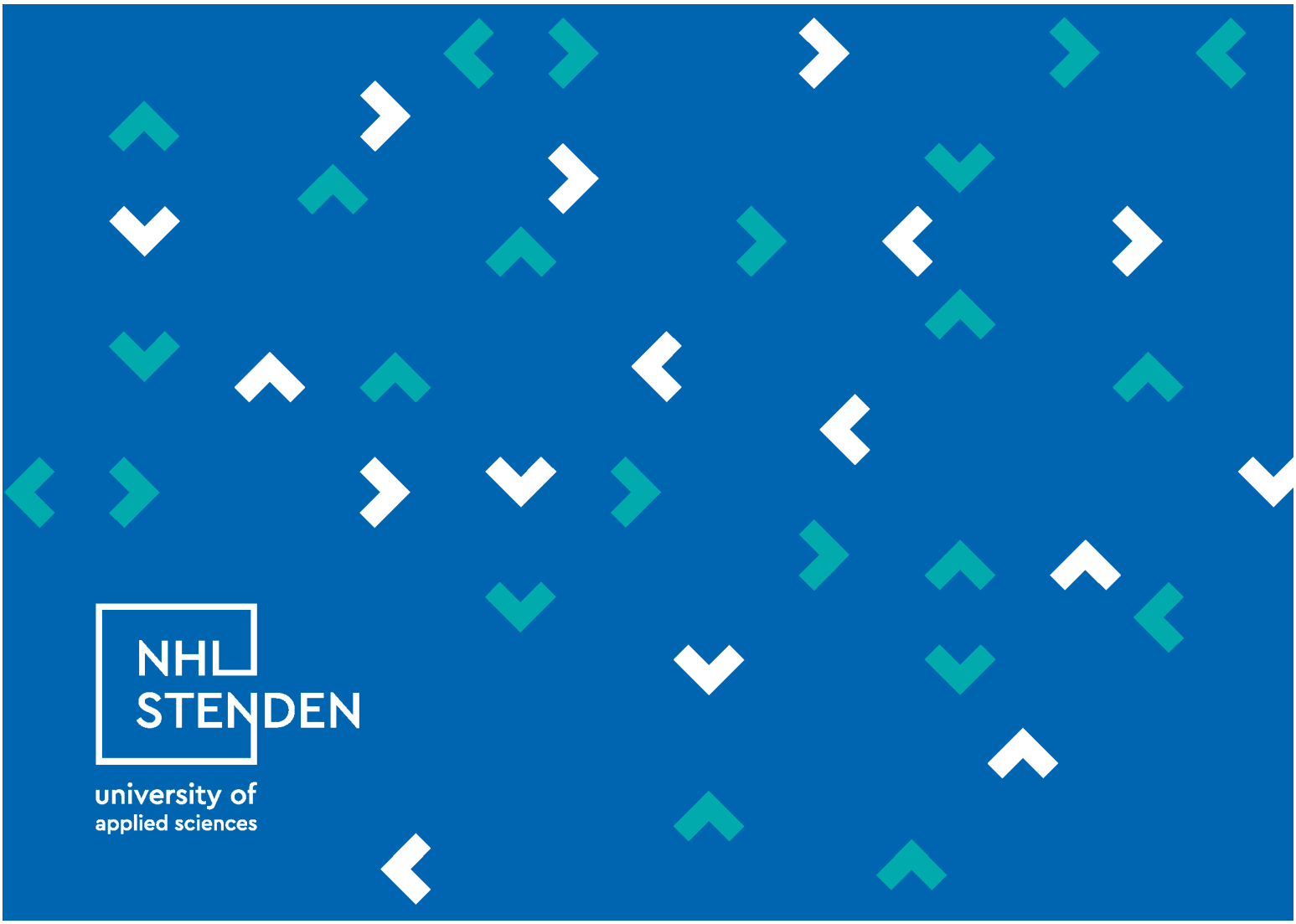
Programmeren 2

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. ■

N.B. 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.

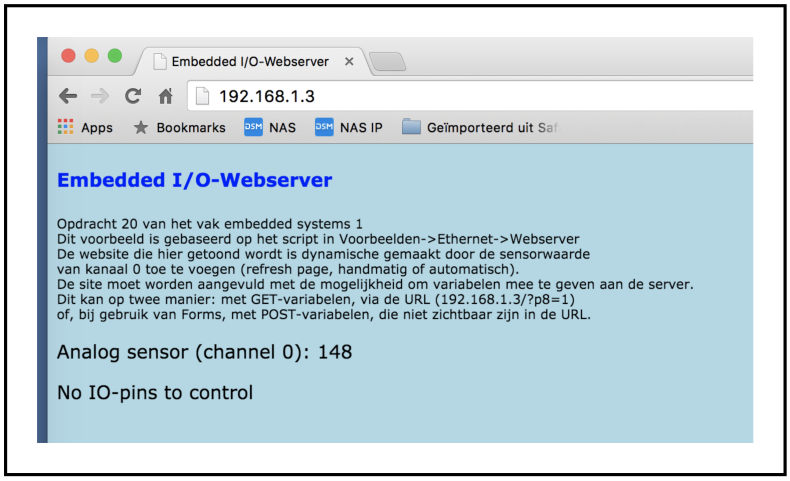


Figure 1.1: Webserver

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.

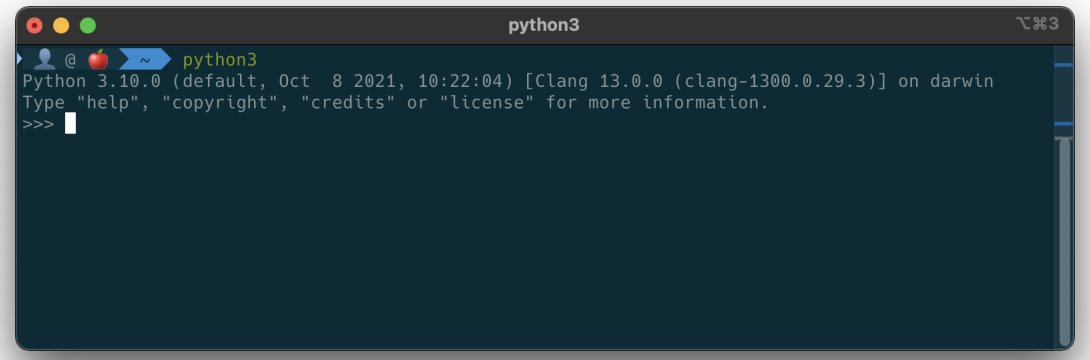


Figure 2.1: De *Python* interpreter, met versie *3.10.0*

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*:

```
>>> x = 12  # x is nu een int
>>> x
12
>>> x = 'abc'  # En vanaf hier een str
>>> x
'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:

```
a = 200
if a > 100:
  print('conditie is True!')
  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:

```
a = 200
if a > 100:
  print('conditie is True!')
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.

```
a = 200
if a > 100:
  print('conditie is True!')
  print('a is groter dan 100')
else:
  print('conditie is False!')
  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:

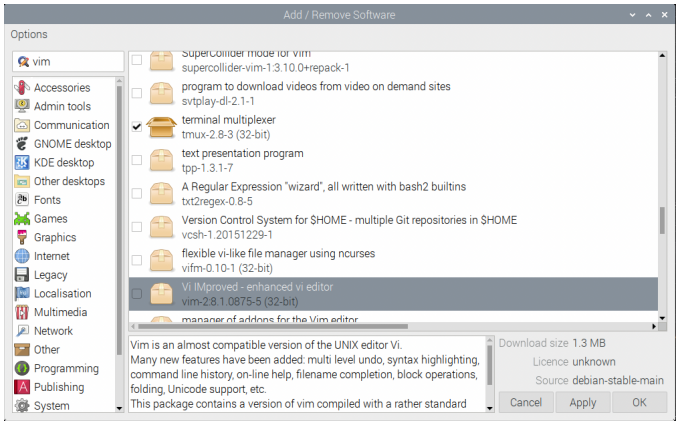


Figure 3.1: *Add/Remove Software* tool

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 update
```

Daarna installeer je het programma in kwestie:

```
sudo apt 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:

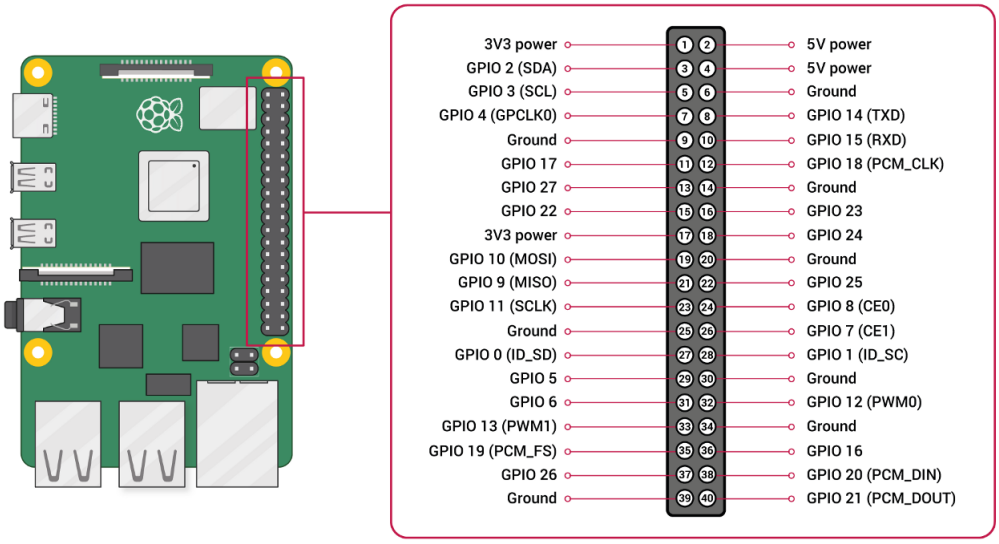


Figure 3.2: Pinout van de *Raspberry Pi*, bron: *https://www.raspberrypi.org/documentation/usage/gpio/*

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:

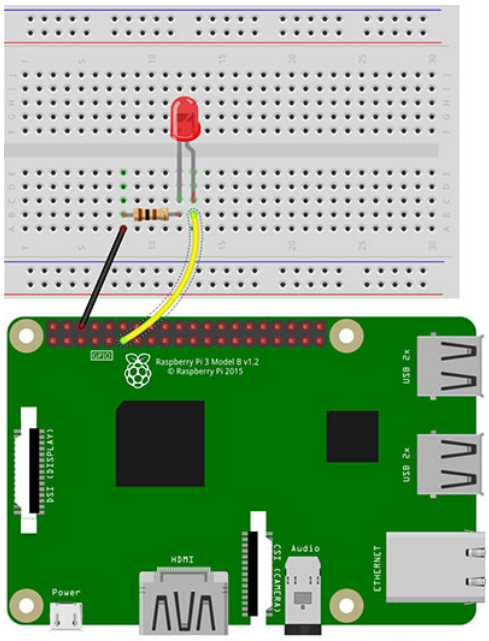


Figure 3.3: LED aangesloten op *GPIO17*

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.

