PO Informatica Python: **Zeeslag**

In deze opdracht gaan we het spel Zeeslag bouwen: een persoon speelt tegen een computer. Het is een uitdagende opdracht. Deel de opdracht op in stukjes. Probeer het spel steeds zo te programmeren dat je tussendoor zo veel mogelijk stukjes kunt testen zonder dat het spel helemaal af hoeft te zijn. Zo weet je zeker dat je altijd een werkend product hebt. Hoe verder je komt, hoe hoger je cijfer. Het is de bedoeling dat je zo veel mogelijk **zelfstandig** aan de slag gaat, zonder te veel hulp van de docent. Help elkaar als je vastloopt. ***Laat zien wat je geleerd hebt, dus ook hoe je fouten uit je eigen code kunt halen.***

# Toelichting opdracht

**Samenwerking**: het wordt **aanbevolen** om in tweetallen te werken (drietallen zijn niet toegestaan).

**Beoordeling**: zie volgende pagina

**Inleveren op**:

1. **Een verslag met daarin**:
   * wie eraan gewerkt heeft;
   * (een foto van) je stroomdiagram;
   * een evaluatie van je project met een korte uitleg van wat wel/niet werkt en wat wel/niet goed ging. Geef een voorbeeld van iets waar je tegenaan liep, hoe je dat probleem opgelost hebt en wat je ervan geleerd hebt;
   * welke uitbreidingen je hebt toegevoegd.
2. **Bestand** met je **Pythoncode** (zet boven in je code jouw/jullie naam in commentaar)

Je mag elkaar helpen, maar iedereen moet zijn eigen code schrijven en inleveren. Kopieer niet van iemand anders! Plagiaat wordt niet getolereerd.

Goed programmeren gaat dus **niet alleen** om een werkende **oplossing**, maar juist om een **degelijke aanpak en oplossing. Laat zien wat je geleerd hebt!** Ga gestructureerd te werk, bedenk uit welke losse onderdelen je programma bestaat en pak ze los van elkaar aan.

# Beoordeling

|  |  |
| --- | --- |
| Je **proces** wordt beoordeeld op: | |
| *Voortgang, Verslag & Ontwerp* | Je programma en verslag zijn compleet en op tijd en op de juiste wijze ingeleverd (zie toelichting opdracht). Je werkt zelfstandig en lost zelf (of met andere leerlingen) fouten in de code op. Je geeft een voorbeeld van iets waar je tegenaan liep en hoe je dat probleem opgelost hebt (en dus wat je ervan geleerd hebt).  Een plan/ontwerp (stroomdiagram) is vooraf gemaakt en staat in je verslag. Je ontwerp maakt duidelijk uit welke brokken je programma is opgebouwd (zie **Deel A**) en wat de belangrijkste variabelen en datastructuren zijn. Je ontwerp wordt gedurende het gehele programmeertraject gebruikt als leidraad. Er is een duidelijke overeenstemming tussen je ontwerp en je code. Je maakt gebruik van versiebeheer. |
| Je **product** wordt beoordeeld op: | |
| *Correctheid, Volledigheid & Originaliteit* | De basis van het programma is af volgens de omschrijving in **Deel B**. Hoe verder je komt qua functionaliteit (**Deel C**: uitbreidingen, originele invulling), hoe hoger je cijfer. Je uitbreidingen staan kort beschreven in je verslag.  Je code voldoet aan de verwachting volgens omschrijving **Deel B**. Je programma is robuust en geeft duidelijke foutmeldingen bij onvoorziene omstandigheden (bv. ongeldige invoer). In je verslag staat wat wel/(nog)niet goed werkt. |
| *Code documentatie & presentatie* | Jouw naam/jullie namen staan bovenaan de code.  **Code** is makkelijk te lezen en begrijpen.  **Commentaar** is aanwezig. Bij blokken code (bv. wat bij een loop of een functie hoort) wordt kort samengevat wat het doel van dat blok is. Zowel keuzes als problemen in je code worden met commentaar toegelicht. Nieuwe constructies die je jezelf hebt eigen gemaakt worden met commentaar toegelicht.  **Namen** beschrijven de bedoeling nauwkeurig en zijn compleet, onderscheidend, beknopt, correct gespeld en hebben consistent gebruik van conventies (camelUpperCase of met liggende streepjes) en constanten zijn met hoofdletters geschreven. Functies zijn met werkwoorden omschreven, variabelen met zelfstandige naamwoorden.  De **opmaak** is consistent en logisch gestructureerd. Vergelijkbare delen van code zijn duidelijk herkenbaar en staan bij elkaar: globale variabelen, definities en hoofdprogramma. |
| *Constructie* | De kwaliteit van de code: deze is duidelijk, efficiënt, elegant, logisch en goed gestructureerd.  De **flow** is eenvoudig. Het meest gebruikelijke pad door de code is duidelijk. Het hoofdprogramma leest als een inhoudsopgave en geeft op hoofdlijnen een duidelijk overzicht van wat het programma doet. Er wordt goed gebruik gemaakt van loops en condities. Coderegels en condities zijn kort, eenvoudig en makkelijk te begrijpen. Er komen geen lange of diep geneste brokken code voor. Er komt geen code dubbel voor. Er komt geen code voor die nooit uitgevoerd wordt. Gebruik van 'harde' waarden (echte getallen) wordt zo veel mogelijk vermeden.  **Functies** hebben één (duidelijk) doel en zijn kort en krachtig. Er wordt gebruik gemaakt van parameters en retourwaarden (om het gebruik van globale variabelen te beperken). Returns worden op logische, eenduidige en overzichtelijke wijze gebruikt. Variabelen worden niet hergebruikt voor verschillende doeleinden. |

