Teaching programming with Jupyterhub and Nbgrader

Gert-Ludwig Ingold
Universität Augsburg

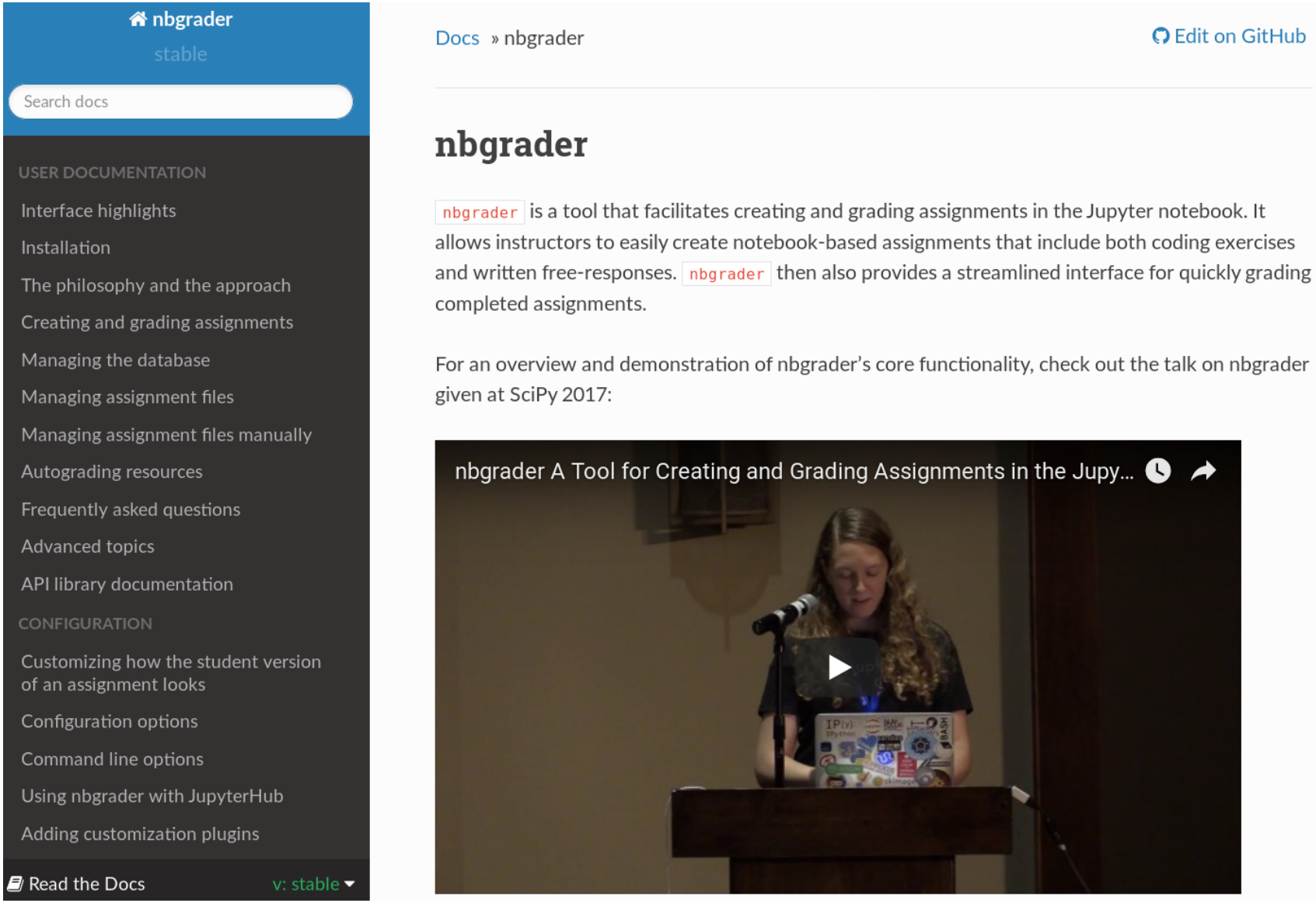
About me

- theoretical physicist at the Universität Augsburg
- teaching programming to physicists and materials scientists since 2010, typically 10+ students
- in two years: mandatory course for 100+ students ⟶ nbgrader
- involved in two Erasmus+ projects on education and computing
  - iCSE4school (2015–2017)
  - Juypter@edu (2017–2019)

