Part I

Object oriented programming with classes

Datatypes and Classes

A datatype, e.g. a `list` binds data and related methods together:

```
my_list= [1,2,3,4,-1]
my_list.<tab> #  show a list of all related methods (
                                in ipython only)
my_list.sort()  # is such a method
my_list.reverse() # .. another method
```

In this unit we show how own dataypes and related methods can be created.
For example

- polynomials
- triangles
- special problems (find a zero)
- etc.

A minimalistic example

```
class Nix:
    pass
```

An object with a certain datatype is said to be an *instance* of that type:

```
a=Nix()
```

Check:

```
if isinstance(a,Nix):
    print('Indeed it belongs to the new class Nix')
```

An example

Let us define a new datatype for rational numbers:

```
class RationalNumber:
    def __init__(self,numerator,denominator):
        self.numerator=numerator              # attribute
        self.denominator=denominator          # attribute
```

It got a method which initializes an instance with two attributes:

```
q=RationalNumber(10,20)      # Defines a new object
q.numerator                  # returns 10
q.denominator                # returns 20
```

__init__ and self

What happens, when

```
q=RationalNumber(10,20)      # Defines a new object
```

is executed?

- a new object with name q is created
- the command q.__init__(10,20) is executed.

self is a placeholder for the name of the newly created instance (here: q)

Adding methods

Methods are functions bound to the class:

```
class RationalNumber:
...
  def convert2float(self):
    return float(self.numerator)/float(self.
                                    denominator)
```

and its use ...

```
q=RationalNumber(10,20)     # Defines a new object
q.convert2float() # returns 0.5
```

Note, again the special role of `self` !

Adding methods (Cont.)

Note:
Both commands are equivalent:

```
RationalNumber.convert2float(q)
q.convert2float()
```

Adding method (Cont.)

A second method ....

```
class RationalNumber:
...
  def add(self, other):
    p1, q1 = self.numerator, self.denominator
    if isinstance(other, RationalNumber):
      p2, q2 =  other.numerator, other.denominator
    elif isinstance(other, int):
      p2, q2 = other, 1
    else:
      raise TypeError('...')
    return RationalNumber(p1 * q2 + p2 * q1, q1 * q2)
```

A call to this method takes the following form

```
q = RationalNumber(1, 2)
p = RationalNumber(1, 3)
q.add(p)    # returns the RationalNumber for 5/6
```

Special methods: Operators

We would like to add RationalNumber number instances just by

```
q + p
```

.

Renaming the method

```
RationalNumber.add
```

to

```
RationalNumber.__add__
```

makes this possible.

Special methods: Operators (Cont.)

Operator	Method	Operator	Method
+	__add__	+=	__iadd__
*	__mul__	*=	__imul__
-	__sub__	-=	__isub__
/	__truediv__	/=	__idiv__
**	__pow__		
==	__eq__	!=	__ne__
<=	__le__	<	__lt__
>=	__ge__	>	__gt__
()	__call__	[ ]	__getitem__

Special methods: Representation

`__repr__` defines how the object is represented, when just typing its name:

```
class RationalNumber:
 ...
  def __repr__(self):
    return f"{self.numerator} / {self.denominator}"
```

Now: q returns 10 / 20.

Reverse operations

So far we can use the class for operations like

$$1/5 + 5/6 \text{ or } 1/5 + 5$$

but $5 + 1/5$ requires that the integer's method `__add__` knows about `RationaNumber`.

Instead of extending the methods of `int` we use reverse operations:

```
class RationalNumber(object)
    ....
     def __radd__(self, other):
         return self + other
```

`__radd__` reverses the role of self and other.

Check the operation $1 + q$

Extending the example

We add a method `shorten` to our example:

```
class RationalNumber:
    ....
    def shorten(self):
        def gcd(a, b):
            # Computes the greatest common divisor
            if b == 0:
                return a
            else:
                return gcd(b, a % b)
        factor = gcd(self.numerator, self.denominator)
        self.numerator = self.numerator // factor
        self.denominator = self.denominator // factor
```

Note, it performs an *inplace* modification of the object.

