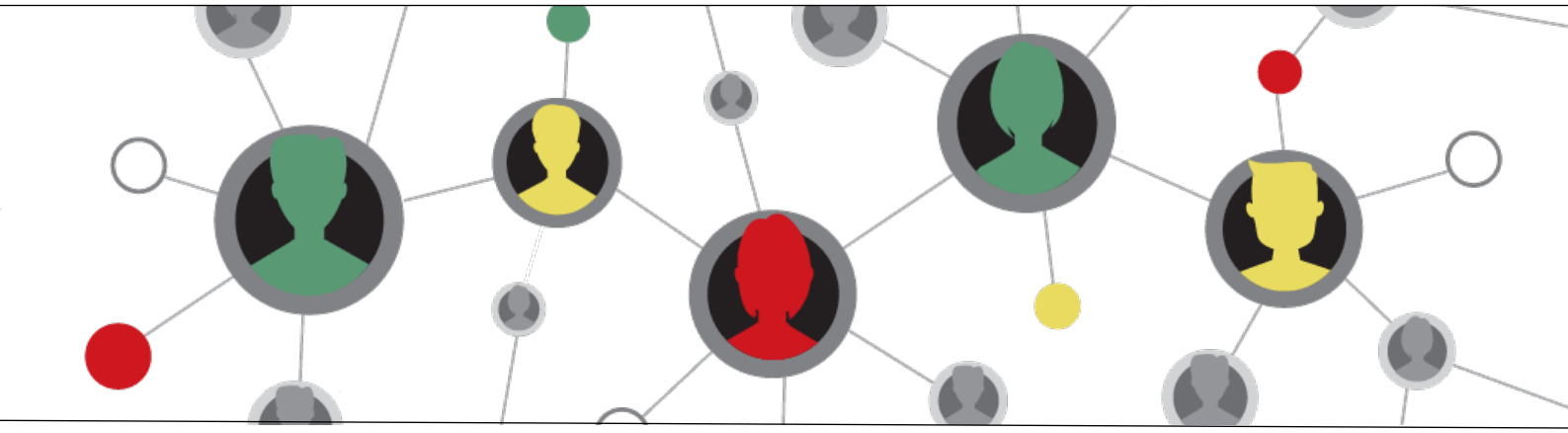
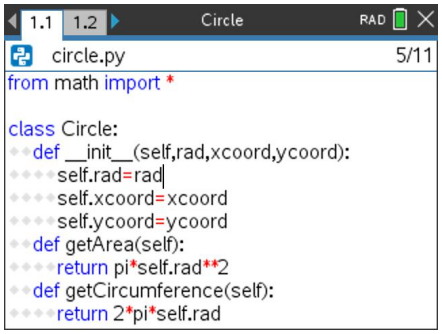
TI Python BootCamp

Deel 5 – Objectgeoriënteerd programmeren





T³ NEDERLAND

Teachers Teaching with Technology™

T³ VLAANDEREN

1. Objectgeoriënteerd

Objectgeoriënteerd programmeren is een systeem gebaseerd op objecten die bestaan uit data/gegevens (attributen) en code (methodes) om deze gegevens te bewerken of verwerken.

In Python worden objecten gemodelleerd als instanties (elementen) van een klasse. Een klasse kan beschouwd worden als een blauwdruk die de definitie van een object bepaalt.

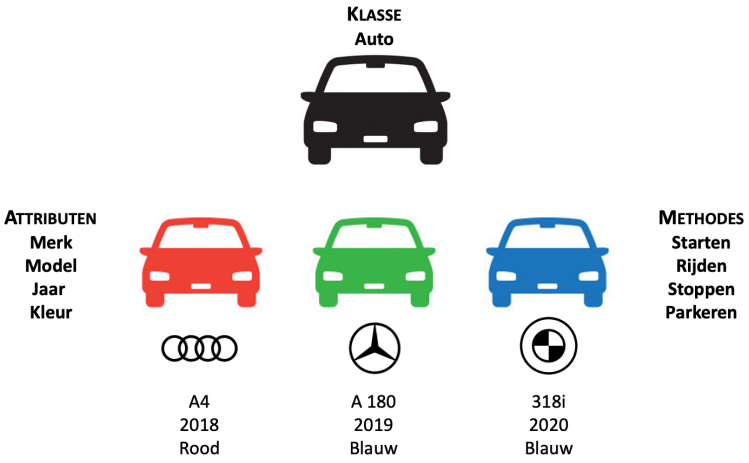
Een klasse bestaat uit:

- **Attributen** die de toestand/eigenschappen van een object vastleggen
- **Methodes** of functies die de toestand van het object aanpassen en het gedrag van een object bepalen.

Een speciale methode (de constructor) instantieert een object met zijn eigen waarden van de attributen; de toestand van een object.

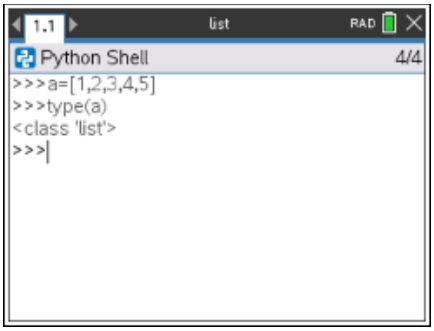
Object Oriented Programming (OOP) laat programmeurs toe hun eigen objecten en bijhorende methodes te construeren en zo hun code modulair op te bouwen.

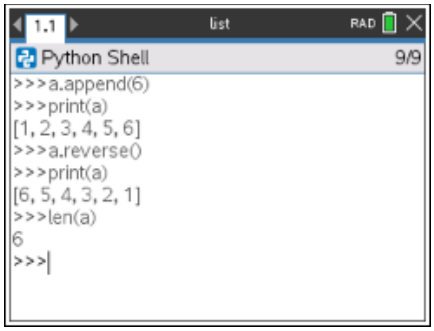
Klassen en objecten kan je vergelijken met objecten uit de ons omringende wereld zoals bv een auto.



In Python ***Everything is an object***.

In deze TI Python Bootcamp hebben we al veel gewerkt met objecten en methodes. Een voorbeeld hiervan is het data-type lijst met de methodes append() & reverse() en de speciale methodes print() & len().





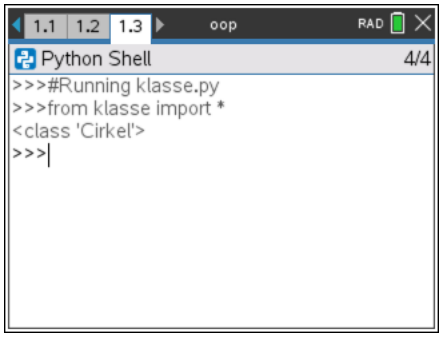
We bekijken de basics van objectgeoriënteerd programmeren in Python.

2. Klassen

User defined objecten kunnen gecodeerd worden met het keyword class.

We illustreren de syntax van het coderen van klassen met het concept cirkel.

```
# Definitie nieuw object type
class Cirkel:
    pass

# Declaratie object van de klasse Cirkel
c=Cirkel()

print(type(x))
```



Het statement pass kan je gebruiken als een placeholder voor code. Wanneer pass wordt uitgevoerd, gebeurt er niets maar het geeft geen error. Lege code-blokken genereren namelijk een error.

Een Python-conventie geeft aan dat een klasse-naam start met een hoofdletter; net zoals we klein letters gebruiken voor variabelen.

3. Attributen

Een cirkel is analytisch volledig bepaald door zijn middelpunt en de straal. Het middelpunt en de straal gaan we coderen als de attributen(eigenschappen) van een object van de klasse Cirkel.

De syntax voor het creëren van een attribuut is b.v. self.rad = rad als onderdeel van de speciale methode __init__().

De methode __init__() initialiseert de attributen van een object en deze methode wordt steeds uitgevoerd bij het aanmaken van een object.

De algemene syntax voor het coderen van de attributen van een klasse is de volgende:

```
class Cirkel:
    def __init__(self,parameter1,parameter2):
        self.parameter1 = parameter1
        self.parameter = parameter2
```

Het keyword **self** representeerd de instantie van de klasse en wordt gebruikt om naar zichzelf te wijzen. Het is niet noodzakelijk de naam self te gebruiken maar het is een afspraak tussen Python-programmeurs.

