Sushi

We will start off with a little bit of background information. According to Wikipedia, sushi (寿司) is a dish from Japan that, in one way or another, consists of a bite of rice of a few centimetres thick, with an ingredient on top of it or in it, such as raw fish or other seafood, but also cooked fish, baked egg or vegetables. The two most popular types of sushi, combined with a few subdivisions, are:

- **maki** — one or more tasty things rolled in seaweed and rice
  - **futomaki** — seaweed on the outside, usually vegetarian
  - **temaki** — cone-shaped role of seaweed filled with rice and tasty things
  - **uramaki** — rice on the outside
- **nigiri** — a hand-formed ball of rice, garnished with something tasty
  - **gunkanmaki** — with a loose or soft topping that is kept in place with a strike of seaweed
  - **temarizushi** — where the topping is only pressed in the rice ball

This kind of hierarchic division magically lends to an implementation of classes and inheritance.

Assignment

If you ever had dinner in a Japanese sushi-restaurant, you have probably had a good laugh with the funny phrases and many mistakes in the English translation of Japanese descriptions of the dishes. We are going to write a simple program that (in theory) can be used by the owner of a sushi-restaurant to compose a menu, completed with English and Japanese (although not in Japanese characters, sorry for that) descriptions of the dishes. Below we explain how you have to construct this program in steps.

1. To start off, make a class `Ingredient`. Two arguments must be given to the initializing method — `japanese` and `english` — that correspond with the Japanese and English names of the ingredients. The argument `english` is optional, and has the value of the argument `japanese` as a standard value, if it is not given (just like when some names on the meno are not translated, and you have to guess what they are). The value of both arguments must be kept as attribute of the objects of the class `Ingredient`.
   (click here for an extra tip)
2. Add two methods to the class `Ingredient`: `__str__(self)` and `english(self)`. Both methods must print a string. The `__str__` method is automatically called when an object of the class `Ingredient` is printed or converted to a string. Make sure that the method `__str__` print the Japanese name of the ingredient, and the method `english` the English translation of the ingredient.
   (click here for an extra tip)
3. We have already made a file based on the data of the website http://www.bento.com/sushivoc.html that lists a number of much-used sushi-ingredients. The first column contains the original Japanese name of the ingredient, and the second column contains the English translation, if it is known. There are a number of Japanese terms for which no English translation was given (intentionally), e.g. `fugu`. For these ingredients, the Japanese name should also be used in the English translation. You can download the file here.
   Write a function `indexIngredients` to which an opened file object must be given as an

argument. This function must index all ingredients of the file in the format of a dictionary, where the Japanese name is used as a key, and the corresponding value is an Ingredient object that can print both the Japanese and the English name of an ingredient. Test the correct functioning of this function before continuing.

4. Now, make a class Sushi that can be used to represent various sushi-dishes. The class Sushi must have an initializing method to which a list of Ingredient objects must be given.
5. Then, add a method __str__(self) to the class klasse Sushi. This method must print a string. The string must list the Japanese name of all ingredients in the dish. The contents of the string is formulated in English, so for example "buri", "buri and tsubugai" or "buri, tsubugai and kanpachi" are the correct ways to print respectively one, two and three ingredients. It does not suffice to simply separate the ingredients by commas.
   (click here for an extra tip)
   (click here for an extra tip)
6. When making a Sushi object, a list of Ingredient objects must be given to the initializing method. To allow the user to make Sushi objects in a simple manner, we ask you to write a function makeSushi. A string that lists the Japanese name of all ingredients in the sushi, must be given to this function as an argument, separated by spaces. The function must first convert this string of names to a list of corresponding Ingredient objects, and must then print a Sushi object that is made on the bases of this list. Of course, when converting the string to the list of Ingredient objects, the dictionary that is printed by the function indexIngredients must be used. Based on the function makeSushi, it is simple to make a Sushi object via a string like "unagi fugu ika sake". This allows testing all code you have written up until now in the following manner:

   ```
   >>> sushi = makeSushi("unagi fugu ika sake")
   >>> print(sushi)
   unagi, fugu, ika and sake
   ```

