Universiteit van Amsterdam

Programmeertalen

Python Puzzles

Bas Terwijn

Python Puzzles

1 Python Puzzels

Onderstaande 14 Python puzzelopgaven helpen om op een object-georiënteerde (OO) manier te leren programmeren met Python. Het is de bedoeling dat je hierbij zelf zoekt naar documentatie en code-voorbeelden om de puzzels op te lossen. Er zijn veel verschillende bronnen hiervoor beschikbaar, https://docs.python.org geeft de officiële documentatie. Gebruik **geen** ChatGPT of andere AI code-generator tools omdat je dan niet zelf met Python leert programmeren wat problemen zal geven op het tentamen.

1.1 Setup

Installeer een editor of Integrated Development Environment (IDE) die ondersteuning heeft voor Python. Je mag zelf een editor kiezen, maar een populaire editor is Visual Studio Code. Via deze link vind je ook instructie voor het installeren van "python" (gebruik versie **3.10** of hoger) en "pip" (gebruik versie 23 of hoger):

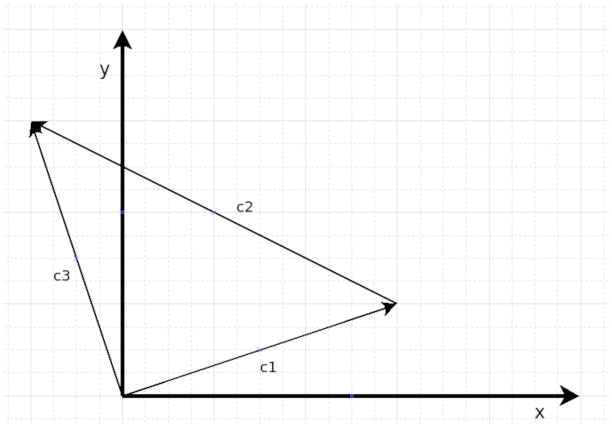
```
$ python --version
Python 3.10
$ pip --version
pip 23.0
```

Op sommige besturingssystemen moet je in plaats daarvan commando "python3" en "pip3" gebruiken voor de juiste versie:

```
$ python3 --version
Python 3.10
$ pip3 --version
pip 23.0
```

2 Object-Oriented Programming

Bij object-oriented programming maken we gebruik van classes om onze eigen types te maken met al bestaande types als bouwstenen. Een class bestaat uit waarden (*instance variables*) en functies (*methodes*). In de programmeertaal C is een *struct* het concept wat het meest lijkt op een *class* in Python. In onderstaande code-voorbeeld zien we een *struct*/*class* genaamd "Coordinate" in C en in Python die de waarden x en y heeft, en functies/methoden heeft die iets met deze waarden doen. In figuur 1 zijn de coordinaten van dit voorbeeld weergegeven in een assenstelsel.



Figuur 1: Coordinaten c1, c2 en c3 als vectoren in assenstelsel.

coordinate.c

```c
#include <stdio.h>
#include <math.h>

typedef struct {
    int x;
    int y;
} Coordinate;

Coordinate init_coordinate(int x, int y) {
    Coordinate self;
    self.x = x;
    self.y = y;
    return self;
}

char* repr(Coordinate self) {
    static char buffer[30]; // reusing static buffer
    sprintf(buffer, "(%d, %d)", self.x, self.y);
    return buffer;
}

Coordinate add(Coordinate self, Coordinate other) {
    return init_coordinate(self.x + other.x, self.y + other.y);
}

double get_distance(Coordinate self) {
    return sqrt(self.x*self.x + self.y*self.y);
}

int main() {
    Coordinate c1 = init_coordinate( 3, 1);
    Coordinate c2 = init_coordinate(-4, 2);
    Coordinate c3 = add(c1, c2);
    printf("c1: %s\n", repr(c1));
    printf("c2: %s\n", repr(c2));
    printf("c3: %s\n", repr(c3));
    printf("c3 distance: %f\n", get_distance(c3));
}
```