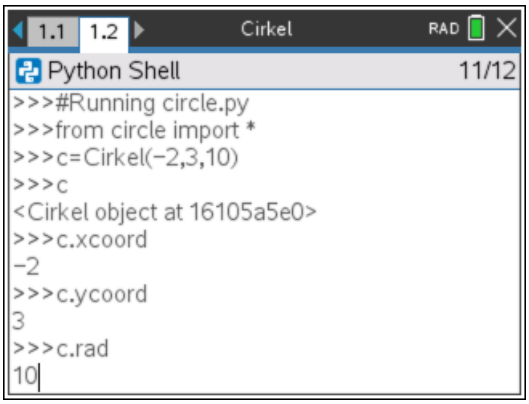
Deze syntax kan wat eigenaardig overkomen (b.v. 3x parameter1). Ook dit is een python-afspraak om telkens driemaal hetzelfde woord te gebruiken. Het is niet noodzakelijk en het zal geen error geven indien niet zo.

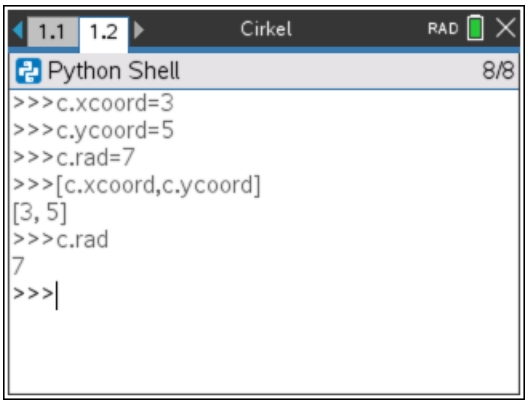
Hieronder twee dezelfde klassen voor het object Cirkel:

```
class Cirkel:
    def __init__(self,xcoord,ycoord,rad):
        self.xcoord = xcoord
        self.ycoord = ycoord
        self.rad = rad
```

```
class Cirkel:
    def __init__(self,xc,yc,straal):
        self.xcoord = xc
        self.ycoord = yc
        self.rad = straal
```

Na het declareren van een cirkel object – c = Cirkel(-3,10,2) wordt een object Cirkel aangemaakt in het geheugen. De attributen(eigenschappen) kunnen als volgt opgeroepen worden en eventueel aangepast.



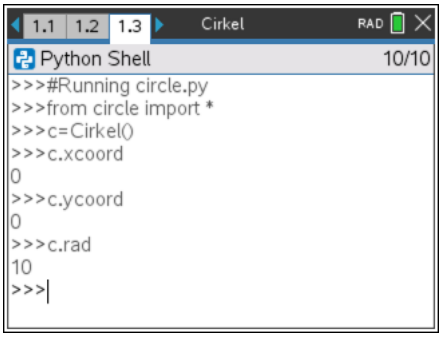


Voor de argumenten van __init__ (alsook voor andere methodes) kan als volgt aan de argumenten een standaard-waarde toegekend worden:

```
def __init__(self,xcoord=0,ycoord=0,rad=10).
```

Een Cirkel-object kan in dit geval gedeclareerd worden zonder argumenten

```
c = Cirkel().
```

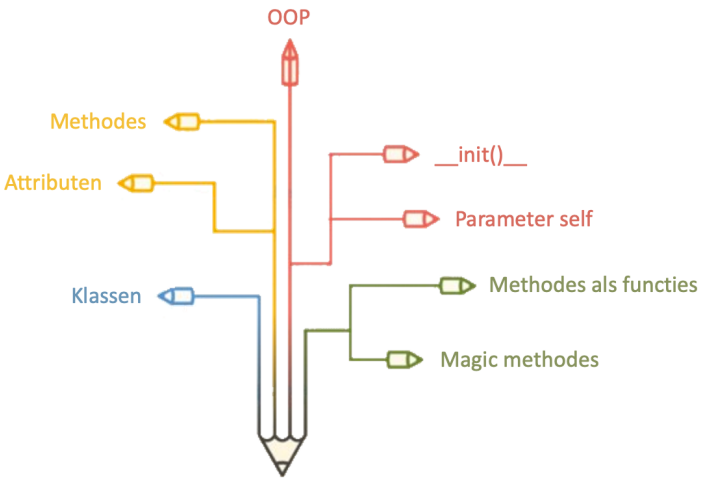


Tot hiertoe hebben we de volgende terminologie van object georiënteerd programmeren geïllustreerd:

- een **klasse**, het model of de standaard van de mogelijkheden wat een object kan zijn en doen
- een **object**, de instantie van een klasse m.a.w. een concrete entiteit van een klasse
- een **instantie**, is als een object maar laten we even terug naar het concept auto:
  - een blauwdruk voor een auto-ontwerp is de klasse-beschrijving
  - alle auto's die op basis van die blauwdruk zijn vervaardigd, zijn objecten van die klasse
  - uw auto die van die blauwdruk is gemaakt, is een instantie van die klasse.

De termen instantie en object worden vaak door elkaar gebruikt en de meest voorkomende term is object.

In het volgende deel bespreken we **methodes** van een klasse.



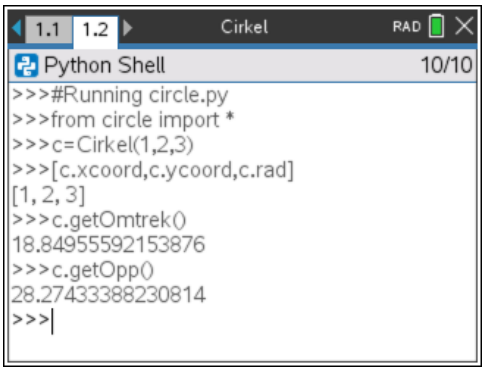
1. Methodes

Methodes zijn functies gedefinieerd in een klasse. Methodes worden gebruikt om operaties uit te voeren op/met de attributen van een klasse.

Een methode is een functie op een object van een klasse die het object zelf aanspreekt met het klasse-argument self.

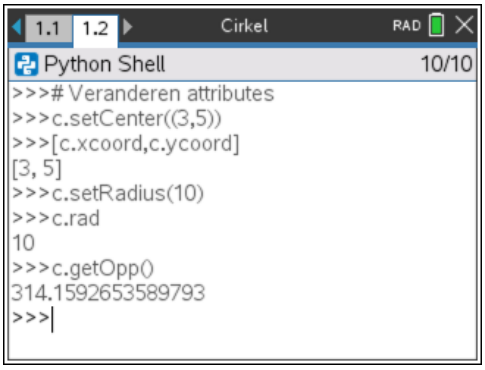
We illustreren enkele methodes voor de klasse Cirkel voor het berekenen van de omtrek en de oppervlakte.

```
from math import *
from ti_draw import *

class Cirkel:
    def __init__(self,xcoord,ycoord,rad):
        self.xcoord = xcoord
        self.ycoord = ycoord
        self.rad = rad

    def getOmtrek(self):
        return 2*pi*self.rad

    def getOpp(self):
        return pi*self.rad**2
```



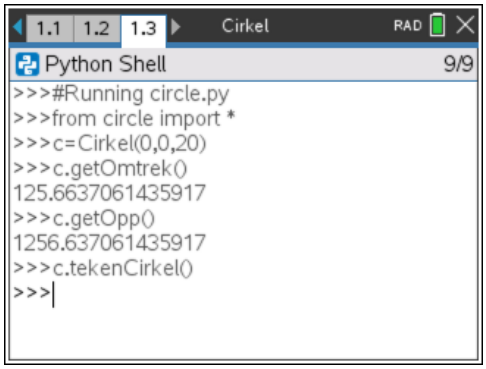
Methodes kunnen ook gebruikt worden om de attributen van een object te veranderen

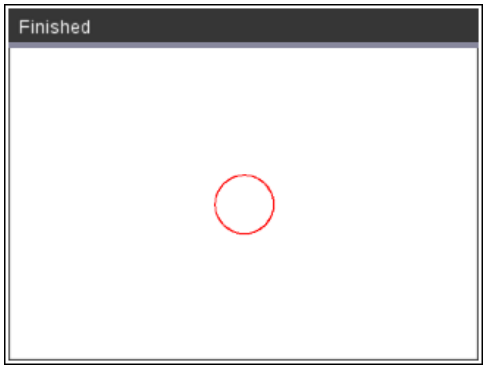
```
    def setRadius(self,newrad):
        self.rad=newrad

    def setCenter(self,newcenter):
        self.xcoord,self.ycoord=newcenter
```



en een object grafisch voor te stellen

```
    def tekenCirkel(self):
        w,h=get_screen_dim()
        set_window(-w/2,w/2,-h/2,h/2)
        set_color(255,0,0)
        draw_circle(self.xcoord,self.ycoord,self.rad)
```





2. Magic methodes

Voor Python-klassen kunnen speciale methodes, magic methodes, geïmplementeerd worden, gebruikmakend van speciale syntax. We maakten al kennis met __init__(), de methode die de attributen van de klasse definieert.

