Collections

March 29, 2021

0.1 Notes

1. The notebooks are largely self-contained, i.e, if you see a symbol there will be an explanation about it at some point in the notebook.
    - Most often there will be links to the cell where the symbols are explained
    - If the symbols are not explained in this notebook, a reference to the appropriate notebook will be provided
2. **Github does a poor job of rendering this notebook**. The online render of this notebook is missing links, symbols, and notations are badly formatted. It is advised that you clone a local copy (or download the notebook) and open it locally.
3. The Numbers notebook may be referred after this notebook for more information on real number, integer, complex number notations.

1 Contents

- Lists or n-tuples
  - Introduction
  - Set of lists
- Aggregation notations
  - Big Sum
  - Big Product
  - Big Union, Intersection, And etc
  - Aggegating over collections
  - Einstein Notations for Tensors

1.1 Importing Libraries

[1]:
```python
import itertools
```

Sets

Introduction

A set is an unordererd collection of objects in which repetition is forbidden. Order does NOT matter. Denoted by

$$\{a, b, c\}$$

Notation: {2,3,7} is a set of three integers

Large sets can be specified using a ellipses to show a continuing pattern: {2,4,6,8, …}

[2]:
```python
set([12,4,21,21])
```

[2]:
```
{4, 12, 21}
```

Some literature introduce concepts using partially ordered sets or Posets. A partial order is any binary relation that is reflexive (each element is comparable to itself), antisymmetric (no two different elements precede each other), and transitive (the start of a chain of precedence relations must precede the end of the chain). A partially oredered set P will be denoted along with its binary relation such as: $(P, \leq)$

Membership

Membership in a set is indicated using $\in$ which is read as 'belongs to' or 'is an element of'. So

$$x \in A$$

can be read as object x is an element of set A.

Note: sometimes the backward $\in$ is used in the same way, such that

$$A \ni x$$

means exactly the same as above. It is sometimes used to represent the phrase 'such that' but its more common to represent 'such that' with **s.t**

[3]:
```python
A = set([5,6,7])
x = 5

x in A
```

[3]: True

The opposite, i.e non-Membership, is indicated using $\notin$ which is read as 'not an element of' or 'does not belog to'. So

$$x \notin A$$

can be read as object x is NOT an element of set A.

[4]:
```python
A = set([5,6,7])
x = 4

x not in A
```

[4]: True

Null set

A set containing no elements is called a null set or an empty set. Denoted most commonly by:

$$\emptyset$$

But it is also generally denoted by

$$O$$

or

$$\varnothing$$

or simply by

$$\{\}$$

[5]:
```python
a = set([])

a == set()
```

[5]: True

Real set

A set containing real numbers is called a real number set. It is represented by:

$$\mathbb{R}$$

So we can represent variable x is a member of the real number set with the representation:

$$x \in \mathbb{R}$$

[6]:
```
R = (float,int)
x = 3.8

type(x) == R[0] or type(x) == R[1]
```

[6]: True

Integer set

A set containing only integers is called an integer set. It is represented by:

$$\mathbb{Z}$$

So we can represent variable x as a member of the integer set with the representation:

$$x \in \mathbb{Z}$$

Nonnegative and negative integers can also be represented, but with additions to this symbol. There are various ways to represent it, commonly used symbols for **non-negative integers** are: $\mathbb{Z}^*, \mathbb{Z}^+, \mathbb{Z}_{\geq 0}$

[7]:
```
Z = int
x = 5

type(x) == Z
```

[7]: True

Complex set

A set containing only complex numbers is called a complex number set. It is represented by:

$$\mathbb{C}$$

So we can represent variable x belonging to the complex number set with the representation:

$$x \in \mathbb{C}$$

For more information, see the Numbers notebook

[8]:
```
C = complex
x = complex(3,5)

print(x.real,x.imag)
type(x) == C
```

```
3.0 5.0
```

[8]: True

Set builders notation

The set builder notation can be used to represent sets in a more consistent manner. The set builder notation will include a couple of statements within curly braces that generally denotes the elements of the built set and it's conditions. For example

$$A = \{x \in \mathbb{Z} : 1 \leq x \leq 100\}$$

This can be read as a set of integers (See: Integer set) where each element is represented using a dummy variable x (i.e all elements of set A are integers). The colon then describes additional conditions for x, which in this case is that x has to be between 1 and 100 **inclusive**.