coordinate.py

```python
import math


class Coordinate:

    def __init__(self, x, y):
        """ Initalises a Coordinate object. For example:
        >>> c1 = Coordinate(1, 2)
        """
        self.x = x
        self.y = y

    def __repr__(self):
        """ Returns the string representation of a Coordinate.
        For example:
        >>> print(Coordinate(1, 2))
        (1, 2)
        """
        return f"({self.x}, {self.y})"

    def __add__(self, other):
        """ Returns a new Coordinate by adding 'self' and
        'other'. This method enables the '+' operator:
        >>> print(Coordinate(1, 2) + Coordinate(3, 3))
        (4, 5)
        """
        return Coordinate(self.x + other.x, self.y + other.y)

    def get_distance(self):
        """ Returns the distance to (0, 0). For example:
        >>> Coordinate(3, 4).get_distance()
        5.0
        """
        return math.sqrt(self.x**2 + self.y**2)


def main():
    c1 = Coordinate(3, 1)
    c2 = Coordinate(-4, 2)
    c3 = c1 + c2
    print("c1:", c1)
    print("c2:", c2)
    print("c3:", c3)
    print("c3 distance:", c3.get_distance())


if __name__ == "__main__":
    main()
```

```
$ gcc coordinate.c -o coordinate -lm
$ ./coordinate
c1: (3, 1)
c2: (-4, 2)
c3: (-1, 3)
c3 distance: 3.162278
```

```
$ python coordinate.py
c1: (3, 1)
c2: (-4, 2)
c3: (-1, 3)
c3 distance: 3.1622776601683795
```

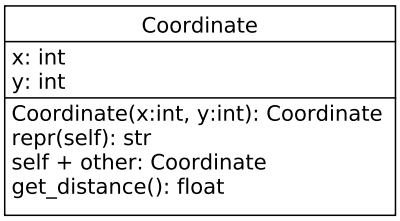
Merk op dat Python niet gebruik maakt van '{' en '}' om het begin en einde van een code block aan te geven. In plaats daarvan gebruikt Python indentatie (tabs). Let dus goed op de indentatie om foutmeldingen en bugs te voorkomen.

2.1 Abstractie

In een class diagram van de Unified Modelling Language (UML) wordt een class weergegeven als een rechthoek met drie delen zoals weergegeven in figuur 2. Deel1 heeft de naam van de class, deel2 heeft de *instance variables* van de class, en deel3 heeft alle *methods* van de class wat ook wel het *interface* van de class wordt genoemd. Dit interface beschrijft hoe de class gebruikt kan worden. Een class kunnen we als bouwsteen gebruiken bij het schrijven van weer andere software. In deze andere sofware kunnen we dan dit interface gebruiken zonder meer te hoeven weten hoe dit interface precies is geïmplementeerd en hoe de informatie binnen de class wordt opgeslagen. Dit noemen we ook wel *abstractie*:

- **Procedurele abstractie:** Het verbergen van de implementatie-details van een bepaalde functie/methode/procedure, zodat de gebruiker alleen bezig hoeft te zijn met het aanroepen van de functie zonder zich druk te hoeven maken over de logica van de functie.
- **Gegevensabstractie**: Het verbergen van de interne implementatie-details van de gegevensstructuur, zodat de gebruiker alleen gegevens hoeft aan te dragen zonder zich druk te hoeven maken over hoe deze worden opgeslagen.

Abstractie is heel belangrijk want het stelt ons in staat om grote en complexe software te schrijven wat niet mogelijk zou zijn als we daarvoor alle implementatie-details van alle bouwstenen zouden moeten blijven onthouden. Een class biedt ons zowel procedurele als gegevensabstractie terwijl een functie ons meestal alleen procedurele abstractie biedt.



Figuur 2: UML class diagram van de Coordinate class.

2.2 Doctest

In de docstring documentatie staan voorbeelden van wat het resultaat is van het aanroepen van een methode. In de docstring van de get_distance() methode staat bijvoorbeeld het voorbeeld:

```
>>> Coordinate(3,4).get_distance()
5.0
```

Dit voorbeeld geeft het resultaat van het aanroepen van deze methode, de afstand van coordinaat (3, 4) tot (0, 0). Deze voorbeelden kunnen we ook als tests gebruiken. Om deze tests uit te voeren gebruiken we het commando:

```
$ python -m doctest coordinate.py
```

Als hierbij niks wordt weergegeven dan zijn alle test geslaagd, elke test die niet slaagt wordt weergegeven. Soms willen we alle tests zien die worden uitgevoerd waarvoor we "-v" (verbose) aan het commando toevoegen:

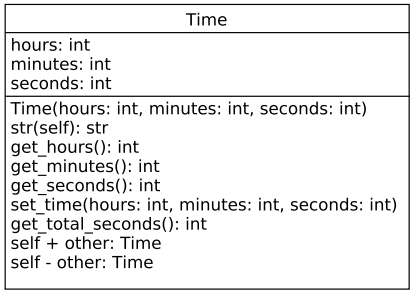
```
$ python -m doctest coordinate.py -v
Trying:
    print(Coordinate(1, 2) + Coordinate(3, 3))
Expecting:
    (4, 5)
ok
```

```
Trying:
    c1 = Coordinate(1, 2)
Expecting nothing
ok
Trying:
    print(Coordinate(1, 2))
Expecting:
    (1, 2)
ok
Trying:
    Coordinate(3,4).get_distance()
Expecting:
    5.0
ok
3 items had no tests:
    coordinate
    coordinate.Coordinate
    coordinate.main
4 items passed all tests:
   1 tests in coordinate.Coordinate.__add__
   1 tests in coordinate.Coordinate.__init__
   1 tests in coordinate.Coordinate.__repr__
   1 tests in coordinate.Coordinate.get_distance
4 tests in 7 items.
4 passed and 0 failed.
Test passed.
```