We bekijken nog twee andere magic methodes:

- __str__() string-representatie van een object
- __len__() geeft de (user defined) lengte van een object

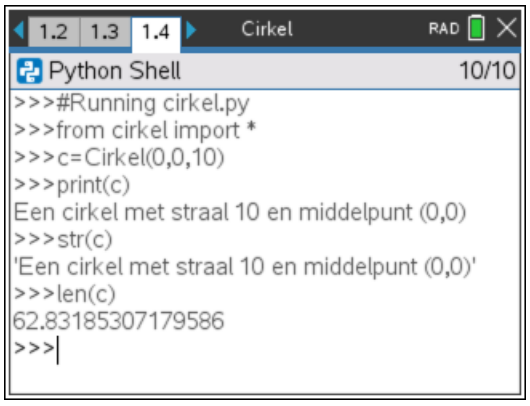
We herschrijven de klasse Cirkel als volgt:

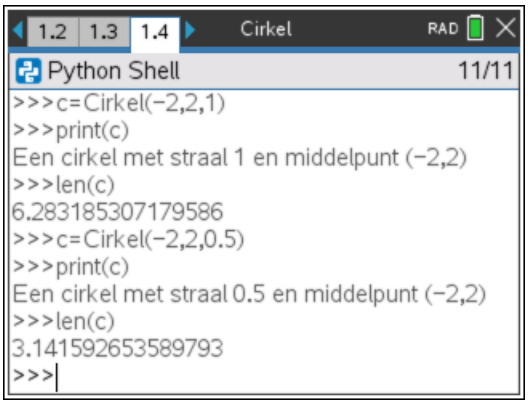
```
from math import *
class Cirkel:
    def __init__(self,xcoord,ycoord,rad):
        self.xcoord=xcoord
        self.ycoord=ycoord
        self.rad=rad
        self.omtrek=2*pi*rad
        self.opp=pi*rad**2
```

En we definiëren de volgende __str__() en __len__ magic methodes voor de klasse Cirkel:

```
    def __str__(self):
        return "Een cirkel met straal {} en middelpunt ({},{})".format(self.rad,self.xcoord,self.ycoord)

    def __len__(self):
        return self.omtrek
```

Deze methodes voeren we als volgt uit op een object:





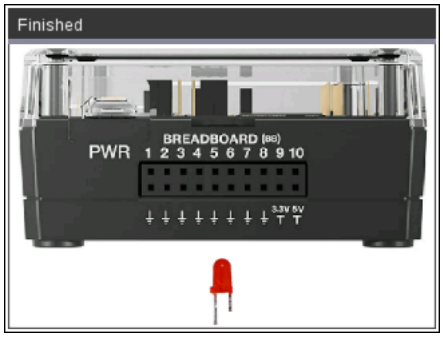
1. Virtuele LEDs

Met de grafische modules en het definiëren van klassen, kunnen we op een vrij makkelijke en gebruiksvriendelijke manier STEM-experimenten simuleren. We bekijken twee voorbeelden: LEDs en een RGB-array.

1.1. Knipperend LED

We starten met het bepalen van de achtergrond. Voor dit voorbeeld een afbeelding van de BB-voorkant van de TI-Innovator™ hub en een afbeelding van een LED. We gebruiken hiervoor de functie background().

```
from ti_draw import *
from ti_image import *

def background():
    hub = load_image("hub")
    hub.show_image(0,0)
    led = load_image("led")
    led.show_image(150,160)
```



OFF

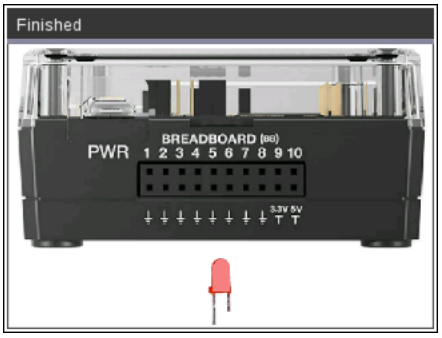
De LED definiëren we als een klasse met als attributen:

- de bijhorende figuur
- de x- en y-coördinaat waar de figuur te tonen

```
class Led():

    def __init__(self):
        self.img = load_image("led")
        self.x = 150
        self.y = 160
```

Als methodes coderen we de functies on() en off(); functies die ook beschikbaar in de TI Hub-modules.

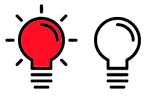
```
    def on(self):
        set_color(255,0,150)
        fill_rect(self.x+3,self.y+5,13,20)
        fill_circle(self.x+9,self.y+6,6)

    def off(self):
        self.img.show_image(self.x,self.y)
```

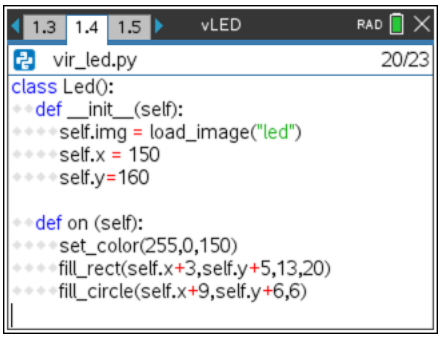


ON

Bovenstaande code bewaren we als het programma vir_led.py zodat we de objecten van de klasse Led() kunnen gebruiken in andere programma's, b.v. voor het knipperen van een LED:

```
from vir_led import *
from ti_system import *
from time import *

use_buffer()

background()
paint_buffer()

led = Led()

while get_key() != "esc":
    led.on() ; paint_buffer()
    sleep(0.3)
    led.off() ; paint_buffer()
    sleep(0.3)
```

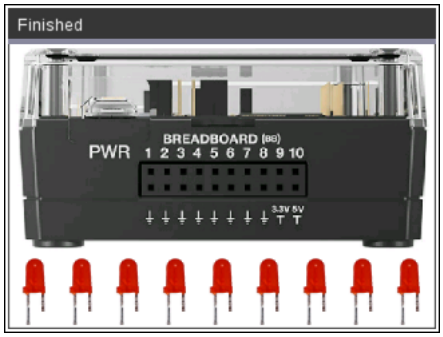




1.2. Lopend LED

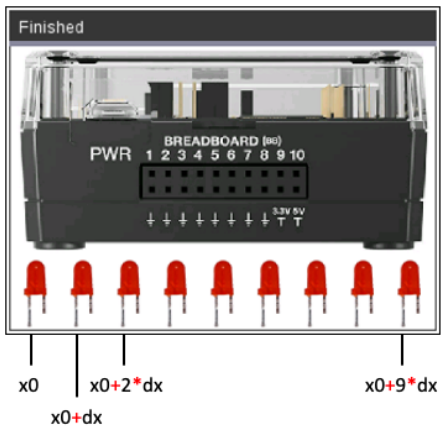
In het volgende voorbeeld gebruiken we 9 LEDs en breiden de achtergrond als volgt uit:

```
def background():
    hub = load_image("hub")
    led = load_image("led")
    hub.show_image(0,0)
    for i in range(9):
        led.show_image(10+i*35,160,160)
```



We passen de attributen van de klasse Led() aan zodat iedere LED bepaalt wordt door een BB-poort: “BB 1” tot en met “BB 9”.

```
class Led():
# port = “BB 1”, “BB 2”, ... of “BB 9”
    def __init__(self,port):
        self.port=port
        self.factor = int(self.port[3])
        self.img = load_image("led")
        self.dx = 35
        self.x0 = 10
        self.x = self.x0+(self.factor-1)*self.dx
        self.y = 160
```

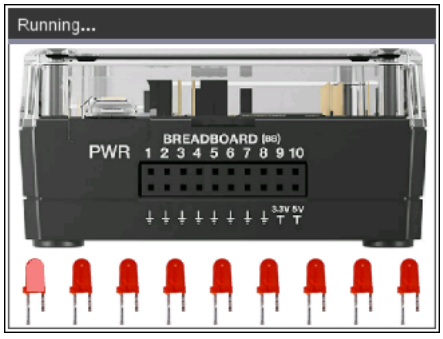


De methodes blijven hetzelfde en we bewaren deze codes als vir_hub.py.