Attributes depending on each other – A guiding example

Let us assume the following class to define a triangle

```
class Triangle:
    def __init__(self, A, B, C):
        self.A = array(A)
        self.B = array(B)
        self.C = array(C)
        self.a = self.C - self.B
        self.b = self.C - self.A
        self.c = self.B - self.A
    def area(self):
        return abs(cross(self.b, self.c))/2
```

An instance of this triangle is created by

```
tr = Triangle([0.,0.], [1.,0.], [0.,1.])
```

and its area is computed by

```
tr.area() # returns 0.5
```

Altering an attribute

Altering an attribute,

```
tr.B = array([1.0, 0.1])
```

does not affect the attribute 'area':

```
tr.area() # still 0.5
```

We have to be able to run "internally" some commands when one attribute is changed in order to modify the others!

For this end we introduce attributes only for internal use, e.g _B and ...

Altering an attribute (Cont.)

... and modify our class

```
class Triangle:
    def __init__(self,A, B, C):
        self._A = array(A)
        self._B = array(B)
        self._C = array(C)
        self._a = self._C - self._B
        self._b = self._C - self._A
        self._c = self._B - self._A
    def area(self):
        return abs(cross(self._c, self._b)) / 2.
    def set_B(self,B):
        self._B = B
        self._a = self._C - self._B
        self._c = self._B - self._A
    def get_B(self):
        return self._B
    B=property(fget = get_B, fset = set_B)
```

What would happen if we change `self._B` to `self.B`?

A triangle has three corners

Python allows us to delete an attribute

```
del tr.B
```

If so, we won't have a triangle any more. So let's prohibit deletion

```
class Triangle:
    ...
    def del_Pt(self):
        raise Exception('A triangle point cannot be
                                        deleted')
    B = property(fget = get_B,
                 fset = set_B,
                 fdel = del_Pt)
```

```
del tr.B # now raises an exception
```

Inheritance - a mathematical concept

Mathematics builds on class inheritance principles:

Example of a mathematical inheritance hierarchy

- *Functions*
  can be evaluated, differentiated, integrated, summed up, ...
- *Polynomials* are functions
  same methods as functions, change representation, truncate, ...
- *Trigonometric polynomials* are polynomials
  same methods as polynomials, ask for real parts, ...

We say *polynomials inherit properties and methods from functions*.

Inheritance - in Python

```
class ODESolver:
    def __init__(self, f, y0, t0, tend, Nsteps=100):
        self.f = f
        self.y0 = y0
        self.interval = [t0, tend]
        self.Nsteps = Nsteps
        self.t = np.linspace(t0, tend, Nsteps+1)
        self.h = (tend-t0)/Nsteps

    def phi(self, tn, yn):
        raise NotImplementedError

    def solve(self):
        t = self.t
        y = np.zeros_like(t)
        y[0] = self.y0
        for n in range(self.Nsteps):
            y[n+1] = y[n] + self.h*self.phi(t[n],y[n])
        return t, y
```

Inheritance - in Python (cont.)

... construct from it two subclasses

```
class Euler(ODESolver):
    def phi(self, t, y):
        return self.f(t,y)

class Heun(ODESolver):
    def phi(self, t, y):
        f = self.f
        h = self.h
        u = y + h*f(t,y)
        return (f(t,y)+f(t+h,u))/2
```

Inheritance - in Python (cont.)

Which we then call

```
def f(t, y):
    return -0.5*y

euler = Euler(f, 15., 0., 10., 20)
euler.solve()

heun = Heun(f, 15., 0., 10., 20)
heun.solve()
```

Part II

More on Lists and other Container Types

Creating sublists

Slicing – `L[i:j]`

Slicing a list between index i and j is forming a new list by taking elements with indices starting at i and ending *just before* j.

Alternatively ...

`L[i:j]` means: create a list by removing the first i elements from L and keeping the next $j - i$ elements (for $j > i \geq 0$) and removing the rest.

