Mathematics with Python

Table of Contents

The idea

To enable users to study some fundamental aspects of maths (sets, relations, functions, numbers and so on) using Python code.

Users can look at the Python code and see how it compares with the mathematical definitions of the ideas involved.

They can also write their own Python code using the classes provided, to check their understanding of these ideas.

The Python code

This is a single Python module (file). It contains a set of relevant classes, and users can add their own code to use them (where it says 'Your code goes here'). The user needs a knowledge of basic Python. See https://python.org.

Execute the code at the command line, in the folder where mython.py is saved, with

python3 mython.py

Or load and run it from an IDE such as Visual Studio or Wings.

Python has a built-in set type. However it is mutable, while mathematical sets are not. We can make new sets from old ones (say, by a union with another set) but we cannot change an existing set. Like we cannot change 28.3. So the code defines and uses a Set class.

Set

This is the fundamental mathematical idea. We say, for example:

A = {4,7,9} meaning A is the set with 3 elements, 4 7 and 9. This is definition by extension – giving a list of the elements. Or

B = {x | x ∈ ℕ and x > 5 } the set of elements x where x ∈ ℕ x is an element of ℕ, the natural numbers, and x is greater than 5. So B = {6,7,8..}. This is an infinite set, defined by intension – giving a rule to decide if something is an element or not.

The elements of a set can be any type. Many examples here are sets of integers, but that is just for simplicity.

Some formal approaches to mathematics use sets as the single fundamental type of object, and then sets can only contain sets, since there is nothing else – everything is a set.

Maths classes are mentioned later.

We can make a set in Python code like:

```
B = Set([1, 2, 4])
print(B)
```

and get output:

```
( 1 2 4  )
```

Mathematical sets can be infinite – such as the set of natural numbers ℕ. These Python sets can only be finite, since the class wraps the elements in a Python list.

We can also supply a name for the set, by

```
B = Set([1, 2, 4], "Set B")
print(B)
```

then we get

```
Set B
```

We can still get the set elements, from the .list field:

```
B = Set([1, 2, 4], "Set B")
print(B.elements)
```

Or slightly more neatly:

```
B = Set([1, 2, 4], "Set B")
print(B.getElements())
```

outputs:

```
Elements of Set B:
1
2
4
```

Sets cannot contain duplicates, so

```
B = Set([1, 2, 1, 2, 4], "Set B")
print(B.list)
```

is also {1,2,4}

We can also make a set by supplying a range and a string expression like this:

```
B = Set(range(1, 6), "2*n")
print(B.elements)
```

producing

```
[2, 4, 6, 8, 10]
```

This is a quick way to produce large sets. The code executes:

```
ℕ = Set(range(0, 1000), "n")
```

to make a set like a finite version of ℕ.

The code starts

```
# coding: utf-8
```

so that we can use Unicode characters like ℕ, ℤ, ℂ and ∩, not just ASCII.

Notation

Unfortunately there is a confusing inconsistency between maths and Python:

Thing	Example	Idea
Maths set	{1,2,3}	A collection of objects – the basis of maths ideas
Python list	[1,2,3]	A linear data structure
Maths ordered pair	(1,2)	Distinct first and second members – because sets are not ordered
Python tuple	(1,2,3)	A read-only linear data structure
Maths tuple	<1,2,3>	A way of talking about something with distinct parts, usually of different type. For example a group is a tuple < S, + > of a set S and a binary operation + with some rules.
Python class		Defines an object type
Maths class		Variation on the 'set' idea

Element of a set

The Set class has a method to find if a set has a given member:

```
B = Set([1, 2, 3, 4])
print(B.hasMember(3)) # True
print(B.hasMember(5)) # False
```

This corresponds to

B = { 1,2,3 }

then

3 ∈ B is true, and

5 ∈ B is false

Equality of sets

Two sets are equal if they contain the same elements.

In other words, everything in the first set must be in the second, and everything in the second must be in the first.

Duplicates are ignored and the order does not matter. For example

```
B = Set([1, 2, 3, 2, 1])
C = Set([3, 2, 1])
print(B == C)) # True
```

Set has a dunder method which is called when == is applied to Set instances:

```
  def __eq__(self, other):
    for _x in self.list:
      if _x not in other.list:
        return False
```

```
    for _x in other.list:
      if _x not in self.list:
        return False
    return True
```

Union of sets

The set of elements in one set or the other:

```
B = Set([1, 2, 3, 4], "B")
C = Set([2, 3, 4, 5], "C")
D = B.union(C)
print(D)  # B ∪ C
print(D.elements)  # [1, 2, 3, 4, 5]
```

The code for union is:

```
  def union(self, other):
    _v = self.elements
    for _x in other.elements:
      if _x not in _v:
        _v.append(_x)
    return Set(_v, str(self) + " ∪ " + str(other))
```

So we start with all the elements in the first set (self.elements), then iterate through the other set by *for _x in other.elements:*, and if we do not already have it *if _x not in _v*, add it to the list by *_v.append(_x)*. Then we make a new set with this list as elements. This avoids getting duplicate elements.

