Week 2 Opgave 1

Bits & Gates

In deze opgaven van week 2 simuleren we gates, circuits, half-adder, full-adder en uiteindelijk een adder voor een register met vier bits.

De opgave `w2o2`, `w2o3` en `w2o4` zijn een uitbreiding op deze opgave. Zorg dus dat de tests slagen voordat je ze gebruikt in de volgende opgave.

Om met `Gate` te kunnen werken hebben we een aantal bouwstenen nodig. Deze zitten in de module `w2_lib.py`. Deze kun je aan je python-file toevoegen door `from w2_lib import *`. In `w2_lib` staan hulpprogrammatuur. Deze kan maar hoef je niet te bestuderen. Nu zijn de objecten van `w2_lib` beschikbaar als bijvoorbeeld `Bit`.

Class

Om `Bits`, `Gates`, `Circuits` en `Registers` gebruiken een python constructie `class`. Hierbij wordt een klasse gedefinieerd en vanuit een klasse kunnen objecten worden aangemaakt. In `w2_lib.py` zijn een aantal klassen gedefinieerd. Een stuk van de `Bit` definitie is:

```
1 class Bit:
2    def __init__(self, bit=None, name=''):
3        self.bit = bit
4        self.name = name
5    def set(self, bit):
6        self.bit = bit
```

`1` De definitie van de klasse met naam `Bit`. We gebruiken bijvoorkeur namen die beginnen met een hoofdletter voor klassen.

`2` De speciale methode `__init__` wordt aangeroepen als een object van uit de klasse wordt gemaakt. Deze functie wordt gebruikt om het object te initialiseren. De default-argumenten zorgen er voor dat niet alle argumenten hoeven meegegeven te worden bij de creatie van het object.

`3-4` De initialisatie van het object. Als geen argumenten worden meegegeven bij de creatie wordt het object geïnitialiseerd met de object-variabele `bit` op `None` en de object-variabele `name` op `""`. De object-variabele worden aangeduid met `self.bit` en `self.name`.

`5` De definitie van een object-methode `set`. Deze heeft als argumenten `self` en `bit`. Bij de aanroep van set is een waarde voor `bit` verplicht (geen default-waarde is gedefinieerd).

`6` De waarde van de object-variable `self.bit` wordt gezet op waarde van het argument `bit`.

We maken een object van een bepaalde klasse door de klasse aan te roepen als een functie. Hierbij kunnen argumenten worden meegegeven zoals gedefinieerd in de `__init__` functie van de klasse.

Probeer eens:

```
>>> from w2_lib import *
>>> b = Bit(1, 'a')
>>> b
<w2_lib.Bit object at 0x104a92e80>
>>> str(b)
'<Bit:a  bit=1>'
>>> b.bit
```

Bit

We hebben natuurlijk bits nodig om iets te kunnen doen met gate. De python-class `Bit` implementeert een Bit. Een bit heeft normaal gesproken twee states '0' en '1'. Voor het gemak voegen we nog een state toe '?'. Deze state betekent ongeïnitialiseerd en helpt ons fouten in een circuit te voorkomen.

Een bit kun je creëren door het statement:

```
b = Bit()
```

Dit creert een ongeinitialiseerd bit, een bit met waarde '?'. met:

```
bTrue = Bit(1); bFalse = Bit(0)
```

Worden respectievelijk een bit met waarde '1' (True) en een bit met waarde '0' (False) gemaakt. Veelal is het handig om je bits een naam te geven.

```
>>> bA = Bit(0, name='A')
>>> print(str(bA))
<Bit:A  bit=0>
>>> print(int(bA))
>>> 0
```

De str-value van Bit geeft informatie over Bit, de int-value geeft de waarde. De waarde opvragen van een ongeïnitialiseerd Bit geeft een foutmelding. Wat de int-waarde van een object is kun je zelf bepalen door een methode `def __int__(self):` aan je object-klasse toe te voegen. Evenzo de str-value met `def __str__(self):`.

```
>>> bA = Bit(name='A')
>>> print(int(bA))
Traceback (most recent call last):
  File "", line 1, in
  File ".../w2_lib.py", line 32, in __int__
    raise ValueError('Uninitialised Bit name={}'.format(self.name))
ValueError: Uninitialised Bit name=A
```