Jupyter@edu

- University of Silesia, Poland (Leader)
- Universität Augsburg, Germany
- Universidade Portucalense Infante D. Henrique, Portugal
- European University Cyprus, Cyprus

nbgrader



nbgrader.readthedocs.io github.com/jupyter/nbgrader/

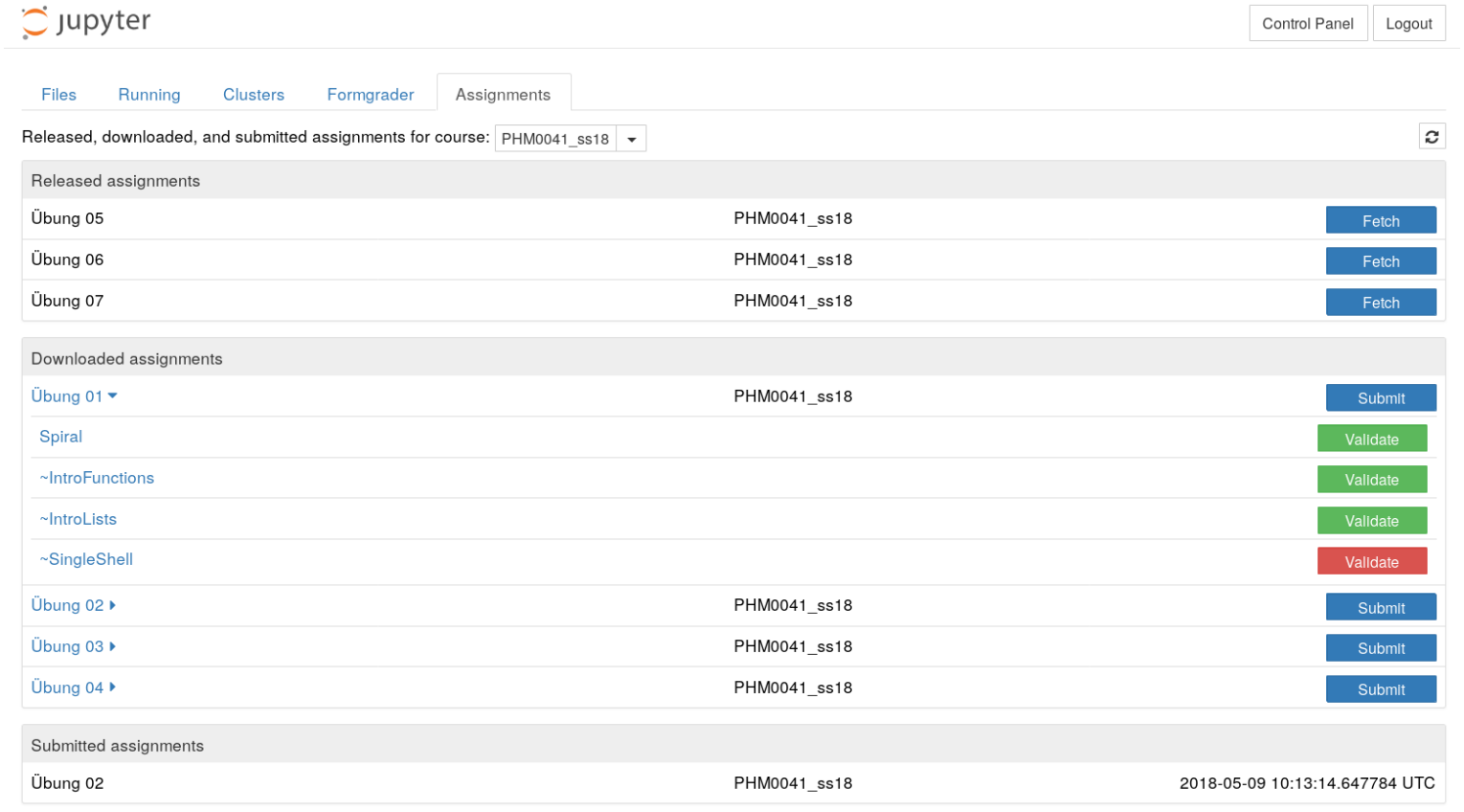
Jupyterhub and Jupyter notebooks
grading
branching paths

Jupyterhub and Jupyter notebooks

grading

branching paths

Jupyterhub and nbgrader



- easily accessible interface to problem sets
- no need to install Python on local computer
- consistent working environment for all students
- *but* may be beginners should gather experience with Python on their own computer

Jupyter notebooks or something else?