Example

```
L = ['C', 'l', 'o', 'u', 'd']
L[1:4] # remove one element and take three from there:
# ['l', 'o', 'u']
```

Partial slicing

One may omit the first or last bound of the slicing:

```
L = ['C', 'l', 'o', 'u', 'd']
L[1:] # ['l', 'o', 'u', 'd']
L[:3] # ['C', 'l', 'o']
L[-2:] # ['u', 'd']     # negative indexing !
L[:-2] # ['C', 'l', 'o']
L[:] # the entire list
```

Mathematical Analogy

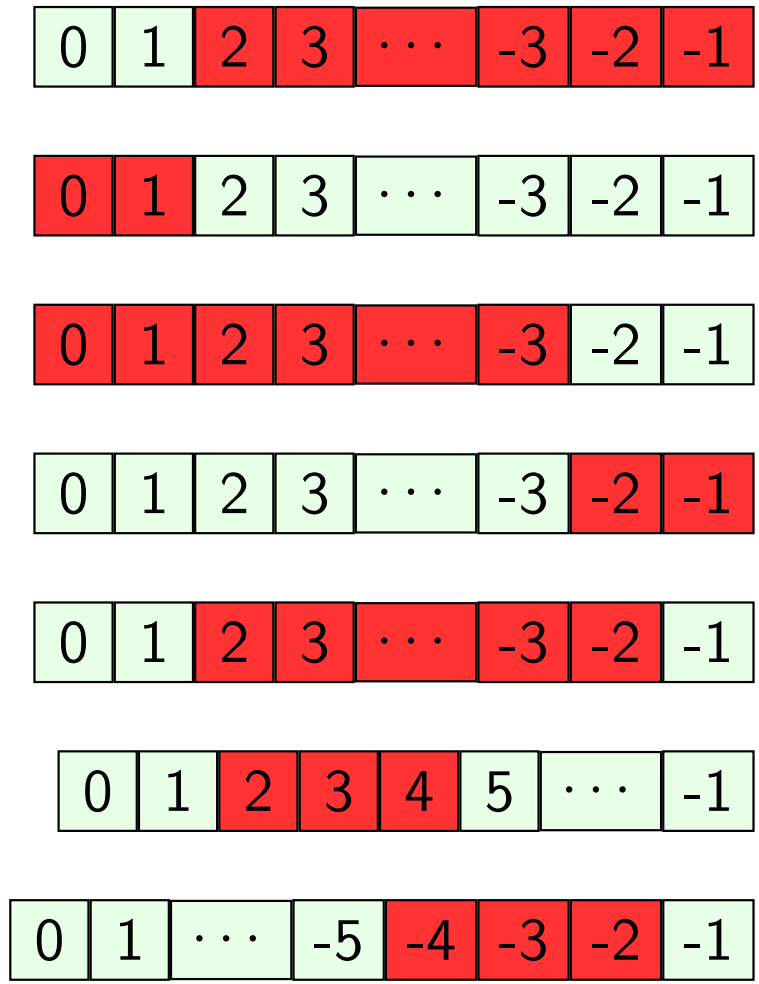
This is similar to half lines in $\mathbb{R}$: $[-\infty, a)$ means: take all numbers strictly lower than a, cf. syntax of `L[:j]`.

Let's sum it up

- `L[i:]` take all elements *except* the i first ones
- `L[:i]` take the i first elements
- `L[-i:]` take the last i elements
- `L[:-i]` to take all elements *except* the i last ones

Examples

- `L[2:]`
- `L[:2]`
- `L[:-2]`
- `L[-2:]`
- `L[2:-1]`
- `L[2:5]`
- `L[-4:-1]`



Dictionaries

A dictionary is an unordered structure of (key, value) pairs. It is similar to a list but the the objects are accessed by *keys* instead of *index*.

One indicates dictionaries by square brackets:

```
homework = {'Anna': True, 'Kerstin': False}

homework['Anna'] # True
# changing a value:
homework['Kerstin'] = True
# deleting an item
del homework['Anna']
```

Note: Any *immutable* (unchangable) object can be used as a key, e.g. strings, numbers, Booleans, etc. are fine, but not lists.

Looping through Dictionaries

A dictionary is an object with the following useful mehods:

`keys`, `items`, `values`

By default a dictionary is considered as a list of *keys*:

```
for key in homework:   # or homework.keys()
  print(f'{k} {homework[key]}')
```

Looping through Dictionaries

A dictionary is an object with the following useful mehods:

`keys`, `items`, `values`

By default a dictionary is considered as a list of *keys*:

```
for key in homework:  # or homework.keys()
  print(f'{k} {homework[key]}')
```

One may also use `items` to loop through keys and values:

```
for key, value in homework.items():
  print('{key} {value}')
```

Looping through Dictionaries

A dictionary is an object with the following useful mehods:

`keys`, `items`, `values`

By default a dictionary is considered as a list of *keys*:

```
for key in homework:  # or homework.keys()
  print(f'{k} {homework[key]}')
```

