lecture_2_numerical_ii

March 26, 2024

1 Lecture 2: Numerical Basics: Part II

1.1 Data Science Fundamentals

1.2 Representation of numbers

##### DSF - University of Glasgow - Chris McCaig - 2023/2024

1.3 Summary

By the end of this unit you should know:

1.4 Arithmetic, broadcasting and aggregation

1.5 Floating point numbers

[ ]:
```
##### JUST IN CASE YOU STILL NEED TO INSTALL ANY OF THESE

#!pip install -U --no-cache https://github.com/johnhw/jhwutils/zipball/master

#!pip install -U scikit-image
#!pip install sympy
#!pip install statsmodels
```

[ ]:
```
try:
    import sympy
    sympy.init_printing(use_latex='png')
except:
    sympy = False

try:
    from Tkinter import *
except ImportError:
    from tk import *
```

[2]:
```
# MAKE SURE YOU RUN THIS CELL! IF IT GIVES AN ERROR FIRST TIME RUN IT AGAIN,␣
 ↪WITHOUT RESTARTING THE KERNEL
```

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt

from jhwutils.matrices import show_boxed_tensor_latex
from jhwutils.matrices import print_matrix
from jhwutils.float_inspector import print_float_html

from jhwutils.image_audio import (
    play_sound,
    show_image,
    load_image_colour,
    load_image_gray,
    show_image_mpl,
    load_sound
)


%matplotlib inline
plt.rc("figure", figsize=(7, 3.5), dpi=140)
```

2 Map: arithmetic on arrays

The major advantage of array representations is to be able to do arithmetic on arrays directly.

Basic arithmetic is computed **elementwise**. This means that a function is applied to each element of an array. There are a few different kind of element wise operations:

- single argument, like `np.tan()` or unary negative `(-x)`
- two argument, like `x+y` or `x-1` or `np.maximum(x,y)`
- and various other cases, like np.where(condition, true_values, false_values)‘

All of these work on arrays without any special syntax. We can simply write expressions using array variables.

```
x + y + 2  # if x and y are arrays, this just works
```

[3]:
```
x = np.array([1, 2, 3, 4])
y = np.array([0, 1, 2, 3])
print_matrix("x", x)
print_matrix("y", y)
```

```
x
 [[1 2 3 4]]
y
 [[0 1 2 3]]
```

[4]:
```
print_matrix("x+y", x+y)
print_matrix("x-y", x-y)
print_matrix("x*y", x*y)
print_matrix("x/y", x/y)
print_matrix("x^y", x**y)
```

```
x+y
 [[1 3 5 7]]
x-y
 [[1 1 1 1]]
x*y
 [[ 0  2  6 12]]
x/y
 [[ inf 2.   1.5  1.33]]
x^y
 [[ 1  2  9 64]]
```

```
/tmp/ipykernel_15072/2865631761.py:4: RuntimeWarning: divide by zero encountered
in divide
  print_matrix("x/y", x/y)
```

[5]:
```
# examples with scalars and arrays
print_matrix("x+1", x + 1)
print_matrix("2x", x * 2)
print_matrix("1/x", 1 / x)
print_matrix("x^2", x**2)
```

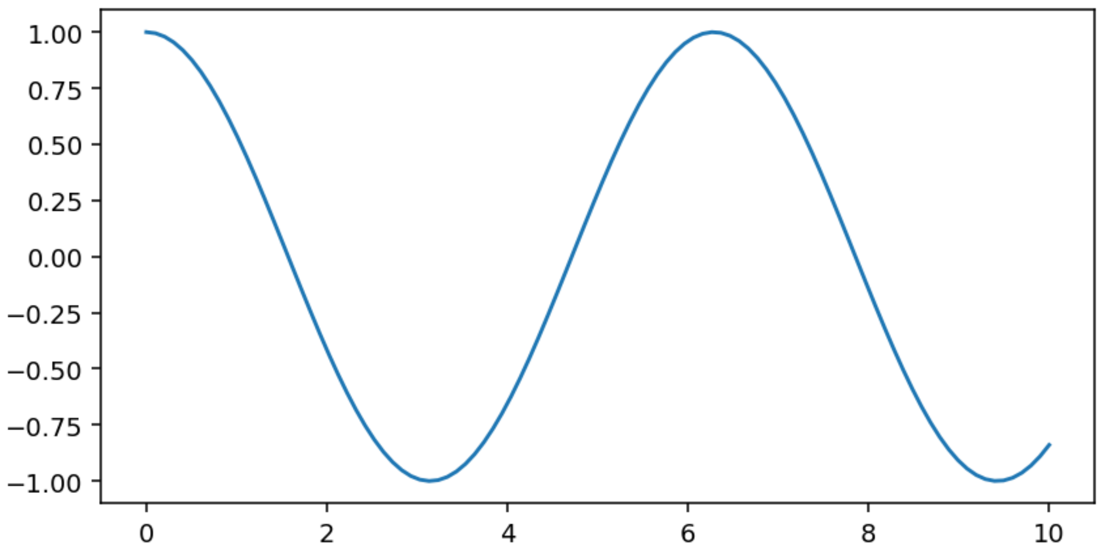
```
x+1
 [[2 3 4 5]]
2x
 [[2 4 6 8]]
1/x
 [[1.   0.5  0.33 0.25]]
x^2
 [[ 1  4  9 16]]
```

2.0.1 Applying simple functions

Standard functions, like `cos` or `tan` can be applied to arrays:

[6]:
```
x = np.linspace(0,10,100)
plt.plot(x, np.cos(x))
```

[6]:
```
[<matplotlib.lines.Line2D at 0x7f4959a73c10>]
```



3 An example: changing volume

If a sound is just an array of values, then we can use array operations to apply changes to a whole sound at once.

For example, scaling (multiplying the values) will change the volume.

[ ]:
```
guitar = load_sound("sounds/guitar.wav")
play_sound(guitar)
```

[ ]:
```
play_sound(guitar*0.2)
```

[ ]:
```
play_sound(np.tanh(guitar*1200)*0.15) # distortion pedal
```

[ ]:
```
play_sound(np.convolve(guitar, np.ones(50)/50)) # bass only
```

[ ]:
```
play_sound(np.convolve(guitar, np.random.normal(0,1, 15000) * np.exp(-np.
 ↪linspace(0, 5, 15000))) * 0.02) # reverb
```

If we wanted to fade out the sound, we'd need to multiply *each element* by a different value (e.g. fading from 0.0 to 1.0).

[ ]:
```
fade = np.linspace(1,0, len(guitar))**2
play_sound(guitar*fade)
```

3.1 Mixing sounds

Mixing sounds simply involves adding them (and possibly reducing the gain)