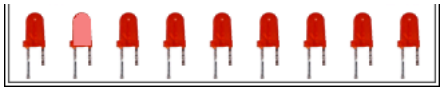
```
    def on(self):
        set_color(255,0,150)
        fill_rect(self.x+3,self.y+5,13,20)
        fill_circle(self.x+9,self.y+6,6)

    def off(self):
        self.img.show_image(self.x,self.y)
```

Een lopend LED coderen we met de klasse Led() als volgt:

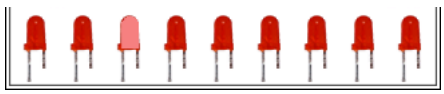
```
from vir_hub import *
from ti_system import *
from time import *

use_buffer()

background()
paint_buffer()

leds = []
for n in range(9):
    leds.append (Led("BB " + str(n+1)))

while get_key() != "esc":
    for vled in leds:
        vled.on() ; paint_buffer() ; sleep(0.2)
        vled.off() ; paint_buffer() ; sleep(0.2)
```



led = LED(“BB 1”) ; led.on()



led = LED(“BB 2”) ; led.on()

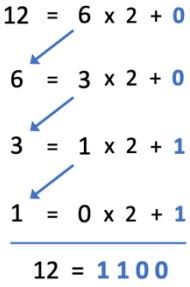


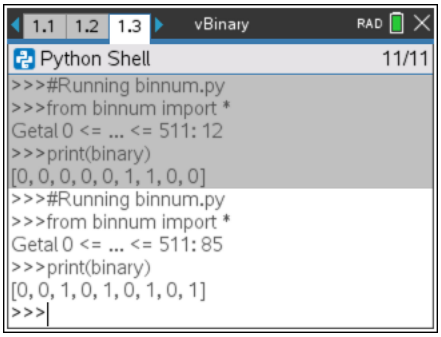
led = LED(“BB 3”) ; led.on()

1.3. Binaire LED-voorstelling

We definiëren eerst de functie dec2bin() voor de conversie van een natuurlijk getal, tussen 0 en 511, naar het corresponderende binair getal; telkens bestaande uit 9 digits.

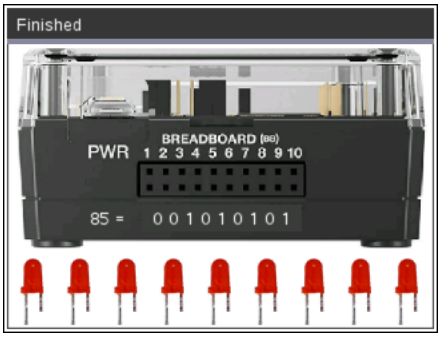
```
from vir_hub import *
from ti_system import *
from time import *

def dec2bin(t):
    bin=[]
    while t//2 != 0:
        bin.append(t%2)
        t=t//2
    bin.append(t%2)
    # Aanvullen met 0'en tot 9 digits
    for i in range(0,9-len(bin)):
        bin.append(0)
    bin.reverse()
    return bin

n=int(input("Getal 0 <= ... < = 511: "))
binary=dec2bin(n)
```

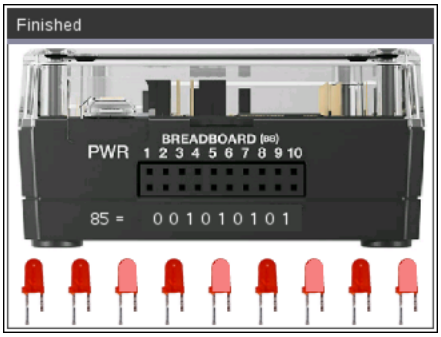




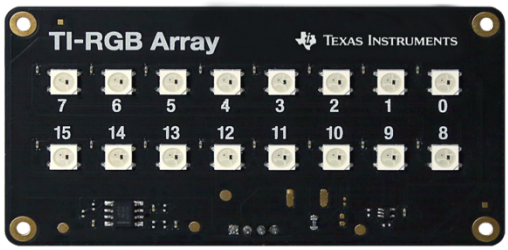
Met onderstaande code en gebruikmakend van vir_hub.py, printen we de binaire voorstelling en visualiseren we deze voorstelling m.b.v. de 9 LEDs.

```
background()

# Printen van de binaire voorstelling van n
set_color(57,60,57)
fill_rect(100,120,120,20)
set_color(175,175,175)
draw_text(62,138,"{} = ".format(n))
for i in range(len(binary)):
    draw_text(109+i*12,138,binary[i])
paint_buffer()

sleep(2)

# Binaire visualisatie met de negen Leds
for i in range(9):
    if binary[i] == 1:
        binled=Led("BB "+str(i+1))
        binled.on()
paint_buffer()
```





2. Virtuele RGB-array



Voor de TI-Innovator™ Hub is een TI-RGB array beschikbaar met 16 programmeerbare LEDs.

In de module TI Hub is een klasse rgb_array() beschikbaar. Voor een object uit deze klasse zijn o.a. de volgende methodes beschikbaar:

- set(led_positie,rood,groen,blauw)
- set_all(rood,groen,blauw)
- all_off()

We simuleren deze TI-RGB array als volgt.

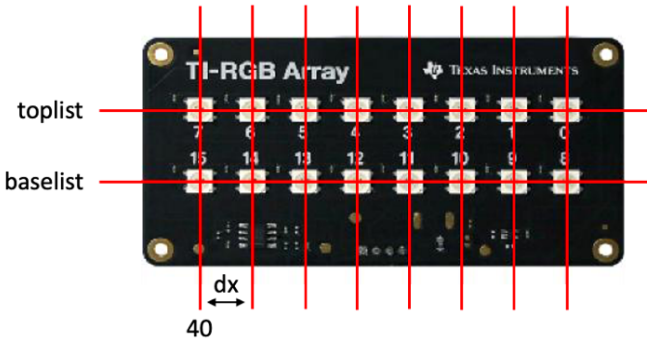
Als achtergrond gebruiken we een afbeelding van de TI RGB array.

```
from ti_image import *
from ti_draw import *
from time import *

def background():
    rgb=load_image("rgb")
    rgb.show_image(0,0)
```

Voor de attributen voor de klasse Array() definiëren we hoofdzakelijk lijsten met de coördinaten van de middelpunten van de cirkels die de LEDs simuleren:

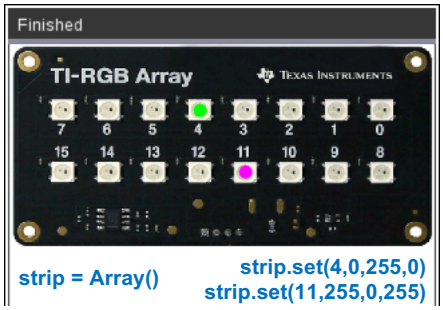
- toplist middelpunten LEDs: 0-7
- baselist middelpunten LEDs: 8-15
- leds middelpunten LEDs: 0-15
- topoff simulatie off van LEDs 0-7
- baseoff simulatie off van LEDs 8-15
- dx afstand tussen LEDs
- rad straal LEDs
- image bijhorende figuur

```
class Array:
    def __init__(self):
        self.dx=34
        self.toplist=[[40+i*self.dx,49] for i in range(8)]
        self.baselist=[[40+i*self.dx,95] for i in range(8)]
        self.leds=self.toplist+self.baselist
        self.topoff=[[41+i*self.dx,48] for i in range(8)]
        self.baseoff=[[42+i*self.dx,95] for i in range(8)]
        self.image=load_image("rgb")
        self.rad=5
```