Dit laat dan voor elke test het verwachte resultaat zien met op het einde een overzicht van het aantal geslaagde en ongeslaagde tests. Gebruik dit commando (meestal zonder "-v") steeds bij de onderstaande puzzels om te testen of je de methoden goed hebt geïmplementeerd.

3 Time

Als puzzel gaan we nu onze eigen alarmklok programmeren. Hiervoor schrijven we eerst een Time class om daarmee op een eenvoudige manier met tijd-waarden om te kunnen gaan. Schrijf voor onderstaande opgaven je eigen implementatie, maak **geen** gebruik van al bestaande code voor tijd-waarden. Het maken van onderstaande opgave 1 t/m 5 resulteert in de Time class zoals weergegeven in figuur 3.



Figuur 3: UML class diagram van de Time class.

Opgave 1: Implementeer de methoden van onderstaande class Time zodat we objecten van type Time kunnen aanmaken. Een Time object heeft 'hours', 'minutes' en 'seconds' en heeft de string representatie van een digitale klok, bijvoorbeeld 21:05:02 voor twee seconden na vijf minuten over negen 's avonds. Voor deze string representatie kun je gebruik maken van f-strings formatting voor "leading zeros" met bijvoorbeeld `f"{123:07}"`.

De methoden met een dubbele underscores "___" zijn speciale methoden en worden dunder methods (of magic methods) genoemd. Dunder methoden maken een bepaalde interactie met de class mogelijk:

- ___init___() zorgt dat we een object kunnen aanmaken, bijvoorbeeld met: `t = Time(8,5,30)`
- ___repr___() zorgt dat een Time object naar een string kan worden geconverteerd, bv: `str(Time(8,5,30))`

alarm_clock.py

```
HOURS_IN_DAY = 24
MINUTES_IN_HOUR = 60
SECONDS_IN_MINUTE = 60


class Time:
    """ Represents a time of day."""

    def __init__(self, hours, minutes, seconds):
        """ Initialises a Time object with integers 'hours', 'minutes' and
        'seconds.
        >>> t = Time(18, 30, 0)
        """

    def __repr__(self):
        """ Returns the string representation of a Time object.
        >>> print( Time(8,5,30) )
        08:05:30
        """

    def get_hours(self):
        """ Returns the hours of the Time object.
        >>> Time(23,0,0).get_hours()
        23
        """

    def get_minutes(self):
        """ Returns the minutes of the Time object.
        >>> Time(0,59,0).get_minutes()
        59
        """

    def get_seconds(self):
        """ Returns the seconds of the Time object.
        >>> Time(0,0,59).get_seconds()
        59
        """
```

Test steeds je implementatie met gebruik van de doctests en door een eigen main functie toe te voegen. Een main functie is geen onderdeel van de Time class en heeft dus geen indentatie (staat dus helemaal links tegen de kantlijn) zoals bijvoorbeeld:

```
def main():
    t1 = Time(9, 30, 5)
    print("t1:", t1)
    print("seconds:", t1.get_seconds())

if __name__ == "__main__": # keep this at the bottom of the file
    main()
```

Opgave 2: Voeg onderstaande methode set_time() toe aan de Time class om de tijd van een al bestaand Time object aan te passen. Zorg daarbij dat de instance variabelen van een time object altijd geldige waarden krijgen, ook als er bijvoorbeeld een negatief aantal seconden wordt geven. Zo zou bijvoorbeeld Time(0,0,90) moeten leiden tot een 00:01:30 tijd, en Time(0,1,-30) in een 00:00:30 tijd. Gebruik hiervoor bijvoorbeeld de divmod() functie. Gebruik deze set_time() methode vervolgens ook in de ___init___() methode zodat ook bij het aanmaken van een nieuw Time object ongeldige waarden automatisch worden gecorrigeerd.