```
[ ]: sax = load_sound("sounds/erhu.wav")
     play_sound(sax)
```

```
[ ]: max_len = min(len(guitar), len(sax))
     # nb: here I slice so that both are the length of the shortest sound
     play_sound(sax[:max_len] + guitar[:max_len])
```

```
[ ]: play_sound(guitar[:max_len]*fade[:max_len]+
                sax[:max_len]*(1-fade[:max_len]))
```

3.2 Map

This is a special case of a **map**: the application of a function to each element of a sequence.

There are certain rules which dictate what operations can be applied together.

- For single argument operations, there is no problem; the operation is applied to each element of the array
- If there are two or more arguments, like in `x + y`, then `x` and `y` must have **compatible shapes**. This means it must be possible to pair each element of `x` with a corresponding element of `y`

3.2.1 Same shape

In the simplest case, `x` and `y` have the same shape; then the operation is applied to each pair of elements from `x` and `y` in sequence.

3.2.2 Not the same shape

If `x` and `y` aren't the same shape, it might seem like they cannot be added (or divided, or "maximumed"). However, NumPy provides **broadcasting rules** to allow arrays to be automatically expanded to allow operations between certain shapes of arrays.

3.2.3 Repeat until they match

The rule is simple; if the arrays don't match in size, but one array can be *tiled* to be the same size as the other, this tiling is done implicitly as the operation occurs. For example, adding a scalar to an array implicitly *tiles* the scalar to the size of the array, then adds the two arrays together (this is done much more efficiently internally than explicitly generating the array).

The easiest broadcasting rule is scalar arithmetic: `x+1` is valid for any array `x`, because NumPy **broadcasts** the 1 to make it the same shape as `x` and then adds them together, so that every element of `x` is paired with a 1.

Broadcasting always works for any scalar and any array, because a scalar can be repeated however many times necessary to make the operation work.

You can imagine that `x+1` is really `x + np.tile(1, x.shape)` which works the same, but is much less efficient:

[7]:
```
x = np.zeros((5, 5))
x + 1
```

[7]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

[8]:
```
# same, but creates large temporary array
x + np.tile(1, x.shape)
```

[8]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

3.3 Broadcasting

So far we have seen: * **elementwise array arithmetic** (both sides of an operator have exactly the same shape) and * **scalar arithmetic** (one side of the operator is a scalar, and the other is an array).

This is part of a general pattern, which lets us very compactly write operations between arrays of different sizes, under some specific restrictions.

Broadcasting is the way in which arithmetic operations are done on arrays when the operands are of different shapes.

1. If the operands have the same number of dimensions, then they **must** have the same shape; operations are done elementwise. `y = x + x`
2. If one operand is an array with fewer dimensions than the other, then if the *last dimensions* of the first array match the shape as the second array, operations are well-defined. If we have a LHS of size (…,j,k,l) and a RHS of (l) or (k,l) or (j,k,l) etc., then everything is OK.

This says for example that:

```
shape (2,2) * shape(2,) -> valid
shape (2,3,4) * shape(3,4) -> valid
shape (2,3,4) * shape(4,) -> valid

shape (2,3,4) * shape (2,4) -> invalid
shape (2,3,4) * shape(2) --> invalid
shape (2,3,4) * shape(8) --> invalid
```

Broadcasting is just automatic tiling When broadcasting, the array is *repeated* or tiling as needed to expand to the correct size, then the operation is applied. So adding a (2,3) array and a (3,) array means repeating the (3,) array into 2 identical rows, then adding to the (2,3) array.

```
[9]: vec4 = np.array([1, 2, 3, 4])
     mat4 = np.zeros((4, 4))  # 4x4 zeros
```

```
[10]: mat3x4 = np.full((3, 4), 8.0)  # 3x4 filled with 8
      vec3 = np.array([1, 2, 3])
      vec4x = np.array([1, 1, 1, 1])
```

```
[11]: print((vec4+1))        # scalar (Rule 2)
      print((vec4 + vec4x))  # elementwise (Rule 1)
```

```
[2 3 4 5]
[2 3 4 5]
```

```
[12]: print((mat4 + mat4))   # elementwise (Rule 1)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
[13]: # note that vec4 is repeated over the rows to make it the same size
      print((mat4 + vec4))   # broadcasting: valid because RHS (vec4) has dimension␣
       ↪matching the last dimension of matt4
```

```
[[1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]]
```

```
[14]: # note that the operation operates across *columns*, i.e the last dimension of␣
       ↪the array
      print((mat3x4 + vec4)) # broadcasting
```