Bit heeft een methode `get(self)` en `set(self, v)` om de waarde op te halen en te zetten. Voor object methoden geld dat deze bij de definite een standaard eerste argument `self` hebben. Dit argument is het object. Via `self` kun je bij variabele en methoden van het object komen. Deze methode worden aangeroepen vanuit het object.

Probeer het zelf:

```
>>> b = Bit(0)
>>> str(b)
'<Bit:   bit=>'
>>> b.set('1')
>>> str(b)
'<Bit:   bit=1>'
>>>
```

Je kunt meer informtie over Bit opvragen door `help(Bit)` .

Letop: Kijk naar het gebruik van `self` in de documentatie.

Sub-class

We gebruiken in python ook veelal de **sub-klasse** constructie.

```
class Gate_21(Gate):
    def __init__(self, ia, ib, oc):
        self.ia = ia
        self.ib = ib
        self.oc = oc
class Nand(Gate_21):
    def _run(self):
        self.oc.set( ( self.ia.get() & self.ib.get() ^ 1 ) )
```

De klasse `Nand` is een sub-klasse van de klasse `Gate_21` . Dit houdt in dat alle methoden die in de klasse `Gate_21` gedefineerd zijn ook in de klasse `Nand` gebruikt kunnen worden. Dus zonder dat een `__init__` methode bij `Nand` gedefinieerd is. Deze bestaat wel door de definitie in de super-klasse (Gate_21).

Gate

De logica wordt gerealiseerd door een Gate. Een gate heeft een of meer input Bits en een of meer output Bits. Op basis van de input Bits en het type Gate worden de output bits gezet.

Er in zijn in `w2_lib.py` twee klasse gedefineerd `Gate_11` en `Gate_21`. Deze zijn het prototype van een single-input single-output gate en een dual-input single-output gate.

Letop: Als niet alle inputs een waarde hebben krijg je een foutmelding.

De methode `_run(self)` voert de berekening uit.

```
class Nand(Gate_21):
    """ The Nand operator """
    def _run(self):
        self.oc.set( ( self.ia.get() & self.ib.get() ^ 1 ) )
```

Een voorbeeld van het gebruik van Gates.

```
>>> bA = Bit(1, 'A')
>>> bB = Bit(0, 'B')
>>> bC = Bit(name='C')
>>> str(bA), str(bB)
('<Bit:A  bit=1>', '<Bit:B  bit=0>')
>>> g = Nand(bA, bB, bC)
>>> g.run()
>>> str(bC)
'<Bit:C  bit=1>'
```

Eerst worden de input Bits (`bA` , `bB`) aangemaakt met de waarde '1' en '0'. En een ongeïnitialiseerde output Bit (`bC`). Vervolgens wordt een Nand-gate aangemaakt. Door de methode `run` aan te roepen op de Gate wordt de operatie uitgevoerd en is het resultaat in `bC` zichtbaar.

Opgave 1

Gates

De file `w1o1.py` bevat de implementatie van een Not-gate en een Nand-gate. De opdracht is om een And-gate, Or-gate, Nor-gate en een Xor-gate te maken. Dit lijkt op opgave `w1o3` , het idee is het zelfde, maar nu moet de functie `_run` worden geïmplementeerd. Deze implementatie lijken op die van Nand, maar de `_run` methode is anders. Merk op dat de gates een sub-klasse zijn van `Gate_1.1` of `Gate_21`

Inleveren

Zorg dat je naam in het programma staat (`__author__`) en lever jouw versie van `w2o1.py` in op VLO. Inleveren voor **vrijdag 15 feb 24:00** (voltijd) of **maandag 18 feb 24:00** (deeltijd).