alarm_clock.py

```
    def set_time(self, hours, minutes, seconds):
        """ Sets the time of the Time object to 'hours', 'minutes',
        and 'seconds' making sure the values are in valid range:
          hours:   [0, HOURS_IN_DAY)
          minutes: [0, MINUTES_IN_HOUR)
          seconds: [0, SECONDS_IN_MINUTE)
        >>> time = Time(0, 0, 0)
        >>> time.set_time(0, 0, 90)
        >>> print(time)
        00:01:30
        >>> time.set_time(0, 0, 3600)
        >>> print(time)
        01:00:00
        >>> time.set_time(0, 0, -1)
        >>> print(time)
        23:59:59
        >>> time.set_time(10, -121, 0)
        >>> print(time)
        07:59:00
        >>> time.set_time(-50, 0, 0)
        >>> print(time)
        22:00:00
        >>> print(Time(10, -120, -150)) # __init__() test
        07:57:30
        """
```

Opgave 3: Voeg ook methode get_total_seconds() toe welke het aantal seconden sinds tijdstip 00:00:00 berekent en returned. Schrijf de implementatie.

alarm_clock.py

```
    def get_total_seconds(self):
        """ Returns the number of seconds since time 00:00:00.
        >>> Time(0,0,1).get_total_seconds()
        1
        >>> Time(0,1,0).get_total_seconds()
        60
        >>> Time(1,0,0).get_total_seconds()
        3600
        >>> Time(13,30,5).get_total_seconds()
        48605
        """
```

Opgave 4: Zoals we al zagen bij de Coordinate class kunnen we het gebruik van de "+" operator op objecten van een class mogelijk maken door de dunder method ___add___() toe te voegen. Voeg deze methode toe en ook de dunder method ___sub___() die het gebruik van de "-" operator mogelijk maakt. Schrijf de implementaties.

alarm_clock.py

```
    def __add__(self, other):
        """ Returns a valid Time objects which is Time objects
        'other' added to 'self'.
        >>> print(Time(0,0,0) + Time(1,2,3))
        01:02:03
        >>> print(Time(13,30,0) + Time(1,46,-45))
        15:15:15
        """

    def __sub__(self, other):
        """ Returns a valid Time objects which is Time objects
        'other' substracted from 'self'.
        >>> print(Time(10,10,10) - Time(1,2,3))
        09:08:07
        >>> print(Time(10,0,0) - Time(1,50,600))
        08:00:00
        """
```

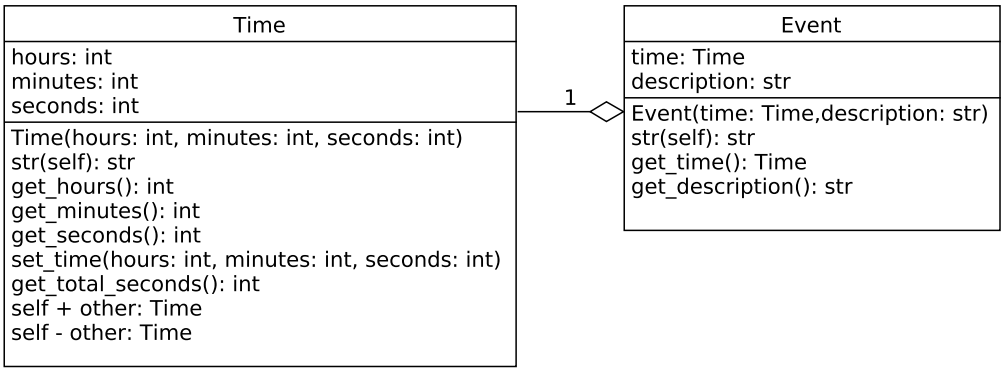
Opgave 5: Voeg ook de functies get_current_hours_minutes_seconds() en now() toe. Dit zijn net als de main() geen methodes maar losse functies en krijgen dus ook geen indentatie. Functie get_current_hours_minutes_seconds() gebruikt zelf weer een functie uit de Python "time" module waardoor we een "import time" regel boven aan de file moeten toevoegen. De "*" operator in de now() functie is geen pointer operator zoals in C, maar een unpack operator die de 3 waarden in de tuple uitpakt zodat er 3 losse argumenten aan de Time() methode (de ___init___() dunder method van class Time) worden meegegeven. Test de now() functie in je main() functie door bijvoorbeeld uit te rekenen hoeveel seconden de huidige dag nog duurt.

alarm_clock.py

```
import time

# <here goes the Time class with its methods>


def get_current_hours_minutes_seconds():
    """ Returns the current (hours, minutes, seconds) as a tuple. """
    t = time.localtime()
    return (t.tm_hour, t.tm_min, t.tm_sec)


def now():
    """ Returns the current time as Time object. """
    return Time(*get_current_hours_minutes_seconds())
```

4 Event

Onze alarmklok heeft verschillende gebeurtenissen die een tijd hebben waarop een alarm moet afgaan, bijvoorbeeld om 07:59:30 moet een alarm afgaan voor de “opstaan” gebeurtenis. Om een gebeurtenis te representeren schrijven we de Event class. De Event class is slechts een container class om de tijd en de beschrijving van een gebeurtenis in op te slaan. Ook al heeft deze class weinig logica, het is toch een nuttige class om te maken omdat het een abstractie toevoegt en het makkelijker maakt om over de code na te denken. De Event class, die we gaan schrijven in onderstaande opgaven, is weergegeven in figuur 4. Hierin is ook schematish de relatie met de Time class weergegeven, een object van de Event class heeft altijd 1 object van de Time class.



