeld met Thonny gewerkt.

Mocht *Thonny* of je favoriete editor nog niet geïnstalleerd zijn, is dit een mooi moment om te kijken dat eigenlijk gaat onder *Linux*. Je kunt hiervoor de grafische *Add/Remove Software*-tool voor gebruiken:

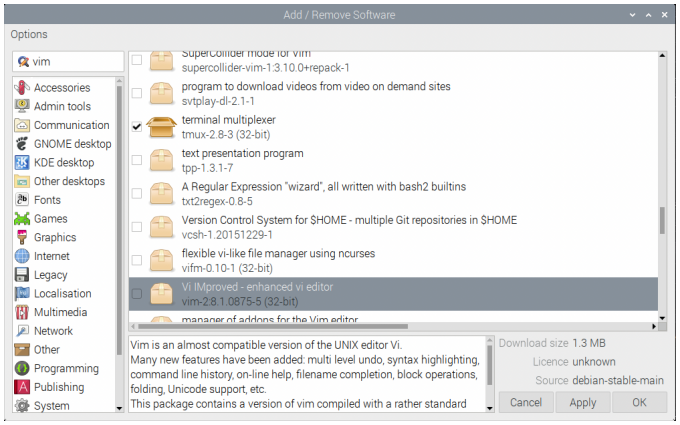


Figure 3.1: *Add/Remove Software* tool

N.B. Een beetje hacker doet dit vaak via de *Terminal*. Ook omdat de *Pi* niet altijd aangesloten is op een scherm, is het ook handig om te weten hoe dit werkt. Allereerst zorg je ervoor dat alle lijsten met software die je kunt installeren up-to-date is:

```
sudo apt-get update
```

Daarna installeer je het programma in kwestie:

```
sudo apt-get install thonny
```

Als dit klaar is, is je vers geïnstalleerde programma te vinden in het *Pi-menu* linksboven.

3.3 Pi header

Zoals je wellicht al gezien had, heeft de *Pi* een 40-pins header aan boord, die sinds *Model 2B* altijd hetzelfde is gebleven. Deze header maakt dat de *Pi* niet alleen een 'leuk klein, maar relatief snel computertje' is, maar ook gemakkelijk hardware kan aansturen en sensors kan uitlezen. Een combinatie die het een krachtig data analyse platform maakt. In afbeelding 3.2, is te zien wat er allemaal op de header zit:

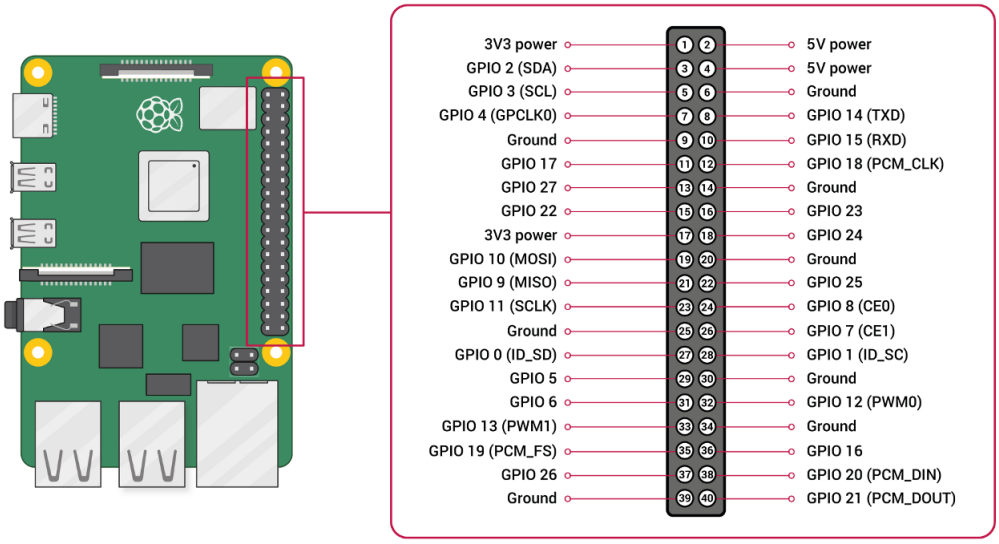


Figure 3.2: Pinout van de *Raspberry Pi*, bron: *https://www.raspberrypi.org/documentation/usage/gpio/*