Parrondo's paradox

Random numbers and fair coin

In order to play the games needed to demonstrate Parrondo's paradox, we need to simulate the flipping of a coin. This can be achieved by generating random numbers. As random numbers on a computer are usually generated by means of a deterministic algorithm, the generated numbers are actually pseudo-random numbers.

Python's standard libary contains a module `random` which provides various ways to generate pseudo-random numbers. The basic way to generate equally distributed floats on an interval $[0.0, 1.0)$ is as follows:

In [ ]:
```
import random
random.random()
```

Call the `random` method a few times to convince yourself that a different number is obtained each time.

In [ ]:
```
random.random()
```

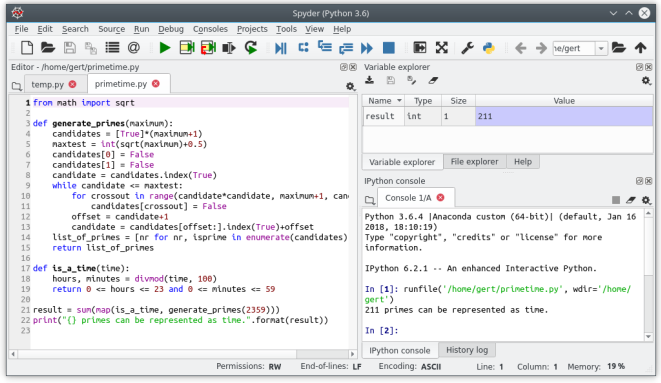
In order to make the generation of the random numbers reproducible, we can set a seed. As argument, an arbitrary integer can be chosen.

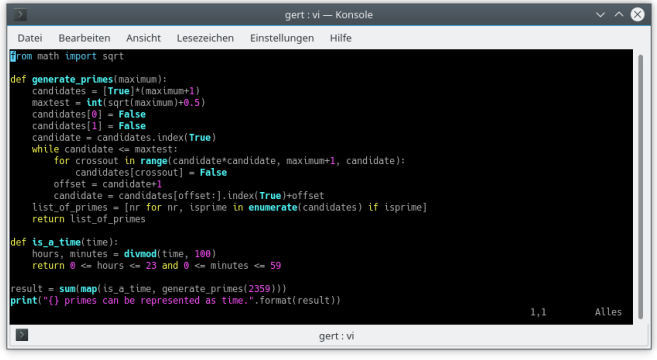
In [ ]:
```
random.seed(0)
for _ in range(10):
    print(random.random())
```

Convince yourself that the cell above always yields the same ten random numbers.

Now, let us make use of what we have learned so far. We want to play a game where we flip a fair coin. We loose if the random number is smaller than 0.5, otherwise we win. Complete the following function which should return the number of winning flips for a given number of total flips.

In [ ]:
```
import random

def flip_fair_coin(ntotal, seed):
    """determine the number of winning flips

       ntotal: total number of flips
       seed:   seed for random number generator
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```





- notebook allows to guide students through a problem set
- students might think that programming in Python implies working with a Jupyter notebook

Jupyterhub and Jupyter notebooks

grading

branching paths

Grading

In [ ]: ID: julia-iteration — Autograded answer

```
def juliaiter(z0, c, threshold, maxiter):
    """Determine number of iterations needed to cross the threshold

       z0:        initial value for z
       c:         complex number in iteration prescription
       threshold: threshold value to be crossed by |z|
       maxiter:   maximum of iterations
    """
    ### BEGIN SOLUTION
    z = z0
    niter = 0
    while niter < maxiter and abs(z) <= threshold:
        niter = niter+1
        z = z**2 + c
    return niter
    ### END SOLUTION
```

-

Test your solution by executing the following two cells. Everything is fine if no error message is displayed.

In [ ]: Points: 1 — ID: test-existence — Autograder tests

```
result = juliaiter(-0.3+0.4j, -0.8+0.156j, 2, 100)
assert result is not None, 'Does your function return a result?'
assert type(result) == type(1), 'The result should be an integer.'
```

In [ ]: Points: 2 — ID: test-correctness — Autograder tests