   Pay attention to the fact that the function makeSushi must also be able to make sushi's based on ingredients that do not appear in the dictionary. In this case, Ingredient objects must be made, solely based on the Japanese names of ingredients.
7. Now, give the class Sushi another method english(self), that prints a description of the sushi-dishes where the names of the ingredients are now translated to English. This method must print a similar string as the one that is printed by the method __str__, but where the ingredients are listed in English and not in Japanese. In this case, however, it is not desirable to call that method __str__ for the implementation of the method english of a Sushi object and translating the ingredients in the printed string one by one. Because a list of Ingredient objects is given when making a Sushi object, it is sufficient to call the method english of the individual Ingredient objects and to draw it up in a correct way with commas and the word and. With this, you are now also able to put the English translation of a sushi-dish on the menu.

   ```
   >>> sushi = makeSushi("unagi fugu ika sake")
   >>> print(sushi.english())
   eel, fugu, squid and salmon
   ```

   Because the methods __str__ and english of a Sushi object display the list of ingredient names in the same way, is it in any case interesting to write a help method that can print a list of ingredient names, no matter what language the ingredients are in.
8. Now make a class Maki that inherits all its attributes from the class Sushi. Objects of the class Maki behave in the exact same way as the objects of the class Sushi, except that instead of

listing the names of the ingredients, now more informative descriptions are given. Here, we want the methods __str__ and english to print a string of the format

[ingredients] rolled in [rice] and [seaweed]

where [ingredients] represents our grammatical list of ingredients, and [rice] and [seaweed] represent two other ingredients that are consistently used in all types of sushi. For these last two ingredients — like it is the case with all other ingredients — it must be assured that the correct language is used on the correct position. These set ingredients, however, are not summed up in the list of ingredients that is given when making sushi. However, they are implied by the type of sushi. You can better make constants for these ingredients that you put in the right place in the class hierarchy. The code fragment below indicates how (but not where) we have defined the set ingredients.

```
rice = Ingredient('su-meshi', 'sushi rice')
seaweed = Ingredient('nori', 'seaweed')
```

9. Now that we already have two kinds of sushi, it is time to revise our way in which these sushi-objects are made. As an initial point we have already written a function makeSushi, that makes a Sushi object based on the string of the Japanese ingredient names. Now, we want to expand this so that the strings "unagi fugu" and "unagi fugu sushi" must be see as the description of an general sushi for which a Sushi object must be made. If the last word of the description should be "maki", however, a Maki object must be made. Because we also want to define other sushi kinds later on, this divide must happen as flexible as possible, by using an implementation that later can easily be expanded. As a general rule, we suppose that a sushi is described by a sequence of Japanese names of ingredients, possibly followed by a specific type of sushi. If no type of sushi is given, a Sushi object must be made, otherwise a sushi-object of the given type must be made.
   (click here for an extra tip)
   We can now also reorganize the program code that takes care of make sushi-objects. Until now, sushi-objects were made based on the function makeSushi, that on its turn used both the function indexIngredients (to make ingredients) and a data structure that represents the different types of sushi. When programming object-oriented, it is better to bundle all these aids in a class SushiMaker. We can parameterize the initialization method of this class with an opened file object from which the translation of sushi-ingredients can be read. Standard, this file object can be opened in order to read the file to which was referred above. Then, sushi objects can be made in the following manner

```
>>> chef = SushiMaker()
>>> sushi = chef.makeSushi('unagi fugu sushi')
>>> print(sushi)
unagi and fugu
>>> print(sushi.english())
eel and fugu
>>> sushi = chef.makeSushi('unagi fugu maki')
>>> print(sushi)
unagi and fugu rolled in su-meshi and nori
>>> print(sushi.english())
eel and fugu rolled in sushi rice and seaweed
```

10. Cool! But now we still have to add a number of sushi varieties to our arsenal. Futomaki, temaki and uramaki are all subvarieties of maki, and can be represented by classes that inherit from the class Maki. The way in which the description of every one of these varieties

as a string must happen is as follows:

Futomaki: "[ingredients] rolled in [rice] and [seaweed], with [seaweed] facing out"
Temaki: "cone of [seaweed] filled with [rice] and [ingredients]"
Uramaki: "[ingredients] rolled in [seaweed] and [rice], with [rice] facing out"

For the implementation of the different descriptions, it could be handy to use a formatstring. If you define, for example, the following string

```
>>> description = "{ingredients} rolled in {rice} and {seaweed}, with {seaweed} facing out"
```

you can do the following

```
>>>  description.format(rice='su-meshi', seaweed='nori', ingredients='unagi fugu')
'unagi fugu rolled in su-meshi and nori, with nori facing out'
```

This is actually a fairly powerful instrument, because it allows to enclose rice and seaweed in the dictionary, even if they don't occur in the format string! with this information in hand, you can now rewrite the basic class Sushi, so that the description happens on the bases of a property of the object that is used as format string in combination with a dictionary of "rice", "seaweed" and "ingredients". This way, it is sufficient that every child class defines an own format string and everything works as it should. Making new child classes is then simply done in the following manner

```
class Futomaki(Maki):

    description = "{ingredients} rolled in {rice} and {seaweed}, with {seaweed} facing out"

class Temaki(Maki):

    description = "cone of {seaweed} filled with {rice} and {ingredients}"
```

Make sure that this works for both descriptions with Japanese and English names of the ingredients, and also make sure that the data structure with the different kinds of sushi in the class SushiMaker is adjusted in a correct way. This way, the following should also work:

```
>>> chef = SushiMaker()
>>> sushi = chef.makeSushi('unagi ohyo uramaki')
>>> print(sushi)
unagi and ohyo rolled in nori and su-meshi, with su-meshi facing out
>>> print(sushi.english())
eel and halibut rolled in seaweed and sushi rice, with sushi rice facing out
>>> sushi = chef.makeSushi('ikura temaki')
>>> print(sushi)
cone of nori filled with su-meshi and ikura
>>> print(sushi.english())
cone of seaweed filled with sushi rice and salmon roe
```

11. We are almost there. The still have to be made a few classes for the last sequence of sushi varieties. Add a class Nigiri that inherits from the class Sushi, and add two classes Gunkanmaki and Temarizushi that inherit from the class Nigiri. Because nigiri normally only has one topping, you must try to benefit from the inheritance to enforce that this condition applies to all nigiri varieties. You do this by providing an adjusted implementation of the method __init__ within the class Nigiri. If this condition is violated, an InvalidSushiError must be raised (you must define this one yourself, seeing as the Python libraries do not quite contain an error of that kind). Moreover, don't forget to call the method __init__ of the basic class when reimplementing the method __init__. The descriptions of the sushi varieties that belong with

these classes are

Nigiri: "hand-formed [rice] topped with [ingredients]"
Gunkanmaki: "[ingredients] on [rice] wrapped in a strip of [seaweed]"
Temarizushi: "[ingredients] pressed into a ball of [rice]"

(click here for an extra tip)
As a last test, the following should now also work

```
>>> chef = SushiMaker()
>>> sushi = chef.makeSushi('fugu ohyo ika unagi')
>>> print(sushi)
fugu, ohyo, ika and unagi
>>> print(sushi.english())
fugu, halibut, squid and eel
>>> sushi = chef.makeSushi('fugu ohyo ika unagi sushi')
>>> print(sushi)
fugu, ohyo, ika and unagi
>>> print(sushi.english())
fugu, halibut, squid and eel
>>> sushi = chef.makeSushi('ika sake gunkanmaki')
Traceback (most recent call last):
InvalidSushiError: Nigiri has only one topping
Traceback (most recent call last):
InvalidSushiError: Nigiri has only one topping
```

Epilogue


This rare snow roll shows that even the wind wants sushi.