Als het goed komen de meeste pin-fucnties bekend voor, want deze zijn vergelijkbaar met de *Arduino*. Zo heeft de *Pi*:

- 28 digitale *GPIO*-pinnen, vergelijkbaar met de digitale pinnen op de *Arduino*.
- 2 *PWM*-pinnen, om analoge signalen mee na te bootsen.
- Een *SPI*-bus (*MOSI, MISO, SCLK*).
- Een I^2C-bus (*SDA, SCL*) (ook wel: *TWI*-bus genoemd).
- Een *UART* (*TXD, RXD*), vergelijkbaar met de *Serial* op de *Arduino*.

Opdracht 3.1 Geen zin om de functie van elk pinnetje te onthouden? Open op je *Pi* een *Terminal*-venster en typ '*pinout*'. ■

3.4 GPIO

Tijd om de pinnen daadwerkelijk te gaan gebruiken. We beginnen maar eens door de *GPIO*-poorten te gebruiken als output, met het aansturen van een LED.

3.4.1 GPIO: output

Sluit een *LED* en een weerstand (van ongeveer 300Ω) aan op *GPIO17*. Zie ook afbeelding 3.3, hieronder:

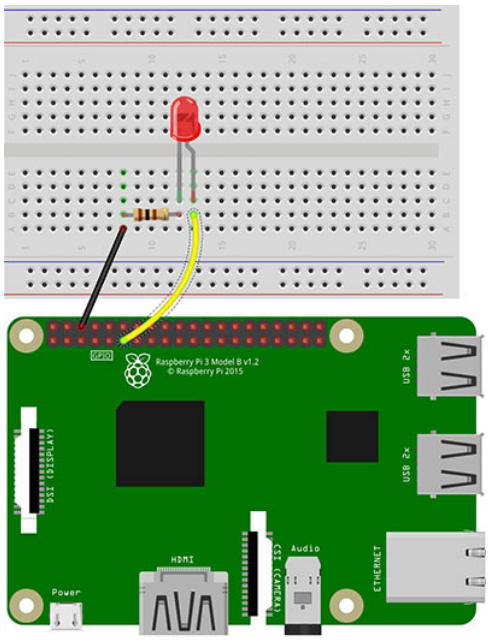


Figure 3.3: LED aangesloten op *GPIO17*

Zoals altijd met programmeren, kan elk probleem opgelost worden op meerdere manieren. We gaan eerst even kijken naar de 'klassieke' manier om *GPIO*-poorten aan te sturen. Open je editor en voer de onderstaande code in, sla het bestand op als een **.py*-bestand.

```
1  import RPi.GPIO  # Nodig voor pinnen te kunnen besturen
2  import time      # Nodig voor sleep()-functie
3
4  led_pin = 17     # LED zit op GPIO17.
5
6  RPi.GPIO.setmode(RPi.GPIO.BCM)          # Benader pinnen a.h.v. hun GPIO-nummer.
7  RPi.GPIO.setup(led_pin, RPi.GPIO.OUT)   # Set led_pin als OUPUT
8
9  while True:
10     RPi.GPIO.output(led_pin, True)    # LED aan!
11     time.sleep(1)                     # Wacht 1 seconde.
12     RPi.GPIO.output(led_pin, False)   # LED uit!
13     time.sleep(1)                     # Wacht 1 seconde.
```

N.B. Bij het runnen krijg je nu waarschijnlijk een warning. Deze is te onderdrukken door `RPi.GPIO.setwarnings(False)` toe te voegen voordat de `setmode()`-functie wordt aangeroepen op regel 6.

Het programma heeft qua opbouw veel weg van wat we gewend waren bij de *Arduino*. Met de eerste twee regels (`import`) zorgen we dat we gebruik kunnen maken van de functies in de `RPi.GPIO`- en de `time`-module. Op regel 6 en 7 worden de module en de poort geconfigureert (dit is vergelijkbaar wat er in de *void setup()* had gestaan). Daarna begint een oneindige lus, die het daadwerkelijke lampje laat knipperen (Dit had bij de *Arduino* in de *void loop()* gestaan).