Intersection

This is similar:

```
B = Set([1, 2, 3, 4], "B")
C = Set([2, 3, 4, 5], "C")
D = B.intersection(C)
print(D)  # B ∩ C
print(D.elements)  # [2, 3, 4]
```

The code is:

```
  def intersection(self, other):
    _v = []
    for _x in self.elements:
      if _x in other.elements:
        _v.append(_x)
    return Set(_v, str(self) + " ∩ " + str(other))
```

so we start with an empty list, then iterate through all the elements in the first set list, and if one is also in the other set list, we add it to the list. Then return a new set with this list.

Set difference

The elements of the first set which are not in the other set:

```
B = Set([1, 2, 3, 4], "B")
C = Set([2, 3, 4, 5], "C")
D = B.difference(C)
print(D)  # B - C
print(D.elements)  # [1]
```

The code is

```
  def difference(self, other):
    _v = []
    for _x in self.elements:
      if _x not in other.elements:
        _v.append(_x)
    return Set(_v, str(self) + " - " + str(other))
```

Symmetric difference

Elements which are in either set but not in both:

```
B = Set([1, 2, 3, 4], "B")
C = Set([2, 3, 4, 5], "C")
D = B.symDiff(C)
print(D)  # B Δ C
print(D.elements)  # [1,5]
```

The code is

```
  def symDiff(self, other):
    """ symmetric difference """
    _v = []
    for _x in self.elements:
      if _x not in other.elements:
        _v.append(_x)
    for _x in other.elements:
      if _x not in self.elements:
        _v.append(_x)
    return Set(_v, str(self) + " Δ " + str(other ))
```

Subset

For example

```
B = Set([1, 2, 4], "Set B")
C = Set([0, 1, 2, 3, 4, 5, 6])
print(B.isSubset(C)) # True
```

The code is:

```
def isSubset(self, other):
    for _x in self.elements:
      if _x not in other.elements:
        return False
    return True
```

We iterate through set elements *for _x in self.elements:* and for each one, if its not in the other set *if _x not in other.elements*, we return false, and the method ends. If the loop completes, every element must be in the other set, and we can return true.

Cardinality of a set

The number of elements in a set. The cardinality of set A is written |A|

For example

```
B = Set([1, 2, 4], "Set B")
print(B.card())  # 3
```

The code is

```
  def card(self):
```

```
    return len(self.list)
```

Sets of sets

A set can contain any type of elements. So a set can be made of sets. For example

```
B = Set([1, 2])
C = Set([3, 4])
D = Set([5, 6])
E = Set([B, C, D])

print(E)  # ( ( 1 2  ) ( 3 4  ) ( 5 6  )  )
print(E.card()) # 3
```

Here E has 3 elements. It is different from {1,2,3,4,5,6} which has 6 elements.

Powerset

The powerset is the set of all subsets.

For example:

```
B = Set([1, 2, 3])
print(B.powerSet().getElements())
```

outputs

```
Elements of ( (  ) ( 3  ) ( 2  ) ( 2 3  ) ( 1  ) ( 1 3  ) ( 1 2  ) ( 1 2
3  )  ):
(  )
( 3  )
( 2  )
( 2 3  )
( 1  )
( 1 3  )
( 1 2  )
( 1 2 3  )
```

For each element we can include it in the subset, or not. Suppose, like here, there are 3 elements in the set. Count from 0 to $2^3 - 1$, in binary:

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

We can use the binary form to tell us which elements to include in the subset. So 011 means miss the first element, include the second, include the third.

So the Python code is:

```
  def powerSet(self):
    n = len(self.elements)  # count of elements
    overall = []  # will be collection of all subsets
    for pattern in range(0, 2**n):  # basis for each subset
      # produce string of bits, pattern in base 2, padded with
      # leading 0 for a width of n bits:
      fString = "0" + str(n) + "b"
      bitString = format(pattern, fString)
      thisList = []  # for this subset..
      for index in range(0, len(bitString)):
        c = bitString[index]
        if c == "1":  # add element if bit is 1
          thisList.append(self.elements[index])
      ss = Set(thisList)  # make subset from list
      overall.append(ss)  # add subset to collection

    return Set(overall)  # make Set from collection
```

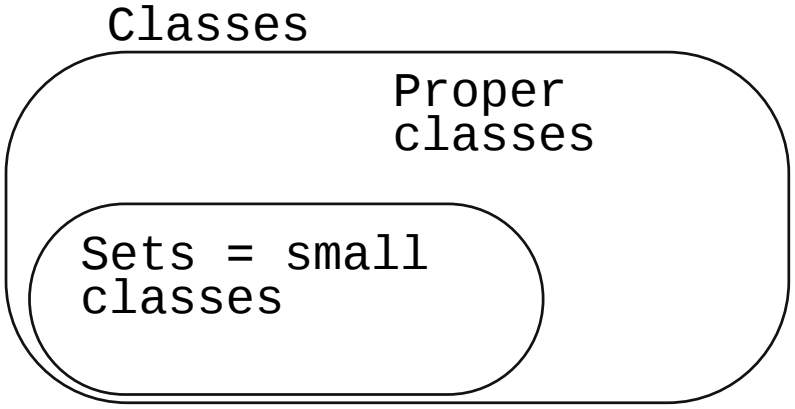
Sets and classes

During the nineteenth century sets became seen as the foundation of mathematics. But some logical paradoxes were found – such as -

The set S of all sets which are not members of themselves.