We beginnen met een klein beetje achtergrondinformatie. Volgens Wikipedia is sushi (寿司) een gerecht uit Japan dat in één of andere vorm altijd bestaat uit een hapje rijst van een paar centimeter dik, met daarop of daartussen de overige ingrediënten, waaronder rauwe vis of andere zeevruchten, maar ook gaar gemaakte vis, gebakken ei of groenten. De twee populairste vormen van sushi, samen met een paar onderverdelingen, zijn:

- **maki** — één of meer lekkere dingen opgerold in zeewier en rijst
  - **futomaki** — zeewier aan de buitenkant, meestal vegetarisch

    - **temaki** — kegelvormige rol zeewier gevuld met rijst en lekkere dingen
    - **uramaki** — rijst aan de buitenkant
- **nigiri** — een met de hand gevormd bolletje rijst, gegarneerd met iets lekkers
    - **gunkanmaki** — met een losse of zachte topping die op zijn plaats gehouden wordt door een strook van zeewier
    - **temarizushi** — waarbij de topping enkel maar geperst wordt in de rijstbal

Dit soort van hiërarchische onderverdeling leent zich wonderwel tot een implementatie met gebruik van klassen en overerving.

Opgave

Als je ooit al eens bent gaan eten in een sushi-restaurant, dan heb je waarschijnlijk ook goed gelachen met de grappige zinsneden en vele spelfouten in de Engelse vertaling van de Japanse omschrijvingen van de gerechten. We gaan een eenvoudig programma schrijven dat (in theorie) door de eigenaar van een sushi-restaurant kan gebruikt worden om een menu op te stellen, compleet met Engelse en Japanse (maar wel niet de echte Japanse karakters, sorry daarvoor) omschrijving van de gerechten. We geven hieronder in verschillende stappen aan hoe dit programma tot stand moet komen.

1. Maak om te beginnen een klasse `Ingredient`. Aan de initialisatiemethode moeten twee argumenten doorgegeven worden — `japans` en `engels` — die corresponderen met de Japanse en Engelse benaming van het ingrediënt. Het argument `engels` is optioneel, en heeft als standaardwaarde de waarde van het argument `japans` indien het niet wordt opgegeven (net zoals op menu's de namen van sommige ingrediënten niet vertaald worden, en je het raden hebt waarvoor ze staan). De waarde van beide argumenten moet bijgehouden worden als attribuut van de objecten van de klasse `Ingredient`.
   (klik hier voor een extra tip)
2. Voeg twee methoden toe aan de klasse `Ingredient`: `__str__(self)` en `engels(self)`. Beide methoden moeten een string teruggeven. De `__str__` methode wordt automatisch aangeroepen wanneer een object van de klasse `Ingredient` wordt afgedrukt of omgezet naar een string. Zorg ervoor dat de methode `__str__` de Japanse benaming van het ingrediënt teruggeeft, en de methode `engels` de Engelse vertaling ervan.
   (klik hier voor een extra tip)
3. We hebben op basis van de gegevens op de website http://www.bento.com/sushivoc.html reeds een bestand aangemaakt dat een aantal veelgebruikte sushi-ingrediënten oplijst. De eerste kolom bevat de originele Japanse benaming van het ingrediënt, en de tweede kolom bevat de Engelse vertaling ervan indien die gekend is. Er zijn een aantal Japanse termen waarvoor (bewust) geen Engelse vertaling werd opgegeven (bijvoorbeeld `fugu`). Voor deze ingrediënten moet de Japanse benaming ook in de Engelse vertaling gebruikt worden. Je kunt het bestand hier downloaden.
   Schrijf een functie `indexeerIngredienten` waaraan een geopend bestandsobject als argument moet doorgegeven worden. Deze functie moet alle ingrediënten uit het bestand indexeren onder de vorm van een dictionary, waarbij de Japanse benaming gebruikt wordt als sleutel, en de bijhorende waarde een `Ingredient` object is dat zowel de Japanse als Engelse benaming van het ingrediënt kan weergeven. Test de correcte werking van deze functie alvorens verder te gaan.
4. Maak nu een klasse `Sushi` die gebruikt kan worden om verschillende sushi-gerechten voor te stellen. De klasse `Sushi` moet een initialisatiemethode hebben waaraan een lijst van

Ingredient objecten moet doorgegeven worden.

