Low-level series manipulation

The series manipulation capabilities that we have seen so far have mostly been of a mathematical nature: arithmetics, special functions, evaluation, substitution, etc. We are now going to see how we can manipulate series objects at a lower-level. Specifically, we will be using methods that act directly onto the terms of the series.

As in the previous lecture, the first step involves importing pyranha's modules and setting up some useful shortcuts.

In [1]:
```
from pyranhapp0x import *
from fractions import Fraction as Frac
```

Accessing series as lists of terms

We can get a representation of the series as a list of terms using the `list` attribute:

In [2]:
```
pt = polynomial.get_type('rational')
x,y,z = pt('x'),pt('y'),pt('z')
2*x + y**2/2 + z**3
```

Out[2]: $\frac{1}{2}y^2 + 2x + z^3$

In [3]:
```
(2*x + y**2/2 + z**3).list
```

Out[3]: `[(Fraction(1, 2), y**2), (Fraction(2, 1), x), (Fraction(1, 1), z**3)]`

Each item in the list is a term, represented as a `tuple`:

In [4]:
```
(2*x + y**2/2 + z**3).list[0]
```

Out[4]: `(Fraction(1, 2), y**2)`

The first element of the tuple is the coefficient:

In [5]:
```
t = (2*x + y**2/2 + z**3).list[0]
t[0]
```

Out[5]: `Fraction(1, 2)`

The second element of the tuple is the other piece of the term (e.g., the monomial, or the trigonometric part for Poisson series):

In [6]:
```
t[1]
```

Out[6]: y^2

Note that the type of the second element of the term tuple is the same type as the original series:

In [7]:
```
type(t[1])
```

Out[7]: `pyranhapp0x._core._polynomial_2`

In [8]:
```
type(x)
```

Out[8]: `pyranhapp0x._core._polynomial_2`

The original term can be reconstructed via multiplication:

In [9]:
```
t[0]*t[1]
```

Out[9]: $\frac{1}{2}y^2$

Being able to access the series as lists of terms opens up interesting possibilities. For instance, we can sort the terms using standard pythonic idioms such as the `sorted()` function:

In [10]:
```
sorted([4,7,2,8,9,0])
```

Out[10]: `[0, 2, 4, 7, 8, 9]`

Sorting is natural when it comes to integer numbers, but in our case we need to sort tuples of non-trivial object, and we want to be flexible (i.e., we want to be able to sort according to different criteria).

In order to do this, we need to use a bit of functional programming. Let's take a look:

In [11]:
```
s = (1 + x)**10
sorted(s.list,key = lambda t: t[0])
```

Out[11]:
```
[(Fraction(1, 1), 1),
 (Fraction(1, 1), x**10),
 (Fraction(10, 1), x),
 (Fraction(10, 1), x**9),
 (Fraction(45, 1), x**2),
 (Fraction(45, 1), x**8),
 (Fraction(120, 1), x**3),
 (Fraction(120, 1), x**7),
 (Fraction(210, 1), x**4),
 (Fraction(210, 1), x**6),
 (Fraction(252, 1), x**5)]
```

Here we are using the `sorted()` builtin function on the list representation of a series, with an extra parameter called `key`. This extra parameter is a *function* tasked with extracting something sortable from the elements of the list. In this case we are extracting the coefficient from the term and using that as a sorting key.

The `lambda` keyword is used do define a function *inline*, i.e., in the same place where it is going to be used. We could have used just a normal function to achieve the same effect:

In [12]:
```
def key_extractor(t):
    return t[0]

sorted(s.list,key = key_extractor)
```

Out[12]:
```
[(Fraction(1, 1), 1),
 (Fraction(1, 1), x**10),
 (Fraction(10, 1), x),
 (Fraction(10, 1), x**9),
 (Fraction(45, 1), x**2),
 (Fraction(45, 1), x**8),
 (Fraction(120, 1), x**3),
 (Fraction(120, 1), x**7),
 (Fraction(210, 1), x**4),
 (Fraction(210, 1), x**6),
 (Fraction(252, 1), x**5)]
```

Notice the differences between a normal function and a lambda function:

- the lambda function is limited to one single line,
- the lambda function does not need the `return` statement.

Let us take a look at another similar example:

In [13]:
```
sorted(s.list,key = lambda t: t[1].degree())
```

Out[13]:
```
[(Fraction(1, 1), 1),
 (Fraction(10, 1), x),
 (Fraction(45, 1), x**2),
 (Fraction(120, 1), x**3),
 (Fraction(210, 1), x**4),
 (Fraction(252, 1), x**5),
 (Fraction(210, 1), x**6),
 (Fraction(120, 1), x**7),
 (Fraction(45, 1), x**8),
 (Fraction(10, 1), x**9),
 (Fraction(1, 1), x**10)]
```

There are other useful Python builtins which work in a similar way:

In [14]:
```
max(s.list,key = lambda t: t[1].degree())
```

Out[14]:
```
(Fraction(1, 1), x**10)
```

In [15]:
```
min(s.list,key = lambda t: t[0])
```

Out[15]:
```
(Fraction(1, 1), 1)
```

Filtering

When working with collections of objects, a useful capability it that of filtering, i.e., selectively discarding elements from a collection. In pyranha, we could implement filtering by operating on the list representation of a series, and then re-assembling it.