Sometimes, instead of a colon a pipe can be used . **This DOES not mean conditioning as used in probability notations**. The meaning is the same as above, the colon and pipe are interchangeable from a notation perspective. The above notation is equivalent to:

$$\{x \in \mathbb{Z} \mid 1 \leq x \leq 100\}$$

Sometimes the conditions can be described in simple words. It has the same meaning as the above as well:

$$\{x \in \mathbb{Z} \mid \text{x is greater than or equal to 1 but less than or equal to 100}\}$$

[9]:
```python
Z = int
A = set(range(1,101))

# Let's check if each element fulfills all the conditions
# of the set builder
condition = []
for i in A:
    condition.append( (type(i) == Z) and (1 <= i <= 100))

all(condition)
```

[9]: True

When context is clear we may skip the preamble, or the preamble may be used to just introduce the dummy variable such that:

$$\{x : 1 \leq x \leq 100\}$$

or

$$\{x \mid 1 \leq x \leq 100\}$$

The conditions can be sophisticated and complex at times and can require careful reading to understand the nuances. For example: some of the conditions may actually be expressions that need to be evaluated such as:

$$C = \{(x, y) : x \in A, y \in B, x + y = 6\}$$

Breaking this down, this set builder notation builds a set of ordered pairs of x and y such that x takes values from set A and y takes values from set B and $x + y$ SHOULD sum to 6, . It should be noted that $(x, y) \neq (y, x)$ since the paranthesis suggests that (x, y) is a list. For more on lists, see: introduction to Lists

Although this set builder notation is readable it can also be written using logic notations, using Logical And $\wedge$:

$$C = \{(x, y) : x \in A \wedge y \in B \wedge x + y = 6\}$$

For more information on Logical And, see the Logic notebook

[10]:
```
Z = int
A = set([-2,4,7])
B = set([2,8,9])

all_x_y_combinations = itertools.product(A,B)

print('All (x,y): ', list(itertools.product(A,B)))

C = set([(x,y) for x,y in all_x_y_combinations if (x + y == 6)])

print('\nC: ', C)
```

```
All (x,y):  [(4, 8), (4, 9), (4, 2), (-2, 8), (-2, 9), (-2, 2), (7, 8), (7, 9),
(7, 2)]

C:  {(-2, 8), (4, 2)}
```

Set equality

Two sets A and B are equal if they have the same elements.

$$A = B$$

[11]:
```
A = set([3,5,6,7])
B = set([3,5,6,7])

A == B
```

[11]: `True`

Subset

Consider two sets A and B. A is considered a subset of B if all elements of A are also elements in B. A may also be equal to B.

Denoted by:

$$A \subseteq B$$

Note: Strict subsets can be denoted by $\subset$. This denotes only subset with no equality of the subsets. Consider sets A and B where A is a subset of B but never equal to B:

$$A \subset B$$

However, this isn't universally accepted and some authors use $\subset$ and $\subseteq$ interchangeably

[12]:
```
A = set([3,5,6,7])
B = set([3,5,6,7,8,9,10])
C = set([3,5,6,7])

A.issubset(B), A.issubset(C), A == B, A == C
```

[12]:
```
(True, True, False, True)
```

Note: It should be noted that $A \subseteq B$ is not the same as $A \in B$, mainly because when we talk about subsets we are talking about **sets** A and B. When we talk about membership $x \in B$ we are talking about **elements** in Sets.

[13]:
```
A = set([4])
B = set([4])

print('A is a subset of B: ',A.issubset(B))

print('\nA belongs to B', A in B)

for x in A:
    print('\nElement in A belongs to B: ', x in B)
```

```
A is a subset of B:  True

A belongs to B False

Element in A belongs to B:  True
```

Superset

Consider two sets A and B. A is considered a superset of B if all elements of B are also elements in A. A may also be equal to B.