```
import math
maxiter = 100
threshold = 2
assert juliaiter(1, 0, threshold, maxiter) == maxiter, 'There is a problem even for c=0.'
z0 = 1.0000001
expected = int(math.log(math.log(threshold)/math.log(z0), 2))+1
assert juliaiter(z0, 0, threshold, maxiter) == expected, 'There is a problem even for c=0.'
z0 = -0.3+0.4j
c = -0.8+0.156j
threshold = abs(z0**8+4*z0**6*c+2*z0**4*(3*c**2+c)+4*z0**2*(c**3+c**2)+c**4+2*c**3+c**2+c)
maxiter = 4
assert juliaiter(z0, c, threshold+1e-6, maxiter) == maxiter, 'Wrong number of iterations'
assert juliaiter(z0, c, threshold-1e-6, maxiter) == maxiter-1, 'Wrong number of iterations'
```

- autograded answer
- autograder tests
- manually graded answer

Functions

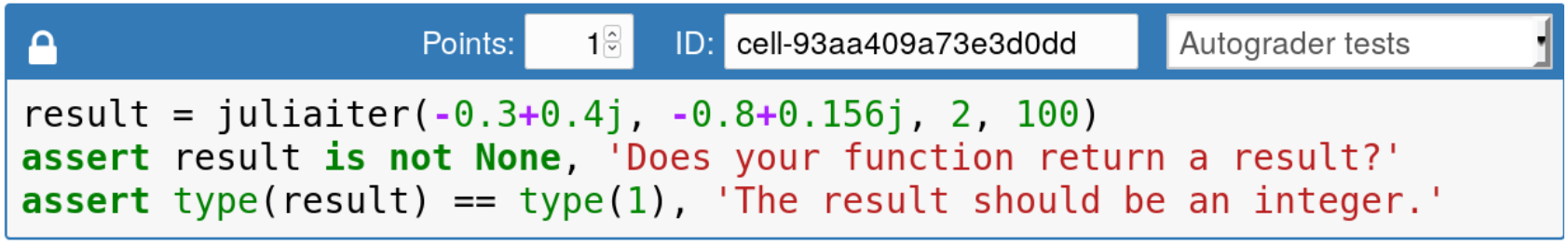
ID: julia-iteration Autograded answer

```
def juliaiter(z0, c, threshold, maxiter):
    """Determine number of iterations needed to cross the threshold

       z0:        initial value for z
       c:         complex number in iteration prescription
       threshold: threshold value to be crossed by |z|
       maxiter:   maximum of iterations