Tot zover de 'klassieke' manier, tegenwoordig zie je steeds meer dat er gebruikt wordt gemaakt van de `gpiozero`-module. En dat ziet er dan als volgt uit:

```
1  import gpiozero
2  import time
3
4  led = gpiozero.LED(17)  # LED aangesloten op GPIO17
5
6  while True:
7      led.on()        # LED aan!
8      time.sleep(1)   # Wacht 1 seconde.
9      led.off()       # LED uit!
10     time.sleep(1)   # Wacht 1 seconde
```

De `gpiozero`-module regelt alle configuratie verder voor je, wat een stuk compactere en meer leesbare code oplevert.

Vind je het vervelend om steeds `time.sleep()` te typen? Vervang dan regel 2 door:

```
1  from time import sleep
```

Je kunt nu gewoon `sleep()` gebruiken, zonder '`time.`' ervoor te hoeven typen. Hetzelfde principe gaat ook op voor `gpiozero.LED()`.

3.4.2 GPIO: input

Het eerstvolgende wat je zou willen doen met de *GPIO*-pinnen is deze natuurlijk gebruiken als input. Dat kan door er een sensor aan te hangen. Let op, de *Raspberry Pi* heeft in tegenstelling tot de *Arduino* geen analoge ingangen, dus de sensor moet een digitaal (hoog of laag) signaal terug geven. In z'n meest basale vorm komt dit overeen met een knopje. Zie Figuur 3.4.

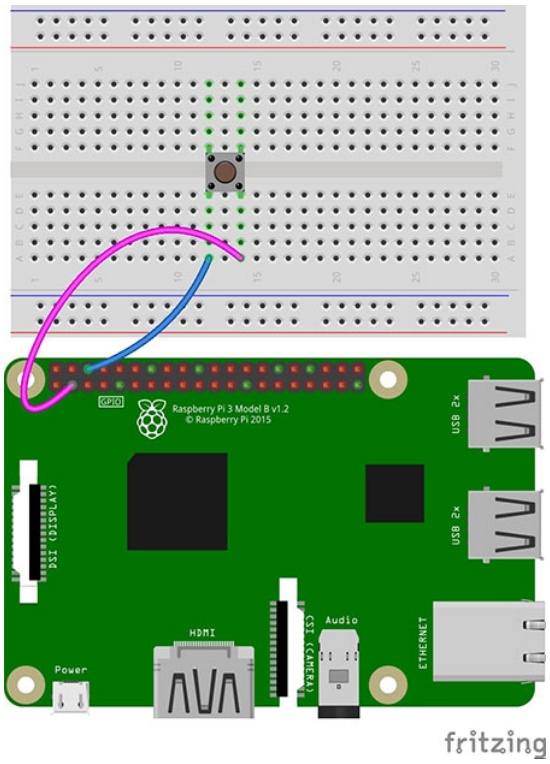


Figure 3.4: Een drukknop aangesloten op *GPIO2*

Ook hier gaan we eerst maar even kijken naar de 'klassieke' manier. Ook omdat dit het meest overeen komt met hoe het op de *Arduino* werkt. Typ het onderstaande programma over en sla het op als een **.py* bestandd:

```
1  import RPi.GPIO  # Nodig voor pinnen te kunnen besturen
2  from time import sleep
3
4  button_pin = 2  # Drukknop zit aangesloten op GPIO2.
5
6  # Configureer button_pin als input:
7  RPi.GPIO.setwarnings(False)
8  RPi.GPIO.setmode(RPi.GPIO.BCM)
9  RPi.GPIO.setup(button_pin, RPi.GPIO.IN)
10
11 while True:
12     if not RPi.GPIO.input(button_pin):  # De knop is omgekeerd, vandaar 'not'.
13         print('Knop ingedrukt!')
14         sleep(0.25)                     # Wacht 250ms.
```

Zodra de knop wordt ingedrukt (we lezen de status uit door `Rpi.GPIO.input()` aan te roepen op regel 12), wordt er iets naar het scherm geschreven dat het gelukt is.