Figuur 4: UML class diagram van de Time en Event class.

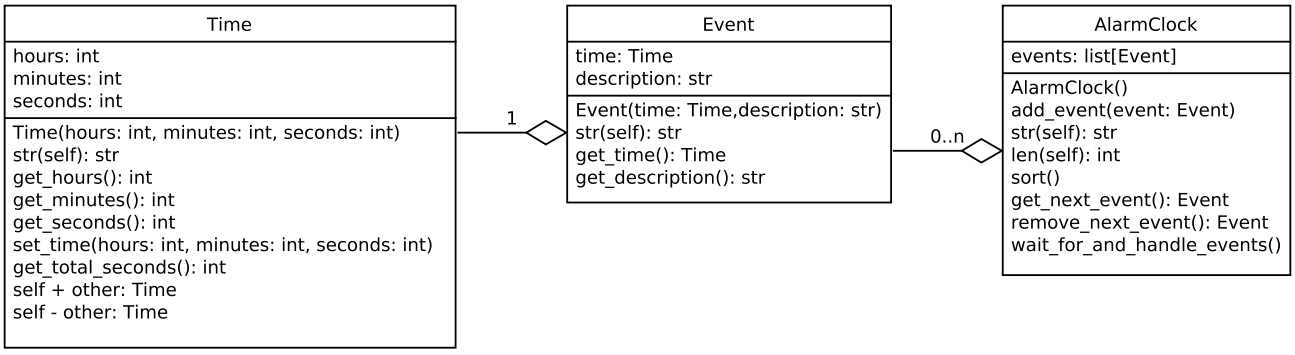
Opgave 6: Voeg onderstaande Event class toe en implementeer de methoden van deze class.

alarm_clock.py

```
class Event:
    """ Represents an event that happens at a certain time."""

    def __init__(self, time, description):
        """ Initialises an Event object with a 'time' object of type Time and a
        'description' of type str.
        >>> event = Event(Time(18, 30, 0), "dinner")
        """

    def __repr__(self):
        """ Returns the string representation of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner") )
        18:30:00 dinner
        """

    def get_time(self):
        """ Returns the time of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner").get_time() )
        18:30:00
        """

    def get_description(self):
        """ Returns the description of an Event object.
        >>> print( Event(Time(18, 30, 0), "dinner").get_description() )
        dinner
        """
```

5 AlarmClock

Nu zijn we klaar om de AlarmClock class te schrijven. Aan een object van class AlarmClock class kunnen verschillende events worden toegevoegd. Bij het bereiken van het tijdstip van een event zal de AlarmClock het event printen en verwijderen. De AlarmClock class is weergegeven in figuur 5 waarbij is weergeven dat een AlarmClock object 0 t/m n Event objecten kan hebben.



Figuur 5: UML class diagram van de Time, Event en AlarmClock class.

Opgave 7: Voeg de AlarmClock class toe en implementeer onderstaande methoden. Gebruik een list om de verschillende events in op te slaan.

alarm_clock.py

```
class AlarmClock:
    """ Represents an alarm clock that can handle events. """

    def __init__(self):
        """ Initialises an AlarmClock object with an empty list of events.
        >>> alarm_clock = AlarmClock()
        """

    def add_event(self, event):
        """ Adds an 'event' to this AlarmClock object, it doesn't return anything.
        >>> alarm_clock = AlarmClock()
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        """

    def __repr__(self):
        """ Returns a string representation of the AlarmClock object.
        >>> alarm_clock = AlarmClock()
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        >>> s = str(alarm_clock)
        >>> "18:30:00" in s
        True
        >>> "dinner" in s
        True
        >>> "breakfast" in s
        False
        """
```

Opgave 8: Voeg de ___len___() dunder method toe aan de AlarmClock class en implementeer deze zodat het het aantal events in een AlarmClock object teruggeeft.

alarm_clock.py

```
    def __len__(self):
        """ Returns the number of events in this AlarmClock object.
        >>> alarm_clock = AlarmClock()
        >>> len(alarm_clock)
        0
        >>> event = Event(Time(18, 30, 0), "dinner")
        >>> alarm_clock.add_event(event)
        >>> len(alarm_clock)
        1
        """
```