```
[[ 9. 10. 11. 12.]
 [ 9. 10. 11. 12.]
 [ 9. 10. 11. 12.]]
```

```
[15]: # broadcasting also works on comparisons
      mat4x = np.array([[1,2,3,4],
                        [4,5,6,7],
                        [8,9,10,11],
                        [12,13,14,15]])
      print((mat4x>vec4))
      # note this has compared [1,2,3,4] to each row of mat4x
```

```
[[False False False False]
 [ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
```

3.3.1 Invalid broadcasting examples

```
[16]: mat3x4 = np.array([[1,5,8,2],
                 [0,2,4,1],
                 [3,3,6,2]])

      print_matrix("mat3x4",mat3x4)
      print_matrix("vec3",vec3)

      print_matrix()

      print((mat3x4 + vec3)) # invalid: last dimensions don't match!
```

```
mat3x4
 [[1 5 8 2]
 [0 2 4 1]
 [3 3 6 2]]
vec3
 [[1 2 3]]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 8
      5 print_matrix("mat3x4",mat3x4)
      6 print_matrix("vec3",vec3)
----> 8 print_matrix()
     10 print((mat3x4 + vec3)) # invalid: last dimensions don't match!

TypeError: print_matrix() missing 2 required positional arguments: 'name' and␣
  ↪'matrix'
```

```
[17]: print((mat4+vec3)) # invalid: last dimensions don't match
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[17], line 1
----> 1 print((mat4+vec3)) # invalid: last dimensions don't match

ValueError: operands could not be broadcast together with shapes (4,4) (3,)
```

```
[18]: print((mat4+mat3x4)) # invalid: arrays have same rank but different shape
```

```
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
Cell In[18], line 1
----> 1 print((mat4+mat3x4)) # invalid: arrays have same rank but different shape

ValueError: operands could not be broadcast together with shapes (4,4) (3,4)
```

3.4 Transposing in broadcasts

Transpose solves one of the problems you might have seen with broadcasting. Imagine we want to add a vector to every row of a matrix. This is easy:

[19]:
```
x = np.zeros((4,3)) # 4 rows, 3 columns
y = np.array([1,1,9]) # 3 element vector, applies to each row
print_matrix("x",x )
print_matrix("y",y)
# this will repeat the 3 element vector into 4 rows, then add
print_matrix("x+y",x+y)
```

```
x
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
y
 [[1 1 9]]
x+y
 [[1. 1. 9.]
 [1. 1. 9.]
 [1. 1. 9.]
 [1. 1. 9.]]
```

But how would we add a vector to each *column*? This would need a 4 element vector, and this cannot be added directly as it violates the broadcasting rules

[20]:
```
z = np.array([1,2,3,4])
print_matrix("x+z", x+z) # this can't work; a 4x3 and a 4 don't have matching␣
 ↪last dimensions
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[20], line 2
      1 z = np.array([1,2,3,4])
----> 2 print_matrix("x+z", x+z) # this can't work; a 4x3 and a 4 don't have␣
 ↪matching last dimensions

ValueError: operands could not be broadcast together with shapes (4,3) (4,)
```

But the **transpose** of `x` is a 3x4 matrix, to which `z` can be added. The result is transposed, so we transpose it back:

```
[21]: print_matrix("(x^T)", (x.T))
```

```
(x^T)
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
[22]: print_matrix("(x^T+z^T)", (x.T + z))
```

```
(x^T+z^T)
 [[1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]]
```

```
[23]: print_matrix("(x^T+z^T)^T", (x.T + z).T)
```

```
(x^T+z^T)^T
 [[1. 1. 1.]
 [2. 2. 2.]
 [3. 3. 3.]
 [4. 4. 4.]]
```

4 Reduction

Reduction (sometimes called **fold**) is the process of applying an operator or function with two arguments repeatedly to some sequence.

For example, if we reduce [1,2,3,4] with `+`, the result is `1+2+3+4 = 10`. If we reduce `[1,2,3,4]` with `*`, the result is `1*2*3*4 = 24`.

Reduction: stick an operator in between elements

```
1 2 3 4
5 6 7 8
```

Reduce on columns with "+":

```
1 + 2 + 3 + 4  =  10
5 + 6 + 7 + 8  =  26
```

Reduce on rows with "+":

```
1 2 3 4
+ + + +
5 6 7 8


=
6 8 10 12
```

Reduce on rows then columns:

```
1 + 2 + 3 + 4
+   +   +   +
5 + 6 + 7 + 8

=
6 + 8 + 10 + 12  = 36
```

Many operations can be expressed as reductions. These are **aggregate** operations.

`np.any` and `np.all` test if an array of Boolean values is all True or not all False (i.e. if any element is True). These are one kind of **aggregate function** – a function that processes an array and returns a single value which "summarises" the array in some way.

- `np.any` is the reduction with logical OR
- `np.all` is the reduction with logical AND
- `np.min` is the reduction with min(a,b)
- `np.max` is the reduction with max(a,b)
- `np.sum` is the reduction with +
- `np.prod` is the reduction with *

[24]:
```python
print("any", np.any([True, False, False]))  # true = True or False or False
print("all", np.all([True, False, False]))  # false = True and False and False
```

```
any True
all False
```

[25]:
```python
x = np.array([1, 2, 3, 4, 5, 6])  # 1 + 2 + 3 + 4 + 5 + 6
print(np.sum(x))
```

```
21
```

[26]:
```python
print(np.prod(x))  # 1 * 2 * 3 * 4 * 5 * 6 = 6!
```

```
720
```

[27]:
```python
print(np.max(x))  # max(max(max(max(max(1,2), 3), 4), 5), 6)
```

```
6
```

Some functions are built on top of reductions: * `np.mean` is the sum divided by the number of elements reduced * `np.std` computes the standard deviation using the mean, then some elementwise arithmetic

[28]:
```python
print(np.mean(x))
print(np.sum(x) / len(x))  # equivalent
```