Denoted by:

$$A \supseteq B$$

Note: Strict supersets can be denoted by $\supset$. This denotes only superset with no equality of the subsets. Consider sets A and B where A is a superset of B but never equal to B:

$$A \supset B$$

However, this isn't universally accepted and some authors use $\supset$ and $\supseteq$ interchangeably

[14]:
```python
A = set([3,5,6,7,8,9,10])
B = set([3,5,6,7])
C = set([3,5,6,7,8,9,10])

A.issuperset(B), A.issuperset(C), A == B, A == C
```

[14]:
```
(True, True, False, True)
```

Union

Consider two sets A and B. The union of A and B are the set of elements that are in both A and B.

Denoted by:

$$A \cup B$$

[15]:
```python
A = set([3,5,6,7,8,9,10])
B = set([3,5,6,        11,12,13,14,15])

A.union(B)
```

[15]:
```
{3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Disjoint Union

Consider two sets A and B. A disjoint union is a single symbol that denotes that A and B DO NOT share common elements and the result is a new set with elements from both A and B.

This may denoted by any of several symbols such as: $\oplus, \uplus, +, \sqcup$ etc and can be an author preference.

For example

$$A_1 \uplus A_2 \uplus ... \uplus A_n = B$$

This means that none of the sets $A_1, A_2, A_3...A_n$ share common elements: i.e

- $A_i \cap A_j = \emptyset \; \forall i \neq j$ (See: For all, Null set, and Intersection)

And that the union of $A_1, A_2, A_3...A_n$ is B - $A_1 \cup A_2 \cup .. \cup A_3 = B$

Note: Such aggegations can be simply denoted using Big notations: see Big Union and Intersection

[16]:
```python
def disjoint_union(A,B):
    """Returns disjoint union if applicable else returns str 'Invalid'"""
    return A.union(B) if A.intersection(B) == set() else 'Invalid'

A = set([      7,8,9,10])
B = set([3,5,6,        11,12,13,14,15])
```

```
disjoint_union(A,B)
```

[16]: {3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

Intersection

Consider two sets A and B. The intersection of A and B are the set of elements that they have in common.

Denoted by:

$$A \cap B$$

[17]:
```
A = set([3,5,6,7,8,9,10])
B = set([3,5,6,        11,12,13,14,15])

A.intersection(B)
```

[17]: {3, 5, 6}

Difference or relative complement

Consider two sets A and B. The difference of A and B is a set of all elements in A that are not in B.

Denoted by:

$$A - B$$

Since this can be ambiguous and may **erroneously** be interpreted as $a - b$ where $a \in A$ and $b \in B$, it can also be denoted as a relative complement:

$$A \setminus B$$

[18]:
```
A = set([3,5,6,7,8,9,10])
B = set([3,5,6,        11,12,13,14,15])

A - B
```

[18]: {7, 8, 9, 10}

Complement

The absolute complement of a set A is all elements that are not in A. This can mean a very large space of elements and is denoted various ways:

$$A^c$$

or

$$\overline{A}$$

or

$$A'$$

Symmetric Difference

Consider two sets A and B, the the symmetric difference of A and B are elements in A or B but not both.

Denoted by:

$$A \Delta B$$

Note that $A \Delta B = (A - B) \cap (B - A)$

[19]:
```
A = set([3,5,6,7,8,9,10])
B = set([3,5,6,        11,12,13,14,15])

A.symmetric_difference(B)
```

[19]:
```
{7, 8, 9, 10, 11, 12, 13, 14, 15}
```

Cardinality

Consider set A, the cardinality is the number of elements in A. Denoted by:

$$\mid A \mid$$

[20]:
```
A = set([3,5,6,7,8,9,10])

cardinality = len(A)

print(cardinality)
```

```
7
```

Cartesian Product

Consider two sets A and B, the cartesian product is the set of all oredered pairs of elements in A and B:

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

Here (a,b) is a list (ordered pair) so $(a, b) \neq (b, a)$. See Lists: Introduction

[21]:
```python
A = set([3,5,6])
B = set([7,8,9])

set(itertools.product(A,B))
```

[21]:
```
{(3, 7), (3, 8), (3, 9), (5, 7), (5, 8), (5, 9), (6, 7), (6, 8), (6, 9)}
```

As an extension of the cartesian product, the notation A^2 is cartesian product of A with A, so it is:

$$A^2 = \{(x, y) : x, y \in A\}$$

[22]:
```python
A = set([3,5,6])