Is S a member of itself? If it is, its not. And if its not, it is.

The solution was the idea of a class (in the mathematical sense not Python OOP). A class was any collection of things, with no restrictions. Sets were one type of class, and had restrictions – so they could not be member sof themselves, and could not be 'the set of all sets'. They can be infinite, like ℕ, but not bigger than that. Sets are called 'small classes', and classes which are not sets are 'proper classes':



Here we are defining sets by extension so can only handle finite sets, and we do not model proper classes.

Ordered pairs

To define relations (like equal and greater than) and functions we need to have pairs of values.

We cannot just use a set {a,b} as an ordered pair, because sets are not ordered, and {a,b} = {b,a}

We can define an ordered pair (a,b) to be the set { {a,b}, {b}}. Then we can tell the difference between the second element and the first. Some examples

{ {1,2} {2} } is the ordered pair (1,2)

{ {4,2} {4} } is the ordered pair (2,4)

{ {7}, {7,2}} is the ordered pair (2,7)

The second item is in the set with one element. The first item is the other element of the other one.

We could define a Python class for OrderedPair. But we can simply use the Python list [a,b] as the ordered pair (a,b).

Cartesian Product

The Cartesian product of a set with another is the set of all ordered pairs with the first from the first set and the second from the second set.

Symbolically,

$A \times B = \{ x = (a,b) \mid a \in A \land b \in B \}$

For example:

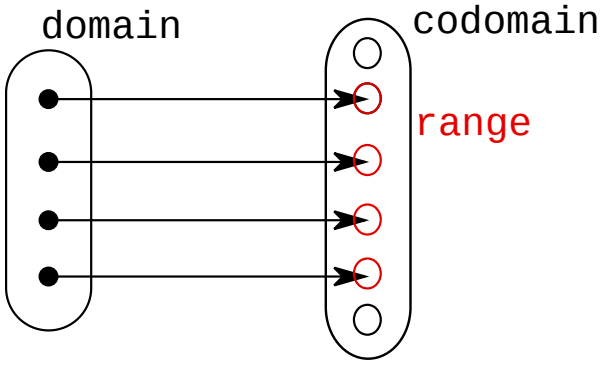
```
B = Set([1, 2])
C = Set([3, 4])
print(B.cartesianProduct(C))   # ( [1, 3] [1, 4] [2, 3] [2, 4]  )
```

The code is

```
  def cartesianProduct(self, other):
    pairs = []
    for el1 in self.elements:
      for el2 in other.elements:
        pairs.append([el1, el2])
    return Set(pairs)
```

Relations

A relation connects one set to another. The 'from' set is the domain, and the 'to' set is the codomain.



Any pairing is possible. One domain element might map to several different codomain elements, or all domain elements could map to one single codomain element.

Each arrow links an ordered pair of elements. In each pair, the first is a domain element, and the second is a codomain.

So a relation is a subset of the Cartesian product of domain and codomain.

The domain and codomain could be the same set.

For example:

```
B = Set([1, 2])
C = Set([3, 4])
D = Relation(B, C, [[1, 3], [2, 4]], "R")
print(D.getElements())
```

Then D is a relation, with domain B and codomain C. 1 maps to 3, and 2 maps to 4. The output is:

```
Elements of Relation R:
[1, 3]
[2, 4]
```

We often write x~y to mean that the pair (x,y) is an element of the relation – in other words, that x~y is true.

We can also create a relation by using a *comprehension* to create the list of pairs.

This is useful if the domain is large and listing all pairs would be tedious. For example

```
B = Set(range(1, 100), "2*n")
C = Set(range(1, 202), "n")
myList = [[x, y] for x in B.elements for y in B.list if y == x + 2]
D = Relation(B, C, myList, "R")
print(D.getElements())
```

Then B={2,4,6..200}

C = {1,2,3..202}

and the output is:

```
Elements of Relation R:
[2, 4]
[4, 6]
[6, 8]
..
[194, 196]
[196, 198]
[198, 200]
```

This is the relation x~y iff y=x+2

Another example of a relation could be

R = { (x,y} | x = y mod 4 }

This uses modular arithmetic. So x and y have the same remainder when divided by 4.