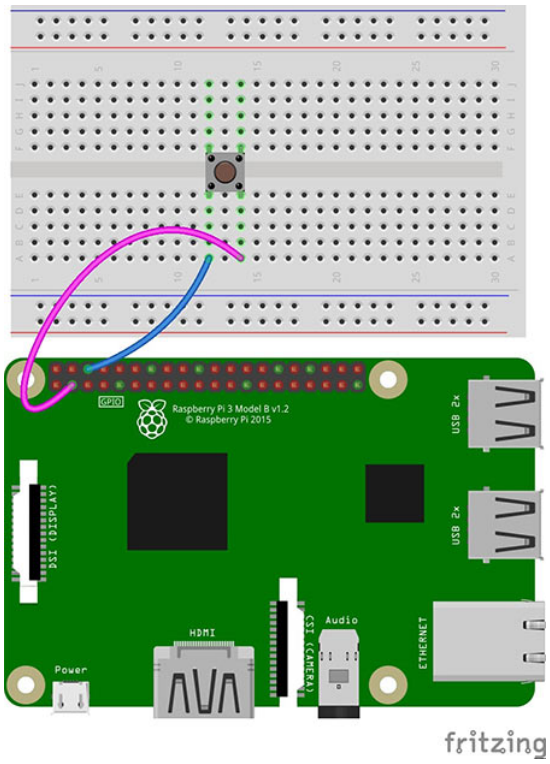


Figure 3.4: Een drukknop aangesloten op *GPIO2*

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 Button, 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 dat 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

We hebben inmiddels een gedegen bassis kennis van *Python* opgedaan gedurende dit vak. We hebben o.a. geleerd over loops, lijsten, functies en IO aansturing in *Python*, en de vershillen gezien met *C*. Een van de concepten die nog onbelicht is gebleven is het zogeheten 'object georiënteerd programmeren' (of *OOP*). Omdat dit een van de belangrijkste en handigste concepten van het programmeren is van de afgelopen decenia, én omdat we stiekem er toch al flink mee gewerkt hebben, gaan we in dit hoofdstuk ook dit thema belichten. Daarnaast gaan we deze week aan de slag met *packages*, handige software bibliotheken die de functionaliteit van onze *Python* omgeving flink kunnen uitbreiden.

5.1 Object georiënteerd programmeren

Zoals je wellicht al zou verwachten, focust het software design principe genaamd 'object georiënteerd programmeren' (OOP), zich vooral op 'objecten'. Objecten bestaan uit variabelen en functies. Waarbij je de variabelen kunt zien als eigenschappen van het object en functies als gedrag ervan. Lekker abstract hè?

Een voorbeeldje maakt het hopelijk wat duidelijker. Stel je voor: je wilt op een object georiënteerde manier naar een cirkel kijken. Allereerst ga je dan nadenken over welke eigenschappen jouw cirkel zou moeten hebben. Een cirkel heeft (in dit voorbeeld) maar één eigenschap: zijn straal. Dit wordt de variabele van ons cirkel-object, onthoud deze even.

Daarnaast gaan we kijken naar het 'gedrag' van ons cirkel-object. Dat wil zeggen, we stellen ons de vraag, wat willen we allemaal met onze cirkel doen? Voor dit voorbeeld wil ik graag de oppervlakte en de omtrek van de cirkel makkelijk kunnen bepalen. Dit worden de twee functies van van ons object:

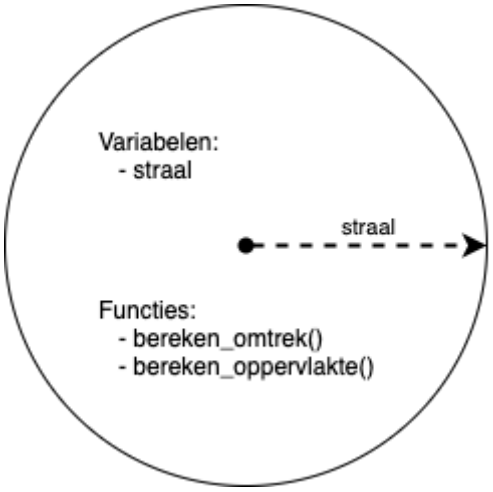


Figure 5.1: Ons cirkel-object heeft een straal, en kan (op basis daarvan) de omtrek en oppervlakte van zichzelf berekenen.

In code kunnen we dat nu op de volgende manier hiervan een blauwdruk van onze cirkel vastleggen:

```
1  from math import pi
2
3
4  class Cirkel:
5      r = 10  # Elke cirkel heeft een straal. In dit geval is r =10
6
7      def bereken_oppervlakte(self):
8          # Berekent de oppervlakte op basis van de klasse waarde r
9          opp = pi * self.r**2
10         return opp
11
12     def bereken_omtrek(self):
13         # Berekent de omtrek op basis van de klasse waarde r
14         omt = pi * self.r * 2
15         return omt
```

We kunnen hier wel even kort doorheen lopen, om het concept wat hier gemaakt is goed te snappen. Allereerst wordt uit de `math` bibliotheek `pi` geïmporteerd, zodat we kunnen rekeken met π. Daarna wordt het interessant want op regel 4 maken we een klasse-definitie van we onze cirkel, via het woordje `class`. Dit kun je zien als een soort blauwdruk van ons object, zodat we die later kunnen gebruiken. De naam van onze `class` is `Cirkel`, want klasse-definities schrijf je altijd met een hoofdletter.

Daarna is er op regel 5 te zien dat we onze `Cirkel` een variabele `r` heeft, deze stelt de straal van de cirkel voor. En we hebben deze een standaard waarde 10 gegeven.

Op regel 7 en 12 worden twee functies gedefinieerd, zoals als het goed is bekend voor moet komen. Het enige dat er een beetje gek aan is, is dat argument `self`. En dit is ook een bijzonder maar erg nuttig ding. Hiermee kan je namelijk de klasse-variabelen benaderen. Dus als je iets wil doen

met de straal van de cirkel, gebruik je in de functies `self.r` (in plaats van enkel `r`). Je hoeft die `self` allleen te typen bij defineren van de klasse-functies. Bij het aanroepen wordt dit argument automatisch door *Python* gevuld en hoef je deze dus ook niet mee te geven dan.

Laten we onze nieuwe klasse-definitie eens gebruiken. Sla het bovenstaande stukje code op als *cirkel.py*. Op deze manier kunnen we 'm nu importeren in een ander stukje code. Hieronder volgt een stukje code waarin onze `Cirkel` daadwerkelijk wordt gebruikt:

```
1  from cirkel import Cirkel  # Importeer onze Cirkel definitie.
2
3  # Maak een object 'Cirkel' aan met de naam 'my_circle'.
4  my_circle = Cirkel()
5
6  # Roep de functies van Cirkel aan, en sla de waardes op:
7  omt = my_circle.bereken_omtrek()
8  opp = my_circle.bereken_oppervlakte()
9
10 # Print de waardes:
11 print(f'Deze cirkel heeft een straal van: {my_circle.r}cm.')
12 print(f'En daarmee een omtrek van: {omt:.2f}cm.')
13 print(f'En heeft een oppervlakte van: {opp:.2f}cm^2.')
```

N.B. Let op: Importeren op deze manier werk alleen als beide bestanden in dezelfde folder staan.

Ook dit stukje code heeft wellicht een beetje uitleg nodig. Op de eerste regel importeren we ons bestand `cirkel` en uit dat bestand willen we onze klasse-definitie `Cirkel` gebruiken. Je kunt dus meerdere klasse-definities in een bestand zetten en je importeert alleen diegene die je wilt gebruiken.

Op regel 4 wordt (eindelijk) daadwerkelijk een object gemaakt van onze klasse-definitie. Het object heeft de naam `my_circle` en is dus van het type `Cirkel`.

N.B. Komt deze regel bekend voor? Toen ik eerder stelde dat we stiekem al vaker met object georiënteerd programmeren hebben gewerkt, is dit waar ik op doelde.
Zo hebben bijv. een tijdje terug al op deze manier een `LED` en een `Button` aangemaakt (`led = LED(17)` en `btn = Button(2)`). De klasse-definities van deze stonden namelijk in de package `gpiozero`. Zelf bij de *Arduino* gebruikten we dit soms op plekken bijv. `Serial.print();`

Na het aanmaken van het object `my_circ`, kunnen we deze gebruiken en bijv. de functies die het heeft aanroepen. Dit gebeurt op regel 7&8 (zie je dat hier niets meer te zien is van de `self`). De waardes die de functies teruggeven worden opgeslagen in de variabelen `omt` en `opp` en deze worden uiteindelijk in de laatste 3 regels op een nette manier naar het scherm geschreven:

```
1 Deze cirkel heeft een straal van: 10cm.
2 En daarmee een omtrek van: 62.83cm.
3 En heeft een oppervlakte van: 314.16cm^2.
```

5.1.1 Constructors

Het voordeel van object geörienteerd programmeren is dat de gemaakte objecten makkelijk herbruikbaar (en uitbreidbaar) zijn. Hierdoor is je code ook makkelijker te onderhouden en is er als het goed is een duidelijke structuur in je projecten terug te zien.

In ons `Cirkel`-voorbeeldje is de herbruikbaarheid van de code nog niet zo duidelijk, omdat de cirkel in dit geval alijd een straal van 10cm heeft. Je zou deze eigenlijk flexibel willen aanpassen naar je wensen. Dit kan heel mooi met een *constructor*. De *constructor* is de functie die wordt aangeroepen door *Python* (dus automatisch) wanneer je een object aanmaakt, dus in dit geval dus bij de regel: `my_circle = Cirkel()`.

Zo'n *constructor* is dus optioneel, we hebben 'm immers nog niet aangemaakt. Laten we er nu eentje maken voor onze cirkel-definitie. Een constructor is gewoon een functie, maar met een vaste (en wat gekke) naam: `__init__(self)` (met 2 keer 2 liggende streepjes). Zie het onderstaande voorbeeld:

```
1  from math import pi
2
3
4  class Cirkel:
5      r = 0  # De straal is standaard 0, je kan 'm zetten met de constructor.
6
7      def __init__(self, r):
8          # Deze gekke functie is dus de constructor.
9          self.r = r  # Deze regel slaat de waarde van r op als classe waarde
10
11     def bereken_oppervlakte(self):
12         # Berekent de oppervlakte op basis van de klasse waarde r
13         return pi * self.r**2
14
15     def bereken_omtrek(self):
16         # Berekent de omtrek op basis van de klasse waarde r
17         return pi * self.r * 2
```

In de bovenstaande code is niet veel veranderd ten opzichte van de vorige versie van *cirkel.py*. De klasse-variabele `r` is nu standaard 0 en dient nu een waarde te krijgen via de constructor. Ook zijn de functies wat compacter geschreven, maar functioneel exact hetzelfde.

De daadwerkelijke constructor zien we op regels 7-9. Zoals gezegd heeft deze een vaste naam, en naast het argument `self` is er ook een argument `r`. Hiermee kun je dus de gewenste straal opgeven als je een object maakt. Laten we dat eens doen:

```
1  from cirkel import Cirkel
2
3  # Maak een Cirkel, met de naam my_circ en een straal van 5:
4  my_circ  = Cirkel(5)
5
6  # Maak een Cirkel, met de naam my_circ2 en een straal van 25:
7  my_circ2 = Cirkel(25)
```

Bij het aanmaken van het objet, bepalen we dus pas de straal van de cirkel. En hebben we op deze manier een flexibel object gemaakt die we steeds opnieuw kunnen gebruiken en makkelijk kunnen uitbreiden. Willen we bijv. een functie wijzigen of toevoegen, dan hoeven we dat maar op een plek te doen: in de klasse-definitie in `cirkel.py`. Overal waar we dan een object ervan gebruiken zal deze nieuwe functionalteit dan beschikbaar komen.

Object georiënteerd programmeren is niet het makkelijkste onderwerp om compleet te begrijpen. Als je hier voor het eerst naar kijkt kan dit wellicht nog een beetje criptisch en abstract overkomen. Daarom geef ik in dit hoofdstuk nog een voorbeeld voordat we verder gaan met andere onderwerpen.