Opdracht 3.2 Wat gebeurt er als je de knop ingedrukt blijft houden? En wat als je de `sleep()` regel verwijderd? ■

Precies hetzelfde kunnen we ook bereiken door de `gpiozero`-module te gebruiken:

```
1 from gpiozero import Button
2 from time import sleep
3
4 button = Button(2)  # Drukknop zit aangesloten op GPIO2.
5
6 while True:
7     if button.is_pressed:
8         print("Knop ingedrukt!")
9         sleep(0.25)
```

Wederom een stuk compacter en to-the-point. Met deze `gpiozero`-module zijn nog veel meer mooie dingen te doen, we komen er later op terug als we leren hoe functies werken in *Python*. Merk op dat we op dit moment nog steeds alles in een eindeloze `while`-loop hebben staan, dat betekend dat de *Pi* enorm druk is met het draaien van ons kleine scriptje, dat kan veel efficiënter!

3.5 Huiswerkopdrachten

Opdracht 3.3 Schrijf een programma waarin je met een `while`-loop de eerste 10 getallen van de tafel van 5 print. ■

Opdracht 3.4 Schrijf een programma waarin je de gebruiker vraagt om een getal, en geef daarna de eerste 10 getallen van de tafel voor dat getal. Gebruik hiervoor een `for`-loop. ■

Opdracht 3.5 Schrijf een programma die de getallen van -10 t/m -1 op scherm print. Gebruik hiervoor alleen de `range()`-functie in de `for`-loop (en natuurlijk een `print()` ;)). ■

Opdracht 3.6 Vraag aan de gebruiker een getal l, en print daarna een rij van sterretjes. Dus bij $l = 7$, geeft het de volgende output:

```
1 *******
```

TIP: Als je niet wilt dat een `print`-functie elke keer een nieuwe regel begint, kun je dit veranderen door het *end*-argument te gebruiken: `print('*', end='')`. ■

Opdracht 3.7 Breid Vraag 2.8 uit. Vraag aan de gebruiker nu 2 getallen: h en l. Print daarna een vierkant van sterretjes ter grootte van $h \times l$. Dus bij $h = 3$ en $l = 5$, geeft het de volgende output:

```
1 *****
2 *****
3 *****
```

TIP: Je kunt loops in loops zetten. ■

Opdracht 3.8 Sluit een LED aan op *GPIO17* en een op *GPIO18*. Zorg ervoor dat de ene 1 keer per seconde knippert en de andere 2 keer per seconde.
TIP: De `sleep()`-functie accepteerd ook argumenten die kleiner zijn dan 1 (oftewel: *floats*), zodat je deze ook kunt gebruiken voor delays van minder dan 1 seconde. ■

Opdracht 3.9 Sluit een LED aan op *GPIO17* en een knop op *GPIO19*. Zorg ervoor dat als de knop wordt ingedrukt, de LED 10 keer gaat knipperen. ■

4. Week 4: Diepere duik

Deze week gaan kijken naar enkele concepten die ons leven als programmeur een stukje makkelijker kunnen gaan maken. We gaan eerst een blik werpen op verzamelingen in de vorm van *lijsten*. Daarna de robuustheid van onze code aanzienlijk verbeteren met `Try` / `Except`-blokken en ten slotte meer structuur aanbrengen met de introductie van *functies*.

4.1 Lijsten

De lijsten in *Python* zijn vergelijkbaar met de array's in *C*, maar dan veel flexibeler in te zetten. Het voordeel van een lijst te gebruiken, is dat je data die bij elkaar hoort mooi kunt groeperen. Maar je kunt ook handige operaties op een lijst uitvoeren, zo kun je 'm bijvoorbeeld sorteren, omkeren, of elementen eruit filteren. Eerst gaan we kijken naar het aanmaken ervan, hieronder volgen een aantal willekeurige lijsten:

```
1  # Lijsten met getallen:
2  cijfers = [1, 2, 3, 5, 10, 15]
3  frac = [0.5, 0.25, 0.2, 0.1, 0.001, 0.0005]
4
5  # Lijsten met strings:
6  namen = ['Piet', 'Jan', 'Klaas']
7  dagen = ['ma', 'di', 'wo', 'do', 'vr', 'za', 'zo']
8
9  # Een lijst met verschillende datatypes in een, kan ook:
10 mix = [13, 'robotica', 3, 2, 'analyse', 3.14, cijfers]
```

Lijsten hebben zoals je hierboven kunt zien een naam en een verzameling aan elementen. De elementen staan tussen de blokhaken [ ], en zijn gescheiden door een komma. Lijsten kunnen elk data type bevatten (er kunnen zelfs verschillende datatypes - en dus ook weer lijsten - in een lijst staan, wat we verder in dit vak niet meer gaan behandelen).