```
3.5
3.5
```

By default, aggregate functions operate over the whole array, regardless of how many dimensions it has. This means reducing over the last axis, then reducing over the second last axis, and so on, until a single scalar remains. For example, `np.max(x)`, if `x` is a 2D array, will compute the reduction across columns and get the max for each row, then reduce over rows to get the max over the whole array.

We can specify the specific axes to reduce on using the `axes=` argument to any function that reduces.

[29]:
```python
x = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print_matrix("x", x)
print("max(x)=", np.max(x))  # reduce on all axes
print_matrix("max_{0}(x)", np.max(x, axis=0))  # reduce on rows
print_matrix("max_{1}(x)", np.max(x, axis=1))  # reduce on columns
print_matrix("max_{0,1}(x)",
             np.max(x, axis=(0, 1)))  # same as all axes in this case
```

```
x
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
max(x)= 9
max_{0}(x)
 [[7 8 9]]
max_{1}(x)
 [[3 6 9]]
```

$max_{0,1}(x) = 9$

[30]:
```python
print_matrix("\\text{mean}_0(x)", np.mean(x, axis=0))  # mean on rows
print_matrix("\\text{mean}_1(x)", np.mean(x, axis=1))  # mean on columns
```

```
\text{mean}_0(x)
 [[4. 5. 6.]]
\text{mean}_1(x)
 [[2. 5. 8.]]
```

5 Accumulation

The sum of an array is a single scalar value. The **cumulative sum** or **running sum** of an array is an array of the same size, which stores the result of summing up every element until that point.

This is almost the same as reduction, but we keep intermediate values during the computation, instead of collapsing to just the final result. The general process is called **accumulation** and it can be used with different operators.

For example, the accumulation of `[1,2,3,4]` with `+` is `[1, 1+2, 1+2+3, 1+2+3+4] = [1,3,6,10]`.

- `np.cumsum` is the accumulation of `+`

- `np.cumprod` is the accumulation of `*`
- `np.diff` is the accumulation of `-` (but note that it has one less output than input)

Accumulations operate on a single axis at a time, and you should specify this if you are using them on an array with more than one dimension (otherwise you will get the accumulation of flattened array).

[31]:
```
print_matrix("x", x)
print_matrix("\\text{cumsum}_0(x)",
             np.cumsum(x, axis=0)) # sum across rows
print_matrix("\\text{cumprod}_1(x)",
             np.cumprod(x, axis=1)) # product across columns
print_matrix("\\text{diff}_0(x)",
             np.diff(x, axis=0)) # difference across rows
print_matrix("\\text{diff}_1(x)", np.diff(x, axis=1)) # difference across␣
 ↪columns
```

```
x
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
\text{cumsum}_0(x)
 [[ 1  2  3]
 [ 5  7  9]
 [12 15 18]]
\text{cumprod}_1(x)
 [[  1   2   6]
 [  4  20 120]
 [  7  56 504]]
\text{diff}_0(x)
 [[3 3 3]
 [3 3 3]]
\text{diff}_1(x)
 [[1 1]
 [1 1]
 [1 1]]
```

- `np.gradient` is like `np.diff` but uses central differences to get same length output, and it computes the gradient over *every* axis and returns them all in a list. It is a very useful function in image processing.

[32]:
```
print_matrix("\\nabla x_0", np.gradient(x)[0])
print_matrix("\\nabla x_1", np.gradient(x)[1])
z = np.array([1,3,8,10,18])
print_matrix("gradient(z**2)",np.gradient(z**2))
print_matrix("gradient(z**2)",z**2)
print_matrix("diff(z**2)",np.diff(z**2))
np.gradient(z)
```

```
\nabla x_0
 [[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
\nabla x_1
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
gradient(z**2)
 [[  8.    31.5   45.5 130.   224. ]]
gradient(z**2)
 [[  1    9   64 100 324]]
diff(z**2)
 [[  8   55   36 224]]
```

[32]:
```
array([2. , 3.5, 3.5, 5. , 8. ])
```

5.1 Finding

There are functions which find **indices** that satisfy criteria. For example, the largest value along some axis.

- `np.argmax()` finds the index of the largest element
- `np.argmin()` finds the index of the smallest element
- `np.argsort()` finds the indices that would sort the array back into order
- `np.nonzero()` finds indices that are non-zero (or True, for Boolean arrays)

Finding indices is of great importance, because it allows us to cross-reference across axes or arrays. For example, we can find the row where some value is maximised (most wheat production) and then find the attribute which corresponds to it (the year when most wheat was produced).

[35]:
```
x = np.array([5,9,0,13,-8,7,2,8,0,-8])
print_matrix("x", x)
# note: argmin/max will tie break on the first occurence
print_matrix("argmin(x)", np.argmin(x))
print_matrix("argmax(x)", np.argmax(x))
```

```
x
 [[ 5  9  0 13 -8  7  2  8  0 -8]]
```

$argmin(x) = 4$

$argmax(x) = 3$

[36]:
```
print("nonzero(x)=", np.nonzero(x))

## argmin is almost the same as this
## but this can return *multiple* minimums, instead of the first
print(np.nonzero(x==np.min(x)))
```

```
nonzero(x)= (array([0, 1, 3, 4, 5, 6, 7, 9]),)
(array([4, 9]),)
```

5.2 Argsorting

argsort finds the indices that would put an array in order. It is an *extremely* useful operation.

[37]:
```
# get the indices that would put x into order
print_matrix('x', x)
print_matrix('\\text{argsort}(x)', np.argsort(x))
```

```
x
 [[ 5  9  0 13 -8  7  2  8  0 -8]]
\text{argsort}(x)
 [[4 9 2 8 6 0 5 7 1 3]]
```

[38]:
```
# hey presto! sorted!
print_matrix('x[\\text{argsort}(x)]', x[np.argsort(x)])
```