Laten we het eens proberen met wat simpele hardware, een LED. We gebruikten vaak de `LED` klasse uit de `gpiozero` package. Stel nu dat die package niet bestond en we het enkel moesten doen met de 'klassieke' `RPi.GPIO` package. Dan zouden we met de kennis die we inmiddels over OOP hebben opgedaan, zelf een LED-klasse maken.

Hierbij dienen we weer eerst af te vragen, wat zijn de eigenschappen en functies van een led. Een led zit altijd aangesloten aan een bepaalde pin en heeft daarnaast een status ('aan' dan wel 'uit'). Qua functionaliteit wil je een led aan kunnen zetten, uit kunnen zetten, en eventueel ook willen schakelen (togglen, uitzetten als hij aan is, en vice versa):

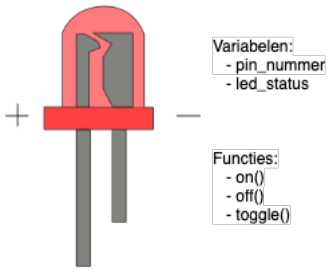


Figure 5.2: Led-object met 'pin_nummer' en een 'status' en kan aan, uit of geschakeld worden.

Als we ons herinneren uit vorige weken, kunnen we met *RPi.GPIO* al deze functionaliteiten uitwerken. We gaan nu deze package gebruiken om een object georiënteerde `Led`-klasse te maken (die functioneel dus vergelijkbaar is met de `LED`-klasse uit *gpiozero*). Sla dit op als `my_led.py`.

```
1  import RPi.GPIO as io
2
3  class Led:
4      led_pin = 0  # Pin is standaard 0, je kan 'm zetten met de constructor.
5      status = False  # De Led staat standaard uit.
6
7      def __init__(self, pin_number, default_state=False):
8          # Constructor van Led, geeft led_pin en status een waarde
9          # en initieerd de GPIO
10
11         self.led_pin = pin_number
12         self.status = default_state
13
14         io.setwarnings(False)  # Onderdruk warnings
15         io.setmode(io.BCM)  # Gebruik BCM voor pinaanduiding
16         io.setup(self.led_pin, io.OUT)  # Set led_pin als OUTPUT
17
18         # Schrijf de default status naar de led:
19         io.output(self.led_pin, self.status)
20
21     def on(self):
22         # Zet de led aan!
23         io.output(self.led_pin, True)
24
25         # Pas status aan:
26         self.status = True
27
```

```
28      def off(self):
29          # Zet de led uit!
30          io.output(self.led_pin, False)
31
32          # Pas status aan:
33          self.status = False
34
35      def toggle(self):
36          # Toggle led. Zet uit als aan, en andersom.
37          self.status = not self.status  # Gooi status om.
38
39          io.output(self.led_pin, self.status)  # Set led.
```

In de code hierboven gebruiken we dus de package *RPi.GPIO* om een OOP versie van een Led te maken. Op regel 1 importeren we deze package. De twee variabelen van de Led, `led_pin` en `status` zijn te vinden onder de klasse-definitie.
In de constructor gebeurt alle setup code. De klasse-variablen krijgen hun waarde (regel 12 en 13). Daarna wordt de poort geïnitieerd als uitgang. En uieindelijk wordt voor de netheid, de huidige status geschreven naar de led pin. Als status nu `True` is, gaat de Led aan. Is deze `False`, blijft hij uit.

N.B. Valt je op dat in constructor de `default_state` al een waarde krijgt? (`False`). Dit is een standaard waarde. Zodat bij het aanmaken van daadwerkelijke object, je het argument `default_state` niet per se mee hoeft te geven. Oftewel, je kunt ons Led-object op deze manieren aanmaken:

```
1  from my_led import Led
2
3  voorbeeld_led = Led(17, False)   # Led op pin 17, standaard uit
4  voorbeeld_led = Led(17)          # Led op pin 17, standaard uit
5  voorbeeld_led = Led(17, True)    # Led op pin 17, standaard aan
```

Daarna volgen de functies van de led. Deze zullen er als het goed is niet verrassend uitzien. `on()` en `off()` zetten de led respectievelijk aan en uit. En de `toggle()` functie zet de led uit, als deze aan was, en aan als deze uit was. Ook wordt overal de nieuwe status van de led opgeslagen.

De klasse-definitie is dan als het volgt bijv. te gebruiken:

```
1  from my_led import Led
2  from time import sleep
3
4  mijn_led = Led(17)  # Led op pin 17
5
6  while True:
7    mijn_led.on()
8    sleep(1)
9    mijn_led.off()
10   sleep(1)
```

Opdracht 5.1 Zie jij verschillen met de code op blz. 28? ■

5.2 Packages

Een *Package* in *Python* is een verzameling van modules. Deze modules kunnen we weer in onze code gebruiken door deze te importeren met `import`. We hbben er in middel al een aantal van gebruikt bijv.: `math, time, gpiozero, RPi`, maar deze worden allemaal al standaard meegeleverd op bijv. de *Raspberry Pi*. Je kunt ze ook heel eenvoudig zelf toevoegen aan je *Python* omgeving, door gebruik te maken van `pip`. Dit programma kun je aanroepen vanaf de terminal:

```
python3 –m pip install <naam_package>
```

N.B. Je hebt ongetwijfeld al een hele lijst geïnstalleerd zonder dat je het wist, je kunt het bekijken door het onderstaande in een terminal in te typen:

```
python3 –m pip list
```

En als je wat schijfruimte wilt vrij maken door packages te verwijderen die je niet langer gebruikt, kan dat met:

```
python3 –m pip uninstall <naam_package>
```

Om een beetje bekend te raken met *pip*, gaan we als voorbeeld een kleine package installeren genaamd `camelcase`:

```
python3 –m pip install camelcase
```

Probeer nu het volgende scriptje te draaien:

```
1 from camelcase import CamelCase      # importeer onze nieuwe aanwinst
2
3 c = CamelCase()                      # Nodig om de package te kunnen gebruiken.
4 txt = 'pip onder de knie krijgen!'   # De tekst die omgezet dient te woren.
5
6 print(c.hump(txt))                   # Pas camelcase toe op de tekst.
```

Als dat lukt, is de module juist geïnstalleerd en heb je de basis van *pip* onder de knie! (Deze packages worden overigens gedownload vanaf *https://pypi.org/*, kijk er gerust eens na en wellicht kom je er interessante tegen tussen de bijna 350.000 packages.)
Het aanpassen van hoofdletters is echter niet al te spannend, dus gaan we verder met een package waar we echt iets aan hebben: *matplotlib*.

5.3 matplotlib

Matplotlib is een verzameling aan tools voor het maken en bewerken van grafieken.

Opdracht 5.2 Installeer Matplotlib ■

Om het te gebruiken in z'n meest simpele vorm zorg je ervoor dat je een reeks x-waardes hebt, en een reeks bijbehorende y-waardes. Deze kun je dan plotten (het doorrekenen van alle data van de grafiek) en de resulterende grafiek kun je tonen op het scherm:

```
1  import matplotlib.pyplot as plt
2
3  # Creeer x en y waardes ahv lijsten:
4  x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5  y = [2, 3, 4, 5, 1, 2, 3, 4, 7, 1]
6
7
8  # Plot de waardes en laat hierna de grafiek zien:
9  plt.plot(x, y)
10 plt.show()
```

Hieruit rolt grafiek 5.3 uit:

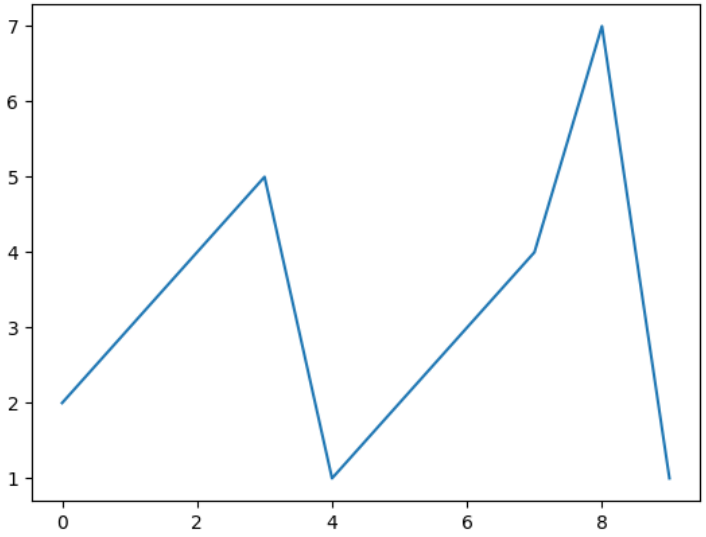


Figure 5.3: Simpele plot met *Matplotlib*

N.B. Kreeg je bij het runnen van bovenstaande code op de *Pi* een error, in de trant van *'Error retrieving accessibility bus address'*? Dan kan het zijn dat je een module in Linux mist, open een terminal en voer in:

```
sudo apt update
sudo apt install at-spi2-core
```

Een grafiek is natuurlijk pas echt bruikbaar als deze netjes is opgemaakt met duidelijke labels voor de assen en eventueel een titel en een legenda. Ook dat is snel gepiept met *matplotlib*. In het onderstaande stuk code voegen we dit toe aan ons eerdere script. Daarnaast is er een extra set met y-waardes toegevoegd, om 2 verschillende meetmomenten na te bootsen.

```
1  import matplotlib.pyplot as plt
2
3  # Creeer x-waardes:
4  x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5
6  # y-waardes voor de eerste en tweede meting:
7  y1 = [2, 3, 4, 5, 1, 2, 3, 4, 7, 1]
8  y2 = [1, 2, 4, 6, 2, 4, 2, 3, 4, 4]
9
10
11 # Plot de eerste meting:
12 plt.plot(x, y1, '--',label='eerste meting')
13
14 # Plot de tweede meting:
15 plt.plot(x, y2, '..', label='tweede meting')
16
17 # Geef de grafiek labels voor beide assen en een titel:
18 plt.xlabel('Tijd in minuten')
19 plt.ylabel('Temperatuur in graden Celcius')
20 plt.title('Gemeten temperaturen')
21
22 # Genereer een legenda (ahv van de ingevoerde labels bij plot()):
23 plt.legend()
24
25 # Laat grafiek zien:
26 plt.show()
```

De bonenstaande code produceert onderstaande grafiek 5.4:

Opdracht 5.3 Zorg dat de eerste lijn rood gekleurd (`color`) wordt, en de tweede lijn groen en gestippeld (`linestyle`).
TIP: Gebruik hiervoor de documentatie van de packge op: *https://matplotlib.org*. ■

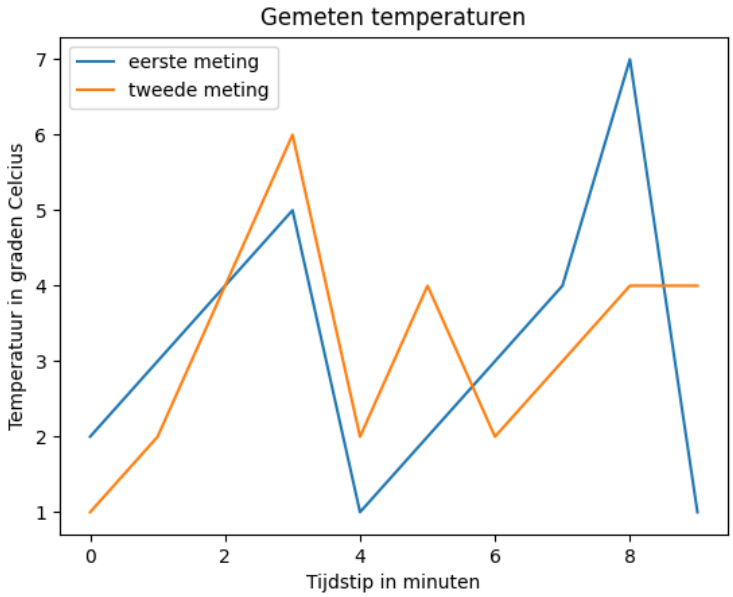


Figure 5.4: Uitgebreidere plot met *Matplotlib*

5.3.1 NumPy

De mogelijkheden van *matplotlib* kunnen enorm uitgebreid worden in combinatie met andere packages. Een voor de hand liggende keuze daarin is bijvoorbeeld *NumPy*. Die een enorm scala aan wiskundige tools met zich meebrengt.