So this could be (0,0) or (0,4) or (0,8).. or (1,5) or (1,9) or (2,6) or (2,10)..

```
D = Set(range(0, 10), "n")
myList = [[x, y] for x in D.elements for y in D.elements if x % 4 == y % 4]
R = Relation(D, D, myList, "R")
print(R.getElements())
```

outputs

```
Elements of Relation R:
[0, 0]
[0, 4]
[0, 8]
[1, 1]
[1, 5]
[1, 9]
..
[8, 0]
[8, 4]
```

```
[8, 8]
[9, 1]
[9, 5]
[9, 9]
```

Range or image

The range of a relation is the codomain elements which are mapped to.

This might be the whole of the codomain, or just a proper subset of it.

The range is sometimes called the image.

For example

```
domain = Set(range(0, 5), "n")
codomain = Set(range(0, 10), "n")
pairs = [[x, y] for x in domain.elements for y in domain.elements if y == x]
r = Relation(domain, codomain, pairs, "r")
print(r.getElements())
```

r is the identity relation : x maps to x. So the elements of r are:

```
Elements of Relation r:
[0, 0]
[1, 1]
[2, 2]
[3, 3]
[4, 4]
```

and the range:

```
print(r.image())
```

is

(0 1 2 3 4)

The code is

```
  def image(self):
    result = []
    for pair in self.elements:
      result.append(pair[1])
    return Set(result)
```

we use *image* because Python already has the keyword *range*

Equivalence relation

This is a type of relation which is a generalisation of the ordinary = for numbers.

So some relations are equivalence relations, while others are not. They might be order relations, for example.

A relation (which we write as ~) is an equivalence relation if it meets three tests, which are chosen because they are how ordinary = works:

1. x ~ x for all x in the domain set. So ~ is *reflexive*

2. if x ~ y, then y~ x. So ~ is *symmetric*

3. if x~y and y~z, then x ~ z. So ~ is *transitive*.

For example

R = { (x,y} | x = y mod 4 }

x and y have the same remainder when divided by 4. So x=4m+k and y=4n+k where $0 \le k \le 3$

then x~x, because x has the same remainder as itself

if x~y then y~x, since if x and y have the same remainder, so do y and x

if x~y and y~z, then x=4m+k, y=4n+k, z=4p+k. So x and z have the same remainder, so x~z.

The Python code confirms this:

```
D = Set(range(0, 10), "n")
myList = [[x, y] for x in D.elements for y in D.elements if x % 4 == y % 4]
R = Relation(D, D, myList, "R")
print(R.isReflexive()) # True
print(R.isSymmetric()) # True
print(R.isTransitive())  # True
print(R.isEquivalence()) # True
```

This works by checking every possibility. For example

```
  def isReflexive(self):
    for x in self.domain.elements:
      if [x, x] not in self.elements:
        return False
    return True
```

Is it a proof?

It depends how 'proof' is defined. It certainly only covers the case for integers 0 to 9, not for $\mathbb{N}$.

Its also wildly inefficient. Transitivity, for example, requires the checking of n^3 cases. It is a brute force approach, totally lacking in elegance. But it is completely direct.

Order relations

An order relation is an abstraction of the 'greater than' relation on ordinary numbers. A relation is an order relation if:

1. x ~ x for all x in the domain set. So ~ is *reflexive*

2. if x ~ y and y~ x. Then x = y. So ~ is *anti-symmetric*

3. if x~y and y~z, then x ~ z. So ~ is *transitive*.

For 2, the idea is that we cannot have both x~y and y~x. Like for ordinary numbers, we cannot have a>b and b>a. So instead of >, we are using ≥, and say that x~y and y~x iff x and y are the same thing.

For example

```
D = Set(range(0, 10), "n")
myList = [[x, y] for x in D.list for y in D.list if x >= y]
R = Relation(D, D, myList, "R")
print(R.isReflexive())  # True
print(R.isSymmetric())  # False
print(R.isTransitive())  # True
print(R.isEquivalence())  # False
print(R.isAntiSymmetric()) # True
```

```
print(R.isOrder())          # True
```

What are the elements of this relation?

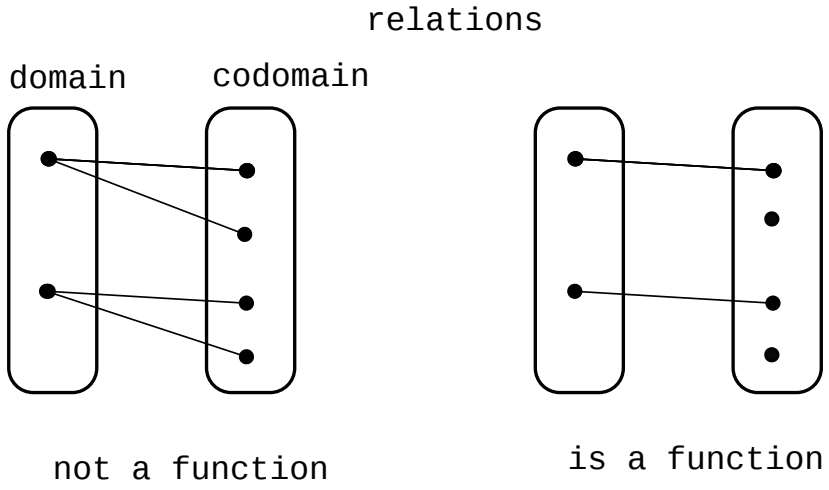
```
Elements of Relation R:
[0, 0]
[1, 0]
[1, 1]
[2, 0]
[2, 1]
[2, 2]
..
[9, 0]
[9, 1]
[9, 2]
[9, 3]
[9, 4]
[9, 5]
[9, 6]
[9, 7]
[9, 8]
[9, 9]
```