```
x[\text{argsort}(x)]
 [[-8 -8  0  0  2  5  7  8  9 13]]
```

---

6 Part II: Floating point

What we are going to do: **understand how floating point ndarrays work, right down to the bits in memory**

6.1 Algebra: a loss of structure

The **algebraic properties** of operators on real numbers (associativity, distributivity, and commutativity) are *not* preserved with the representation of numbers that we use for computations. The approximations used to store these numbers efficiently for computational purposes means that:

$$ab \neq ba, a + b \neq b + a, \text{etc.}$$

$$a(b + c) \neq ab + bc$$

$$a(bc) \neq (ab)c$$

Of course, most of the time, a good representation will be very close to these properties and they will almost hold, or at least hold for many specific examples. *But they are not preserved in general.*

[39]:
```
print(1e120 + 1e-120 + -1e120)
```

```
0.0
```

[40]:
```
print(1e120 + -1e120 + 1e-120)
```

```
1e-120
```

7 Number types

There are different representations for numbers that can be stored in arrays. Of these, **integers** and **floating point numbers** are of most relevance. Integers are familiar in their operation, but floats have some subtlety.

7.1 Integers

Integers represent whole numbers (no fractional part). They come in two varieties: signed and unsigned. In memory, these are (normally!) stored as binary 2's complement (for signed) or unsigned binary (for unsigned).

Most 64 bit systems support operations on at least the following integer types:

name	bytes	min	max
int8	1	-128	127
uint8	1	0	255
int16	2	-32,768	32,767
uint16	2	0	65,535
int32	4	-2,147,483,648	2,147,483,647
uint32	4	0	4,294,967,295
int64	8	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807
uint64	8	0	18,446,744,073,709,551,615

An operation which exceeds the bounds of the type results in **overflow**. Overflows have behaviour that may be undefined; for example adding 8 to the int8 value 120 (exceeds 127; result might be 127, or -128, or some other number). In most systems you will ever see, the result will be to wrap around, computing the operation modulo the range of the integer type.

NumPy allows integer arrays, although we won't use them extensively:

[41]:
```
int_array = np.array([100, 110, 120], dtype=np.int8)

print(int_array)
print(int_array + 20)  # beware!
```

```
[100 110 120]
[ 120 -126 -116]
```

[42]:
```
uint_array = np.array([100, 110, 120], dtype=np.uint8)
print(uint_array)

print(uint_array + 20)  # ok
```

```
[100 110 120]
[120 130 140]
```

[43]:
```
print(uint_array + 150)  # wrap, but through 0 this time
```

```
[250   4  14]
```

8 Floats

-*The other kind of floats* by *siaronj* shared *CC BY*

8.1 Floating point representation

Integers have (very) limited range, and don't represent fractional parts of numbers. Floating point is the most common representation for numbers that may be very large or small, and where fractional parts are required. Most modern computing hardware supports floating point numbers directly in hardware.

(side note: floating point isn't **required** for fractional representation; *fixed point* notation can also be used, but is much less flexible in terms of available range. It is often faster on simple hardware like microcontrollers; however modern CPUs and GPUs have extremely fast floating point units)

The die of the 8087, the first Intel hardware floating point unit. From http://www.righto.com/2018/09/two-bits-per-transistor-high-density.html

Unlike integers, floating point numbers have some surprising properties which can cause **numerical issues**.

8.1.1 A number in [1.0, 2.0) and a shift

All floating point numbers are is a compact way to represent numbers of very large range, by allowing a fractional number with a standardised range (*mantissa*, varies from 1.0 to just less than 2.0) with a scaling or stretching factor (*exponent*, varies in steps of powers of 2).

Floating point numbers can be thought of as numbers between 1.0 and 2.0, that can be shifted and stretched by doubling or halving repeatedly

The advantage of this is that the for a relatively small number of digits, a very large range of numbers can be represented. The precision, however, is variable, with very precise representation of small numbers (close to zero) and coarser representation of numbers far from 0.

8.1.2 Sign, exponent, mantissa

A floating point number is represented by three parts, each of which is in practice an integer. Just like scientific notation, the number is separated into an exponent (the magnitude of a number) and a mantissa (the fractional part of a number).

Scientific notation

For example, in scientific notation, `5340.2` is writen `5.3402 * 10^3` (or `5.3402e3`). Likewise, `0.00051` is written `5.1 * 10^(-4)` or `5.1e-4`. There is always *exactly* one digit before the decimal point; the "shift" to put the decimal in the right place is written in the exponent portion.

```
  5.3402     * 10 ^ 3
  [mantissa] * 10 ^ [exponent]
```

The advantage of this is that for a relatively small number of digits, a very large range of numbers can be represented. The precision, however, is variable, with very precise representation of small numbers (close to zero) and coarser representation of numbers far from 0.

Binary floating point In binary floating point, calculations are done base 2, and every number is split into three parts. These parts are:

- the **sign**; a single bit indicating if a number is positive or negative
- the **exponent**; a signed integer indicating how much to "shift" the mantissa by
- the **mantissa**; an unsigned integer representing the fractional part of the number, following the 1.

A floating point number is equal to:

```
sign * (1.[mantissa]) * (2^exponent)
```

The leading one Note that a leading 1 is inserted before the mantissa; this is because it is unnecessary to represent the first digit, as we know the mantissa represents a number between 1.0 (inclusive) and 2.0 (exclusive). Instead, the leading one is *implicitly* present in all computations.

The mantissa is always a positive number, stored as an integer such that it would be shifted until the first digit was just after the decimal point. So a mantissa which was stored as a 23 digit binary integer $00100111010001001000101_2$ would really represent the number $1.00100111010001001000101_2$ The exponent is stored as a positive integer, with an implied "offset" to allow it to represent negative numbers.

For example, in `float32`, the format is:

```
1     8      23
sign  exp.   mantisssa
```

The exponents in `float32` are stored with an implied offset of -127 (the "bias"), so exponent=0 really means exponent=-127.

So if we had a `float32` number

```
  1 10000011 00100111010001001000101
```

What do we know?

1. The number is negative, because leading bit (sign bit) is 1.
2. The mantissa represents $1.00100111010001001000101_2 = 1.153389573097229_{10}$
3. The exponent represents $2^{131-127} = 2^4 = 16$ ($1000011_2 = 131_{10}$), because of the implied offset.

So the number which is represented can be computed as follows:

```
[44]: sign = "1"
      exponent = "10000011"
      mantissa = "00100111010001001000101"
```

```python
exponent = int(exponent, 2) - 127  # compensate for bias
mantissa = 1.0 + int(mantissa, 2) / 2 ** len(mantissa)  # convert to 1.xxxx␣
↪format

print("exponent (decimal, including bias)", exponent)
print("mantissa (decimal, including leading 1)", mantissa)

if sign == "1":
    number = -1 * mantissa * 2 ** exponent
else:
    number = mantissa * 2 ** exponent

print("number", number)
```

```
exponent (decimal, including bias) 4
mantissa (decimal, including leading 1) 1.153389573097229
number -18.454233169555664
```

8.1.3 IEEE 754

The dominant standard for floating point numbers is IEEE754, which specifies both a representation for floating point numbers and operations defined upon them, along with a set of conventions for "special" numbers.

The IEEE754 standard types are given below:

Name	Common name	Base	Digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias	E min	E max	Notes
binary16	Half precision	2	11	3.31	5	4.51	2^4−1 = 15	−14	+15	not basic
binary32	Single precision	2	24	7.22	8	38.23	2^7−1 = 127	−126	+127	
binary64	Double precision	2	53	15.95	11	307.95	2^10−1 = 1023	−1022	+1023	
binary128	Quadruple precision	2	113	34.02	15	4931.77	2^14−1 = 16383	−16382	+16383	
binary256	Octuple precision	2	237	71.34	19	78913.2	2^18−1 = 262143	−262142	+262143	not basic

Floats, doubles Almost all floating point computations are either done in **single precision** (**float32**, sometimes just called "float") or **double precision** (**float64**, sometimes just called "double").

float32 **float32** is 32 bits, or 4 bytes per number; **float64** is 64 bits or 8 bytes per number.

GPUs typically are fastest (by a long way) using **float32**, but can do double precision **float64** computations at some significant cost.

float64 **float64** is 64 bits, or 8 bytes per number. Most desktop CPUs (e.g. x86) have specialised **float64** hardware (or for x86 slightly odd 80-bit "long double" representations).

Exotic floating point numbers Some GPUs can do very fast **float16** operations, but this is an unusual format outside of some specialised machine learning applications, where precision isn't critical. (there is even various kinds of **float8** used occasionally).

float128 and **float256** are very rare outside of astronomical simulations where tiny errors matter and scales are very large. For example, JPL's ephemeris of the solar system is computed using `float128`. Software support for `float128` or `float256` is relatively rare. NumPy does not support `float128` or `float256`, for example (it seems like it does, but it doesn't).

IEEE 754 also specifies **floating-point decimal** formats that are rarely used outside of specialised applications, like some calculators.

8.2 Binary representation of floats

We can take any float and look at its representation in memory, where it will be a fixed length sequence of bits (e.g. float64 = 64 bits). This can be split up into the sign, exponent and mantissa. Let's look at some examples:

[45]:
```
print_float_html(1.0)
```

```
<IPython.core.display.HTML object>
```

[46]:
```
print_float_html(4.0)
```

```
<IPython.core.display.HTML object>
```

[47]:
```
print_float_html(5.0)
```

```
<IPython.core.display.HTML object>
```

[48]:
```
print_float_html(0.25)
```

```
<IPython.core.display.HTML object>
```

[49]:
```
print_float_html(1.0 / 3.0)
```

```
<IPython.core.display.HTML object>
```

[50]:
```
print_float_html(2000000.01)
```

```
<IPython.core.display.HTML object>
```

[51]:
```
print_float_html(6.02e23)  # Avogadro's number (approximately)
```

```
<IPython.core.display.HTML object>
```

```
[52]: print_float_html(1e-90)
```

```
<IPython.core.display.HTML object>
```

```
[53]: print_float_html(1.5e300)
```

```
<IPython.core.display.HTML object>
```

8.2.1 Integers in floats

For **float64**, every integer from -2^{53} to 2^{53} is precisely representable; integers outside of this range are not represented exactly (this is because the mantissa is effectively 53 bits, including the implicit leading 1).