set(itertools.product(A,repeat=2))
```

[22]:
```
{(3, 3), (3, 5), (3, 6), (5, 3), (5, 5), (5, 6), (6, 3), (6, 5), (6, 6)}
```

A further extension of the cartesian product, the notation A^n is the set of all n-tuples of elements of A:

$$n \in \mathbb{Z}_{>0},\ A^n = \{(a_1, a_2, ..., a_n) : a_1, a_2, ..., a_n \in A\}$$

(See: Integer set)

[23]:
```python
A = set([3,5])
n = 3

set(itertools.product(A,repeat=n))
```

[23]:
```
{(3, 3, 3),
 (3, 3, 5),
 (3, 5, 3),
 (3, 5, 5),
 (5, 3, 3),
 (5, 3, 5),
 (5, 5, 3),
 (5, 5, 5)}
```

Power Set

Consider a set A, the power set of A is the set of all subsets of A. Denoted by the Weierstrass symbol:

$$\wp(A)$$

or

$$2^A$$

[24]:
```python
def powerset(A):
    return itertools.chain.from_iterable(itertools.combinations(A, i) for i in␣
 ↪range(len(A)+1))

A = set([2,3,4])

set(powerset(A))
```

[24]:
```
{(), (2,), (2, 3), (2, 3, 4), (2, 4), (3,), (3, 4), (4,)}
```

Set Exponentiation

##Need to add

[ ]:
```python

```

Infimum, Supremum, Max, Min

If A is a set of real numbers, then the largest and smallest numbers are shown as:

$$\max A, \min A$$

However, it is not always possible to evaluate the largest and/or smallest of elements of a set. For example: let $B = \{x \in \mathbb{R}| -1 < x < 1\}$

It can be seen $max B,min B $ **cannot** be evaluated since -1 and 1 do not belong to the set B. In such cases, to evaluate the bounds of B we can instead use the concept of Supremum (also called *least upper bound*) and Infimum (also called *greatest lower bound*). These are denoted as

$$\sup B = 1, \inf B = -1$$

The supremum and infimum are properties **OF** the set B and may not neccesarily be **IN** the set B. It may also be denoted using:

$$\text{lub} B = 1, \text{glb } B = -1$$

[25]:
```python
A = set({2,8,11})

min(A), max(A)
```

[25]:
```
(2, 11)
```

[26]:
```python
B = {-0.9999999, -0.5, 0, 0.5, 0.9999999} #This is a representation but imagine␣
 ↪+-0.9999999999.....
```

```python
print('B: ', B)
print('inf(B): -1, sup(B): 1')
```

```
B:  {-0.9999999, -0.5, 0.9999999, 0, 0.5}
inf(B): -1, sup(B): 1
```

Note 1:

The concept of supremum and infimum applies to all partially ordered sets (posets, See: Introduction to sets).

- All such sets have a supremum and infimum but it may not have a maximum and/or minimum.
- So if the max and min can be defined for a poset A, then $\max A = \sup A$ and $\min A = \inf A$

For example consider the set $\mathbb{R} \cup \{-\infty, \infty\}$ (which is read as all real numbers including plus and minus infinity). This set does not have a max or min but the sup and inf are ∞ and $-\infty$ respectively. This assumes a standard where we recognize infinities as valid sups and infs.

Another example may be

$$A = \{x^{-1} | \, x \in \mathbb{N}\}$$

where $\mathbb{N}$ is the set of natural numbers (**see the Numbers notebook for more details on natural numbers**). In this case it is easy to see for

$$\max A = \sup A = 1, \text{at } x = 1$$

But the minimum is not defined since that will occur at a very large value of x. We do know that it will be bounded by 0 although 0 will never belong in the set A. So

$$\min A = ?, \; \inf A = 0$$

Lists or n-tuples

Introduction

Note: Here we are talking about the mathematical definition of List.

A list is an ordered collection of objects in which repetition is permitted. Lists are denoted using paranthesis:

$$A = (2, 7, 2, 8, 11)$$

Order matters in lists and repetiton is allowed such that $(a_1, a_1, a_2) \neq (a_1, a_2, a_1)$

A list of n elements is sometimes called an $n-$tuple.

[27]:
```python
# A = (a2, a1, a1)
A = [2,1,1]
# B = (a1, a2, a2)
B = [1,2,2]
```