One may also use `items` to loop through keys and values:

```
for key, value in homework.items():
  print('{key} {value}')
```

You can also print just the values by using the `values` method:

```
for value in homework.values():
  print('{value}')
```

Dictionaries in this course

Dictionaries are mainly used for

- providing functions with arguments in a compact way (see Unit 4).
- to collect options of a method:
  `{'tol':1.e-3,'step size':0.1, 'maxit':1000}`

Sets

Sets are containers which mimic sets in their mathematical sense:

Definition

A *set* is a collection of well defined and distinct objects, considered as an object in its own right. (Wikipedia)

```
fruitbasket = set(['apple', 'pear', 'banana'])
```

The most important operation is `in`, meaning $\in$ (is element of):

```
'plum' in fruitbasket   # returns False
'pear' in fruitbasket   # returns True
```

Operations on Sets

Operations on sets are $A \cap B$, $A \cup B$ and A/B (intersection, union and relative complement):

```
fruitbasket = set(['apple', 'pear', 'banana'])
rotten = set(['pear'])
exotic = set(['mango', 'kiwi'])
emptyset = set([])          # the empty set
# Are all fruits edible (non-rotten)?
emptyset == fruitbasket.intersection(rotten)
# Add some diversity
extendedbasket = fruitbasket.union(exotic)
# The edible fruits
goodfruits = fruitbasket - rotten # rel complement
```

The following syntax can also be used:

```
  exotic = {'mango', 'kiwi'}
```

Compare with dictionaries: `{'a':'mango', 'b':'kiwi'}`

No duplicate elements

"...distinct objects ..." (see mathematical definition)

This is reflected in Python by

```
set({'apple', 'apple', 'pear'})
 == set({'apple', 'pear'}) # True
```

Tuples

Definition

A *tuple* is an *immutable* list. Immutable means that it cannot be modified.

Example

```
my_tuple = (1, 2, 3) # our first tuple!
my_tuple = 1, 2, 3   # parentheses not required
len(my_tuple) # 3, same method as for lists

my_tuple[0] = 'a' # error! tuples are immutable

singleton = 1, # note the comma
singleton = (1,) # preferred for readability
len(singleton) # 1
```

Packing and unpacking

One may assign several variables at once by *unpacking* a list or tuple:

```
a, b = 0, 1 # a gets 0 and b gets 1
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing for lists
```

Packing and unpacking

One may assign several variables at once by *unpacking* a list or tuple:

```
a, b = 0, 1 # a gets 0 and b gets 1
a, b = [0, 1] # exactly the same effect
(a, b) = 0, 1 # same
[a,b] = [0,1] # same thing for lists
```

The swapping trick!

Use packing and unpacking to *swap* the contents of two variables.

```
a, b = b, a
```

Returning Multiple Values

A function may return several values:

```
def argmin(L): # return the minimum and index
  ...
  return minimum, minimum_index      # a tuple

min_info = argmin([1, 2, 0])
min_info[0] # 0
min_info[1] # 2
# often unpacking is used directly
m, i = argmin([1, 2, 0]) # m is 0, i is 2
```

A final word on tuples

- Tuples are *nothing else* than immutable lists

A final word on tuples

- Tuples are *nothing else* than immutable lists
- In most cases lists may be used instead of tuples

A final word on tuples

- Tuples are *nothing else* than immutable lists
- In most cases lists may be used instead of tuples
- The parentheses free notation is nice but *dangerous*, you should *use parentheses when you are not sure*:

```
a, b = b, a # the swap trick, equivalent to:
(a, b) = (b, a)
# but
1, 2 == 3, 4 # returns (1, False, 4)
(1, 2) == (3, 4) # returns False
```

Conversions

Containers can be converted to other container types:

From	To	Command
List	Tuple	`tuple([1, 2, 3])`
Tuple	List	`list((1, 2, 3))`
List, Tuple	Set	`set([1,2,3])`, `set((1,2,3))`
Set	List	`list({1,2,3})`
Dictionary	List	`{'a':4}.values()`
List	Dictionary	–

Summary and Overview

Type	Access	Order	Duplicate Values	Mutability
List	index	yes	yes	yes
Tuple	index	yes	yes	no
Dictionary	key	no	yes	yes
Set	no	no	no	yes