```
[54]: print("Float\t", 1.0 * 2 ** 53 - 1)

      print("Integer\t", 2 ** 53 - 1)  # exactly the same
```

```
Float    9007199254740991.0
Integer  9007199254740991
```

```
[55]: print("Float\t", 1.0 * 2 ** 53 + 1)
      print("Integer\t", 2 ** 53 + 1)  # not the same!
```

```
Float    9007199254740992.0
Integer  9007199254740993
```

```
[56]: print(f"Float\t {1.0 * 7 ** 33 + 1:.8f}")
      print("Integer\t", 7 ** 33 + 1)  # very different!
```

```
Float    7730993719707444064475414528.00000000
Integer  7730993719707444524137094408
```

8.3 Special features of floats

As well as their huge range, floats have some special properties that are critical for numerical computations.

8.3.1 Float exceptions

Float operations, unlike integers, can cause *exceptions* to happen during calculations. These exceptions occur at the *hardware* level, not in the operating system or language. The OS/language can configure how to respond to them (for example, Unix systems send the signal SIGFPE to the process which can handle it how it wishes).

There are five standard floating point exceptions that can occur.

- **Invalid Operation** Occurs when an operation without a defined real number result is attempted, like 0.0 / 0.0 or sqrt(-1.0).

- **Division by Zero** Occurs when dividing by zero.
- **Overflow** Occurs if the result of a computation exceeds the limits of the floating point number (e.g. a `float64` operations results in a number > 1e308)
- **Underflow** Occurs if the result of a computation is smaller than the smallest representable number, and so is rounded off to zero.
- **Inexact** Occurs if a computation will produce an inexact result due to rounding.

Each exception can be **trapped** or **untrapped**. An untrapped exception will not halt execution, and will instead do some default operation (e.g. untrapped divide by zero will output infinity instead of halting). A trapped exception will cause the process to be signaled in to indicate that the operation is problematic, at which point it can either halt or take another action.

Usually, `invalid operation` is trapped, and `inexact` and `underflow` are not trapped. `overflow` and `division by zero` may or may not be trapped. NumPy traps all except `inexact`, but normally just prints a warning and continues; it can be configured to halt and raise an exception instead.

[63]:
```
np.array(0.0) / np.array(0.0)  # invalid operation (results in nan)
```

```
/tmp/ipykernel_15072/3186114846.py:1: RuntimeWarning: invalid value encountered
in divide
  np.array(0.0) / np.array(0.0)  # invalid operation (results in nan)
```

[63]:
```
nan
```

[64]:
```
np.array(1.0) / np.array(0.0)  # divide by zero
```

```
/tmp/ipykernel_15072/424328097.py:1: RuntimeWarning: divide by zero encountered
in divide
  np.array(1.0) / np.array(0.0)  # divide by zero
```

[64]:
```
inf
```

[65]:
```
np.array(2.0) / np.array(3.0)  # inexact (not trapped by NumPy)
```

[65]:
```
0.6666666666666666
```

[66]:
```
np.array(100.0) * np.array(1e307)  # overflow (results in inf)
```

```
/tmp/ipykernel_15072/4094892618.py:1: RuntimeWarning: overflow encountered in
multiply
  np.array(100.0) * np.array(1e307)  # overflow (results in inf)
```

[66]:
```
inf
```

[67]:
```
np.array(0.000001) * np.array(
    1e-307
)  # underflow (results in smallest possible float, by default)
```

```
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
Cell In[67], line 1
----> 1 np.array(0.000001) * np.array(
      2     1e-307
      3 )  # underflow (results in smallest possible float, by default)

FloatingPointError: underflow encountered in multiply
```

[68]:
```
np.seterr(under="raise")  # enable underflow trap
np.array(000000.1) * np.array(
    1e-307
)  # underflow (results in smallest possible float, by default)
```

```
---------------------------------------------------------------------------
FloatingPointError                        Traceback (most recent call last)
Cell In[68], line 2
      1 np.seterr(under="raise")  # enable underflow trap
----> 2 np.array(000000.1) * np.array(
      3     1e-307
      4 )  # underflow (results in smallest possible float, by default)

FloatingPointError: underflow encountered in multiply
```

8.4 Special numbers: zero, inf and NaN

8.4.1 Zero: +0.0 and -0.0

IEEE 754 has both positive and negative zero representations. Positive zero has zero sign, exponent and mantissa. Negative zero has the sign bit set.

Positive and negative 0.0 compare equal, and work exactly the same in all operations, except for the sign bit propagating.

[69]:
```
print_float_html(+0.0)  # this is an all zero bit pattern
```

```
<IPython.core.display.HTML object>
```

[70]:
```
print_float_html(-0.0)  # this has the sign bit set
```

```
<IPython.core.display.HTML object>
```

[71]:
```
0.0 == -0.0
```

[71]:
```
True
```

```
[72]: -0.0 < 0.0
```

```
[72]: False
```

```
[73]: print_float_html(2 * -0.0)  # sign bit propagates
```

```
<IPython.core.display.HTML object>
```

```
[74]: print_float_html(0.0 * 2)  # sign bit propagates
```

```
<IPython.core.display.HTML object>
```

```
[75]: print_float_html(-0.0 ** 2) # sign propagates (even here!)
```

```
<IPython.core.display.HTML object>
```

8.4.2 Infinity: $+\infty$ and -∞

IEEE 754 floating point numbers **explicitly** encode infinities. They do this using a bit pattern of all ones for the exponent, and all zeros for the mantissa. The sign bit indicates whether the number is positive or negative.

```
[76]: # np.inf is a constant equal to infinity
      print_float_html(np.inf)  # positive infinity
```

```
<IPython.core.display.HTML object>
```

```
[77]: print_float_html(-np.inf)  # negative infinity
```

```
<IPython.core.display.HTML object>
```

```
[78]: np.inf + 1  # infinity + anything = infinity
```

```
[78]: inf
```

```
[79]: np.inf * 2  # same with other operations
```

```
[79]: inf
```

```
[80]: np.inf * 0  # but infinity * 0  is undefined
```

```
[80]: nan
```

```
[81]: np.inf - np.inf # not 0!
```

```
[81]: nan
```

```
[82]: np.inf / np.inf  # as is infinity / infinity
```

[82]: nan

[83]:
```
print_float_html(-0.0 * np.inf)  # infinity has a sign
```