# DEEL A: ONTWERP

Voordat we gaan beginnen met het spel ga je eerst een stroomdiagram tekenen waarin je het spel in logische delen opbreekt. Gebruik de onderstaande uitleg van deel B om je ontwerp te maken. Hieruit blijkt welke onderdelen achter elkaar uitgevoerd worden, waar herhaling zit en wat de voorwaarden voor herhaling zijn. Ook blijkt daaruit welke dingen je moet onthouden, oftewel welke variabelen of constanten je hebt. Maak een lijstje van de gegevens die je moet opslaan/onthouden en wat hun typen zijn (bv. String gebruikersgok). Mogelijk zul je gaandeweg aanpassingen maken en tot nieuwe inzichten komen. Pas je stroomdiagram daarop aan.

Voor het digitaal tekenen van een stroomdiagram kun je gebruik maken van de volgende tool: <https://www.informatica-actief.nl/streamtool2/Stream.html> (maak een screenshot maken met WINDOWS+SHIFT+S) of teken je ontwerp op papier en maak daar een foto van.

**Aanpak voor het ontwerp:**

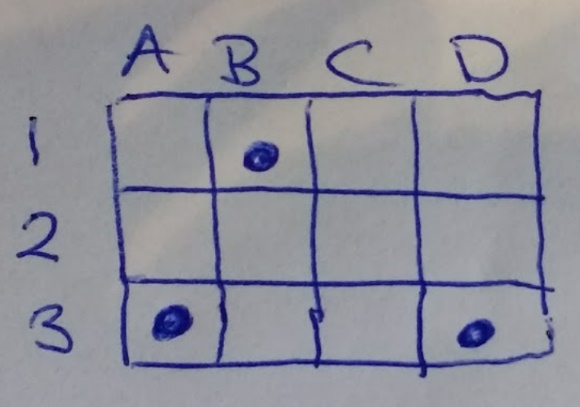
* speel het spel eerst op papier. Schrijf de stappen op die de computer doet;
* maak een schets van je bord met daarop de coördinaten. Bedenk hoe je dat in je programma gaat opslaan, dus wat voor datastructuur je gaat gebruiken (tip: je kunt een lijst van lijsten gebruiken met daarin een teken voor een leeg veld, een schip of een geraakt schip). Schrijf ook op welke gegevens je steeds moet bijhouden en controleren, dus welke variabelen je nodig hebt;
* teken een stroomschema met op hoog niveau (weinig details) een beschrijving van de verschillende onderdelen van je spel. Een aantal voorbeelden: toonBord, plaatsSchepen, vraagSpelerOmCoordinaten, verwerkSchot, isSpelAfgelopen … Deze onderdelen schrijf je straks als aparte functies. Deze kun je ook los van elkaar testen. Als je met zijn tweeën werkt, kun je hiermee ook goed je taken verdelen (nadat je samen een start hebt gemaakt);
* lever dit in als je ontwerp.

# DEEL B: BASISPROGRAMMA

**Programmeeromgeving**: Je werkt in Visual Studio Code (VSC). Er zijn handleidingen beschikbaar voor de installatie van VSC en ook tools voor samenwerken (Live Share, GitHub), vraag je docent daarnaar.

**Begin simpel! Zorg eerst dat de basis van je programma werkt en breid het daarna uit.**

*Weet je niet hoe je moet beginnen? Op de laatste pagina staan wat tips stap voor stap uitgelegd.*

Maak een vereenvoudigde versie van het spel “Zeeslag”. Een speler speelt tegen de computer. De computer creëert (in het geheugen) een matrix (lijst van lijsten) van drie rijen hoog en vier kolommen breed. De rijen zijn genummerd 1, 2 en 3, en de kolommen hebben de letters A, B, C en D. De computer verstopt in **drie** van de cellen een “oorlogsschip.” Ieder schip is precies één cel groot. De schepen mogen elkaar noch horizontaal, noch verticaal, noch diagonaal raken.