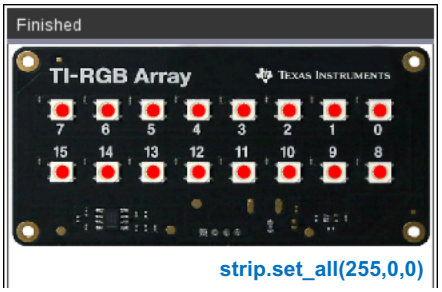


Als methodes coderen we de volgende functionaliteit; ook beschikbaar in de module TI Hub.

```
    def set(self,p,r,g,b):
        set_color(r,g,b)
        use_buffer()
        if p<=7:
            fill_circle(self.toplist[7-p][0],self.toplist[7-p][1],self.rad)
        else:
            fill_circle(self.baselist[15-p][0],self.baselist[15-p][1],self.rad)
        paint_buffer().
```



```
    def set_all(self,r,g,b):
        set_color(r,g,b)
        use_buffer()
        for i in range(16):
            fill_circle(self.leds[i][0],self.leds[i][1],self.rad)
        paint_buffer()
```



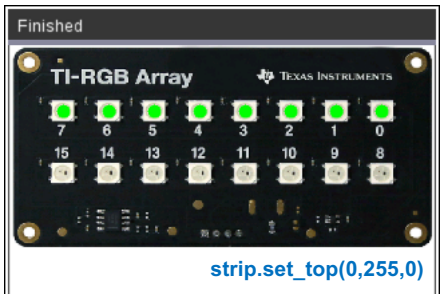
strip.set_all(255,0,0)

```
    def all_off(self):
        self.image.show_image(0,0)
```

strip.all_off(255,0,0)

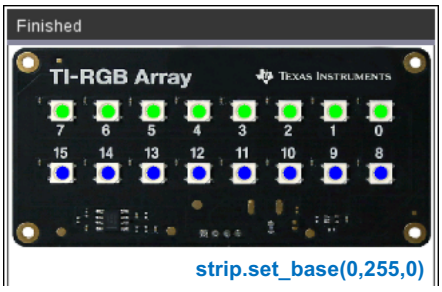
Als extra functionaliteit voegen we de volgende methodes toe en bewaren alle code in vir_rgb.py.

```
    def set_top(self,r,g,b):
        set_color(r,g,b)
        use_buffer()
        for i in range(8):
            fill_circle(self.toplist[i][0],self.toplist[i][1],self.rad)
        paint_buffer()
```



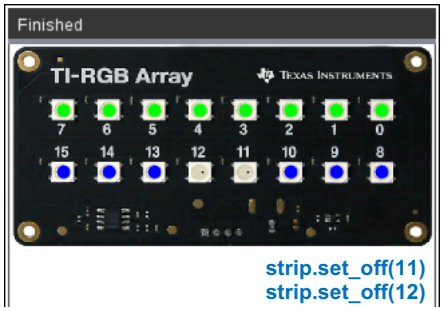
strip.set_top(0,255,0)

```
    def set_base(self,r,g,b):
        set_color(r,g,b)
        use_buffer()
        for i in range(8):
            fill_circle(self.baselist[i][0],self.baselist[i][1],self.rad)
        paint_buffer()
```



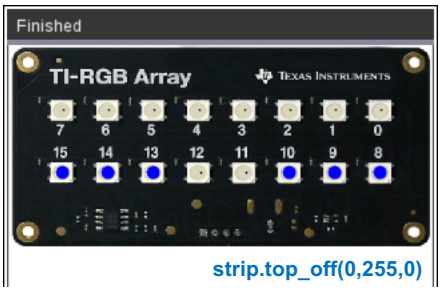
strip.set_base(0,255,0)

```
    def set_off(self,p):
        set_color(217,214,187)
        use_buffer()
        if p<=7:
            fill_circle(self.toplist[7-p][0],self.toplist[7-p][1],self.rad)
            set_color(0,0,0)
            fill_circle(self.topoff[7-p][0],self.topoff[7-p][1],1)
        else:
            fill_circle(self.baselist[15-p][0],self.baselist[15-p][1],self.rad)
            set_color(0,0,0)
            fill_circle(self.baseoff[15-p][0],self.baseoff[15-p][1],1)
        paint_buffer()
```



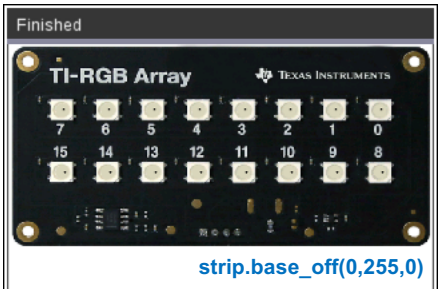
strip.set_off(11)
strip.set_off(12)

```
    def top_off(self):
        set_color(217,214,187)
        use_buffer()
        for i in range(8):
            fill_circle(self.toplist[i][0],self.toplist[i][1],self.rad)
        set_color(0,0,0)
        for i in range(8):
            fill_circle(self.topoff[i][0],self.topoff[i][1],1)
        paint_buffer()
```



strip.top_off(0,255,0)

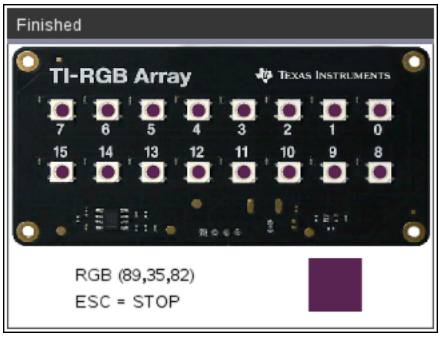
```
    def base_off(self):
        set_color(217,214,187)
        use_buffer()
        for i in range(8):
            fill_circle(self.baselist[i][0],self.baselist[i][1],self.rad)
        set_color(0,0,0)
        for i in range(8):
            fill_circle(self.baseoff[i][0],self.baseoff[i][1],1)
        paint_buffer()
```

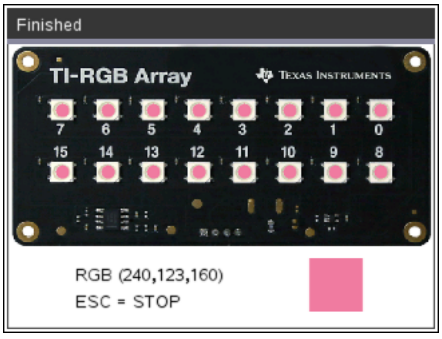


strip.base_off(0,255,0)

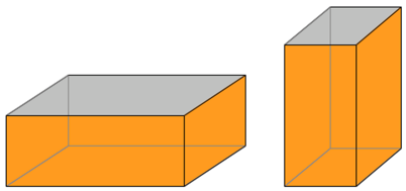
We eindigen met de onderstaande at random kleurenanimatie.

```
from vir_rgb import *
from ti_system import *
from random import *

background()
strip=Array()

use_buffer()

while get_key() != "esc":
    r=randint(0,255)
    g=randint(0,255)
    b=randint(0,255)
    set_color(r,g,b)
    strip.set_all(r,g,b)
    fill_rect(225,160,40,40)
    set_color(255,255,255)
    fill_rect(50,160,150,50)
    set_color(0,0,0)
    draw_text(50,180,"RGB ({},{},{})".format(r,g,b))
    draw_text(50,200,"ESC = STOP")
    paint_buffer()
    sleep(2)
```





1. Een balk



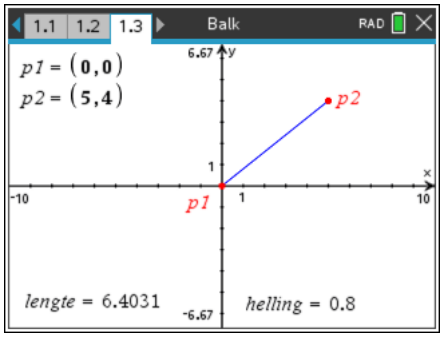
Definieer een klasse Balk() met
- als attributen de lengte, breedte en hoogte en
- als methodes het volume en de oppervlakte

2. Lengte en helling van een segment

a. Definieer een klasse Segment() met
   - als attributen de coördinaten van begin- en eindpunt als tuples en
   - als methodes de lengte en de helling van het segment

b. Codeer een methode die de coördinaten van het begin- en eindpunt van een segment uitvoeren naar TI-Nspire CX-variabelen waarmee het segment in Graphs getekend wordt.

c. Codeer een methode die de coördinaten van het begin- en eindpunt van een segment in de TI-Nspire CX-applicatie Graphs naar een object van de klasse Segment() importeren.