Lijsten zijn bijv. heel krachtige dingen om in te zetten in combinatie met `for`-loops:

```
1 kleuren = ['geel', 'rood', 'groen', 'oranje']
2
3 for kleur in kleuren:
4   print(f'He, nu issie weer {kleur}!')
```

In deze `for`-loop worden alle elementen van *kleuren* een voor een opgeslagen in de variabele *kleur* (regel 3). En die wordt dan weer gebruikt op regel 4. De output van het bovenstaande stukje code is als volgt:

```
1 He, nu issie weer geel
2 He, nu issie weer rood
3 He, nu issie weer groen
4 He, nu issie weer oranje
```

Zoals is gezegd, zijn er een aantal handige operaties uit te voeren op een lijst:

```
1  >>> a = [3,2,4,1]
2  >>> len(a)
3  4
4  >>> a.append(5)
5  >>> a
6  [3, 2, 4, 1, 5]
7  >>> len(a)
8  5
9  >>> a.reverse()
10 >>> a
11 [5, 1, 4, 2, 3]
12 >>> a.sort()
13 >>> a
14 [1, 2, 3, 4, 5]
```

Opdracht 4.1 Maak zelf een lijst aan. Ga na wat de functies `count()`, `pop()` en `remove()` doen. ■

4.2 Try ... Except

Je bent ze inmiddels ongetwijfeld tijdens het maken van de huiswerkopdrachten (of elders) tegengekomen: *errors*. Bijv. bij het opvragen van gegevens bij de gebruiker, via de `input()`-functie:

```
1 x = input('Geef een cijfer: ')
2 x = int(x)  # Zet de input-string om naar een int, en sla dit weer op in 'x'.
3 x += 1  # Tel er 1 bij op
4 print(f'Een hoger: {x}')
```

Deze code runt doorgaans prima:

```
1 Geef een cijfer: 12
2 Een hoger: 13
```

Maar wat nu als de gebruiker iets verkeerds intypt? Bijv. een letter:

```
1 Geef een cijfer: q
2 Traceback (most recent call last):
3   File "/Users/stefan/Code/Python/err_test.py", line 2, in <module>
4     x = int(x)  # Zet de input-string om naar een int, en sla dit weer op in 'x
                                                                          '.
5 ValueError: invalid literal for int() with base 10: 'q'
```

Dan krijg je dus een *error* voor je kiezen, in dit geval een `ValueError`. Bijkomend nadeel: het programma sluit direct af (crasht). Gelukkig geeft `Python` bij errors vaak genoeg informatie waarmee je de bug je in je code kunt oplossen. Hierboven kun je lezen dat er iets mis gaat op regel 2: `x = int(x)`, op het moment dat we de `str` *q* om proberen te zetten naar een `int`. En dat is best te verklaren natuurlijk, *q* is domweg geen getal.

Tip: Mocht je nu op zoek zijn naar de *ASCII*-waarde van een karakter, gebruik dan de `ord()`-functie:

```
1 >>> ord('a')
2 97
3 >>> ord('b')
4 98
```

Een gebruiker kan nou eenmaal iets verkeerds intypen en het zou vrij suf zijn als dan elke keer je programma crasht. Vandaar dat er in *Python* functionaliteit zit om dit soort fouten van buitenaf af te vangen en op te reageren. Dit doen we door onze kritische code (in dit geval het omzetten van een `str` naar een `int`: `x = int(x)`) in een `try` / `except` blok te zetten. En dat ziet er als volgt uit:

```
1 x = input('Geef een cijfer: ')
2 try:
3   x = int(x)   # Zet de input-string om naar een int, en sla dit weer op in 'x'.
4   x += 1   # Tel er 1 bij op
5   print(f'Een hoger: {x}')
6 except ValueError:
7     print('Dat was geen getal, probeer het nog eens')
```

Hier 'proberen' we de `str` te interpreteren als een `int`. Gaat dat goed, dan wordt er netjes de rest van de code uitgevoerd. Krijg je een error, dan wordt het stuk code na de `except` uitgevoerd, en wordt de gebruiker vriendelijk verzocht 'even normaal te doen' en het nog eens te proberen.