    """
    ### BEGIN SOLUTION
    z = z0
    niter = 0
    while niter < maxiter and abs(z) <= threshold:
        niter = niter+1
        z = z**2 + c
    return niter
    ### END SOLUTION
```

- need to introduce functions very early in the course
  but special aspects (no arguments, no return value, default arguments, keyword arguments, …) not needed
- students get used to logically structured code early on
  but they do not do it themselves
- include docstrings to make task well defined
  students get used to the idea that a function contains a docstring
  but they are not writing the docstring by themselves

Grading with tests

- test-driven development
  but tests are not developed by the students
- tests allow to give feedback through error messages
  but students tend to rely on this feedback instead of developing their own critical view on their code
- “Stupid mistakes” will be made not only at the beginning
  - “trivial” standard tests (are results returned, do they have the correct type, . . . ?)
  - + tests of specific functionality which should not disclose the solution



Jupyterhub and Jupyter notebooks

grading

branching paths

Coping with heterogeneity

- students without programming experience and (sometimes) code writing apprehension
- students with knowledge in another programming language
  code often does not look very pythonic
  example: loop over objects vs. loop over indices

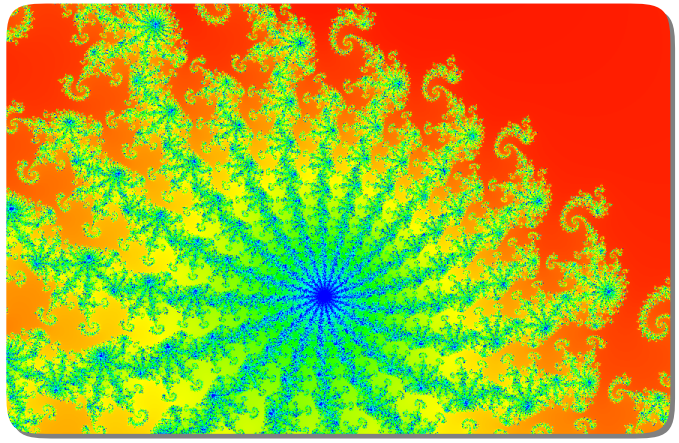
students with programming experience should not be bored and should obtain an interesting result from their code

- *but* scientific applications are often considered as an extra mental burden

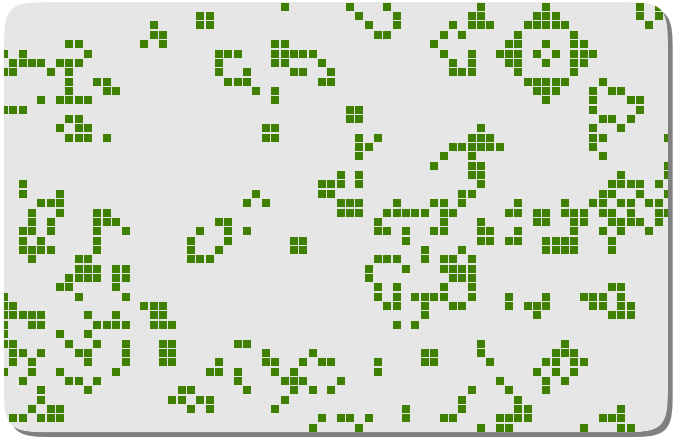
Examples of problem sets

09	72	43	44	45	46	47	48	49	50	8
08	71	42	21	22	23	24	25	26	51	8
07	70	41	20	7	8	9	10	27	52	8
06	69	40	19	6	1	2	11	28	53	8
05	68	39	18	5	4	3	12	29	54	8
04	67	38	17	16	15	14	13	30	55	8
03	66	37	36	35	34	33	32	31	56	8

selected problems from
Project Euler (`projecteuler.net`)



Julia set



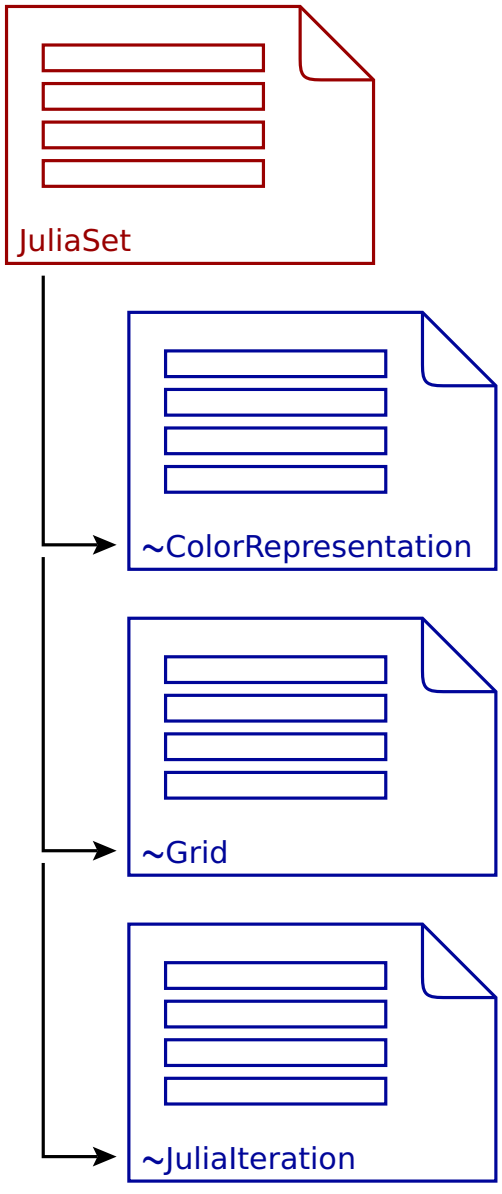
Conway's game of life

