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.

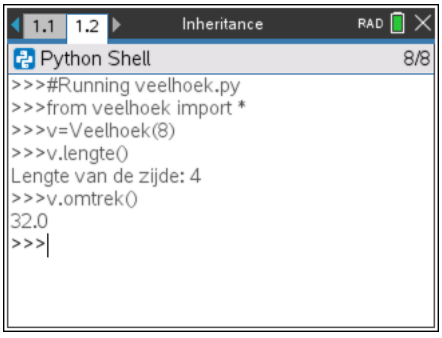
De python-syntax is de volgende:

```
class BasisKlasse():
    Blok

class AfgeleideKlasse(BasisKlasse):
    Blok
```

Voorbeeld
Omtrek van een regelmatige veelhoek

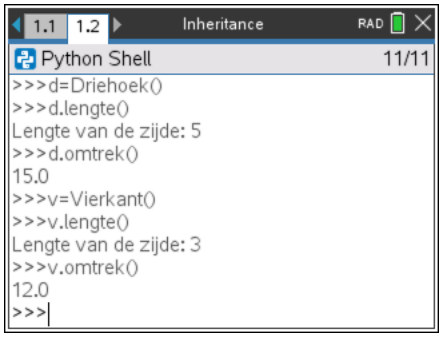
We definiëren als basis-klasse de klasse Veelhoek() met als attributen het aantal zijden en als methodes input van de lengte van de zijde en het berekenen van de omtrek.

```
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
```

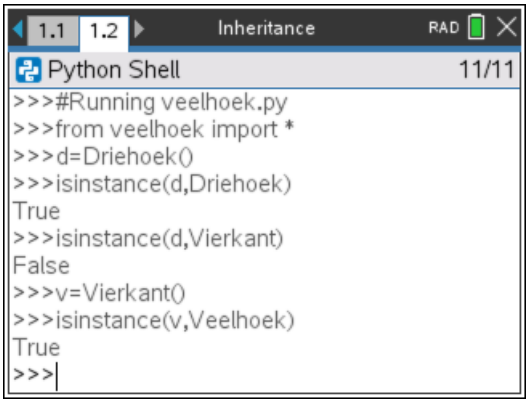


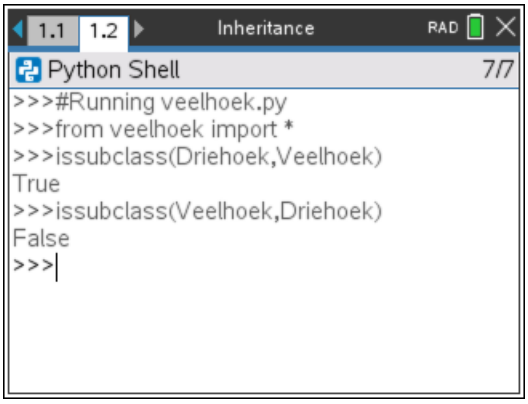
Als afgeleide klassen definiëren we de subklassen Driehoek(), Vierkant(), Vijfhoek() en Zeshoek(). Al deze klassen kunnen gebruik maken van de methodes van de klasse Veelhoek().

```
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.

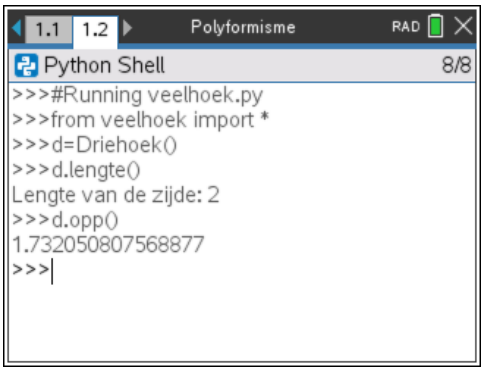


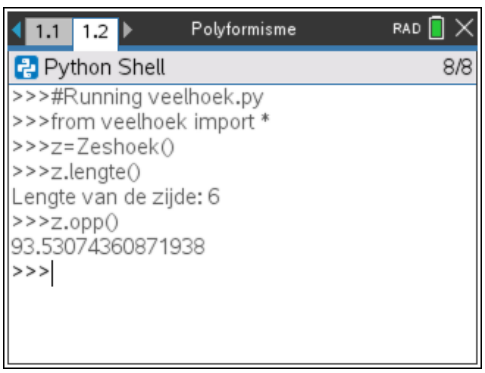


1.2. Polymorfisme

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.

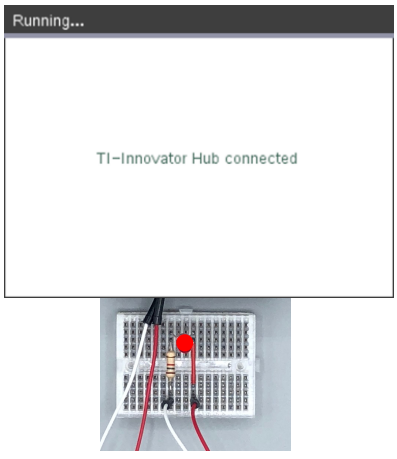
Stap 1	het try-blok controleert de connectie met een TI-Innovator Hub
Stap 2	if connected >> ti_hub import else >> vir_led import
Stap 3	uitvoeren experiment

Dit geeft het volgende voorbeeld.

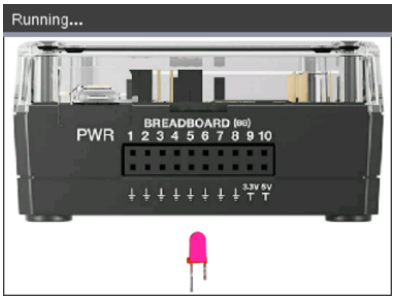
```
from ti_system import *
from ti_draw import *
from time import *