However, the usage pattern was deemed common enough to warrant a separate `filter()` method. Recall our series object `s`:

In [16]:
```
s
```

Out[16]: $1 + 10x + 45x^2 + 120x^3 + 210x^4 + 252x^5 + 210x^6 + 120x^7 + 45x^8 + 10x^9 + x^{10}$

Let's say we want to retain only terms whose degree is less than 5:

In [17]:
```
s.filter(lambda t: t[1].degree() < 5)
```

Out[17]: $1 + 10x + 45x^2 + 120x^3 + 210x^4$

Here we are passing to the `filter()` method a function which is expected to accept a term as input value (in the same format as in the list representation of the series) and return a `True` / `False` value. `filter()` will iterate over each term of the series and run the function with each term as an argument: if the return value is `True` then the term will be kept, otherwise it will be discarded from the series.

Let's filter out the terms with odd coefficients:

In [18]:
```
s.filter(lambda t: t[0] % 2 == 0)
```

Out[18]: $10x + 10x^9 + 120x^3 + 210x^4 + 252x^5 + 210x^6 + 120x^7$

We can retain only those terms which evaluate to at least a certain magnitude:

In [19]:
```
s.filter(lambda t: (t[0] * t[1]).evaluate({'x':.5}) > 3)
```

Out[19]: $10x + 45x^2 + 120x^3 + 210x^4 + 252x^5 + 210x^6$

In these examples, the use of a lambda function is just a matter of convenience, clarity and economy of typing. The exact same effect can be achieved with a normal function:

In [20]:
```
def my_filter_function(t):
    return (t[0] * t[1]).evaluate({'x':.5}) > 3

s.filter(my_filter_function)
```

Out[20]: $10x + 45x^2 + 120x^3 + 210x^4 + 252x^5 + 210x^6$

Lambdas are best employed when the function is short and simple. For anything that does not fit on a single line, it is usually better to define an external separate function and use that instead.

Transforming

Thus far we have operated on polynomials, i.e., series whose coefficients are numerical objects. Things get more complicated when the coefficients themselves are series, such as in Poisson series:

In [21]:
```
pst = poisson_series.get_type('polynomial_rational')
a,b,c,d = pst('a'),pst('b'),pst('c'),pst('d')
ps = (a/2*math.sin(b+c) + d/4*math.cos(2*b-c))**5
ps
```

Out[21]: $\frac{1}{512} a^5 \sin\left(5b + 5c\right) + \left(\frac{15}{512} a^3 d^2 + \frac{15}{4096} a d^4 + \frac{5}{256} a^5\right) \sin\left(b + c\right) + \left(-\frac{5}{512} a^3 d^2 - \frac{5}{512} a^5\right) \sin\left(3b + 3c\right) + \frac{5}{1024} a^4 d \cos\left(6b + 3$

Here we can get the list representation of the series and perform filtering operations, but only at the level of Poisson series terms:

In [22]:
```
ps.list[0]
```

Out[22]: `(1/512*a**5, sin(5b+5c))`

What if we wanted to filter terms in the coefficients (e.g., retain only powers of a up to 3)? We need to introduce another primitive operation, `transform()`. Let's start with a silly example:

In [23]:
```
ps.transform(lambda t: (2*t[0],t[1]))
```

Out[23]: $\frac{1}{256} a^5 \sin\left(5b + 5c\right) + \left(\frac{15}{256} a^3 d^2 + \frac{15}{2048} a d^4 + \frac{5}{128} a^5\right) \sin\left(b + c\right) + \left(-\frac{5}{256} a^3 d^2 - \frac{5}{256} a^5\right) \sin\left(3b + 3c\right) + \frac{5}{512} a^4 d \cos\left(6b + 3c\right)$

This is an (inefficient) way of multiplying the series by 2. `transform()` takes as an argument a function that is supposed to receive a term as input and return a (possibly transformed) term as output.

Let's now get back to the initial task of filtering on the polynomial coefficients. What we need to do is to `transform()` each term of the Poisson series leaving the trigonometric part intact and `filter()`-ing terms in the coefficients. Let's start with the filtering function on the polynomials:

In [24]:
```
lambda u : u[1].degree(['a']) <= 3;
```

For each term of the polynomial coefficient, we retain only those whose degree in a is no greater than 3. We can now chain up this function to the `filter()` and `transform()` methods:

In [25]:
```
ps.transform(lambda t: (t[0].filter(lambda u : u[1].degree(['a']) <= 3),t[1]))
```

Out[25]:
$$\left(\frac{15}{512}a^3d^2+\frac{15}{4096}ad^4\right)\sin\left(b+c\right)-\frac{5}{512}a^3d^2\sin\left(3b+3c\right)+\left(\frac{15}{1024}a^2d^3+\frac{5}{8192}d^5\right)\cos\left(2b-c\right)-\frac{15}{2048}a^2d^3\cos\left(4b+c\right)-\frac{5}{10}$$