each m pairs with all n such that m ≥ n

Functions

Python allows functions to be defined. But that does not make clear what functions are, in terms of sets. So we define our own Function class.

A function is a type of relation in which each domain element maps to only one codomain element:



so we cannot have any two-to-one pairs. Formally:

x~y and x~z implies y=z

The code starts:

```
class Function(Relation):
  def __init__(self, domain, codomain, pairs, name):

    self.domain = domain
    self.codomain = codomain
    if not isinstance(pairs, list):
      raise TypeError("Function error : not a list " + str(pairs))
    for pair in pairs:
      x = pair[0]
      y = pair[1]
```

```
      for others in pairs:
        if others[0] == x:
          if others[1] != y:
            raise TypeError("Function error :" + str(x) + " maps to 2
values")

    self.elements = pairs
    self.name = "Function " + str(name)
```

For example

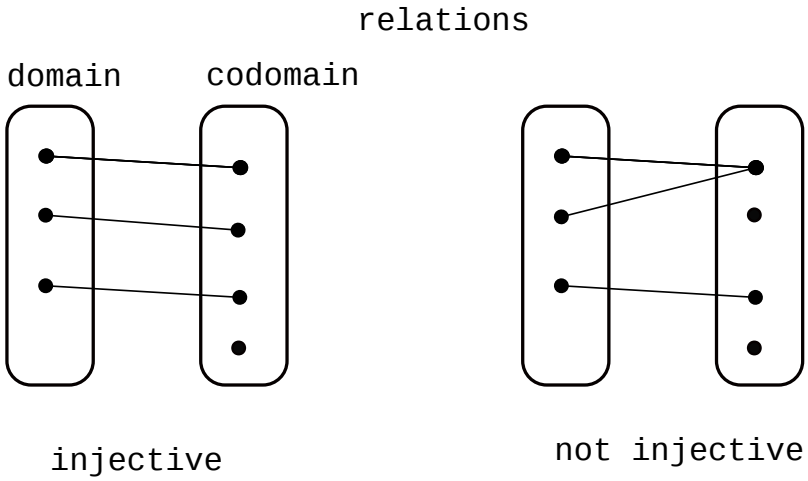
```
domain = Set(range(0, 4), "n")
p = [[0, 1], [1, 2], [2, 0], [3, 3]]
f1 = Function(domain, domain, p, "f1")
print(f1.getElements())
```

outputs:

```
Elements of Function f1:
[0, 1]
[1, 2]
[2, 0]
[3, 3]
```

Injective one-one functions

A function is injective if it is one-one – that is, not two-one or more. A codomain element cannot be mapped to by two different domain elements:



In other words, a function is injective iff

f(a)=f(b) iff a=b

For example

```
domain = Set(range(0, 4), "n")
p = [[0, 1], [1, 2], [2, 2], [3, 3]]
f1 = Function(domain, domain, p, "f1")
print(f1.isInjective())
```

This is not injective because both 1 and 2 map to 2.

The code is:

```
  def isInjective(self):
    for pair in self.elements:
      x = pair[0]
      y = pair[1]
```

```
      for others in self.elements:
        if others[1] == y and others[0] != x:
          return False
    return True
```

Surjective onto functions

A function is surjective if the domain is mapped to the whole of the codomain. So the range is the codomain. The function maps the domain onto the codomain.

For example

```
domain = Set(range(0, 4), "n")
p = [[0, 1], [1, 2], [2, 0], [3, 3]]
f1 = Function(domain, domain, p, "f1")
print(f1.getElements())
print(f1.isSurjective())
```

outputs

```
Elements of Function f1:
[0, 1]
[1, 2]
[2, 0]
[3, 3]

True
```

The code is

```
  def isSurjective(self):
    return self.codomain.equals(self.range())
```

Bijection

A function is a bijection if it is injective and surjective – one-one and onto

Numbers as sets – the natural numbers

Python has number types, as integer and floats. In bytecode they translate to native code numbers which the processor has instructions to handle.

But the idea is to write and use Python code to make clear the mathematical nature of the ideas involved. This includes defining all mathematical things in terms of sets. So we write Python code for numbers.

One such type is the natural numbers 0,1,2,3..

