Ranking für kartesische Produkte in lexikographischer Ordnung.

```
def rank_product_lex(the_tuple, the_maxima):
    """Bestimme die _Nummer_ des gegebenen Tupels
    innerhalb des cartesischen Produkts
    [the_maxima[0]] x [the_maxima[1]] x ..."""
    # Für unsere Zwecke ist es besser, "mit dem Zählen bei
    # Null zu beginnen", auch wenn das nicht unserer Gewohnheit
    # entspricht: Wenn wir konsequent "mit dem Zählen bei Null
    # beginnen", dann machen wir auch alle Einträge unseres Tupels
    # um 1 kleiner; wir führen diese einfache "Transformation"
    # durch und illustrieren zugleich die "list comprehension",
    # eine sehr nützliche syntaktische Feinheit von Python:
    tup = [x-1 for x in the_tuple]
    # Nun wandeln wir das Resultat in einen numpy-Vektor
    # mit Datentyp int (englisch "integer": "ganze Zahl")
    # um (daß tup nach dieser Umwandlung ein anderer
    # Datentyp ist, ist in Python kein Problem - jedenfalls
    # kein _syntaktisches_ ;-):
    tup = np.array(tup)
    # Das könnte man auch in eine Zeile zusammenfassen:
    # tup = np.array([x-1 for x in the_tuple], dtype=int)

    # Drehe die Liste the_maxima um, hänge vorne die Zahl 1
    # an, und verwandle die Liste in einen numpy-Vektor
    # mit Datentyp int:
    aux = np.array([1]+the_maxima[::-1], dtype=int)
    # Überschreibe den eben erzeugten Vektor mit den
    # "kumulativen Produkten", das ist der Vektor
    # (aux[0], aux[0]*aux[1], aux[0]*aux[1]*aux[2], ...):
    aux = np.cumprod(aux)
    # Überschreibe den eben erzeugten Vektor mit dem
    # "umgedrehten" Vektor (dessen Elemente in umgekehrter
    # Reihenfolge erscheinen), wobei das erste Element
    # weggelassen wird ("Slicing" funktioniert bei numpy-arrays
    # genauso wir bei Python-Listen):
    aux = np.flip(aux)[1:]
    # Das könnte man auch in eine Zeile zusammenfassen:
    # aux=np.flip(np.cumprod(np.array([1]+the_maxima[::-1],dtype=int)))[1:]

    # Mathematisch ist die "Nummer" (wenn wir bei Null zu zählen
    # beginnen) gleich dem _inneren Produkt_ von tup und aux
    # (es lohnt sich nachzudenken, _warum_ das so ist!); da
    # wir aber zunächst bei unserer Gewohnheit bleiben und
    # das Zählen bei Eins beginnen wollen, addieren wir 1:
    return np.inner(tup, aux) + 1
```

(Wenn man die ganzen Kommentare weglässt und Zeilen zusammenfasst, dann hat diese Funktion nur drei Zeilen!)