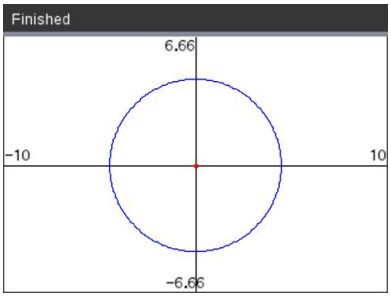
3. Kegelsnede

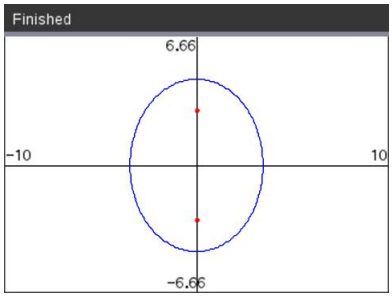
Programmeer een klasse Kegelsnede() met als attributen/argumenten a, b en $c > 0$ met a, b, c parameters van de vergelijking $\frac{x^2}{a} + \frac{y^2}{b} = c$.

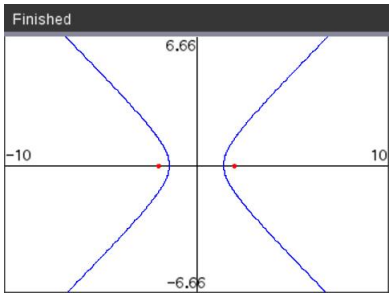
Deze vergelijking stelt een cirkel, ellips of hyperbool voor afhankelijk van de waarde van a, b, c :

- $a = b > 0$ ⇒ de vergelijking bepaalt een cirkel
- $a, b > 0$ en $a \neq b$ ⇒ de vergelijking bepaalt een ellips
- $a \cdot b < 0$ ⇒ de vergelijking bepaalt een hyperbool
- anders ⇒ de vergelijking bepaalt geen kegelsnede

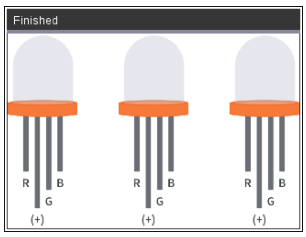
Definieer methodes die op basis van de argumenten a, b en c de kegelsnede bepaalt, onderstaande karakteristieken genereert en de kegelsnede plot in een orthonormaal assenstelsel.







4. Een virtuele RGB-LED



"OUT1" "OUT 2" "OUT3"

Ontwikkel het volgende virtuele RGB-led-experiment.

a. Download een figuur van een RGB-led en creëer een background met driemaal de figuur van de RGB-led.

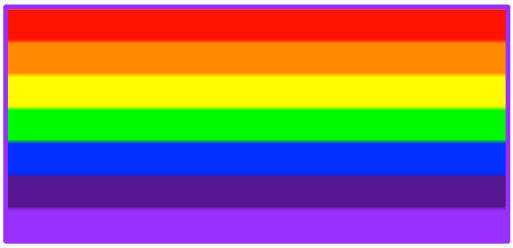
b. Definieer een klasse RGB() met als argument de poort – "OUT 1", "OUT 2", "OUT 3" – die aangeeft welke figuur er bij een object van de klasse RGB correspondeert, respectievelijk links, midden of rechts.

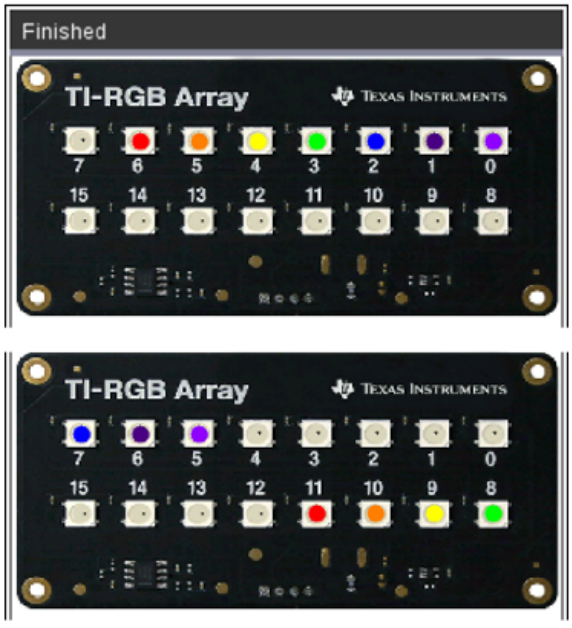
c. Codeer de methode set() die iedere led apart een rgb-kleur geeft en de methode off() die iedere led kan uitzetten.

5. Een kleurig looplicht voor de virtuele RGB-array

Genereer met de module vir_rgb.py (zie BootCamp Deel 5) een gekleurd looplicht, b.v. met de kleuren van de regenboog. Voor de kleuren van de regenboog kan gebruik gemaakt worden van een lijst met de rgb-kleuren:

[[255,0,0],[255,127,0],[255,255,0],[0,255,0],[0,0,255],[75,0,130],[143,0,255]]





Als uitbreiding, voeg een parameter toe die de lengte van het looplicht (het aantal LEDs) aanpast.

1. Enkele karakteristieken van OOP

1.1. Inheritance (overerving)

Overerving is een manier om nieuwe klassen te definiëren op basis van bestaande klassne. Een nieuwe child-klasse die overerft van een bestaande parent-klasse, neemt bepaalde attributen en methodes over van de parent-klasse. M.a.w. de child-klasse gebruikt code van de parent-klasse maar kan ook nieuwe attributen en methodes toevoegen. Terminologie die hier ook wel gebruikt wordt is basis-klasse en afgeleide (sub)klasse.

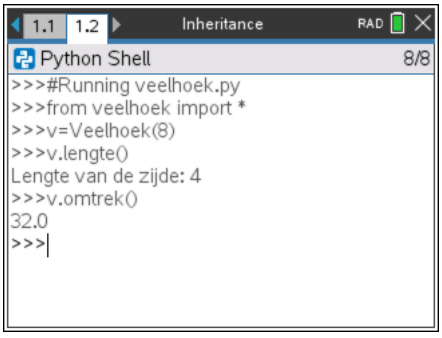
De python-syntax is de volgende:

```
class BasisKlasse():
    Blok

class AfgeleideKlasse(BasisKlasse):
    Blok
```

Voorbeeld
Omtrek van een regelmatige veelhoek

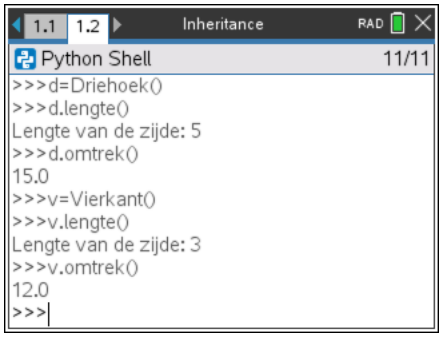
We definiëren als basis-klasse de klasse Veelhoek() met als attributen het aantal zijden en als methodes input van de lengte van de zijde en het berekenen van de omtrek.

```
class Veelhoek():
    def __init__(self, aantal_zijden):
        self.aantal_zijden = aantal_zijden
    def lengte(self):
        self.lengte = float(input("Lengte van de zijde: "))
    def omtrek(self):
        return self.aantal_zijden*self.lengte
```