The set of natural numbers is usually defined by the Peano axioms:

1. The set of natural numbers exists (and is usually called $\mathbb{N}$)

2. There exists a function which maps every element of $\mathbb{N}$ to another element. This mapping is usually called the successor function, and we write succ(n). In ordinary terms this is ‘the number after n”, or in other words, n+1

3. There is one element of ℕ which is not the successor of another element. In other words there is one element which does not have an element 'before' it. We call this 0.

So ℕ contains 0. And the successor of 0, which we call 1. And the successor of 1, called 2. And so on.

This version starts at 0. Some people take ℕ to start at 1.

Zermelo numerals

Peano gives 3 *properties* which the natural numbers must have. But it does not say what they actually *are*.

This can be done in several ways. One simple way is the Zermelo numerals, defined as:

0 = the null set, { }

1 is {0}, so its { {} }

2 is {1}, so it is { { {} } }

and in general

succ(n) is { n }

so 0 is the empty set, and the next number is the set with one element - the previous number.

In Python:

```
class NaturalNumber():
  def __init__(self, n):
    self.name = "Zermelo " + str(n)
    if n == 0:
      self.elements = []
    else:
      self.elements = [NaturalNumber(n - 1).elements]

  def succ(self):
    s = NaturalNumber(0)
    s.elements = [self.elements]
    s.name = "Zermelo succ( " + self.name + " )"
    return s
```

For example:

```
zero = NaturalNumber(0)
print(zero.elements)
n = zero.succ()
print(n.elements)
m = n.succ()
print(m.elements)
```

outputs

```
[]
[[]]
[[[]]]
```

so 0 = { }

n = {{}} = { 0 } = 1

m={{{}}} = { 1 } = 2

Peano arithmetic

As well as defining what the natural numbers are, in terms of sets, we also need to define how we do arithmetic with them – in such a way as to match what happens with 'normal' numbers.

We define 'add' like this:

a + 0 = a

a + succ(b) = succ(a+b)

The first form tells us how to add, if the second number is zero.

The second means in effect a+(b+1) = (a+b)+1. In other words we add the smaller number (b) instead of the bigger (b+1). This is a recursive call to +, with the second number smaller. hat will recurse until the second number is zero – when the first form applies.

In Python, we can write a pred method, as the reverse of succ, and code __add__. This is the dunder method used when we say +:

```
class NaturalNumber:
def __init__(self, n):
self.name = "Zermelo " + str(n)
if n == 0:
self.elements = []
else:
self.elements = [NaturalNumber(n - 1).elements]

def succ(self):
..

def pred(self):
if len(self.elements) == 0:
raise TypeError("Zero has no predecessor ")
else:
s = NaturalNumber(0)
s.elements = self.elements[0]
s.name = "Zermelo pred( " + self.name + " )"
return s

def __add__(self, other):
if len(other.elements) == 0:
return self
else:
return self.succ().__add__(other.pred())
```

Then for example:

```
one = NaturalNumber(1)
n = one + one
print(n.elements)
```

outputting

```
[[[]]]
```

In other words 1 + 1 = 2

The integers

The natural numbers are unsigned. In effect, they are all positive.

The signed integers, positive and negative, are a different set, usually called $\mathbb{Z}$, for the German Zahlen.

To represent Z as sets, we need to use ordered pairs, with the pair (a,b) representing a-b, where a and b are in $\mathbb{N}$. Then if b is larger than a, we have negative numbers.

This allows us to define Z in terms of sets.

We can code it in Python using a tuple as the ordered pair:

```
class Z:

  def __init__(self, a, b):
    # check integers
    if not isinstance(a, int):
      raise TypeError("Making z : need integer, got " + str(a))
    if not isinstance(b, int):
      raise TypeError("Making z : need integer, got " + str(b))

    self.pair = (a, b)  # Python tuple

  def __add__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Z(a + c, b + d)

  def __mul__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Z(a * c + b * d, a * d + b * c)

  def __sub__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Z(a - c, b - d)

  def __eq__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return (a - b) == (c - d)

  def __str__(self):
    return str(self.pair[0]) + "," + str(self.pair[1]) + " = " +
str(self.pair[0] - self.pair[1])
```

as :

```
z1 = Z(2, 4)  # = -2
z2 = Z(4, 2)  # = +2
z3 = z1 + z2
print(z3)  # 6,6 = 0
z3 = z1 - z2
print(z3)  # -2,2 = -4
z3 = z1 * z2
print(z3)  # 16,20 = -4
z4 = Z(6, 8) # = -2
print(z1 == z4) # True
```

We code the dunder methods __add__ and so on.

We do not do divide, since this is not closed for Z. For example, -3/4 is not an integer.

Equivalence classes

For example

```
z1 = Z(2, 4)   # = -2
z2 = Z(3, 5)   # = -2
z3 = Z(4, 6)    # = -2
z4 = Z(6, 4)     # = +2
print(z1 == z2) # True
print(z2 == z3) # True
print(z3 == z4) # False
```

(2,4) and (3,5) and (4,6) are all equal to each other.