Opdracht 5.4 Installeer NumPy ■

We kunnen met de combinatie *NumPy* en *matplotlib* bijv. elke wiskunde functie die we kunnen bedenken op een makkelijke en overzichtelijke wijze plotten. Superhandig voor je wiskunde huiswerk ;). Als voorbeeldje volgt hieronder een klein scriptje die $y = sin(x)$ plot op een bereik van $-\pi$ tot π:

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Genereer x-waardes, van -pi tot pi, met stapjes van 0.1:
5  x = np.arange(-np.pi, np.pi, 0.1)
6
7  # Bereken y-waardes door sinus uit te rekenen voor alle x-waardes:
8  y = np.sin(x)
9
10 # Plot de sinus en laat grafiek zien:
11 plt.plot(x, y)
12 plt.show()
```

De bonenstaande code produceert onderstaande grafiek 5.7:

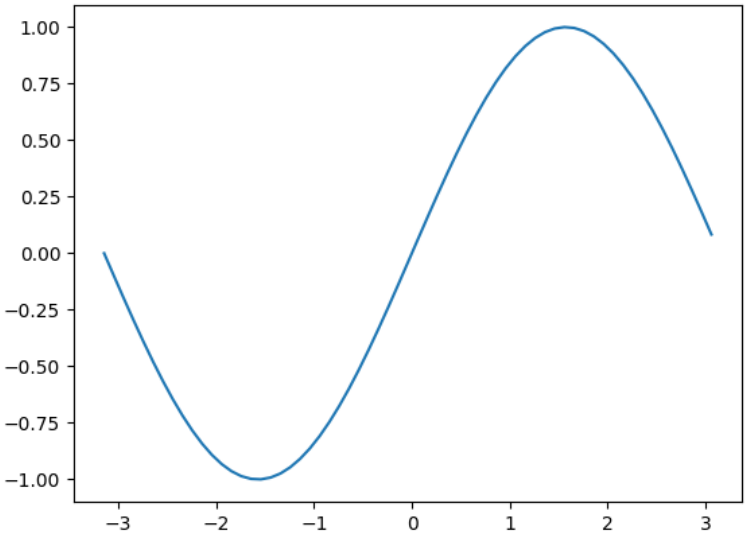


Figure 5.5: Een sinus-functie tekenen met *Matplotlib* en *NumPy*.

Op regel 5 wordt hier een reeks aan getallen gegenereerd door de `np.arange()`-functie. Valt je op dat deze functie nagenoeg hetzelfde eruit ziet (en werkt!) als de ingebouwde `range()`-functie? Het verschil zit 'm in dat `np.arange()` ook reeksen met komma-getallen (floats) kan genereren. Om een soortgelijke reden gebruiken we op regel 7 `np.sin()` in plaats van `math.sin()`. De sinus functie uit *NumPy* kan namelijk de sinus van een hele reeks getallen uitrekenen en dit weer teruggeven als een reeks getallen. De sinus-functie uit `math` kan enkel de sinus uitrekenen op 1 getal per keer.

5.4 Pandas

Het tekenen van grafieken wordt een stuk intteressanter als je veel input data hebt. Die kun je bijvoorbeeld krijgen door een sensor periodiek te meten, maar ook door een spreadsheet in te laden. De package *pandas* gaat je bij dat laatste flink helpen.

Opdracht 5.5 Installeer de package *Pandas* en z'n verschillende parsers:

```
python3 –m pip install pandas odfpy openpyxl xlrd
```

De laatste drie packages die hier worden geïnstalleerd zijn resp. nodig voor support van *.ods*- (LibreOffice), *.xslx*- (Microsoft Excel) en *.xls*- (Oude Microsoft Excel) bestanden. Mocht je bepaalde bestandsformaten niet gaan gebruiken, kun je die uiteraard overslaan. ■

Naast de *Python* packages ben je natuurlijk ook een programma nodig dat spreadsheets kan maken en bewerken. Op je laptop gebruik je hier ongetwijfeld Microsoft Excel voor. Voor de Raspberry Pi moet je hier waarschijnlijk nog even een programma voor installeren. Voor een complete office suite variant, kun je bijv. LibreOffice installeren. Dit is een gratis, en een heel complete vervanger voor Microsoft Office (die overigens ook prima draait op je laptop ;)).

Heb je nu geen zin in een grote download, kun je ook kiezen voor Gnumeric. Dat is gewoon een basic spreadsheet editor zonder poespas, maar wel met alle functionaliteit in huis die we tijdens dit vak nodig zijn.

Opdracht 5.6 Installeer op basis van je voorkeur één spreadsheet programma op je Pi:

```
sudo apt install libreoffce
sudo apt install gnumeric
```

■

Als voorbeeld heb ik de volgende spreadsheet aangemaakt: Een tabel met twee kolommen: 'Tijdstip' en 'Waarde', met elk 50 rijen/waardes:

	A	B
1	Tijdstip	Waarde
2	0	0,69314718
3	1	1,09861229
4	2	1,38629436
5	3	1,60943791
6	4	1,79175947
7	5	1,94591015
8	6	2,07944154
9	7	2,19722458
10	8	2,30258509
11	9	2,39789527
12	10	2,48490665
13	11	2,56494936
14	12	2,63905733
15	13	2,7080502

Figure 5.6: Voorbeeld spreadsheet gemaakt in Excel.

Deze spreedsheet is dan met *pandas* als volgt heel eenvoudig in te laden:

```
1 import pandas as pd
2
3 # Lees spreadsheet in:
4 df = pd.read_excel('excel1.xlsx')
5
6 # Schrijf alle gevonden kolommen naar het scherm:
7 for name in df.columns:
8     print(name)
```

Heb je nu een spreadsheet met meerdere tabladen gemaakt en wil je een specifiek blad inladen met de naam 'Blad1'? Dat kan prima. Pas dan regel 4 aan van de code naar:
`df = pd.read_excel('excel1.xlsx', sheet_name='Blad1')`

Dit stukje code leest de *.xlsx*-bestand in, en print daarna alle namen van de kolommen op het scherm:

```
1 Tijdstip
2 Waardes
```

Deze namen kunnen we daarna gebruiken om de daadwerkelijke data te benaderen en bijvoorbeeld te printen met *matplotlib*:

```
1  import pandas as pd
2  import matplotlib.pyplot as plt
3
4  # Lees spreadsheet in:
5  df = pd.read_excel('excel1.xlsx')
6
7  # Laad de x en y waardes in op basis van kolomdata:
8  x = df['Tijdstip']
9  y = df['Waarde']
10
11 # Extra's: Laat een grid zien op de achtergrond:
12 plt.grid()
13
14 # Set x en y-label:
15 plt.xlabel('Tijdstip')
16 plt.ylabel('Waarde')
17
18 # Plot en print grafiek:
19 plt.plot(x, y)
20 plt.show()
```

Dit bovenstaande stukje code levert uiteindelijk de volgende grafiek op:

Nerdy tip: Je kunt ook in plaats van zelf de x- en y-labels een naam te geven, dit af laten hangen van de gevonden kolomnamen:

```
1 plt.xlabel(df.columns[0])
2 plt.ylabel(df.columns[1])
```

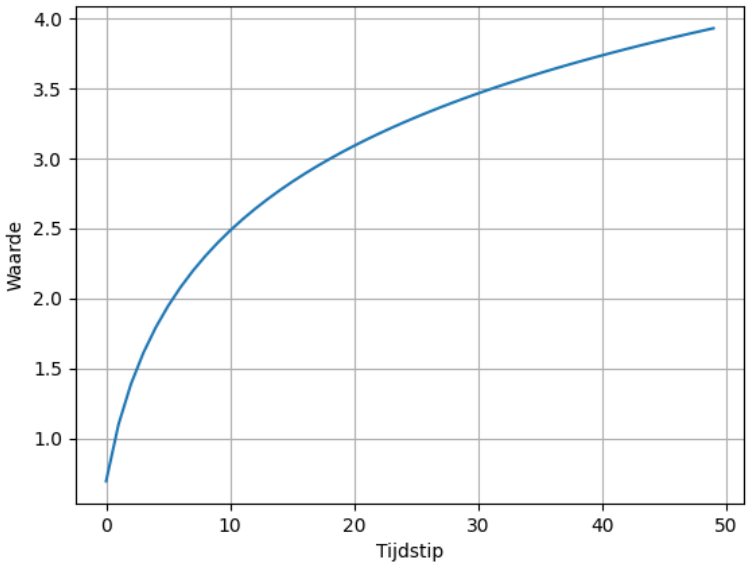


Figure 5.7: Plot gemaakt vanuit spreadsheet.

Andersom kan natuurlijk ook: Je hebt data in je *Python* script, en dat wil je opslaan als een spreadsheet. Dat is wat we met het volgende stukje code doen:

```
1  import pandas as pd
2
3  # Gegeven enkele x en y waardes:
4  x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5  y = [2, 3, 4, 5, 2, 3, 4, 5, 2, 3]
6
7  # 'zip' de beide lijsten in elkaar:
8  data = zip(x, y)
9
10 # Maak een data-frame aan op basis van data, en geef de kolommen een naam:
11 df = pd.DataFrame(data, columns=['Tijdstip', 'Waardes'])
12
13 # Definieer writer, met bestandsnaam en bestandstype:
14 writer = pd.ExcelWriter('test.xlsx')
15
16 # Verwerk de data-frame als excel-structuur:
17 df.to_excel(writer, index=False)
18
19 # Sla de excel-structuur op als daadwerklijk excel-bestand:
20 writer.save()
```

Het gegenereerde bestand kunnen we openen met ons spreadsheet-programma, en zal er dan zo uitzien:

	A	B
1	Tijdstip	Waardes
2	0	2
3	1	3
4	2	4
5	3	5
6	4	2
7	5	3
8	6	4
9	7	5
10	8	2
11	9	3
12		

Figure 5.8: Spreadsheet gegenereerd vanuit *Python*

5.5 Huiswerkopdrachten

Opdracht 5.7 Maak een klasse *Persoon* aan. Geef 'm als variabelen een *naam*, *leeftijd* en een *geslacht*. Geef hem een functie `zeg_hoi()`, waarmee hij/zij zichzelf kan voorstellen (met een `print()`-statement). ■

Opdracht 5.8 Maak op basis van de Led-voorbeeldklasse een klasse aan voor een RGB led (een drie kleuren-led). Doe dit op basis van *RPi.GPIO* met 3 verschillende leds (een voor elke kleur). Geef de RGB naast een constructor, een aantal fucties: bijv. `rood()`, `groen()`, `blauw()`, `uit()` en bedenk zelf ook nog kleur-combinaties a.h.v. de 3 leds. ■

Opdracht 5.9 Op regel 8 van het de laatste voorbeeld code gebruiken we de `zip()`-functie. Ga na in wat deze functie precies doet met 2 lijsten. ■

Opdracht 5.10 Plot (in 1 grafiek) met *matplotlib* en *NumPy* de functies:
$y_1 = \sin(2 \cdot x)$ en $y_2 = 4 \cdot \cos(4 \cdot x)$ tussen -2π en 2π. Zorg ook voor een duidelijke legenda. ■

Opdracht 5.11 Vraag van de gebruiker steeds een nummer, net zolang tot deze 'klaar' intypt. Voeg alle getallen die de gebruiker intypte steeds toe aan een lijst en print hier (als de gebruiker klaar is) een nette grafiek van. (Begin voor de x-waardes te tellen bij 0, en tel er steeds 1 bij op, voor elk ingevoerd getal). ■

Opdracht 5.12 Breid het het programma van opdracht 5.11 uit, zodat het programma ook een Excel sheet opslaat met alle ingevoerde cijfers. ■

Opdracht 5.13 Maak een eigen klasse *Xlsx_writer* met de volgende variabelen:
- filename → Een bestandnaam (bijv. "file.xlsx").
- x_waardes → Een lijst met x-waardes.
- y_waardes → Een lijst met y-waardes.
- x_label → (Optioneel) Een string met een label voor de x-waardes.
- y_label → (Optioneel) Een string met een label voor de y-waardes.

Geef 'm een functie: `sla_op()`, waarmee je alle data verwerkt en het bestand opslaat als een excel-sheet. Verwerk deze klasse in je programma van 5.12. ■

6. Week 6: Arduino met Pi Koppelen

In deze laatste week van het vak Programmeren 2, gaan we ons eerst nog wat verder verdiepen in de wereld van object geörienteerd programmeren (OOP). Daarnaast stevenen we af op onze eindopdracht van dit vak.

6.1 Overerven

Met het vorige hoofdstuk heb je eerste introductie gehad in de wereld van OOP. We hebben op een nieuwe abstracte manier naar problemen gekeken. Ook werd er toen verteld over de voordelen van OOP, zo kon je de code makkelijk onderhouden (je hoeft maar op 1 plek de klasse-definitie/blauwdruk aan te passen, en overal waar je 'm gebruikt wordt deze verandering overgenomen). Maar ook werd er gesteld dat de gemaakte code eenvoudig uitbreidbaar is, en op dat voordeel hebben we nog niet naar gekeken. Als we nu ons geheugen opfrissen door even te kijken naar de gemaakte `Cirkel`-klasse:

```
1  from math import pi
2
3  class Cirkel:
4      r = 0 # De straal is standaard 0, je kan 'm zetten met de constructor.
5
6      def __init__(self, r):
7          self.r = r # Deze regel slaat de waarde van r op als klasse waarde
8
9      def bereken_oppervlakte(self):
10         # Berekent de oppervlakte op basis van de klasse waarde r
11         return pi * self.r**2
12
13     def bereken_omtrek(self):
14         # Berekent de omtrek op basis van de klasse waarde r
15         return pi * self.r * 2
```

We hadden een klasse `Cirkel` gemaakt die als eigenschap z'n straal had, en waarop we de functies `bereken_oppervlakte()` en `bereken_omtrek()` op kunnen uitvoeren. We kunnen nu deze klasse-definitie nu ook gebruiken om andere klasse te definiëren die gebasseerd zijn op deze

cirkel.

Als we bijvoorbeeld een klasse willen maken voor een `Bol` en we gaan hiervoor beredeneren wat zijn eigenschappen en z'n functies zijn. Zul je zien dat deze veel zullen overeenkomen met die van de `Cirkel`. Qua eigenschappen kunnen we een bol definieren op basis van een straal en op basis van die straal kunnen we de omtrek, oppervlakte en inhoud van de bol berekenen. Grafisch weergegeven:

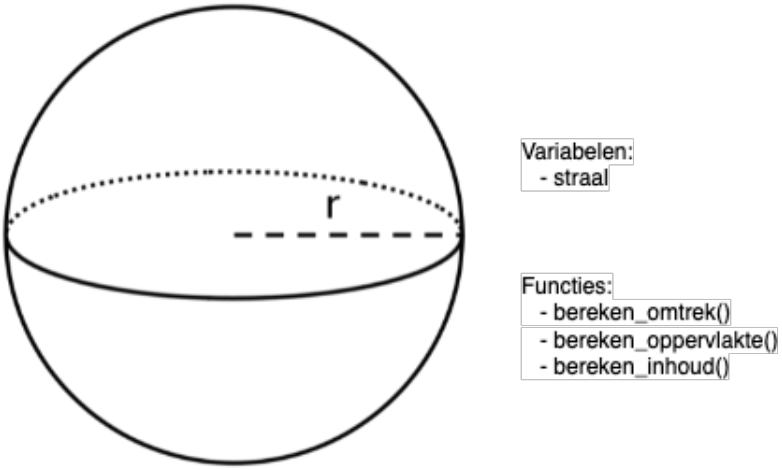


Figure 6.1: Ons bol-object heeft een straal, en kan (op basis daarvan) de omtrek, oppervlakte en inhoud van zichzelf berekenen.

Zowel de klasse-variabele `r`, als 2 van de 3 functies* komen overeen met die van `Cirkel`. Als we dus een klasse-definitie voor een `Bol` willen maken is het handiger om deze te baseren op `Cirkel` dan helemaal opnieuw te beginnen, dat scheelt werk! Onze klasse-definitie van een `Bol` ziet er dan zo uit:

```
1  from cirkel import Cirkel
2  from math import pi
3
4
5  class Bol(Cirkel):
6      # Klasse genaamd Bol, gebasseerd op Cirkel
7
8      def bereken_inhoud(self):
9          # Berekent de inhoud op basis van z'n straal:
10         return 4/3 * pi * self.r**3
```

Valt het je op hoe kort dit bestand is? Op regel 5 gebeurt alle 'magie', hier wordt namelijk gesteld dat we een nieuwe klasse genaamd `Bol` willen maken, maar dat we deze willen baseren op `Cirkel`. *Python* regelt nu onderwater dat de bol-klasse een klasse-variabele `r` voor de straal krijgt, en ook kan hij gebruik maken van de functies (waaronder de constructor) die in `Cirkel` zitten.

Het enige wat wij zelf nog moeten doen is de nieuwe functie uitschrijven die de inhoudt van de bol berekend op basis van z'n straal, met de volgende formule: $V_{bol} = \frac{4}{3}\pi r^3$.

*Wellicht heb je 'm al door: de oppervlakte van een bol berekenen je op een andere manier dan van een cirkel, hier komen we later op terug

N.B. Een klasse basseren op een andere klasse noemen we ook wel een *subklasse* maken. `Bol` is in dit geval een *subklasse* van `Cirkel`. Andersom is `Cirkel` de *superklasse* van `Bol`.

Deze klasse-definitie kunnen we daarna gebruiken in andere programma's op de bekende manier:

```
1  from bol import Bol
2
3  mijn_bol = Bol(10)  # Maak een bol aan met een straal van 10.
4
5  omt = mijn_bol.bereken_omtrek()
6  opp = mijn_bol.bereken_oppervlakte()
7  inh = mijn_bol.bereken_inhoud()
8
9  print(f"Omtrek: {omt:.2f}")
10 print(f"Oppervlakte: {opp:.2f}")
11 print(f"Inhoud: {inh:.2f}")
```

Wat het onderstaande als uitvoer geeft:

```
1 Omtrek: 62.83
2 Oppervlakte: 314.16
3 Inhoud: 4188.79
```

We maken hier een object genaamd `mijn_bol` aan, die van het type `Bol` is. Doordat onze Bol klasse-definitie geen constructor heeft, wordt de constructor van z'n superklasse aangeroepen: die van `Cirkel`. Daarna worden de functies `bereken_omtrek()` en `bereken_oppervlakte()` aangeroepen. Deze zijn ook niet te vinden in de `Bol` klasse-definitie, dus worden die van `Cirkel` hier uitgevoerd. De laatste functie `bereken_inhoud()` is wel te vinden in `Bol`, dus die wordt aangeroepen zoal we gewend zijn.

Python zoekt dus eerst in de klasse-definitie van een subklasse naar klasse-variabelen en functies, en als ze daar niet te vinden zijn kijkt hij naar de superklasse ervan. (zijn ze daar ook niet vinden, dan krijg je een `AttributeError` ;)).

In onze `Bol` zit echter nog wel een bug, de oppervlakte van een bol, bereken je op een andere manier dan van een cirkel. Voor een cirkel geldt: $O_{cirkel} = \pi r^2$, voor een bol: $O_{bol} = 4\pi r^2$.

Gelukkig kunnen we in een subklasse functies van de superklasse makkelijk overschrijven:

```
1  from cirkel import Cirkel
2  from math import pi
3
4
5  class Bol(Cirkel):
6      # Klasse genaamd Bol, gebasseerd op Cirkel
7
8      def bereken_oppervlakte(self):
9          # Berekent de oppervlakte op basis van de klasse waarde r
10         return 4 * pi * self.r**2
11
12     def bereken_inhoud(self):
13         # Berekent de inhoud op basis van z'n straal:
14         return 4/3 * pi * self.r**3
```

Als we nu onze `Bol` gebruiken, zoals boven aan deze pagina, komen er wel de juiste waardes uit:

```
1 Omtrek: 62.83
2 Oppervlakte: 1256.64
3 Inhoud: 4188.79
```

In dit geval zoekt Python eerst naar de functie `bereken_oppervlakte()` in `Bol`, en vindt 'm daar al, hij zoekt dan niet meer verder. Op deze manier kun je dus functies 'vervangen' van de superklasse in de subklasse.

We doen nog een voorbeeldje. Want we kunnen het overerven ook andersom benaderen. Stel je voor je wil op een OOP manier een kat, een hond en een koe beschrijven. In dit voorbeeld heeft elk dier een aantal variabelen en een aantal functies:

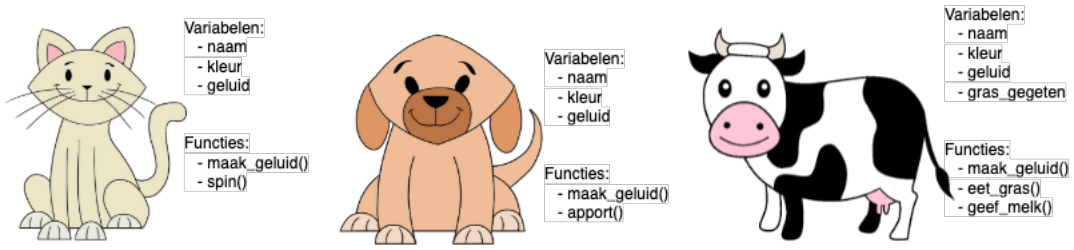


Figure 6.2: Elk dier (kat, hond en koe) heeft z'n eigen eigenschap en functies.

Deze zou je nu los van elkaar gaan uit kunnen werken in code, maar je kunt ook eerst kijken naar de overeenkomsten. Deze overeenkomsten zou je namelijk kunnen verzamelen in een basis superklasse (bijv. 'dier') , waarop je de andere dieren op basseert.

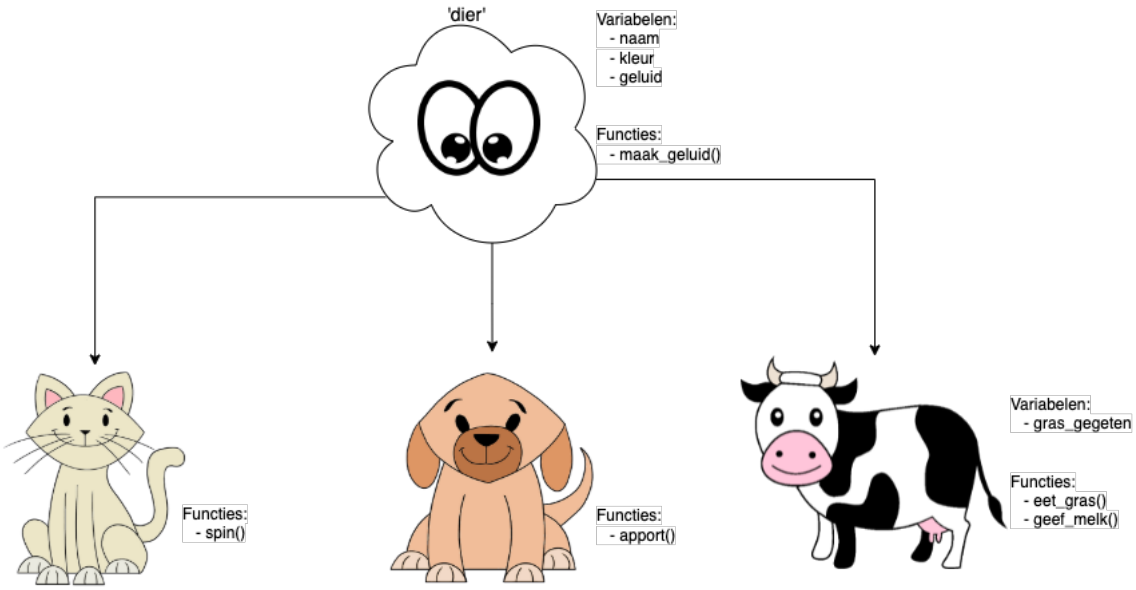


Figure 6.3: Elk dier (kat, hond en koe) heeft z'n eigen eigenschap en functies, maar ook veel overeenkomsten met de basisklasse.

Laten we dit eens uit gaan werken in code. In de klasse-definitie van het basis-Dier (*dier.py*), vinden we dus de `naam`, de `kleur` en het `geluid` dat het dier maakt. Ook zit hierin een functie `maak_geluid()` die het geluid van het dier op het scherm print:

```
class Dier:
    # Basis beschrijving voor een dier
    naam = ''     # String voor de naam van het dier
    kleur = ''    # String voor de kleur van het dier
    geluid = ''   # String voor het geluid dat het maakt

    def __init__(self, naam, kleur, geluid):
        # Constructor om een dier-object te maken.
        self.naam = naam
        self.kleur = kleur
        self.geluid = geluid

    def maak_geluid(self):
        print(self.geluid + "!")
```

In de overige dieren hoeven we dus alleen de functies en variabelen uit te werken die niet in de superklasse `Dier` zitten. Voor het gemak hebben we ze alledrie in hetzelfde bestand gezet:

```
from dier import Dier

class Kat(Dier):
    # Kat-klasse gebasseerd op generiek dier.

    def __init__(self, naam, kleur):
        # Roep constructor van superklasse aan, maar met kat-geluid:
        super().__init__(naam, kleur, 'Miauw')

    def spin(self):
        print('Purrrrrr...')

class Hond(Dier):
    # Honden-klasse gebasseerd op generiek dier.

    def __init__(self, naam, kleur):
        # Roep constructor van superklasse aan, maar met hond-geluid:
        super().__init__(naam, kleur, 'Waf woef')

    def apport(self):
        print('Bal! Bal! Bal!')

class Koe(Dier):
    # Koe-klasse gebasseerd op generiek dier.
    gegeten_gras = 0  # Hoeveelheid gegeten gras

    def __init__(self, naam, kleur):
        # Roep constructor van superklasse aan, maar met koe-geluid:
        super().__init__(naam, kleur, 'Mmmmmmboe')

    def eet_gras(self):
        self.gegeten_gras += 1  # Eet 1 gras.
        print('Om nom nom')

    def geef_melk(self):
        # Geef 1 melk, kost 1 gras.
        if(self.gegeten_gras > 0):
            self.gegeten_gras -= 1
            print('Hier, alsjeblieft: 1 melk!')
        else:
            print('Eerst gras eten..')
```

In de constructor van elk dier gebeurt nog iets bijzonders. Hier wordt namelijk met de hand de constructor van de superklasse `Dier` aangeroepen. Op deze manier worden alle klasse-variabelen met de goede waarde gezet. Nu kunnen we onze gemaakte dieren gebruiken:

```
1  from dieren import Kat, Hond, Koe
2
3  sjors   = Kat('Sjors', 'bruin')          # Een bruine kat genaamd Sjors
4  sjimmie = Hond('Sjimmie', 'knalgroen')   # Een chemische hond genaamd Sjimmie
5  bertha7 = Koe('Bertha 7', 'Wit')         # Een witte koe genaamd Bertha 7
6
7  sjors.maak_geluid()
8  sjimmie.maak_geluid()
9  bertha7.maak_geluid()
10
11 sjors.spin()
12
13 sjimmie.apport()
14
15 bertha7.geef_melk()
16 bertha7.eet_gras()
17 bertha7.geef_melk()
```

De uitvoer laat zich raden (met toegevoegde witregels):

```
1  Miauw!
2  Waf woef!
3  Mmmmmmboe!
4
5  Purrrrrr...
6
7  Bal! Bal! Bal!
8
9  Eerst gras eten..
10 Om nom nom
11 Hier, alsjeblieft: 1 melk!
```

N.B. Wat nu als we een functie proberen uit te voeren van een ander dier?
Bijv.: `bertha7.spin()`. Dan krijg je een error:
`AttributeError:'Koe'object has no attribute 'spin'.`
De klasses die afgeleid zijn van een superklasse hebben geen weet van andere afgeleide klasses.

6.2 Protocollen

Genoeg over abstracte concepten als OOP gehad. Tijd om eens te gaan verdiepen in hoe we onze Raspberry Pi en onze Arduino met elkaar kunnen laten communiciëren. Dit kan op verschillende manieren, beide hebben een vrij uiteenlopend scala aan communicatie mogelijkheden aan boord. We gaan in dit vak aan de slag met twee: via de seriële poort (ook wel: UART) en via Ethernet.

6.2.1 UART

Communicatie via UART of seriële poort wordt op de beide bordjes geregeld door de usb-poort. Je kunt ook bepaalde pinnen op de headers gebruiken, maar gemakshalve blijven we in dit hoofdstuk bij de usb-poorten.

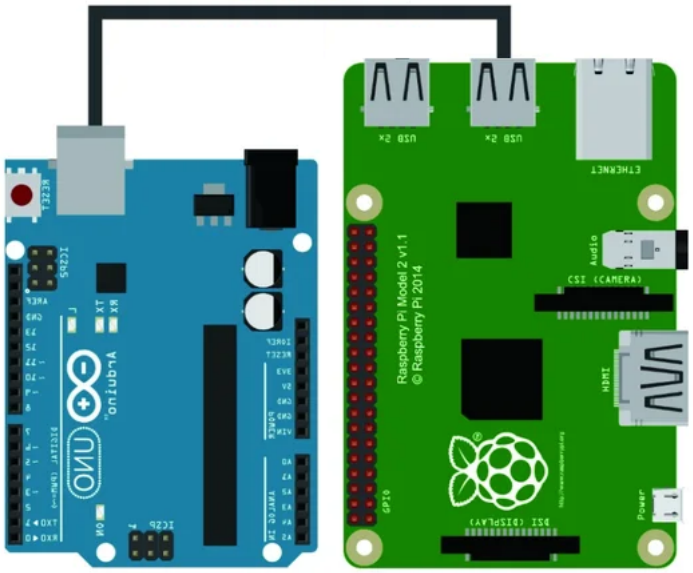


Figure 6.4: De *Raspberry Pi* gekoppeld aan de *Arduino* met een USB kabel.

Opdracht 6.1 Sluit de *Arduino* via USB aan op de *Raspberry Pi*. ■

Onder Windows is de seriële poort, zoals je weet, te vinden als COM*X*, waarbij *X* een letter is (*COM1*, *COM5*, etc.). Omdat er op onze *Raspberry Pi* een Linuxvariant (Raspberry Pi OS) draait, is het verstandig om even stil te staan bij hoe de seriële poort daar werkt.

6.2.2 Seriële poort in Linux

Linux zit nogal anders in elkaar dan Windows. De menubalk die ineens bovenin zit, is het eerste wat in het oog schiet, maar dat is slechts het topje van de ijsberg. Wellicht heb je wel eens een file-explorer gestart, en dan zag het volgende:

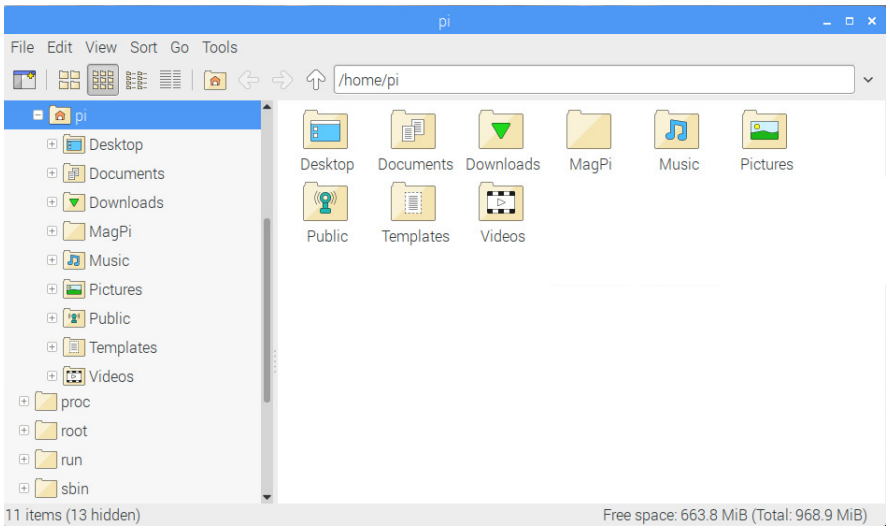


Figure 6.5: De home folder map met de bekende folders

Dat komt waarschijnlijk redelijk bekend voor, de *home* folder (voor de gebruiker *pi*) lijkt nog redelijk op de folder *Mijn Documenten* op Windows. Maar ga nu eens twee mapjes hoger. Je kom nu uit bij de zogeheten *root-folder*, aangegeven met: '*/*'. Deze is vergelijkbaar met de *C:* schijf onder Windows. Waarschijnlijk vraag je je af wat dit allemaal voor vreemde folder zijn, en waar zit dan bijv. Linux geïnstalleerd er is namelijk geen '*/linux*'-folder (zoals er onder windows wel een '*C:\Windows*'-folder is). Tja, dat is dus compleet anders hier. We gaan er niet te diep op in (want dat is een vak an sich), maar een beetje kennis ervan is wel handig als je hardware wil aansturen/uitlezen zoals de seriële poort.

Onder Linux is 'alles een bestand'. Dus niet alleen je standaard tekst-bestanden, plaatjes, video's etc. maar alles. Dus bijv. alle programma's die je draait zijn te vinden onder de '*/proc*'-folder (het tooltje *top* en *htop* halen bijv. hier al hun informatie weg). Informatie over de computer zelf (zoals het aanwezige geheugen en de CPU) kun je vinden in */sys*. En alle aangesloten hardware zijn te vinden onder */dev* (van devices).
Bijv. je harde schijf zit hier (in het geval van de Pi, als */dev/mmcblk0*). Heb je een muis aangesloten? Dan is die te vinden hier als */dev/input/mouse0*. De aangesloten *Arduino* is hier ook te vinden. De vraag is alleen welk bestand moet ik hiervoor hebben?

Het handige daarvoor is om de *Arduino* even los te koppelen en een terminal te openen op de *Pi*. Typ nu het volgende in:

```
ls /dev/tty*
```

Dit geeft een (flinke) lijst met apparaten. Sluit nu de *Arduino* weer aan en voer de bovenstaande regel opnieuw uit. Het apparaat wat er nu bij gekomen is, is de *Arduino*. Onthou deze.

N.B. De *Arduino* zal waarschijnlijk zitten op */dev/ttyUSB0* of */dev/ttyACM0*.

Om de seriële poort uit te lezen zijn is het natuurlijk wel handig dat de *Arduino* daar iets mee doet. We maken even snel een klein programmaatje op de Arduino die precies dat doet en laden dat erop:

```
void setup()
{
  Serial.begin(9600);
}

void loop()
{
  Serial.println("Hallo vanaf Arduino!");
  delay(1000); // Wacht een seconde.
}
```

Opdracht 6.2 Laad het bovenstaande programma op je *Arduino*. Je kunt 'm ook gewoon programmeren op de *Pi* door daar de Arduino-IDE te gebruiken. Is hij toch nog niet geïnstalleerd? Dan:

```
sudo apt install arduino
```

■

6.2.3 PySerial

Tijd om te kijken of we de data die nu gestuurd wordt door de *Arduino* kunnen lezen met de *Pi* via *Python*. Dit gaan we doen met een handige package genaamd *PySerial*.

N.B. Als het goed is, is deze package standaard geïnstalleerd op de *Pi*, zoniet (of als je 'm ook graag op je pc hebben) dan even onderstaande uitvoeren in een terminal:

```
python3 –m pip install pyserial
```

Onderstaande stuk code is alles wat we hiervoor nodig hebben:

```
import serial
from time import sleep

# Open poort, die daarna te gebruiken is als 'ser':
with serial.Serial('/dev/tty0', 9600, timeout=1) as ser:
    sleep(2)                    # Wacht tot Arduino opnieuw is opgestart.
    for i in range(10):
        data = ser.readline()   # Lees tot aan een nieuwe regel ('\n')
        data = data.decode()    # Zet bytes om naar een string
        data = data.strip()     # Verwijder nieuw regel ('\n')
        print(f'{i}: {data}')   # Schrijf string naar scherm
```

We maken hierin een object genaamd `ser` aan, van het type `serial.Serial`. Dit object handeld alle communicatie af met de daadwerkelijke seriële poort. We moeten 'm alleen even vertellen met welke poort en met welke snelheid we willen praten. Voordat dat we iets met de poort doen moeten

we even wachten.
Door een wellicht wat onhandige ontwerpkeuze start de *Arduino* opnieuw op als je de seriële poort opent. Hier is weinig aan te doen, als je hier niet wacht kan het zijn dat je een aantal berichten mis loopt, doordat de *Arduino* nog aan het opstarten is terwijl jij al berichten verwacht. Daarna lezen we de poort 10 keer uit en schrijven wat we terugkrijgen naar het scherm. Hetgeen wat we uitlezen is nog geen `str`, maar `bytes`, vandaar dat we de gelezen data nog even moeten omzetten.

Het keyword `with` is hier nieuw. We kunnen het programma ook schrijven zonder:

```
1  import serial
2  from time import sleep
3
4  # Open poort, die daarna te gebruiken is als 'ser':
5  ser = serial.Serial('/dev/tty0', 9600, timeout=1)
6
7  sleep(2)                    # Wacht tot Arduino opnieuw is opgestart
8
9  for i in range(10):
10     data = ser.readline()   # Lees tot aan een nieuwe regel ('\n')
11     data = data.decode()    # Zet bytes om naar een string
12     data = data.strip()     # Verwijder nieuw regel ('\n')
13     print(f'{i}: {data}')   # Schrijf string naar scherm
14
15 ser.close()  # Sluit poort
```

Maar in dat geval moeten we als we klaar zijn met de poort, de functie `close()` aanroepen zodat de poort netjes wordt afgesloten. Met `with` gebeurt dit automatisch als het programma ermee klaar is. Het is aan te bevelen om het via `with` te doen.

Als we deze code uitvoeren zullen we 10 keer worden begroet door onze *Arduino*, sympathiek!

```
1  0: Hallo vanaf Arduino!
2  1: Hallo vanaf Arduino!
3  2: Hallo vanaf Arduino!
4  3: Hallo vanaf Arduino!
5  4: Hallo vanaf Arduino!
6  5: Hallo vanaf Arduino!
7  6: Hallo vanaf Arduino!
8  7: Hallo vanaf Arduino!
9  8: Hallo vanaf Arduino!
10 9: Hallo vanaf Arduino!
```

Nu het uitlezen van data vanaf de *Arduino* is gelukt, gaan we ons inleren hoe we het de communicatie de andere kant op kunnen opzetten. Hiervoor zijn we een Arduino programma nodig die data op de seriële poort inleest, en hier iets mee doet. In het volgende voorbeeld checkt ons programma of er een 1 of een 0 binnen komt, en zet op basis daarvan een Led (aangesloten op pin $d13$) aan of uit.

```
1  const int led_pin = 13;              // Led, aangesloten op d13
2
3  void setup()
4  {
5    Serial.begin(9600);
6    pinMode(led_pin, OUTPUT);
7  }
8
9  void loop()
10 {
11   // Als er data is:
12   if(Serial.available() > 0)
13   {
14     // Lees deze in:
15     char data = Serial.read();
16     if(data == '1')
17     {
18       digitalWrite(led_pin, HIGH);  // Doe led aan bij een '1'
19     }
20     else if(data == '0')
21     {
22       digitalWrite(led_pin, LOW);   // Doe led uit bij een '0'
23     }
24   }
25 }
```

Op de *Pi* draaien we de volgende code, die eerst het character '1' stuurt en een seconde later het character '0':

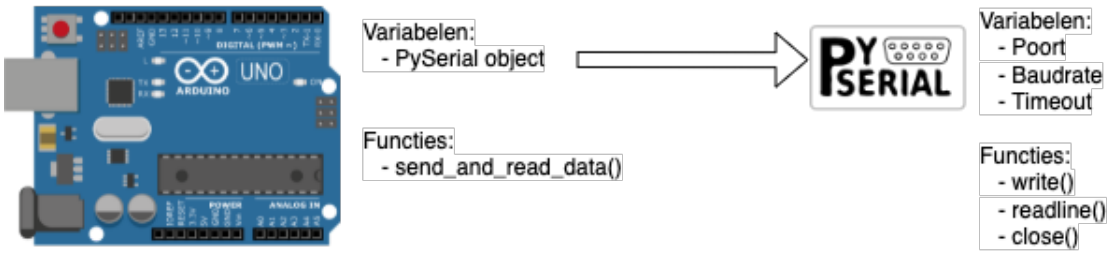
```
1  import serial
2  from time import sleep
3
4  with serial.Serial('/dev/cu.usbmodem1411401', 9600, timeout=.1) as ser:
5      sleep(2)  # Wacht op reset Arduino
6
7      ser.write(b'1')                                  # Stuur '1': Led aan
8      data = ser.readline()                            # Lees binnengekomen data uit
9      print(f'Ontvangen: {data.decode().strip()}')     # Maak data op en print
10
11     sleep(1)  # Even wachten tot Led weer uit moet.
12
13     ser.write('0'.encode())                  # Stuur '0': Led uit
14     data = ser.readline().decode().strip()   # Lees data uit en maak op
15     print(f'Ontvangen: {data}')              # Print data naar scherm
```

An sich is deze code redelijk recht-toe-recht-aan. Maar een belangrijk detail moet je zeker niet over het hoofd zien, op regel 7 en 13 sturen we de 1 en 0. Maar dit is geen standaard `str`, maar `bytes`. Intern werkt *PySerial* enkel met `bytes`. Dus je strings moet je even omzetten voor je ze verstuurd of ontvangt. Dit kan door er een `b` voor te zetten, of door de functie `encode()` aan te roepen op je string.
De ontvangen data moet je dus ook even opmaken (omzetten van `bytes` naar `str` via `decode()` en newline verwijderen met `strip()`). Dit kan direct bij het lezen (regel 14), maar ook gerust bij het printen (regel 8). Van de *Arduino* krijgen we de volgende info terug:

```
1  Ontvangen: Led on!
2  Ontvangen: Led off!
```

OOP variant

Wat nu als we onze eerder opgedane kennis van OOP toe gaan passen op onze *Arduino* communicatie. Dan moeten we eerst weer de standaardvragen stellen. Welke eigenschappen/variabelen heeft onze *Arduino*? En welke functionaliteit moet hij bezitten?
De arduino moet kunnen communiceren met een seriële poort, dus daar is hij een `Serial`-object voor nodig (die weer een poortnaam, baudrate en een timeout moet weten en waar we dan weer de functies `write()`, `readline()` en `close()` van gebruiken). En de *Arduino* heeft op dit moment enkel een functie nodig die data kan zenden en dan direct een antwoord afvangt en teruggeeft. Dat kan in 1 functie.



Als we dit in code uitwerken krijgen we de volgende klasse-definitie:

```
1  import serial
2  from time import sleep
3
4  class Arduino():
5      serial = None  # Poort waarmee gepraat kan worden.
6
7      def __init__(self, port, speed, timeout=.1):
8          # Constructor voor Arduino, open poort:
9          self.serial = serial.Serial(port, speed, timeout=timeout)
10         sleep(2)  # Wacht op reset Arduino
11
12     def send_and_read_data(self, data):
13         # Stuurt een string en geeft string terug:
14         self.serial.write(data.encode())  # Stuur string
15         data = self.serial.readline()     # Ontvang data
16         return data.decode().strip()      # return data
17
18     def __del__(self):
19         # Destructor. Sluit poort als object niet meer nodig is
20         self.serial.close()
```

In de constructor (`__init__()`) van onze Arduino wordt de seriële poort (`serial`) geopend. Voor de poortname, snelheid (baudrate) en timeout, worden de waardes gebruikt die meegegeven worden in de constructor. Daarna wordt er 2 gewacht, zodat de daadwerkelijke *Arduino* even opnieuw kan opstarten.

Op regel 12 is de enige echte functie van onze Arduino klasse: `send_and_read_data()`, deze ontvangt een `str`, zet deze om naar `bytes` en verstuurd 'm over de seriële poort. Daarna wacht hij tot er data binnengekomen is met `readline()`, en deze data wordt weer opgemaakt (omgezet van `bytes` naar `str` en gestript) en teruggegeven.

op regel 18 is weer een nieuw iets te zien, een destructor (`__del__()`). Dit is de tegenhanger van de constructor. Waar de constructor wordt aangeroepen door *Python* als je een nieuw object aanmaakt (`mijn_object =Object(..)`), wordt de destructor aangeroepen als het object niet meer nodig is (bijv. als je programma klaar is, of wellicht heb je het object in een functie of if-statement aangemaakt en is die functie/statement afgerond). Dankzij deze destructor wordt de seriële poort altijd netjes afgesloten.

Je hoeft constructors en destructors dus alleen aan te maken. Deze worden intern en automatisch aangeroepen door *Python*.

We hebben bovenstaande code opgeslagen in een bestand genaamt `arduino.py`. In een nieuw programma kunnen we deze dus weer importeren en gebruiken:

```
1  from arduino import Arduino
2
3  # Maak nieuw object Arduino aan:
4  my_arduino = Arduino('/dev/ACM0', 9600)
5
6  # Stuur '1' en print ontvangen data:
7  data = my_arduino.send_and_read_data('1')
8  print(f'Ontvangen: {data}')
9
10 sleep(1)  # Even wachten
11
12 # Stuur '1' en print ontvangen data:
13 data = my_arduino.send_and_read_data('0')
14 print(f'Ontvangen: {data}')
15 # Hier wordt dus de Arduino-destructor aangeroepen.
```

Dit laat ook weer mooi een ander voordeel van OOP zien: abstractie. Alle technische details zitten verwerkt in onze `Arduino`-klasse, waardoor ons bovenstaande hoofdprogramma daardoor heel recht-toe-recht-aan blijft.

Opdracht 6.3

Voeg aan de arduino-klasse 2 functies toe: `on()` en `off()`, die lamp aan en uit laten gaan.

TIP: i.p.v. stukjes code kopiëren kun je ook `self.send_and_read_data()` gebruiken ;)

■

6.2.4 Ethernet

Naast een USB poort voor seriële communicatie hebben de *Raspberry Pi* en de *Arduino* een ethernetpoort (De *Arduino* is hier uiteraard wel een schild voor nodig).

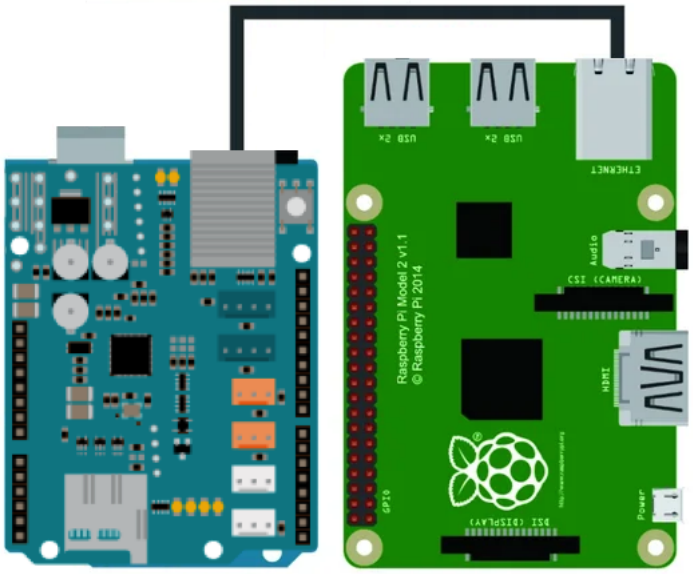


Figure 6.6: De *Raspberry Pi* gekoppeld aan de *Arduino*+shield met een Ethernet kabel.

Opdracht 6.4 Sluit de *Arduino* met een Ethernetkabel aan op de *Raspberry Pi*. ■

We gaan eerst even bekijken hoe je vanuit *Python* via ethernet data kan versturen. Hiervoor zijn we uiteraard een stukje *Arduino*-code voor nodig die die data kan ontvangen en bijv. naar de zijn seriële poort kan schrijven.

```
#include <Ethernet.h>

byte mac[] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
byte ip[] = {192,168,1,3};
EthernetServer server = EthernetServer(23);  // Gebruik poort 23 (telnet)

void setup()
{
  Serial.begin(9600);      // Init Serial
  Ethernet.begin(mac,ip); // Init EthernetShield
  server.begin();
}

void loop()
{
  EthernetClient client = server.available();    // Wacht op een nieuwe client
  if(client)
  {
    while(client.available() > 0)                // Als client is verbonden
    {
      char data = client.read();                 // Haal verstuurde data op
      Serial.print(data);                        // en schrijf naar het scherm
    }
  }
}
```

Opdracht 6.5 Laad het bovenstaande programma op je *Arduino*. ■

Voor het *Python* programma maken we gebruik van de `socket` package, deze zit standaard in *Python* (net als bijv. `math` en `time`) en kun/hoef je dus niet los installeren. Het jammere daaraan is dat het daarom geen kek logo heeft waarmee ik dit document kan opfleuren ;)

Onderstaande programma gebruikt de `socket` package om een bericht te sturen:

```
1 import socket
2
3 arduino_address = '192.168.1.3'
4 arduino_port    = 23
5
6 with socket.socket() as s:
7     s.connect((arduino_address, arduino_port))   # Open connectie
8     s.send(b'Hallo vanuit Python!\n')            # Stuur bericht
```

Het is inderdaad maar een kort programma. Maak een `socket`-object aan (regel 6), zet de connectie op (regel 7) op basis van het gegeven IP addres en poort en verstuur het bericht daarnaar toe. Omdat we weer een `with`-constructie gebruiken, hoeven we wederom niet de connectie zelf af te sluiten (`s.close()`), maar wordt dit dankzij `with` automatisch afgehandeld.

Nu dit werkt, rest natuurlijk de vraag hoe het de andere kant om werkt. In het geval dat de *Arduino* iets stuurt en de *Raspberry Pi* dit moet ontvangen. Ook hiervoor zijn we weer een simpel programmaatje nodig die zodra iemand connectie heeft gemaakt met de *Arduino* begint met het sturen van data. In dit geval doet het programma dat eindeloos door, totdat de andere kant de connectie verbreekt. Als het goed is moet ook dit programma bekend voorkomen, gezien het een hele uitgeklede versie van de *WebserverIO.ino* uit Hoofdstuk 1.

```
1  #include <Ethernet.h>
2
3  byte mac[] = {0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF};
4  byte ip[] = {192,168,1,3};
5  EthernetServer server = EthernetServer(23); //Gebruik poort 23 (telnet)
6
7  void setup()
8  {
9    Ethernet.begin(mac,ip); // init EthernetShield
10   server.begin();
11 }
12
13 void loop()
14 {
15   EthernetClient client = server.available();
16
17   if(client)                                    // Wacht op een nieuwe client
18   {
19     while(client.connected())                   // Zolang client is verbonden:
20     {
21       client.println("Hallo vanuit Arduino!");  // Stuur bericht
22     }
23   }
24 }
```

De bijbehorende code aan de *Python* kant is iets minder triviaal als bij het ontvangen.

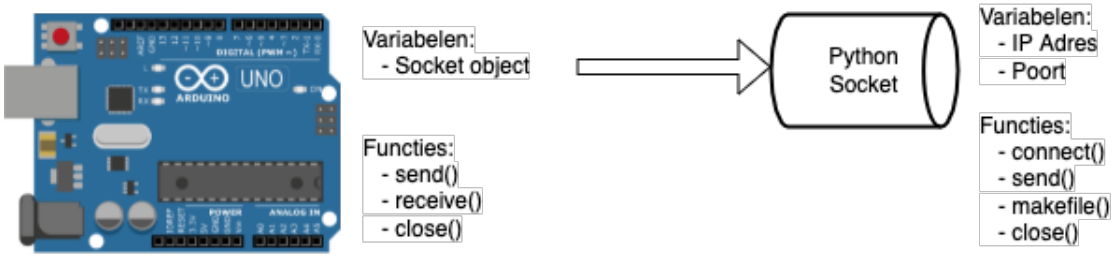
```
1  import socket
2
3  arduino_address = '192.168.1.3'
4  arduino_port    = 23
5
6  with socket.socket() as s:
7      # Open connectie:
8      s.connect((arduino_address, arduino_port))
9
10     # Om de connectie op te zetten, moeten we tenminste iets sturen:
11     s.send(b'Ja! Komt u maar')
12
13     # Lees de binnengekomen data als 'bestand':
14     with s.makefile() as f:
15         for i in range(10):
16             # Lees telkens 1 regel en print deze:
17             data = f.readline()
18             print(f'{i}: {data.strip()}')
```

We beginnen op dezelfde manier: we maken een `socket`-object aan (regel 6), zet de connectie op (regel 8) op basis van het gegeven IP addres en poort. We kunnen pas iets ontvangen nadat we als 'client' (de *Arduino* is in dit geval de server), iets verstuurd hebben. Dus dat doen we op regel 11, het maakt verder niet uit wat dit is.

Daarna wordt er een beetje een truc toegepast. Normaal zou via dit soort verbindingen enkel een stroom aan data krijgen, en moet je van te voren aangeven hoeveel bytes je precies wil lezen. Maar in dit geval (op regel 14) zeggen we dat we de binnengekomen data willen lezen alsof het een bestand is (`f`). Het voordeel daarvan is dat we die wel regel voor regel kunnen lezen. En dat is wat er daarna gebeurt, we lezen 10 keer een regel uit ons (virtuele) bestand `f` en schrijven die naar het scherm. Waar we de vriendelijke begroeting van onze *Arduino* zullen zien.

OOP variant

Ook de communicatie die verloopt met de *Arduino* via het Ethernet kunnen we prima benaderen op een object georiënteerde manier. Als we weer nagaan wat de eigenschappen en functies zijn die nodig zijn. We willen op z'n minst iets versturen en iets ontvangen met de *Arduino*, en daarvoor zijn we een `socket`-object nodig, met een IP adres en een poort nummer. Oftewel:



Als we dit weer uitwerken in een klasse-definitie krijgen we het volgende:

```
1  import socket
2
3  class ArduinoSocket:
4      socket = None
5
6      def __init__(self, address, port):
7          self.socket = socket.socket()          # Maak socket object aan
8          self.socket.connect((address, port))   # Open connectie
9
10     def send(self, msg):
11         self.socket.send(msg.encode())         # Zet om naar bytes en verstuur
12
13     def receive(self, msg='Ja! toe maar'):
14         # Om de connectie op te zetten, moeten we tenminste iets sturen:
15         self.socket.send(msg.encode())
16
17         # Lees de binnengekomen data als 'bestand':
18         with self.socket.makefile() as f:
19             data = f.readline()                # Lees 1 regel
20             return data.strip()                # en geef deze terug
21
22     def close(self):
23         self.socket.close()                    # Sluit connectie
```

In de constructor zorgen we ervoor dat er een `socket`-object wordt aangemaakt op basis van het meegegeven IP adres en de poortnummer. Daarna wordt de connectie gestart. De functies `send()` en `receive()` spreken voor zich.
In plaats van een mooie destructor, hebben hier een normale `close()`-functie. Dit komt omdat streams (wat dit soort dit soort communicaties zijn), niet makkelijk afgerond kunnen worden in de destructor zoals we bij de seriële poort deden. Het gaat voorbij de scope van dit vak om daar helemaal in te duiken, maar voor nu gaan we er vanuit dat we in deze implementatie zelf even de connectie moeten sluiten.

Deze `arduino_socket`-klassse kunnen we daarna op dze manier gebruiken:

```
1  from arduino_socket import ArduinoSocket
2
3  # Open connectie met Arduino:
4  mijn_arduino = ArduinoSocket('192.168.1.3', 23)
5
6  # Haal 10 keer een bericht op:
7  for i in range(10):
8      data = mijn_arduino.receive()
9      print(f'{i}: {data}')
10
11 # Sluit na afloop de connectie:
12 mijn_arduino.close()
```

En daarmee zijn we aan het eind gekomen van de theorie van dit vak. Hopelijk voelen jullie je nu een behoorlijke hacker en kunnen jullie met deze abstracte manieren om problemen op te lossen uit de voeten en daarmee mooie dingen gaan maken!

6.3 Huiswerkopdrachten

Opdracht 6.6

■

Opdracht 6.7

■

Opdracht 6.8

■

Opdracht 6.9

■

Opdracht 6.10

■

Opdracht 6.11

■

Opdracht 6.12

■

Opdracht 6.13

■

Index