Elke error die je tegenkomt kun je op deze manier afvangen. Let daar wel bij op, dat niet elke error afgevangen hoeft te worden. Fouten die je zelf als programmeur maakt horen daar vaak niet bij, het gaat meer om fouten van 'buitenaf'. Bijv. je probeert connectie te maken met een server, maar die is op het moment uit de lucht. Dan zul je wellicht iets in de trant van een `ConnectionError` of een `TimeOutError` krijgen, die zijn handig om af te vangen. Je wilt niet dat door iets van buitenaf je programma crasht. Voor de liefhebber: hier is een lijstje te zien van de errors die standaard te vinden zijn in *Python*: *https://docs.python.org/3/library/exceptions.html*.

4.3 Functies

Functies waren we ook al tegengekomen in *C*, in *Python* is de opzet ervan hetzelfde: Een blok aan code, die alleen wordt uitgevoerd als de functie wordt 'aangeroepen'. Je maakt ze aan ('definieert' ze) met `def`, zie ook het onderstaande voorbeeld:

```
1 def my_function():
2     print('Hallo vanuit een functie!')
3
4 my_function()
5 print('Hallo vanuit het hoofdprogramma!')
6 my_function()
```

Dit produceert de volgende output:

```
1 Hallo vanuit een functie!
2 Hallo vanuit het hoofdprogramma!
3 Hallo vanuit een functie!
```

Je ziet in het programma op de eerste 2 regels de gemaakte functie, na het woordje `def`. De naam van de functie is *my_function*, alle regels die daarna volgen en ingesprongen zijn (tab), horen bij deze functie. In dit geval is dit maar 1 regel, namelijk een `print()`.

Daarna wordt deze functie 'aangeroepen', dat gebeurt voor het eerst op regel 4. Dat doe je dus door de naam van de functie te typen, gevolgd door 2 haakjes (). Even later op regel 6 gebeurt hetzelfde nog eens.

Dit was een voorbeeld van een functie zonder argumenten. Vaak zul je ook functies zien en maken die dat juist wel hebben. Een argument is iets wat je meegeeft met de functie. Dat kunnen er zoveel zijn als je wilt, en ook van elk datatype. Hieronder een voorbeeld met 2 argumenten, van het type `str`:

```
1 def my_function(naam, vraag):
2     print(f'Hallo {naam}! {vraag}?')
3
4 my_function('Henk', 'Hoe gaat het')
5 my_function('Piet', 'Waddup')
```

De uitoer zal in dit geval zijn:

```
1 Hallo Henk! Hoe gaat het?
2 Hallo Piet! Waddup?
```

Valt je op je dat nog steeds nergens het datatype hoeft aan te geven? *Python* snapt dat het om `str` gaat, omdat je ze als zodanig meegeeft op regel 4 en 5.

N.B. De functie zal het ook prima doen als je 'm aanroept met bijv. getallen:
`my_fucntion(1, 2)`

```
1 Hallo 1! 2?
```

Python gaat heel flexibel met om datatypes. Omdat alles wat in de functie *my_function* gebeurt met de argumenten *naam* en *vraag* ook prima gedaan kan worden met getallen, krijg je geen errors. In *C* vlogen nu de errors en warnings om je oren.