== is an equivalence relation.

We have an infinite set of integers, { (2,4), (3,5), (4,6), (5,7).. } which are all equivalent. Symbolically

$\forall$ x,y $\in \mathbb{N}$ (x,y) = (x,y+2)

This is called an equivalence class. Any 2 members of { (2,4), (3,5), (4,6), (5,7).. } are equivalent to each other.

We have an infinite set, made of all possible pairs (a,b).

The equivalence relation splits this into some different sets. One such set is { (2,4), (3,5), (4,6), (5,7).. }, where all the pairs are representations of -2. A different set is { (2,5), (3,6), (4,7), (5,8).. }, all of which are representations of -3.

The equivalence relation *partitions* all possible pairs into a set of *equivalence classes*. No pair is in more than one set. No pair is not in one set.

In fact an element of $\mathbb{Z}$ *is* an equivalence class. For example +2 is the set {(2,0), (3,1), (4,2)..}. So =2 is the set { (a+2,a) } for all a $\in \mathbb{N}$.

Our Python code here is not quite correct. It gives us single ordered pairs, rather than the equivalence class, which is what an integer actually is. But that set is infinite, and we can only handle finite sets.

This is the case for all equivalence relations – they partition the underlying set.

Rational numbers

Examples of rational numbers are 1/3 and 7/8 and 12/7 and -43/7.

The set of rational numbers is usually called $\mathbb{Q}$, for quotient.

We can represent a rational number as an ordered pair of integers (which in turn we can represent as pairs of natural numbers, which are themselves sets).

Like this:

```
  class Rational:

  def __init__(self, a, b):
    # check integers
    if not isinstance(a, int):
      raise TypeError("Making rational : need integer, got " + str(a))
    if not isinstance(b, int):
      raise TypeError("Making rational : need integer, got " + str(b))
```

```
    if b == 0:
      raise TypeError("Making rational : got 0 denominator ")

    if b < 0:  # make numerator -ve
      a = -a
      b = -b
    self.pair = (a, b)

  def __add__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Rational(a * d + c * b, b * d)

  def __mul__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Rational(a * c, b * d)

  def __sub__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Rational(a * d - c * b, b * d)

  def __truediv__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    return Rational(c * d, a * b)

  def __eq__(self, other):
    a = self.pair[0]
    b = self.pair[1]
    c = other.pair[0]
    d = other.pair[1]
    # reduce to lowest form
    f = gcd(a, b)
    a = a / f
    b = b / f
    f = gcd(c, d)
    c = c / f
    d = d / f

    return a == c and b == d

  def __str__(self):
    return str(self.pair[0]) + "/" + str(self.pair[1])
```

The dunder methods for add subtract multiply and divide are defined so that we get the 'expected' result. For example

a/b + c/d = (ad+cb) / (cd)

This stores the pair as it is – for example 4/8 is stored as the pair (4,8), not in lowest terms ½.

When checking if two numbers are equal, the obvious way would be to check if a/b = c/d. But this would do floating point arithmetic, which is not exact. So we reduce both to lowest forms, by dividing by their greatest common divisors, then checking a=c and b=d. We find the gcd using Euclid's algorithm:

```
def gcd(a, b):
  """ greatest common divisor by Euclid """
  if a < 0:
    a = -a
  if b < 0:
    b = -b
  while not b == 0:
    t = b
    b = a % b
    a = t
  return a
```

Examples of using the Rational class:

```
z1 = Rational(2, 4)  # 2/4
z2 = Rational(3, 5)  # 3/5
z3 = z1 + z2
print(z3)       # 22/20
z4 = Rational(11, 10)
print(z4 == z3)   # True
```

The == is an equivalence relation, which partitions the set of all pairs. So that for example (1,2), (3,6) and (4,8) are in the same equivalence class.

Binary Operations

A binary operation is an abstraction of the simple idea of adding 2 numbers. Like we take 2 numbers, 3 and 5, add them, and get 8.

Abstracting this we have:

- a set to chose 2 elements of - the natural numbers, or any set
- some way to combine them to get a 3rd element – by adding, or multiplying, or anything, since they might not be numbers (could be functions, for example)
- the new element we have made is also an element of this set – in other words, it is *closed*.

So a binary operation has an underlying set S, and is a function S X S → S

Some people write the operation as + or . We will write it as x◦y, meaning what you get by combining x and y. We use ◦ as a reminder this might not be normal addition.

An example would be S = {0,1,2), with the operation being addition modulo 3. So for example 0◦2=2, and 2◦2 = 1.