```
A == B
```

[27]: False

[28]:
```
#Equivalently from a mathematics context

# A = (a2, a1, a1)
A = (2,1,1)
# B = (a1, a2, a2)
B = (1,2,2)

A == B
```

[28]: False

When listing 2-tuples using paranthesis, the notation for it: (a, b) may look like an open interval (**See Numbers notebook for more info on intervals**).

To ensure better representation for $n-$tuples and clear distinction between open intervals, some authors also denote it using angle brackets:

$$\langle a, b\rangle$$

This is not limited to 2-tuples, it can be used for n-tuples: $\langle a, b, c, d, e\rangle$ which is equivalent to (a, b, c, d, e) shown above.

Set of lists

A set consisting of ordered pairs is quite commonly used in math literature. This is denoted using set builder notation with an $n-$tuple in the preamble.

$$C = \{(a, b)|\ a \in A, b \in B\}$$

[29]:
```
A = set([2,3,4])
B = set([5,6,7])

C = set(list(itertools.product(A,B)))

print('C: ', C)
print('type of C: ', type(C))
print('\nExample of (a,b) in C: ', C.pop())
print('type of (a,b): ', type(C.pop()))
```

```
C:  {(2, 7), (3, 7), (4, 6), (4, 5), (2, 6), (3, 6), (2, 5), (4, 7), (3, 5)}
type of C:  <class 'set'>
```

```
Example of (a,b) in C:  (2, 7)
type of (a,b):  <class 'tuple'>
```

Aggregation symbols

Big Sum

The symbol $\sum$ is used to represent a sum of a collection. The sum notation is typically used as:

$$\sum_{i=\text{start}}^{stop} \text{expression involving i}$$

For example:

$$\sum_{i=1}^{5} x^i$$

which after explicitely expanding the sum will look like:

$$\sum_{i=1}^{5} x^i = x^1 + x^2 + x^3 + x^4 + x^5$$

[30]:
```python
def sum_x_1_to_5(x):

    sum_var = 0
    for i in range(1,6):
        sum_var += x**i
        print('x: ',x,',i: ',i,',sum: ',sum_var)
    return sum_var

sum_x_1_to_5(3)
```

```
x:  3 ,i:  1 ,sum:  3
x:  3 ,i:  2 ,sum:  12
x:  3 ,i:  3 ,sum:  39
x:  3 ,i:  4 ,sum:  120
x:  3 ,i:  5 ,sum:  363
```

[30]: 363

If the stop condition is ∞, the summation keeps going on without end:

$$F(\infty) = \sum_{n=0}^{\infty} \frac{1}{2^n} = 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ...$$

When computing this sum, it's obviously impossible to keep summing to infinity, the computation will never end. In such cases we can define a tolerance level δ, that is the increase in magnitude of the sum for an iteration reaching which the computation will be terminated.

$$F(i) - F(i-1) \leq \delta$$

In this case

$$F(i) - F(i-1) = \frac{1}{2^i}$$

so the tolerance value for termination will be:

$$\frac{1}{2^i} \leq \delta$$

For more information defining functions F, see the Functions notebook

[31]:
```
def sum_1_by_2_n(tol):

    sum_var = 0
    i = 0
    current_iter = 1000 #Dummy value to begin the algorithm
    while current_iter > tol:
        current_iter = (1/(2**i))
        sum_var += current_iter
        i += 1
        print('Current iteration value: ', current_iter,', tolerance: ',tol,',␣
 ↪sum: ',sum_var)