Dan rest ons nog een ding om in te duiken, functies die iets teruggeven. Het kan zijn dat een functie een bepaalde operatie uitvoerd, en dat je het resultaat daarvan graag wil gebruiken verderop in je programma. Dat kan, net als in *C* met `return`. Zie ook het volgende voorbeeld:

```
1  import math
2
3  def oppervlakte(r):
4      # Berekend de oppervlakte van een cirkel ahv de straal.
5      opp = r**2 * math.pi
6      return opp
7
8  o5 = oppervlakte(r=5)
9  print(f'Oppervlakte cirkel, bij straal=5:  {o5}')
10
11 o10 = oppervlakte(10)
12 print(f'Oppervlakte cirkel, bij straal=10: {o10}')
13
14 print(f'Oppervlakte cirkel, bij straal=15: {oppervlakte(15)}')
15
16 o20 = oppervlakte(20)
17 print(f'Oppervlakte cirkel, bij straal=20: {o20:.2f}')
```

In tegenstelling tot *C* hoef je ook hier geen datatype op te geven, dat wordt voor je geregeld. Enkel de functie afsluiten met een `return`-statement is voldoende om de functie een waarde terug te laten geven.

De functie `oppervlakte()` berekend de oppervlakte van een cirkel a.h.v. de meegegeven straal `r`. En hij wordt op vier verschillende manieren aangeroepen:

- Allereerst op regel 8 met een straal van $r = 5$, hier wordt ook de naam van het argument (r) meegegeven, dat leest makkelijker, maar is niet nodig. Het antwoord van de functie wordt opgeslagen in variabele $o5$ en naar het scherm gescreven.

- Daarna wordt de functie aangeroepen met een straal van $r = 10$, het antwoord hiervan wordt daarna opgeslagen in de variabele $o10$, die dan geprint wordt op het scherm.

- Daarna op regel 14, wordt het aanroepen van de functie (met $r = 15$) en het printen van de teruggegeven waarde gecombineerd tot een regel code.

- En tenslotte wordt de functie nog een keer aangeroepen voor $r = 20$ op regel 16. Bij het printen (regel 17) gebeurt iets bijzonders, daar wordt namelijk aangegeven dat de uitvoer maar 2 kommagetallen moet laten zien (dit komt door de `:.2f` in `print`).

De uitvoer van het programma is dan ook:

```
1  Oppervlakte cirkel, bij straal=5:  78.53981633974483
2  Oppervlakte cirkel, bij straal=10: 314.1592653589793
3  Oppervlakte cirkel, bij straal=15: 706.8583470577034
4  Oppervlakte cirkel, bij straal=20: 1256.64
```

Zoals ik vorige week stelde, kun je met functies in combinatie met de `gpiozero`-module leuke dingen doen. Zo kun je dus bijv. functies aanroepen, als er een bepaalde actie wordt uitgevoerd word, zoals het indrukken en het loslaten van een knop:

```
1  from gpiozero import Buttons
2  from signal import pause
3
4  def ingedrukt():
5    print('knop ingedrukt!')
6
7  def losgelaten():
8    print('knop weer losgelaten!')
9
10 btn = Button(2)   # Drukknop zit op GPIO2
11
12 btn.when_pressed = ingedrukt     # Koppel functie bij het indrukken.
13 btn.when_released = losgelaten   # Koppel functie bij het loslaten.
14 pause()                          # Doe verder niets meer, maar sluit niet af.
```

Opdracht 4.2 Voor het programma uit, en ga voor jezelf na wat er precies gebeurt. ■

N.B. Weet je nog uit Programmeren 1, dat knoppen last kunnen hebben van het zogeheten *bouncen*, waardoor het lijkt dat ze veel vaker ingedrukt worden, dan ze daadwerkelijk worden. Mocht je daar nu ook last van hebben, dan kun je met `gpiozero` vrij makkelijk de bounce-tijd instellen. Vervang regel 10 door:

```
1  # Drukknop zit op GPIO2 met 100ms bounce time:
2  btn = Button(2, bounce_time=0.1)
```

De laatste regel `pause()`, zorgt dat het programma verder niets meer uitvoert, (behalve het afhandelen van de knop) maar ook niet afsluit. In tegenstelling tot de `while True:` uit het vorige hoofdstuk, waardoor de *Raspberry Pi* intensief gebruikt werd, kan hij in dit geval rustig andere taken doen.