Using mython:

```
A = Set([0,1,2])
t=[[x,y,z] for x in A.elements for y in A.elements for z in A.elements if z
== (x+y) % 3 ]
b=BinaryOp(A, t)
print(b)
```

has output

```
0 • 0 = 0
0 • 1 = 1
0 • 2 = 2
1 • 0 = 1
1 • 1 = 2
1 • 2 = 0
2 • 0 = 2
2 • 1 = 0
2 • 2 = 1
```

The code starts:

```
class BinaryOp:
  """ A binary operation """
  def __init__(self, domain, values):
    """ values should be a list of triples [x,y,z] such that
    x•y = z """
    # check arguments
    if not isinstance(domain, Set):
      raise TypeError(str(domain) + "should be a Set")
    if not isinstance(values, list):
      raise TypeError(str(values) + "should be a list")
    for t in values:
      if t[0] not in domain.elements or t[1] not in domain.elements or t[2]
not in domain.elements:
        raise Exception(str(t)+ " value not in domain")

    self.domain=domain
    self.values=values

  def apply(self, x,y):
    """ return x∘y"""
    for triple in self.values:
      if x==triple[0] and y==triple[1]:
        return triple[2]
    raise TypeError("Binary operation - no value for " + str(x)+" and
"+str(y))

  ..
  def __str__(self):
    result=""
    for t in self.values:
      result+=str(t[0])+" • "+str(t[1])+" = "+str(t[2])+"\n"
    return result
```

Cayley table

This is a way to show the results of a binary operation. We have a table, with rows for the first operand, columns for the second, and cell entries for the result.

For example on {0,1,2) with addition mod 3

	0	*1*	*2*
0	0	1	2
1	1	2	0
2	2	0	1

so 1 • 2 =0

A binary operation might match some formula, but it can be arbitrary such as

	0	*1*	*2*
0	0	0	2
1	0	1	2
2	2	2	1

using mython this is:

```
A = Set([0,1,2])
t=[[0,0,0], [0,1,0], [0,2,2], [1,0,0], [1,1,1], [1,2,2], [2,0,2], [2,1,1],
[2,2,1] ]
b=BinaryOp(A, t)
print(b)
```

outputting

0 • 0 = 0

0 • 1 = 0

```
0 • 2 = 2
1 • 0 = 0
1 • 1 = 1
1 • 2 = 2
2 • 0 = 2
2 • 1 = 1
2 • 2 = 1
```

Identity element

A binary operation might have an element which leaves all elements unchanged – like adding 0 to a number.

If the identity is e, then

e•x = x•e = x for all x in S.

For example, for the last one.

```
A = Set([0,1,2])
t=[[0,0,0], [0,1,0], .. ]
b=BinaryOp(A, t)
print(b.identity()) # 1
```

In the Cayley table above, the rows and columns for 1 ar eteh same as the row and column headings.

The code for identity is

```
  def identity(self):
    """ Returns left identity, or None"""
    for x in self.domain.elements:
      ok=True
      for y in self.domain.elements:
        if self.apply(x,y)!=y:
          ok=False
      if ok:
        return x
    return None
```

The left identity e_1 means

$e_1 • x = x$

and the right identity e_2 is

$x • e_2 = x$

‘The identity’ means if the left and right identities are the same.

Inverse

The idea of this is like -x for addition. If we combine x and -x we get 0, the identity:

x + -x = 0

So if there is an identity e, the inverse y of an element x is such that

x • = y • x = e

For our add mod 3 operation, the inverse of 2 is 1:

```
A = Set([0,1,2])
t=[[x,y,z] for x in A.elements for y in A.elements for z in A.elements if z
== (x+y) % 3 ]
b=BinaryOp(A, t)
print(b.inverse(2)) # 1
```

Here the identity is 0, and 2 • 1 = 1 • 2 = 0

For this table:

	0	*1*	*2*
0	0	0	2
1	0	1	2
2	2	2	1

the identity is 1, but 0 does not have an inverse. In the 0 row, there is no 1, so there is no y such that

0 • y = 1

The code for inverse is

```
  def inverse(self,x):
    i=self.identity()
    if i==None:
      raise Exception("Finding inverse, but there is no identity " +
str(x))
    for y in self.domain.elements:
      if self.apply(x,y)==i:
        return y
```

Associativity

An operation is associative if

(x • y) • z = x • (y • z)

for all x y and x.

For example, + is associative, as

(1+2)+3 = 3+3 = 6

1+(2+3) = 1 +5 = 6

But subtraction is not
(3-2)-1 = 1-1 = 0

3-(2-1) = 3-1 = 2

Add mod 3 is associative:

```
A = Set([0,1,2])
t=[[x,y,z] for x in A.elements for y in A.elements for z in A.elements if z
== (x+y) % 3 ]
b=BinaryOp(A, t)
print(b.isAssoc()) # True
```

The code is

```
  def isAssoc(self):
    """ Return true iff this is associative """
    for x in self.domain.elements:
      for y in self.domain.elements:
        for z in self.domain.elements:
          a=self.apply(self.apply(x,y),z)
          b=self.apply(x, self.apply(y,z))
          if not a==b:
            return False
    return True
```

As usual this is a ‘brute force’ approach. It checks for every x y and z using 3 nested loops.