```
3.14159265358979323846264
33832795028841971693993751
05820974944592307816406286
20899862803482534211706798
21480865132823066470938446
09550582231725359408128481
11745028410270193852
```

π to a few thousand digits

`https://github.com/marcinofulus/jupyter4edu/tree/master/augsburg/exercises`

Branching paths



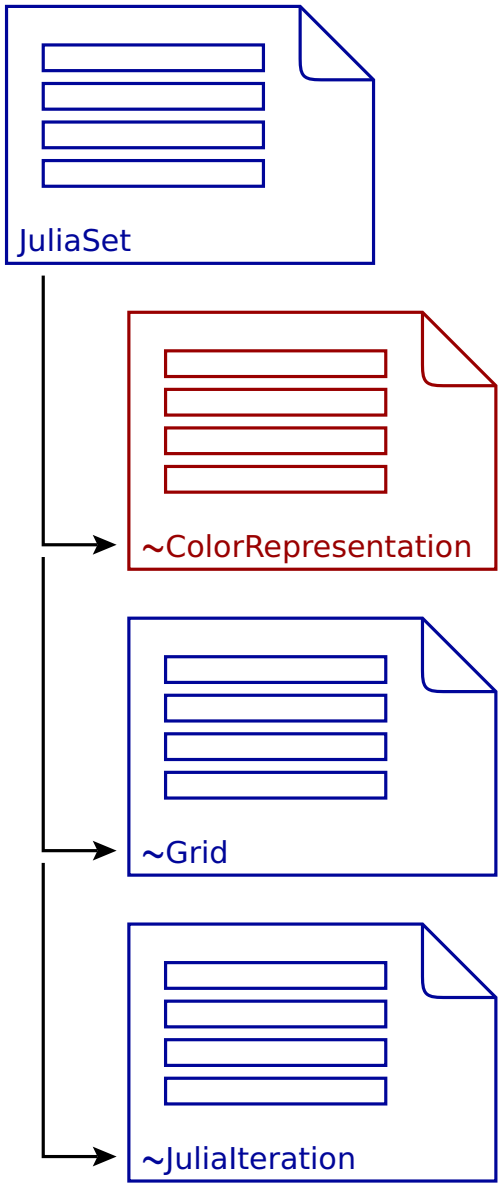
should not set the gridsize too large. It is a good idea to start e.g. with a 100×100 grid. Once your code is working, you can increase the number of grid points. For practical purposes, it makes sense to define a threshold for the absolute value of z beyond which you assume that the series for a given starting value z_0 diverges. A good value could be 2. You should also limit the number of iterations. Here, a good value could be 600. Feel free to play around with these values once your code works.

If you want to tackle the problem without any further help, you can jump over the following three points which are intended to give you some help by splitting the complete problem into smaller parts.

1. Representation of a number by a color
2. Equally spaced numbers on a grid
3. Iteration prescription

In [ ]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

def plot(data, ndim, cmap):
    """Create a 2d graphics from a list of data points

       data: list of ndim² data between 0 and 1
       ndim: side length of grid
       cmap: matplotlib color map mapping interval from 0 to 1 to color

    """
```

Branching paths




```
def colorbar(cmapname):
    gradient = np.linspace(0, 1, 256)
    gradient = np.vstack((gradient, gradient))
    fig = plt.imshow(gradient, aspect=0.05, extent=(0, 1, 0, 1), cmap=plt
    fig.axes.get_yaxis().set_visible(False)

def display_color(x, cmapname):
    rgbcolor = cm.get_cmap(name=cmapname)(float(x), bytes=True, alpha=Fal
    print(rgbcolor)
    return SVG('''<svg height="100" width="100">
                    <rect x="1" y="1" width="50" height="50" fill="rgb(
                </svg>'''.format(*rgbcolor))
```

We start with one of the colormaps provided by the `matplotlib` library called `hot`:

In [3]:
```
colorbar('hot')
```

In the following cell, you can change the argument of the function `display_color` between 0 and 1 and explore how the color changes.