De computer laat het bord zonder schepen zien. De computer vraagt de speler te “schieten” op een cel. De speler doet dat door een kolomletter en rijcijfer in te geven (bijvoorbeeld: "D3"). Als de cel waarop de speler schiet niks bevat, zegt de computer *Mis!*. Als de cel een schip bevat, zegt de computer *Raak!* en markeert op het bord dat het schip is beschadigd. Als de speler erin geslaagd is alle schepen tot zinken te brengen, laat de computer zien hoeveel schoten er nodig waren en eindigt het programma. Anders wordt het bovenstaande herhaald.

Hieronder staat uitgelegd hoe je de programmeeromgeving in moet stellen. Daaronder staat waar het programma aan moet voldoen en staan er tips om het programma te maken.

**Waar het programma aan moet voldoen en extra tips**

* Om te helpen bij het debuggen van het spel, laat je de computer bij de start het bord tonen waarbij je kunt zien welke cellen een schip bevatten. Druk tussendoor ook variabelen af naar de console.
* Maak je programma gebruikersvriendelijk. Druk het bord netjes af, met letters boven de kolommen en getallen bij de rijen. Heet je gebruiker van harte welkom, leg de regels uit en vertel hoeveel van welke schepen er liggen. Accepteer zowel ‘A’ als ‘a’. Controleer op juiste invoer van de gebruiker (geldige coördinaten). Geef begrijpbare foutmeldingen en geef de gebruiker de kans om opnieuw coördinaten in te voeren, zonder dat het als een schotpoging meetelt. Als de gebruiker schiet op een cel waarop al eerder geschoten is, geef dan een waarschuwing en tel het niet mee als een schot.
* Houd de score van meerdere potjes bij. Houd statistieken bij, zoals: topscores, het minst aantal schoten nodig om te winnen en meer. Houd deze in een bestand bij zodat de speler zijn punten niet kwijtraakt elke keer als hij het programma afsluit.
* Geef een gebruiker de kans op revanche als hij verloren heeft.
* Gebruik versiebeheer en label tussendoor je code als onderdelen werken.

# DEEL C: UITBREIDINGEN

* Maak je spel uitdagender. Vraag de gebruiker hoe groot het bord moet zijn. Voeg ook grotere schepen toe. Zet de schepen op willekeurige plekken neer (doe dit alleen met schepen van grootte 1). Voeg een functie toe waarbij de schepen van grootte 1 per toeval geplaatst worden, zonder dat je controleert of ze elkaar raken. Als dat eenmaal werkt, voeg je code toe die ervoor zorgt dat de schepen elkaar niet raken (noch verticaal, horizontaal noch diagonaal).
* Laat de gebruiker tussendoor stoppen en sla de spelstand op in een bestand. Bij het hervatten van het spel wordt de stand uit het bestand ingelezen.
* Maak er een twee-persoonsspel van.
* Laat de gebruiker de moeilijkheid kiezen of bouw meerdere levels in.
* Geef de gebruiker steeds andere complimenten als hij het goed doet.
* Geef de gebruiker een tip als die het niet zo goed doet.
* Zorg dat de grootte van het bord past bij het aantal schepen. Het spel wordt er niet leuker op als je een bord van 30 bij 30 hebt en maar een schip.
* Combineer wat je geleerd hebt over Turtle in onderwerp 1.6 met het spel wat je gemaakt hebt om het bord te tekenen. Houd het venster gedurende de spel open door aan het einde van je code turtle.done() te gebruiken.
* Bedenk zelf nog creatieve toevoegingen.

# Tips (stap-voor-stap uitgelegd)

Stap 1. Breng structuur aan in je code.

|  |
| --- |
| #GLOBALE VARIABELEN  #FUNCTIEDEFINITIES  #HOOFDPROGRAMMA |

Stap 2. Zet je algoritme uit je stroomdiagram om in commentaar. Dit wordt je hoofdprogramma.

|  |
| --- |
| #HOOFDPROGRAMMA  #maak een leeg bord  #vul bord met willekeurige schepen  #toon bord met schepen op het scherm  #zolang spel niet is afgelopen, doe dan:  #vraag speler om invoer  #tel poging  #verwerk schot: controleer of raak/mis, vertel gebruiker, pas bord aan  #toon bord met schepen op het scherm  #spel afgelopen: geef gebruiker een compliment |

Stap 3. Schrijf een functie die een leeg bord maakt. Als parameters krijgt deze twee getallen mee die aangeven hoe groot het bord moet worden. De functie maakt dan een lijst in een lijst en levert deze op met een return.

|  |
| --- |
| #FUNCTIE DEFINITIES  def maakLeegBord(aantal\_rijen, aantal\_kolommen):  #Stap 1.  #maak eerst een leeg bord  #maak een lege rij  #vul die rij met het juist aantal kolommen  #plak rij aan bord (met .append)  #print("controle rij goed geprint: ", bord)  #Stap 2.  #herhaal bovenstaande voor het aantal rijen dat je nodig hebt  print("leeg bord gemaakt")  return bord |

Tip: Kijk eventueel naar onderwerp 12.6.