5. Voeg vervolgens een methode __str__(self) toe aan de klasse Sushi. Deze methode moet een string teruggeven. Deze string moet de Japanse benaming van alle ingrediënten in het gerecht oplijsten. De inhoud van de string zelf is geformuleerd in het Engels, dus bijvoorbeeld "buri", "buri and tsubugai" of "buri, tsubugai and kanpachi" zijn de correcte manieren om respectievelijk één, twee en drie ingrediënten af te drukken. Het volstaat dus niet om de namen van de ingrediënten louter door komma's van elkaar te scheiden.
   (klik hier voor een extra tip)
   (klik hier voor een extra tip)
6. Bij het aanmaken van een Sushi object moet een lijst van Ingredient objecten aan de initialisatiemethode doorgeven worden. Om de gebruiker toe te laten op een eenvoudige manier Sushi objecten aan te maken, vragen we je om een functie maakSushi te schrijven. Aan deze functie moet een string als argument doorgegeven worden, die de Japanse benaming van alle ingrediënten van de sushi oplijst, van elkaar gescheiden door spaties. De functie moet deze string van namen eerst omzetten naar een lijst van de corresponderende Ingredient objecten, en moet daarna een Sushi object teruggeven dat wordt aangemaakt op basis van deze lijst. Uiteraard moet bij het omzetten van de string naar de lijst van Ingredient objecten gebruik gemaakt worden van de dictionary die teruggegeven wordt door de functie indexeerIngredienten. Op basis van de functie maakSushi wordt het eenvoudig om een Sushi object aan te maken via een string zoals "unagi fugu ika sake". Dit maakt het mogelijk om alle code die je tot nu toe hebt geschreven op de volgende manier te testen

   ```
   >>> sushi = maakSushi("unagi fugu ika sake")
   >>> print(sushi)
   unagi, fugu, ika and sake
   ```

   Let hierbij ook op het feit dat de functie maakSushi ook sushi's moet kunnen maken op basis van ingrediënten die niet voorkomen in de dictionary. In dit geval moeten Ingredient objecten aangemaakt worden, enkel op basis van de Japanse benaming van het ingrediënt.
7. Geef de klasse Sushi nu ook een methode engels(self), die een omschrijving van het sushi-gerecht teruggeeft waarbij de benaming van de ingrediënten nu vertaald is naar het Engels. Deze methode moet dus een gelijkaardige string teruggeven als deze die door de methode __str__ wordt teruggegeven, maar waarbij de ingrediënten in het Engels en niet in het Japans worden opgelijst. In dit geval is het echter niet wenselijk om voor de implementatie van de methode engels van een Sushi object die methode __str__ van het object aan te roepen en de ingrediënten in de teruggegeven string één voor één te vertalen. Aangezien bij het aanmaken van een Sushi object een lijst van Ingredient objecten wordt doorgegeven, volstaat het om de methode engels van de individuele Ingredient objecten aan te roepen en deze op een correctie manier op te maken met komma's en het woord and. Hiermee ben je nu ook in staat om de Engelse vertaling van een sushi-gerecht op het menu te zetten.

   ```
   >>> sushi = maakSushi("unagi fugu ika sake")
   >>> print(sushi.engels())
   eel, fugu, squid and salmon
   ```

   Aangezien de methoden __str__ en engels van een Sushi object de lijst van ingrediëntnamen op eenzelfde manier weergeven, is het in dit geval ook interessant om een hulpmethode te schrijven die een lijst van ingrediëntnamen kan weergeven, ongeacht de taal waarin deze ingrediënten benoemd worden.