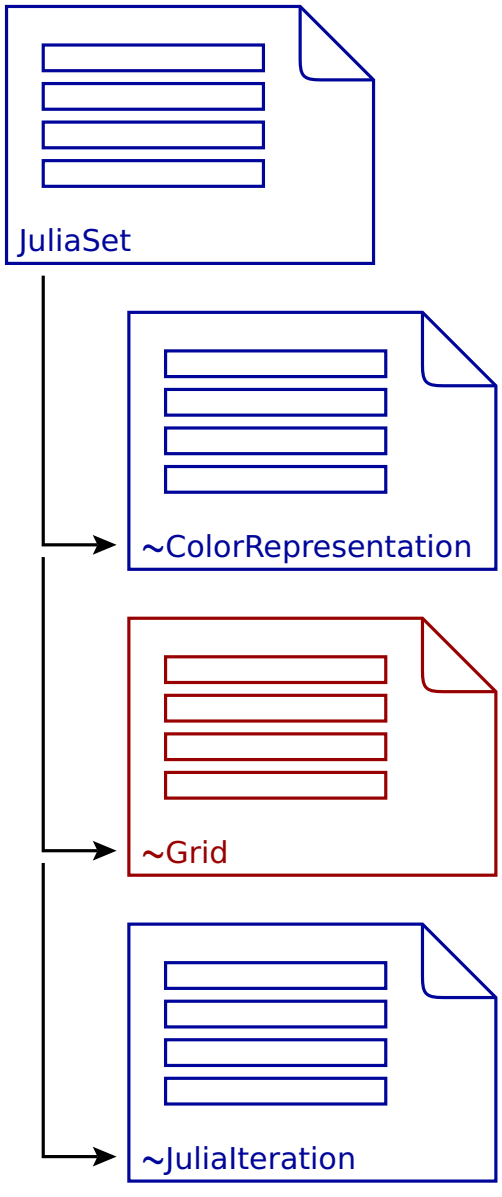
In [4]:
```
display_color(0.9, 'hot')
```

```
(255, 255, 156)
```

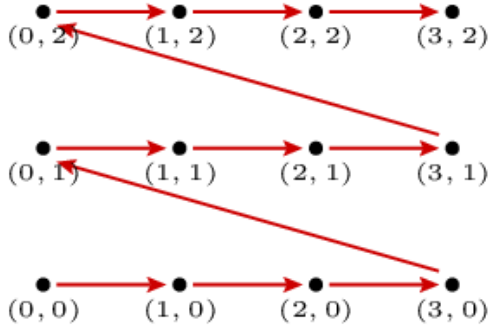
Out[4]:

The `matplotlib` library provides many more colormaps, like e.g. `viridis`. Take a look at Colormaps in Matplotlib for more information. Try out a couple of other colormaps.

Branching paths



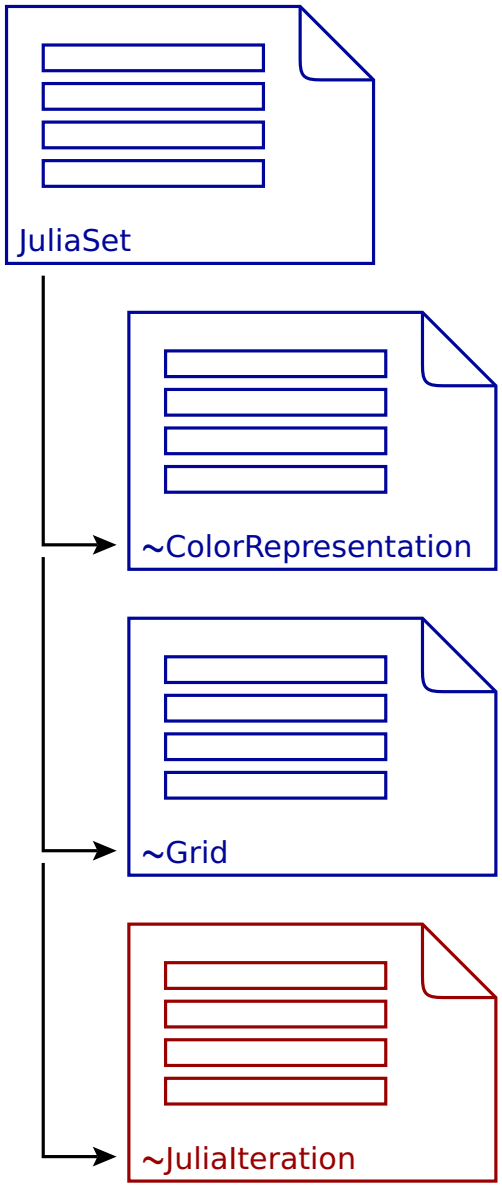
If the code for generating a one-dimensional grid is working, you are ready to extend it to two-dimensional grids. Walk the grid horizontally before making a vertical step like indicated in the following figure.