Opgave 9: Voeg de sort() methode toe aan de AlarmClock class die alle events in een AlarmClock object van laag naar hoog sorteert op tijd. Lees voor de implementatie over het sorteren van een lijst. Omdat er geen volgorde voor Event objecten gedefinieerd is, is het handig om de key function te gebruiken om aan te geven welke waarde dan wel gebruikt moet worden voor het bepalen van de volgorde van events. Gebruik deze sort() methode in de add_event() methode zodat alle events automatisch op volgorde worden gezet nadat een nieuw event wordt toegevoegd.

alarm_clock.py

```
    def sort(self):
        """ Sorts the events by time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> s = str(alarm_clock)
        >>> s.find("event1") < s.find("event2")
        True
        """
```

Opgave 10: Voeg de get_next_event() en remove_next_event() methoden toe aan de AlarmClock class om het eerst volgende event op te vragen dan wel te verwijderen. Implementeer deze methoden.

alarm_clock.py

```
    def get_next_event(self):
        """ Returns the next event with the smallest time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.get_next_event().get_description()
        'event2'
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> alarm_clock.get_next_event().get_description()
        'event1'
        """

    def remove_next_event(self):
        """ Removes and returns the next event with the smallest time.
        >>> alarm_clock = AlarmClock()
        >>> alarm_clock.add_event( Event(Time(0, 0, 2), "event2") )
        >>> alarm_clock.add_event( Event(Time(0, 0, 1), "event1") )
        >>> alarm_clock.remove_next_event().get_description()
        'event1'
        >>> alarm_clock.remove_next_event().get_description()
        'event2'
        """
```

Opgave 11a: Implementeer nu de AlarmClock class door de laatste wait_for_and_handle_events() methode aan de class toe te voegen. Deze methode wacht tot de tijd van elke event aanbreekt en print en verdwijdert deze event dan. Gebruik voor de implementatie de sleep(n) functie om n seconden te slapen. De wait_for_and_handle_events() methode moet pas returnen wanneer alle events geprint en verwijderd zijn.

alarm_clock.py

```
    def wait_for_and_handle_events(self):
        """ Wait for each event to pass and then print the event. """
```

Test deze methode vervolgens in de main() functie, bijvoorbeeld met:

```
def main():
    alarm_clock = AlarmClock()
    alarm_clock.add_event(Event(now() + Time(0, 0,  7), "eat some breakfast"))
    alarm_clock.add_event(Event(now() + Time(0, 0, 12), "off to work"))
    alarm_clock.add_event(Event(now() + Time(0, 0,  2), "good morning, wake up"))
    # debug here later with print('alarm_clock:', alarm_clock) and mg.s()
    alarm_clock.wait_for_and_handle_events()
```

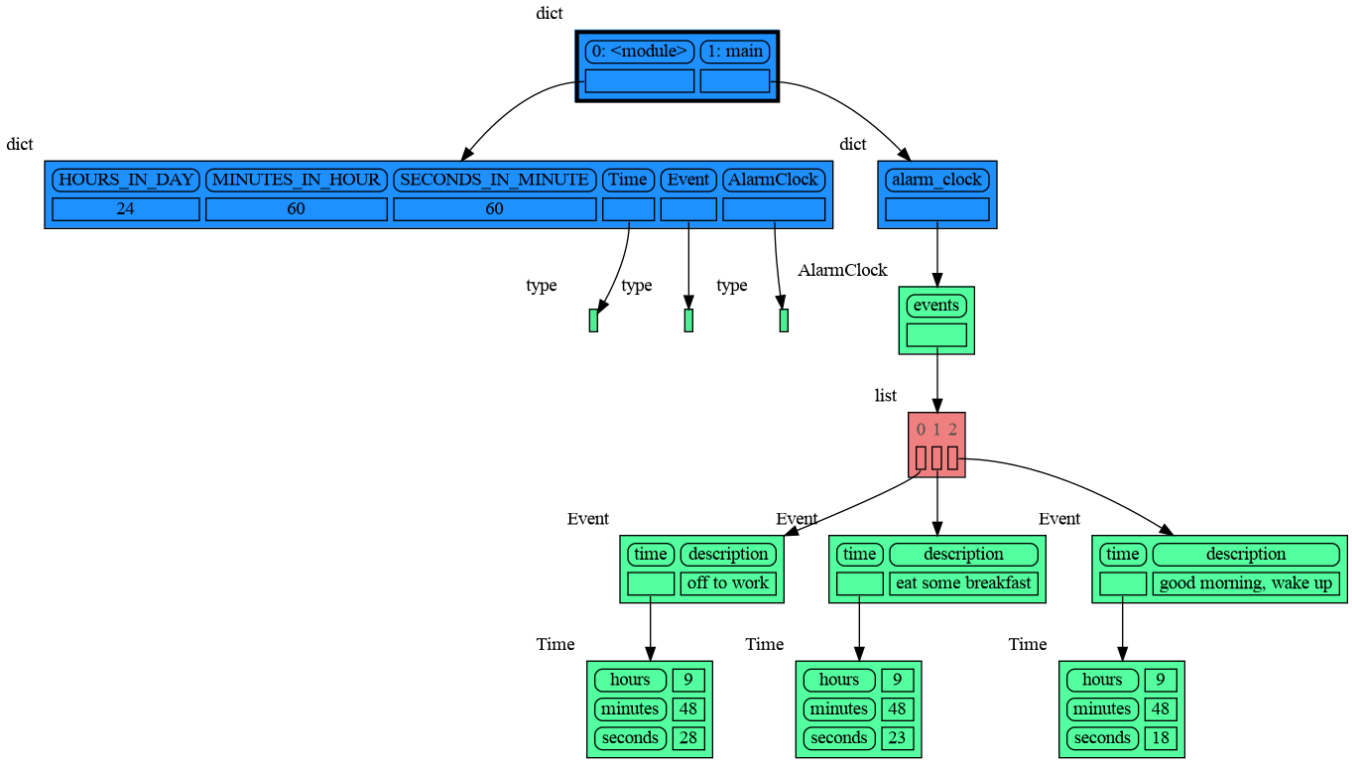
wat dan elke event op de aangegeven tijd print en dus na 12 seconden ongeveer deze output geeft:

```
$ python alarm_clock.py
ALARM: 14:38:11 good morning, wake up
ALARM: 14:38:16 eat some breakfast
ALARM: 14:38:21 off to work
```