N.B. Het mooie van het kunnen koppelen van functies aan gebeurtenissen van een sensor (zoals hier in het geval van de drukknop), is dat dat ook functies kunnen zijn die gebruikt worden door actuatoren, zoals bijv. een LED. Een LED heeft een `on()` - en een `off()`-functie. Dus het koppelen van een LED aan een drukknop kan zo:

```
1  from gpiozero import Buttons, LED
2  from signal import pause
3
4  btn = Button(2)   # Drukknop zit op GPIO2
5  led = LED(17)     # LED zit op GPIO17
6
7  # Koppel het indrukken van de knop met led.on():
8  btn.when_pressed = led.on
9  # Koppel het loslaten van de knop met led.off():
10 btn.when_released = led.off
11
12 # Doe verder niets meer, maar sluit niet af:
13 pause()
```

Afsluitend aan dit hoofdstuk gaan we eens wat dingen combineren die we geleerd hebben. Je kunt namelijk elementen uit een lijst filteren, als je daarvoor een filter-functie voor aanmaakt. Bijv. je hebt een bepaalde temperatuursensor, waar je elk uur de temperatuur mee meet, maar deze geeft af en toe een foute waarde (bijv. $+100°C$ op een winterdag).

```
1  def temp_filter(temp):
2      # Filter-functie, voor temp groter dan 90 graden.
3      if (temp > 90):
4          return False
5      else:
6          return True
7
8  # Lijst met gemeten temperatuur in celcius:
9  gemeten_temp = [15.2, 14.4, 14.2, 99.9, 13.8, 12.6, 100.1]
10
11 # Pas filter toe:
12 gefilterde_temp = filter(temp_filter, gemeten_temp)
13
14 # Print de overgebleven waardes op het scherm:
15 print('De gefilterde temperaturen zijn:')
16 for t in gefilterde_temp:
17     print(t)
```

In het bovenstaande voorbeeld, wordt een filter-functie aangemaakt `temp_filter(temp)`. Die een `False` teruggeeft als de meegegeven temperatuur `temp` groter is dan 90, en anders `True`.

Op regel 12 wordt die filterfunctie daadwerkelijk toegepast op de lijst `gemeten_temp` en de nieuwe lijst die dat opleverd wordt opgelsagen in `gefilterde_temp`

4.4 Huiswerkopdrachten

Opdracht 4.3 Maak een lege lijst aan, en vul deze daarna met de getallen 1 t/m 100. Gebruik hiervoor een loop en de `append()` functie.
Tip: Een lege lijst maak je als volgt aan: `lijst =[]` ■

Opdracht 4.4 Maak een filter-functie waarmee je alle getallen onder de 30 uit een lijst kunt filteren. Pas deze toe op de lijst die je gemaakt hebt bij opdracht 4.3. ■

Opdracht 4.5 Maak een filter-functie waarmee je oneven getallen uit een lijst kunt filteren. Pas deze toe op de lijst die je gemaakt hebt bij opdracht 4.3. ■

Opdracht 4.6 Maak een functie `schreeuw` die een meegegeven str omzet naar hoofdletters, en deze daarna weer terug geeft met `return`. ■

Opdracht 4.7 Wat als je nu de functie die net gemaakt hebt bij 4.6, aanroept met een getal (bijv. `schreeuw(5)`)?
Breidt de functie uit met een `try` / `except`, zodat deze niet langer crasht. ■

Opdracht 4.8 Sluit 4 LEDs aan op de *GPIO17*, *GPIO18*, *GPIO3* en *GPIO5* pins. Configureer ze, en zet ze daarna in een lijst. Ga met een loop door de lijst heen, en zet ze een voor een 1 seconde aan, en dan weer uit. ■

Opdracht 4.9 Maak een programma die 10 keer aan de gebruiker vraagt om een dier in te voeren. Voeg elk van deze dieren toe aan een lijst. Filter alle diernamen die minder dan 3 letters hebben en print daarna de lijst in alfabetische volgorde op het scherm.
Tip: Je bepaalt de lengte van een string (of lijst) met de functie: `len(mijn_string)`. ■

5. Week 5: Data analyse in Python

Volgt nog..

6. Week 6: Arduino met Pi Koppelen

Volgt nog...

Index