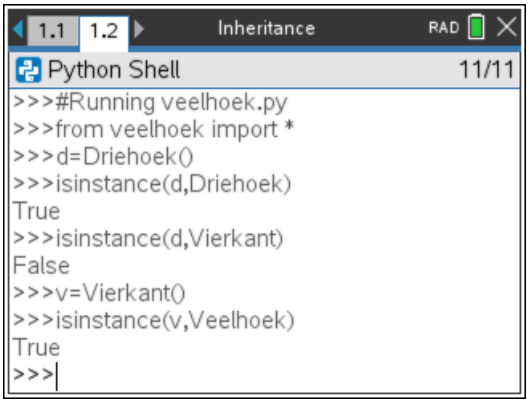


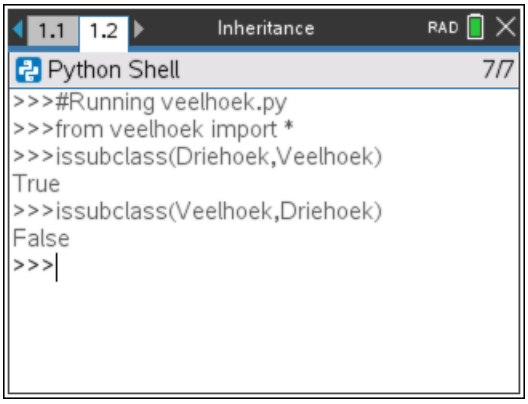
Als afgeleide klassen definiëren we de subklassen Driehoek(), Vierkant(), Vijfhoek() en Zeshoek(). Al deze klassen kunnen gebruik maken van de methodes van de klasse Veelhoek().

```
class Driehoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,3)
class Vierkant(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,4)
class Vijfhoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,5)
class Zeshoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,6)
```



Met het statement isinstance() kan je checken of een object een instantie van een klasse en met issubclass() controleer je klasse-inheritance.

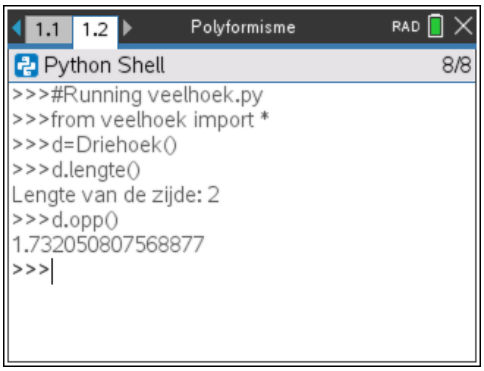


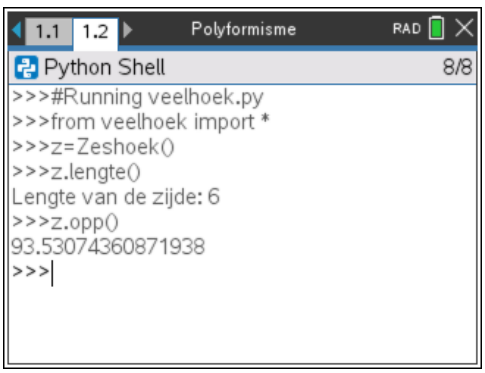


1.2. Polymorfisme

Voor de subklassen van Veelhoek() definiëren we de methode opp() voor de oppervlakte van de veelhoek.

```
from math import *
class Driehoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,3)
    def opp(self):
        return sqrt(3)/4*self.lengte**2
class Vierkant(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,4)
    def opp(self):
        return lengte**2
class Vijfhoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,5)
    def opp(self):
        return (5*self.lengte**2)/(4*sqrt(5-2*sqrt(5)))
class Zeshoek(Veelhoek):
    def __init__(self):
        Veelhoek.__init__(self,6)
    def opp(self):
        return 3*sqrt(3)*self.lengte**2/2
```





Voor iedere klasse definiëren we dezelfde methode met telkens een andere implementatie. Het uitvoeren van de methode runt telkens een andere code. Dit noemen we polymorfisme (veelvormigheid).

De algemene formule $self.opp = \frac{self.aantal\_zijden \cdot self.lengte}{4\tan\left(\frac{\pi}{self.aantal\_zijden}\right)}$ voor de klasse was ook een optie.

2. To Hub, or not to Hub0

Met de try: en except: statements kunnen we de code laten detecteren of de TI-Innovator™ Hub is aangesloten of niet.

Het try-blok zal een exception genereren indien de hub niet is aangesloten en dan wordt het except-blok uitgevoerd.

Indien de hub is aangesloten coderen we dat de module TI Hub wordt ingeladen en indien niet zelf gedefinieerde functionaliteit die een hub-experiment simuleert.

Als voorbeeld bekijken we een knipperend led.

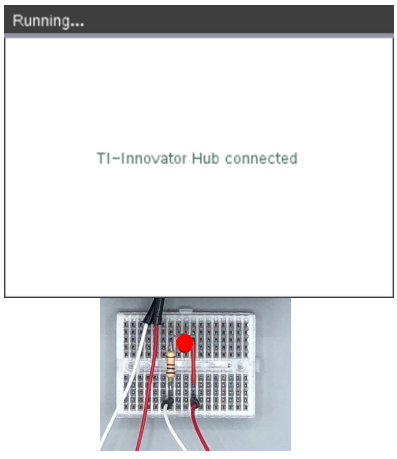
Stap 1	het try-blok controleert de connectie met een TI-Innovator Hub
Stap 2	if connected >> ti_hub import else >> vir_led import
Stap 3	uitvoeren experiment

Dit geeft het volgende voorbeeld.

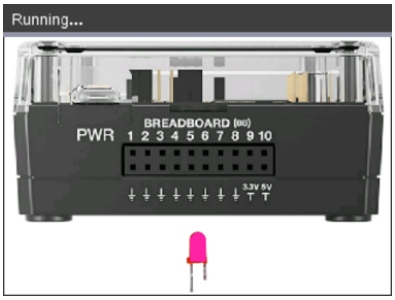
```
from ti_system import *
from ti_draw import *
from time import *

set_color(255,0,0)
draw_text(45,106,"Checking connection TI-Innovator Hub")

try:
    from ti_hub import *
```

```
# TI-Innovator Hub connected
    clear()
    set_color(52,91,77)
    draw_text(75,106,"TI-Innovator Hub connected")
    myled=led("BB 1")
```

```
# No TI-Innovator Hub connected
except:
    from vir_led import *
    background()
    myled=Led("BB 1")
```




```
while get_key() != "esc":
    myled.on()
    sleep(0.3)
    myled.off()
    sleep(0.3)
```



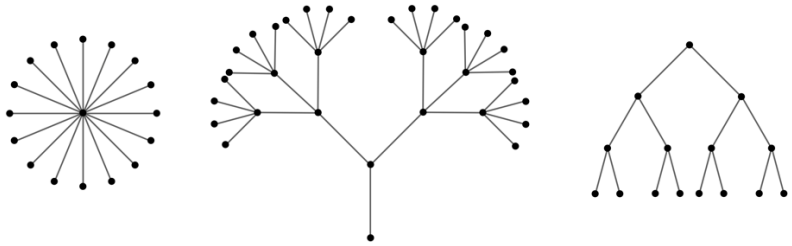
3. Prim-algoritme

Het Prim-algoritme is een algoritme om de minimale opspannende boom van een graaf te bepalen.

Even wat grafen-terminologie.

Een acyclische graaf een graaf is zonder cykel (= pad met lengte groter dan nul van een punt naar zichzelf); ook wel een bos genoemd. Een boom is samenhangende acyclische graaf.