Stap 4. Schrijf een functie die een bord afdrukt. Als parameters geef je de lengte en breedte van de bord mee.

Stap 5. Schrijf een functie dat een bord toevoegt met een schip op een willekeurige plek. Deze functie krijgt een leeg bord als parameter, en levert met een return een gevuld bord op.

Tip: Gebruik print om de coördinaten ter controle op het scherm af te drukken.

Stap 6. Schrijf een functie die de gebruiker om invoer vraagt. Deze functie levert een lijst op met twee getallen daarin: de x-coördinaat en de y-coördinaat van de poging.

Tips:

* Stel de gebruiker twee vragen. Eerst de kolom, dan de rij. De kolom is een letter en die moet je omzetten in een getal. Maak een lijst waar je ze allebei in zet en lever deze met een return op.
* Controleer meteen of het een valide schot is (met andere woorden: of het schot wel op het bord is). Zo niet, vraag de gebruiker opnieuw om invoer.
* Voor als je programma helemaal werkt: bij onderdeel 13 leer je hoe je een letter kan omzetten in een cijfer. Daarmee kan je je code veel eleganter maken.

Stap 7. Schrijf een functie die het schot van de gebruiker verwerkt. Deze krijgt als parameters mee: het bord en de lijst met x- en y-coördinaat uit stap 6. De functie levert het bord op waarin het schot is verwerkt. Geef ook meteen feedback aan de gebruiker of het schot mis was of raak.

Stap 8:Het spel moet gaan herhalen. Bedenk wanneer het is afgelopen. Gebruik voor de herhaling een while-loop met logische vlag, zie onderwerp 7.

Stap 9. Bij stap 5 heb je een schip kunnen plaatsen. Deze willen we uitbreiden met meerdere schepen. Elke keer als een nieuwe willekeurige coördinaat berekend is, moet je eerst controleren of dat wel mag. Een coördinaat wordt afgekeurd als er al een schip staat of als er een schip schuin of naast staat. Een mogelijke aanpak is om, als je een boot ergens plaatst, ook meteen aan te geven welke plekken daarna leeg moeten blijven, bijvoorbeeld door op de gereserveerde plekken een ander teken neer te zetten. Hier zie je een voorbeeld waarbij ‘S’ een schip is en ‘L’ een gereserveerde lege plek is.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
| L | L |  |  |
| S | L |  |  |
| L | L |  |  |

Schrijf een functie die, gegeven een willekeurige coördinaat en een bord, de juiste gereserveerde plekken markeert. De functie levert het bord met ‘gereserveerde’ vakjes op als retourwaarde.

Tips:

* Maak op papier een schets van de coördinaten die niet goed zijn. Stel bijvoorbeeld dat de willekeurige coördinaat (1,3) is. Op welke coördinaten mag nu geen boot staan? Bijvoorbeeld, links ervan mag geen schip staan, dus niet op coördinaat (0,3). Stel dat (1,3) nou niet ‘echte’ getallen zijn, maar de willekeurige variabelen uit je programma (willekeurige\_x, willekeurige\_y), dan kun je berekenen dat coördinaat (willekeurige\_x – 1, willekeurige\_y) niet mag.
* Als de gereserveerde coördinaat op het bord voorkomt en dus bijvoorbeeld niet op coördinaat (-1,2) want die ligt links van het bord, dan zet je daar een speciaal teken.

Bijvoorbeeld, bij een gereserveerde coördinaat (willekeurige\_x – 1, willekeurige\_y) moet je controleren of willekeurige\_x – 1 wel groter dan of gelijk is aan 0. In pseudocode ziet dit er zo uit:

|  |
| --- |
| **Als** (willekeurige\_x – 1) groter dan of gelijk is aan 0 **en** er staat niet al iets op die plek:  Plaats op coördinaat (willekeurige\_x – 1, willekeurige\_y)een ‘L’ |

Stap 10. Breid je code uit stap 5 uit om meerdere schepen op een bord te kunnen plaatsen. Elke keer als een nieuwe willekeurige coördinaat berekend is, moet je eerst controleren of die coördinaat wel mag. Hiervoor gebruik je je functie uit stap 8. Als de coördinaat goed is (de functie levert True op), dan plaats je het schip. Zo niet, dan bereken je opnieuw een willekeurige coördinaat. Ga zo door tot het juiste aantal schepen geplaatst is.

Stap 11. Je bent super goed bezig… kijk nu of je het verder zelf kan afmaken!