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)
```



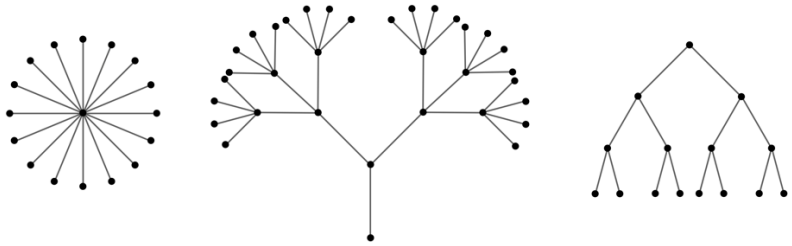
3. Prim-algoritme

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.

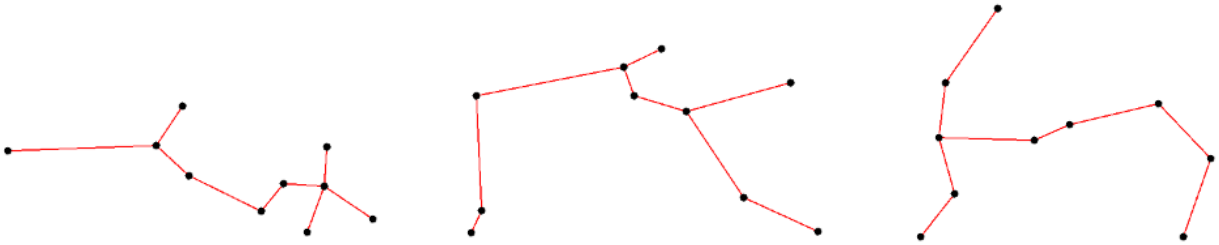
Hieronder wat bomen die samen een bos vormen:



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.

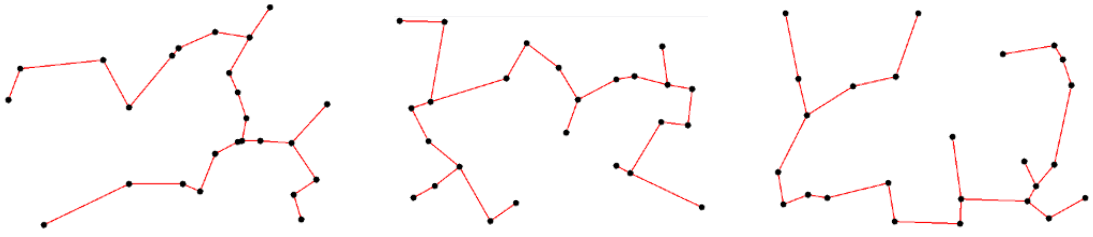
Het Prim-algoritme bepaalt de minimale (gewogen) opspannende boom, m.a.w. de boom met minimale gewicht, kost of afstand. We coderen een Prim-algoritme met als gewicht de Euclidische afstand tussen de punten.

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:

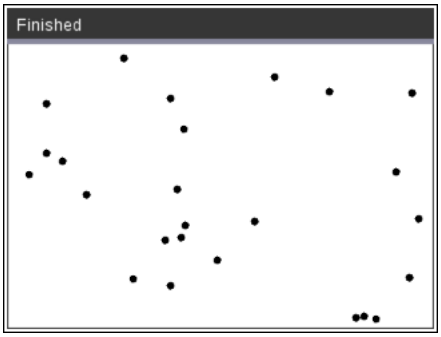
```
from ti_draw import *
from random import *
from math import *

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)
```