8. Maak nu een klasse Maki die al zijn attributen overerft van de klasse Sushi. Objecten van de klasse Maki gedragen zich op precies dezelfde manier als objecten van de klasse Sushi, behalve dat in plaats van enkel de namen van de ingrediënten op te lijsten, er nu een meer informatieve omschrijving moet gegeven worden. Hierbij willen we de methoden __str__ en engels telkens een string laten teruggeven van de vorm

   [ingredienten] rolled in [rijst] and [zeewier]

   waarbij [ingredienten] onze grammaticale lijst van ingrediënten voorstelt, en [rijst] en [zeewier] twee andere ingrediënten voorstellen die consistent gebruikt worden over alle types van sushi heen. Voor deze laatste twee ingrediënten moet — net zoals dat trouwens voor alle andere ingrediënten ook het geval is — verzekerd worden dat de correcte taal gebruikt wordt op de correcte plaats. Deze vaste ingrediënten worden echter niet opgesomd in de lijst van ingrediënten die wordt opgegeven bij het aanmaken van sushi. Ze worden daarentegen geïmpliceerd door het type van sushi. Je kunt dus best constanten aanmaken voor deze vaste ingrediënten die je op de juiste plaats opneemt in de klassehiërarchie. Onderstaand codefragment geeft alvast aan hoe (maar niet waar) wij deze vaste ingrediënten gedefinieerd hebben.

   ```
   rijst = Ingredient('su-meshi', 'sushi rice')
   zeewier = Ingredient('nori', 'seaweed')
   ```

9. Nu we reeds over twee soorten sushi beschikken, wordt het tijd om onze manier waarop deze sushi-objecten aangemaakt worden te herzien. Als vertrekpunt hebben we reeds een functie maakSushi geschreven, die een Sushi object aanmaakt op basis van een string van Japanse ingrediëntnamen. We willen dit nu uitbreiden zodat de strings "unagi fugu" en "unagi fugu sushi" gezien worden als de omschrijving van een algemene sushi waarvoor er een Sushi object moet aangemaakt worden. Als het laatste woord van de omschrijving echter "maki" zou zijn, dan moet er een Maki object aangemaakt worden. Aangezien we straks ook nog andere soorten sushi willen definiëren, moet dit onderscheid bovendien zo flexibel mogelijk gebeuren, door gebruik te maken van een implementatie die later makkelijk kan uitgebreid worden. Als algemene regel veronderstellen we dat een sushi wordt omschreven door een reeks van Japanse benamingen van ingrediënten, mogelijks gevolgd door een speciek type van sushi. Indien geen type van sushi wordt opgegeven, dan moet een Sushi object aangemaakt worden, anders moet een sushi-object van het opgegeven type aangemaakt worden.
   (klik hier voor een extra tip)
   We kunnen nu meteen ook de programmacode die zorgt voor het aanmaken van sushi-objecten een beetje herorganiseren. Tot nu toe werden sushi-objecten aangemaakt op basis van de functie maakSushi, die op zijn beurt enerzijds gebruik maakt van de functie indexeerIngredienten (om ingrediënten aan te maken) en van een datastructuur die de verschillende types van sushi voorstelt. Als we dan toch objectgericht aan het programmeren zijn, dan is het beter om al deze hulpmiddelen te bundelen in een klasse SushiMaker. We kunnen de initialisatiemethode van deze klasse parameteriseren met een geopend bestandsobject waaruit de vertaling van de sushi-ingrediënten kan gelezen worden. Standaard kan dit bestandsobject geopend worden voor het inlezen van het bestand waarnaar hierboven verwezen wordt. Dan kunnen sushi-objecten op de volgende manier aangemaakt worden

   ```
   >>> chef = SushiMaker()
   >>> sushi = chef.maakSushi('unagi fugu sushi')
   ```

```
>>> print(sushi)
unagi and fugu
>>> print(sushi.engels())
eel and fugu
>>> sushi = chef.maakSushi('unagi fugu maki')
>>> print(sushi)
unagi and fugu rolled in su-meshi and nori
>>> print(sushi.engels())
eel and fugu rolled in sushi rice and seaweed
```

10. Cool! Maar we moeten nu wel nog een aantal soorten sushi toevoegen aan ons arsenaal. Futomaki, temaki en uramaki zijn allemaal ondersoorten van maki, en kunnen dus voorgesteld worden door klassen die erven van de klasse Maki. De manier waarop de omschrijving van elk van deze soorten als string moet gebeuren is de volgende:

    ```
    Futomaki: "[ingredienten] rolled in [rijst] and [zeewier], with [zeewier] facing out"
    Temaki: "cone of [zeewier] filled with [rijst] and [ingredienten]"
    Uramaki: "[ingredienten] rolled in [zeewier] and [rijst], with [rijst] facing out"
    ```

    Voor de implementatie van deze verschillende omschrijvingen kan het handig zijn om gebruik te maken van een formaatstring. Als je bijvoorbeeld de volgende string definieert

    ```
    >>> omschrijving = "{ingredienten} rolled in {rijst} and {zeewier}, with {zeewier} facing out"
    ```

    dan kan je het volgende doen

    ```
    >>>  omschrijving.format(rijst='su-meshi', zeewier='nori', ingredienten='unagi fugu')
    'unagi fugu rolled in su-meshi and nori, with nori facing out'
    ```

    Dit is eigenlijk een behoorlijk krachtig instrument, omdat het toelaat om rijst en zeewier in te sluiten in de dictionary, zelfs als ze niet voorkomen in de formaatstring! Met deze informatie in de hand, kan je nu de basisklasse Sushi herschrijven, zodat de omschrijving gebeurt op basis van een eigenschap van het object die als formaatstring gebruikt wordt in combinatie met een dictionary van "rijst", "zeewier" en "ingredienten". Op die manier volstaat het dat elke kindklasse een eigen formaatstring definieert en alles werkt zoals het moet. Nieuwe kindklassen aanmaken kan dan eenvoudigweg op de volgende manier gebeuren

    ```
    class Futomaki(Maki):

        omschrijving = "{ingredienten} rolled in {rijst} and {zeewier}, with {zeewier} facing out"

    class Temaki(Maki):

        omschrijving = "cone of {zeewier} filled with {rijst} and {ingredienten}"
    ```

    Zorg er voor dat dit zowel werkt voor omschrijvingen met Japanse en Engelse benamingen van de ingrediënten, en zorg er ook voor dat de datastructuur met de verschillende soorten sushi in de klasse SushiMaker op een gepaste manier wordt aangepast. Op deze manier moet het volgende nu ook werken:

    ```
    >>> chef = SushiMaker()
    >>> sushi = chef.maakSushi('unagi ohyo uramaki')
    >>> print(sushi)
    unagi and ohyo rolled in nori and su-meshi, with su-meshi facing out
    >>> print(sushi.engels())
    eel and halibut rolled in seaweed and sushi rice, with sushi rice facing out
    >>> sushi = chef.maakSushi('ikura temaki')
    >>> print(sushi)
    ```

```
cone of nori filled with su-meshi and ikura
>>> print(sushi.engels())
cone of seaweed filled with sushi rice and salmon roe
```

11. We zijn er bijna. Er moeten nog een paar klassen aangemaakt worden voor een laatste reeks sushisoorten. Voeg een klasse Nigiri toe die erft van de klasse Sushi, en voeg twee klassen Gunkanmaki en Temarizushi toe die erven van de klasse Nigiri. Omdat nigiri normaalgezien maar één topping heeft, moet je voordeel zien te halen uit de overerving om af te dwingen dat deze voorwaarde geldt voor alle soorten nigiri. Dit doe je door een aangepaste implementatie van de methode __init__ te voorzien binnen de klasse Nigiri. Indien deze voorwaarde geschonden wordt, dan moet een InvalidSushiError opgeworpen worden (deze moet je zelf definiëren, aangezien de Python bibliotheken niet zo volledig zijn dat dit soort fouten reeds bestaat). Vergeet bij het herimplementeren van de methode __init__ ook niet om de methode __init__ van de basisklasse aan te roepen. De omschrijvingen van de soorten sushi die horen bij deze klassen zijn

```
Nigiri: "hand-formed [rijst] topped with [ingredienten]"
Gunkanmaki: "[ingredienten] on [rijst] wrapped in a strip of [zeewier]"
Temarizushi: "[ingredienten] pressed into a ball of [rijst]"
```

(klik hier voor een extra tip)
Als laatste test, moet nu ook het volgende werken

```
>>> chef = SushiMaker()
>>> sushi = chef.maakSushi('fugu ohyo ika unagi')
>>> pr
```