The output in this case should be the list `[(0, 0), (1, 0), (2, 0), (3, 0), (0, 1), (1, 1), (2, 1), (3, 1), (0, 2), (1, 2), (2, 2), (3, 2)]`.

In [ ]:
```
def grid2d(xmin, xmax, nxpts, ymin, ymax, nypts):
    """Generate a list of coordinate tuples (x, y) on a two-dimensional gric
       The points are equally spaced in each of the two directions between
       (and including) the respective bounds. The grid is first walked along
       the horizontal (x) direction, then along the vertical (y) direction.

       xmin: lower bound in horizontal direction
       xmax: upper bound in horizontal direction
       nxpts: number of points in horizontal direction
       ymin: lower bound in vertical direction
       ymax: upper bound in vertical direction
       nypts: number of points in vertical direction
    """
```

Branching paths



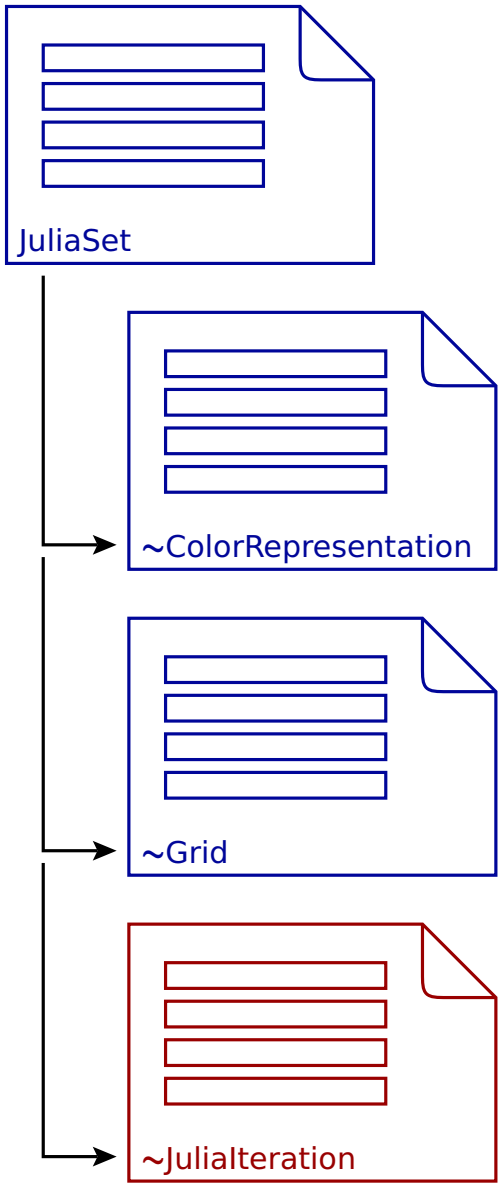
In [ ]:
```
def juliaiter(z0, c, threshold, maxiter):
    """Determine number of iterations needed to cross the threshold

       z0:        initial value for z
       c:         complex number in iteration prescription
       threshold: threshold value to be crossed by |z|
       maxiter:   maximum of iterations
    """
    # YOUR CODE HERE
    raise NotImplementedError()
```

Test your solution by executing the following two cells. Everything is fine if no error message is displayed.

In [ ]:
```
result = juliaiter(-0.3+0.4j, -0.8+0.156j, 2, 100)
assert result is not None, 'Does your function return a result?'
assert type(result) == type(1), 'The result should be an integer.'
```

In [ ]:
```
import math
maxiter = 100
threshold = 2
assert juliaiter(1, 0, threshold, maxiter) == maxiter, 'There is a problem e
z0 = 1.0000001
expected = int(math.log(math.log(threshold)/math.log(z0), 2))+1
assert juliaiter(z0, 0, threshold, maxiter) == expected, 'There is a problem
z0 = -0.3+0.4j
c = -0.8+0.156j
threshold = abs(z0**8+4*z0**6*c+2*z0**4*(3*c**2+c)+4*z0**2*(c**3+c**2)+c**4+
maxiter = 4
assert juliaiter(z0, c, threshold+1e-6, maxiter) == maxiter, 'Wrong number o
assert juliaiter(z0, c, threshold-1e-6, maxiter) == maxiter-1, 'Wrong number
```

Branching paths



- individual path through problem possible
- *but* notebooks are opened in new tabs, it is easy to lose track