```
<IPython.core.display.HTML object>
```

8.4.3 NaN:

NaN or **Not A Number** is a particularly important special "number". NaN is used to represent values that are invalid; for example, the result of 0.0 / 0.0. All of the following result in NaN:

- `0 / 0`
- `inf / inf` (either positive or negative inf)
- `inf - inf` or `inf + -inf`
- `inf * 0` or `0 * -inf`
- `sqrt(x)`, if x<0
- `log(x)`, if x<0
- Any other operation that would have performed any of these calculations internally

NaN has several properties:

- it **propagates**: any floating point operation involving NaN has the output NaN. (almost: `1.0**nan==1.0`).
- any comparison with NaN evaluates to false. NaN is not equal to anything, **including itself**; nor is it greater than or lesser than any other number. It is the only floating point number not equal to itself.
- NaN, however, is *not* equivalent to False in Python

It is both used as the *output of operations* (to indicate where something has gone wrong), and deliberately as a *placeholder in arrays* (e.g. to signal missing data in a dataset).

NaN has all ones exponent, but non-zero mantissa. Note that this means there is *not* a unique bit pattern for NaN. There are $2^{52}-1$ different NaNs in `float64` for example, all of which behave the same.

[84]:
```
# np.nan is a constant equal to nan
print_float_html(np.nan)
```

```
<IPython.core.display.HTML object>
```

[85]:
```
print(np.nan * 5)  # nan propagates
```

```
nan
```

[86]:
```
print(np.sin(np.nan))  # every operation involving nan evaluates to nan
```

```
nan
```

[87]:
```
print(np.nan > 5)  # comparisons are always false
```

```
False
```

[88]:
```
print(np.nan == np.nan)  # comparisons are always false, even equality to␣
 ↪itself!
```

```
False
```

[89]:
```
if np.nan:
    print("NaN is truthy")
else:
    print("NaN is falsey")
```

```
NaN is truthy
```

`np.isnan(a)` tests if an array has `nan` in it (and is exactly the same as `a!=a`, because of NaN's comparison property)

[90]:
```
num = np.array([1, 0, 1])
den = np.array([1, 0, 0])

quot = num / den
print(quot)
```

```
[ 1. nan inf]
```

```
/tmp/ipykernel_15072/1832728363.py:4: RuntimeWarning: divide by zero encountered
in divide
  quot = num / den
/tmp/ipykernel_15072/1832728363.py:4: RuntimeWarning: invalid value encountered
in divide
  quot = num / den
```

[91]:
```
print(np.isnan(quot))
```

```
[False  True False]
```

[92]:
```
print(quot != quot)
```

```
[False  True False]
```

NaN as a result It is very common experience to write some numerical code, and discover that the result is just NaN. This is because NaN propagates – once it "infects" some numerical process, it will spread to all future calculations. This makes sense, since NaN indicates that no useful operation can be done. However, it can be a frustrating experience to debug NaN sources.

The most common cause is **underflow** rounding a number to 0 or **overflow** rounding a number to +/-`inf`, which then gets used in one of the "blacklisted" operations.

[93]:
```
x = np.random.normal(1, 0.4, 60)
print(np.cumsum(x * np.log(x)))
```

```
[ 0.05621192  0.03883596 -0.29886705 -0.66578084 -1.03147253 -0.9093631
 -0.48905585 -0.53487304 -0.23650001 -0.44443551 -0.16830321  0.34965261
  0.63117679  0.26331022  0.34221248 -0.0174366   0.99249012  1.57497258
  1.32271166  1.20896875  1.03098587  0.75724594  0.54710687  0.96160156
  0.7013951   0.66114096  0.55864035  0.96550771  1.45373773  1.59439563
  2.14634875  1.79668898  1.4676386   1.29185955  1.3095174          nan
         nan         nan         nan         nan         nan         nan
         nan         nan         nan         nan         nan         nan
         nan         nan         nan         nan         nan         nan
         nan         nan         nan         nan         nan         nan]
```

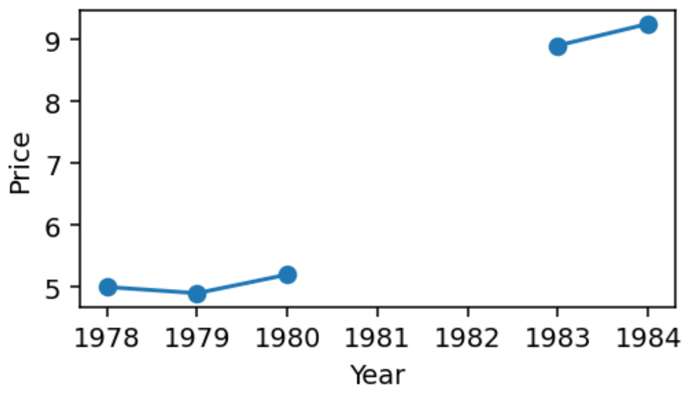
```
/tmp/ipykernel_15072/2266186294.py:2: RuntimeWarning: invalid value encountered
in log
  print(np.cumsum(x * np.log(x)))
```

NaN as a mask Sometimes NaN is used to mask parts of arrays that have missing data. While there is specialised support for masked arrays in some languages/packages, NaNs are available everywhere and don't require any special storage or data structures.

For example, plotting data with NaN's in it results in gaps:

[94]:
```
plt.rc("figure", figsize=(4, 2), dpi=140)
year = [1978, 1979, 1980, 1981, 1982, 1983, 1984]
price = [5.0, 4.9, 5.2, np.nan, np.nan, 8.9, 9.25]  # no data for 1981 or 1982
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(year, price, "-o")
ax.set_xlabel("Year")
ax.set_ylabel("Price")
```

[94]:
```
Text(0, 0.5, 'Price')
```



9 Resources for this lecture

- **From Python to Numpy** http://www.labri.fr/perso/nrougier/from-python-to-numpy/ *recommended reading*

```
[ ]:
```