Opgave 11b: Om Python code te debuggen kunnen we tijdelijke print statements toevoegen zoals bijvoorbeeld `print('alarm_clock:', alarm_clock)` in de main() functie na het toevoegen van de events met `add_event()` calls. Hiermee kunnen we een idee krijgen van de staat van het programma op dat punt. Maar een betere manier hiervoor is om gebruik te maken van Python package memory_graph:

1. volg de installatie-instructie van memory_graph
2. voeg "`import memory_graph as mg`" boven in de code toe
3. voeg "`mg.s()`" toe in de code op de plekken waar je de staat van het programma wilt zien

Als we bijvoorbeeld `mg.s()` toevoegen na de `add_event()` calls en het programma uitvoeren dan stopt het programma daar met een "Press <Enter> to continue..." boodschap en wordt de hele call stack op dat punt weergegeven in een graaf zoals in figuur 6. Als de graaf niet verschijnt of als er andere problemen zijn met PDF uitvoer (bv bij gebruik van Adobe Acrobat Reader), voeg dan bovenaan nog `mg.render_filename = 'memory_graph.png'` toe, gebruik `mg.brs()` in plaast van `mg.s()`, en open zelf de "memory_graph.png" file die na uitvoeren ontstaat in de lokale directory.



Figuur 6: De call stack wat de staat van het programma weergeeft, nuttig bij debuggen.

6 Speech Synthesis

We hebben nu onze eigen Time class geschreven als nuttige oefening van object-georiënteerd programmeren, maar in het algemeen is het beter om eerst te kijken of Python zelf niet al een class heeft die doet wat je wilt zoals bijvoorbeeld de “struct_time” class in de time module. Python biedt heel veel verschillende modules en classes die je gemakkelijk kunt installeren en gebruiken. Stel dat we bijvoorbeeld ook “speech synthesis” aan onze alarmklok willen toevoegen om de event-beschrijving uit te spreken. Na een korte websearch vinden we dat we na installeren van Python module “gtts” en “pygame” met commando:

```
$ pip install gtts pygame
```

we met deze code gemakkelijk speech synthesis kunnen uitvoeren en afspelen:

speech_synthesis.py

```
from gtts import gTTS
import pygame


def text_to_speech(text):
    tts = gTTS(text=text, lang='en')
    filename = "speech.mp3"
    tts.save(filename)
    pygame.mixer.init()
    pygame.mixer.music.load(filename)
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        pass

def main():
    text_to_speech("Hello, this is a speech synthesis test.")


if __name__ == "__main__":
    main()
```

Opgave 12: Voeg deze speech synthesis aan de AlarmClock class toe zodat wanneer de tijd van een event aanbreekt niet alleen de event wordt geprint maar ook de event-beschrijving wordt uitgesproken.