    return sum_var

sum_1_by_2_n(0.0005)
```

```
Current iteration value:  1.0 , tolerance:  0.0005 , sum:  1.0
Current iteration value:  0.5 , tolerance:  0.0005 , sum:  1.5
Current iteration value:  0.25 , tolerance:  0.0005 , sum:  1.75
Current iteration value:  0.125 , tolerance:  0.0005 , sum:  1.875
Current iteration value:  0.0625 , tolerance:  0.0005 , sum:  1.9375
Current iteration value:  0.03125 , tolerance:  0.0005 , sum:  1.96875
Current iteration value:  0.015625 , tolerance:  0.0005 , sum:  1.984375
Current iteration value:  0.0078125 , tolerance:  0.0005 , sum:  1.9921875
Current iteration value:  0.00390625 , tolerance:  0.0005 , sum:  1.99609375
Current iteration value:  0.001953125 , tolerance:  0.0005 , sum:  1.998046875
Current iteration value:  0.0009765625 , tolerance:  0.0005 , sum:  1.9990234375
Current iteration value:  0.00048828125 , tolerance:  0.0005 , sum:
1.99951171875
```

[31]: 1.99951171875

Big Product

The symbol $\prod$ is used to represent products of a collection and it is analogous to the big sum. The big product notation is typically used as:

$$\prod_{i=\text{start}}^{stop} \text{expression involving i}$$

For example:

$$\sum_{i=0}^{5}(2i+1)$$

which after explicitely expanding the sum will look like:

$$\sum_{i=0}^{5}(2i+1) = 1 \times 3 \times 5 \times 7 \times 9 \times 11$$

[32]:
```
prod = 1

for i in range(6):
    prod *= (2*i+1)
    print('Iteration: ',i,', value: ',(2*i+1),', product: ',prod)

prod
```

```
Iteration:  0 , value:  1 , product:  1
Iteration:  1 , value:  3 , product:  3
Iteration:  2 , value:  5 , product:  15
Iteration:  3 , value:  7 , product:  105
Iteration:  4 , value:  9 , product:  945
Iteration:  5 , value:  11 , product:  10395
```

[32]: 10395

Big Union, Intersection, And etc

In fact, analogous to the Big sum and Big product notation, most other operators may be denoted for aggregation by using the giant form of it with a dummy index.

For example: Assume A_1, A_2, A_3 are sets, then the Big Union can be used to show aggregation of several unions

$$\bigcup_{i=1}^{3} A_i = A_1 \cup A_2 \cup A_3$$

The Big Intersection can be used to show aggregation of several intersections

$$\bigcap_{i=1}^{3} A_i = A_1 \cap A_2 \cap A_3$$

If p_1, p_2, p_3 are Booleans, Big And shows aggregation of several Logical Ands (see Logic notebook)

$$\bigwedge_{i=1}^{3} p_i = p_1 \wedge p_2 \wedge p_3$$

Aggegating over collections

Unitl now all aggegration operations traverses a continious stretch of integers. Sometimes, we want to traverse over other forms of indices. In such cases, we can indicate traversals over sets by showing membership of the index:

$$\sum_{i \in A} i^2$$

This can be expanded explicitely by defining set A, for example consider set $A = \{3, 6, 1.7, 5\}$, then

$$\sum_{i \in A} i^2 = 3^2 + 6^2 + 1.7^2 + 5^2$$

Note: This is true for all the above aggegation notations

[33]:
```
A = set([3,6,1.7,5])

sum_var = 0
for i in A:
    sum_var += i**2
    print('i: ',i,', sum: ',sum_var)

sum_var
```

```
i:  1.7 , sum:  2.8899999999999997
i:  3 , sum:  11.89
i:  5 , sum:  36.89
i:  6 , sum:  72.89
```

[33]: 72.89

Einstein notations

In some contexts, especially in Tensor algebra, it is convenient to write sums without explicitely writing the Summation symbol $\sum$. **The Einstein or Tensor notation has several rules**, but in general a dummy index that is repeated is assumed to be the variable on which the summation is done. The summation is done from 1 to the order of the tensor.

For example consider a $m \times m$ square matrix A and vectors u and v of dimension m. Matrix vector multiplication can then be denoted using Einstein notation by

$$u_i = A_{ij} v_j$$

which is equivalent to:

$$u_1 = \sum_{j=1}^{j=3} A_{1j} v_j,$$

$$u_2 = \sum_{j=1}^{j=3} A_{2j} v_j,$$

$$u_3 = \sum_{j=1}^{j=3} A_{3j} v_j$$

This was a simple example to introduce the concept. Please see the wikipage for more rules: (https://en.wikipedia.org/wiki/Einstein_notation#Matrix-vector_multiplication)

[34]:
```python
# Einstein Notation: u_i = A_ij * v_j

A = [[1,2,3],
     [4,5,6],
     [7,8,9]]

v = [2,2,2]
u = [0,0,0] #initializing

for i in range(len(u)):
    u[i] = sum([ a*v[j] for j,a in enumerate(A[i])])

u
```

[34]:
```
[12, 30, 48]
```