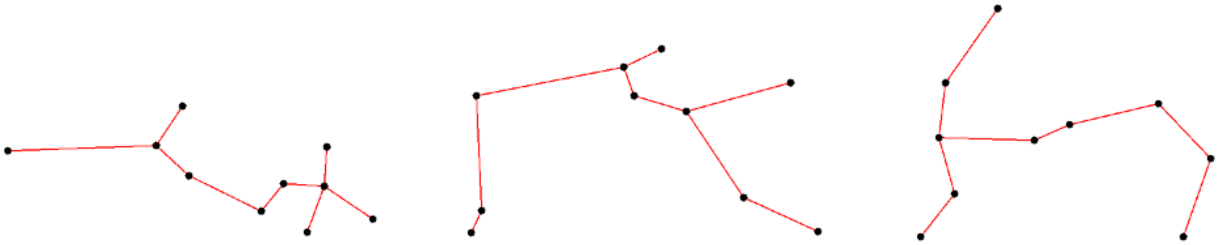
Hieronder wat bomen die samen een bos vormen:



Een opspannende boom is een deelgraaf die alle punten van de graaf bevat. Indien we aan de punten een gewicht toekennen, bv de kost of de afstand van een wandeling tussen twee punten, spreken we een gewogen boom.

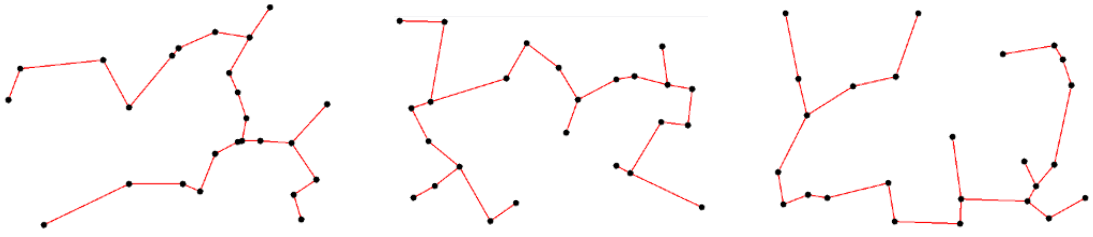
Het Prim-algoritme bepaalt de minimale (gewogen) opspannende boom, m.a.w. de boom met minimale gewicht, kost of afstand. We coderen een Prim-algoritme met als gewicht de Euclidische afstand tussen de punten.

Voor een graaf tekenen we de Euclidisch minimale opspannende boom (EMST – Euclidean minimum spanning tree). Een EMST verbindt een verzameling punten met lijnen zodat de totale lengte van de lijnen minimaal is en zodat ieder punt kan bereikt worden vanuit ieder ander punt door deze lijnen te volgen.



We bouwen het algoritme als volgt op voor een verzameling van punten:

1. Start een boom met een willekeurig punt p (verbonden) en een lijst met punten die nog niet toegevoegd zijn aan de boom (niet verbonden). Bij de start zijn dit alle punten uitgezonderd het willekeurig gekozen punt p.
2. Zoek voor dit punt p het punt q uit de lijst van niet verbonden punten waarvoor geldt dat de afstand minimaal is.
    a. Verbind de punten p en q,
    b. verwijder q van de lijst van niet verbonden punten en
    c. voeg q toe aan de verbonden punten van de boom.
3. Herhaal stap 2 voor alle verbonden punten tot de er geen niet verbonden punten meer zijn



We starten met het definiëren van objecten, methodes en functies:

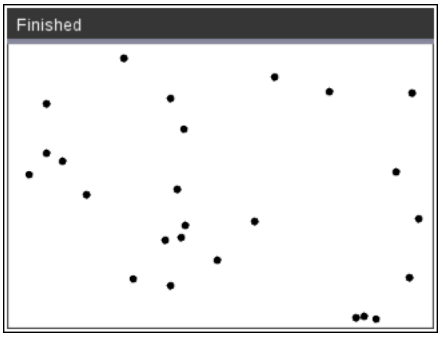
```
from ti_draw import *
from random import *
from math import *

class Punt():
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def teken(self):
        set_color(0,0,0)
        fill_circle(self.x,self.y,r)

def afstand(p,q):
    return sqrt((p.x-q.x)**2+(p.y-q.y)**2)

def lijn(p,q):
    set_color(255,0,0)
    draw_line(p.x,p.y,q.x,q.y)

def teken(punten):
    for p in punten:
        p.teken()

def emstBoom(punten):
    verbonden=[ ]
    niet_verbonden=[ ]
    for p in punten:
        niet_verbonden.append(p)
    rand = randint(0,len(punten)-1)
    p = punten[rand]
    verbonden.append(p)
    niet_verbonden.remove(p)
    while niet_verbonden !=[ ]:
        a=500
        for p in verbonden:
            for q in niet_verbonden:
                d=afstand(p,q)
                if d<a:
                    a=d
                    p1=p
                    q1=q
        lijn(p1,q1)
        verbonden.append(q1)
        niet_verbonden.remove(q1)
```

We bepalen het scherm en generen de punten,

```
w,h = get_screen_dim()
r,n = (3,25)

punten = [ ]
for i in range(n):
    x = randint(r,w-r)
    y = randint(r,h-r)
    punten.append(Punt(x,y))
```

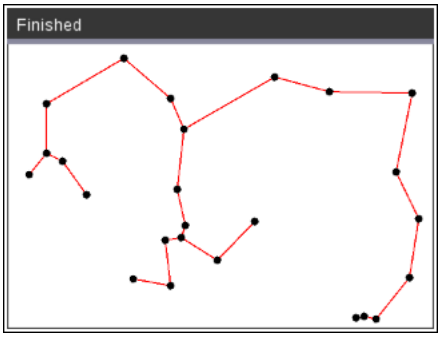
tekenen de punten en

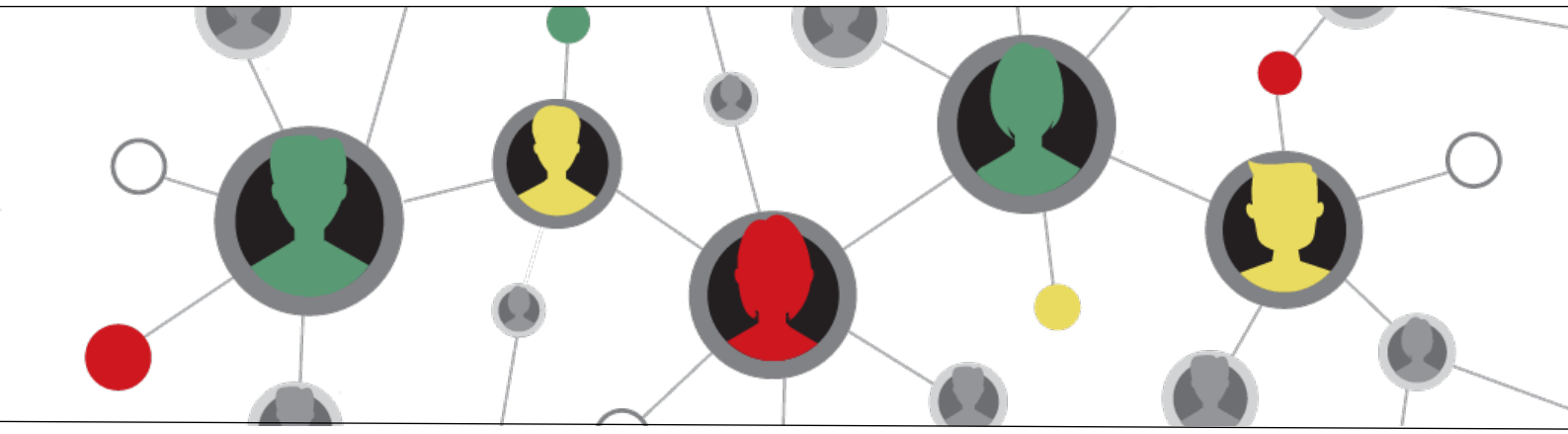
```
w,h = get_screen_dim()
r,n = (3,25)

punten = [ ]
for i in range(n):
    x = randint(r,w-r)
    y = randint(r,h-r)
    punten.append(Punt(x,y))

teken(punten)
```



bepalen de EMST-boom,

```
emstBoom(punten)
```





T³ NEDERLAND T³ VLAANDEREN

www.wil-depython.be
www.wil-depython.nl

Texas Instruments