6.1 Decoupling

Het is **geen** goed idee om de speech synthesis code simpelweg aan de AlarmClock class toe te voegen, want dan gebruikt een alarmklok altijd speech synthesis terwijl we dat in andere software, waar we ook de AlarmClock class gebruiken, misschien wel helemaal niet willen. Het is veel beter om de AlarmClock niet te koppelen met de code die nodig is om een event af te handelen. Een manier om dat te doen is om een functie mee te geven aan de wait_for_and_handle_events() methode die verantwoordelijk is voor de afhandeling van events. Ditzelfde principe zagen we al eerder bij de sort() methode of bij de sorted() functie waarbij we een "key" functie konden meegeven die de volgorde van events bepaalde.

Onderstaande voorbeeld laat zien hoe we zelf een process_numbers() functie kunnen schrijven waaraan we een number_handler() functie kunnen meegeven die dan vervolgens gebruikt wordt om elk nummer af te handelen. De process_numbers() functie is hierdoor niet gekoppeld met de code voor de afhandeling en we kunnen daarom dus ook gemakkelijk verschillende afhandelingen gebruiken:

number_handler.py

```
def process_numbers(numbers, number_handler):
    print("process_numbers:")
    for i in numbers:
        number_handler(i)


def double_the_number(i):
    print(i * 2)


def repeat_three_times(i):
    print(str(i) * 3)


def main():
    numbers = [1, 2, 3, 4, 5]
    process_numbers(numbers, double_the_number)
    process_numbers(numbers, repeat_three_times)
    process_numbers(numbers, lambda i: print(i * i))


if __name__ == "__main__":
    main()
```

Opgave 13: Voeg een event_handler argument toe aan de wait_for_and_handle_events() methode van AlarmClock zodat deze class niet gekoppeld is aan de event-afhandeling.

```
def wait_for_and_handle_events(self, event_handler):
```

Roep vervolgens in de main() functie deze methode aan met een event_handler die elk event print en uitspreekt wanneer het daar tijd voor is.

7 Gegevensabstractie

De alarmklok software moet ook op digitale horloges gebruikt kunnen worden, maar omdat digitale horloges weinig geheugen hebben moet daarvoor wel eerst de Time class worden aangepast zodat het maar 1 integer in plaats van 3 integers gebruikt om de tijd op te slaan. Zo zou de tijd 07:59:30 niet als:

```
self.hours = 7
self.minutes = 59
self.seconds = 30
```

maar bijvoorbeeld als:

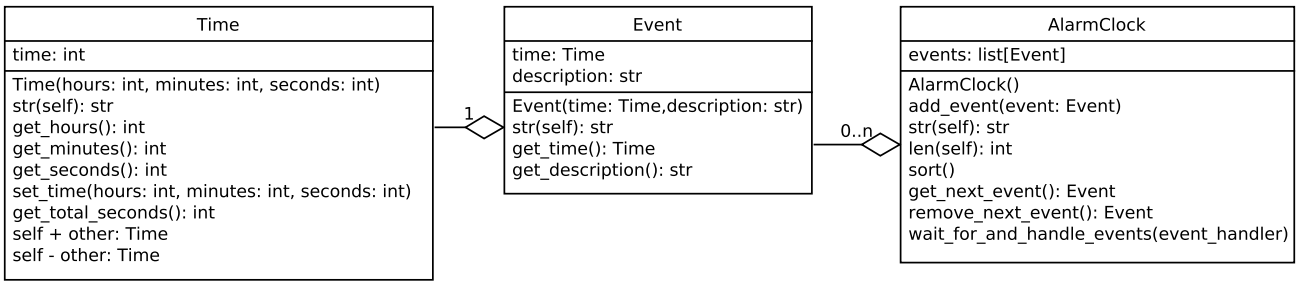
```
self.time = 75930
```

in een object van de Time class moeten worden opgeslagen. Aangezien de Event en AlarmClock classes uitsluitend gebruikmaken van de interface van de Time class en niet van de interne representatie afhankelijk zijn, blijven deze classes onaangetast door veranderingen in de interne implementatie van de Time class. Dit betekent dat aanpassingen beperkt kunnen blijven tot de Time class zelf. Dit illustreert een belangrijk voordeel van abstractie in objectgeoriënteerd programmeren: het mogelijk maken van veranderingen **achter** een interface zonder de rest van de code te hoeven aanpassen.

Opgave 14: Herschrijf de implementatie van de Time class zodat het maar 1 integer gebruikt om de tijd op te slaan en herschrijf de implementatie van de corresponderende methoden zodat deze aanpassing achter het interface van de Time class verborgen blijft en dus de Event en AlarmClock classes correct blijven werken.

8 UML Class Diagram

Het uiteindelijke UML class diagram van de alarmklok software ziet er dan uit zoals in figuur 7:



Figuur 7: UML class diagram van de alarmklok software.