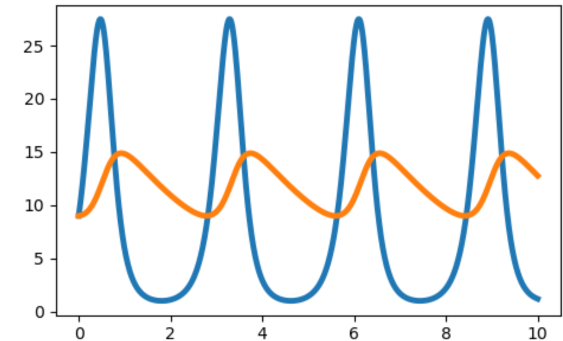
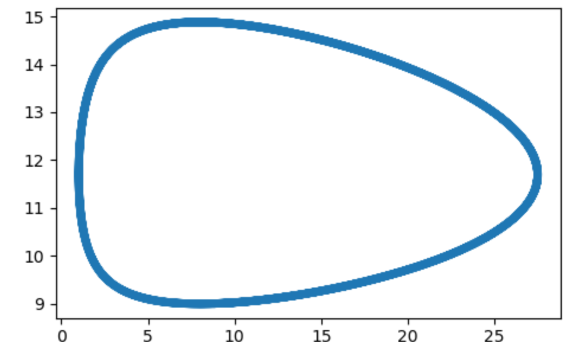
Learning outcomes

- Learning fundamental programming skills in Python
- Describe simple dynamic systems with equations (Mathematical model)
- Simulate dynamic systems using Python (numerical solutions)
- Visualize the results in different ways
- Interpret and analyze results from simulations





Why do we program?

- Computers are excellent at calculating/sorting things very fast
- Humans are (sometimes) good at thinking but not very fast in calculating/sorting
- Programming: Telling your computer to do iterative & annoying tasks
- Programming languages: Translating human instructions to a language that the computer “understands”

Why do we use Python?

- Open source and object oriented programming language
- Many programming languages are good for certain purposes
- Python can be used for almost everything
- Python: accessible syntax and useful packages
- Many scientific and commercial programs choose Python as programming language
  - NASA, Google, Youtube, Reddit, Instagram, Video Games etc.

Lets get started – Where do we program?

- Coding in python
  - Write a piece of text in any text editor that formulates the code you want to run
- Running a code
  - Pass the piece of text to a “Interpreter” that reads the text and translates the commands and operations to the computer (typically in the Console)
- Instead of writing code in any text editor, we prefer IDEs (Integrated Development Environment)
  - include very helpful tools for coding

Integrated Development Environments

- Many different IDEs available
- Most common ones:
  - PyCharm
  - Jupyter Notebook
  - Spyder
- This lecture is being taught in Spyder
  - If you already know how to program in Python and prefer an other IDE, feel free to use it
- Task: Open Spyder on your devices

Spyder (Python 3.5)

File Edit Search Source Run Debug Consoles Tools View Help

/home/student

Editor - /home/student/.spyder2-py3/temp.py

temp.py

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7
8
```

Object inspector

Source Console Object

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Object Inspector.*

New to Spyder? Read our tutorial

Optional Tools

Object inspector | Variable explorer | File explorer

IPython console

Console 1/A

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
%guiref   -> A brief reference about the graphical user interface.

In [1]:
```

Console

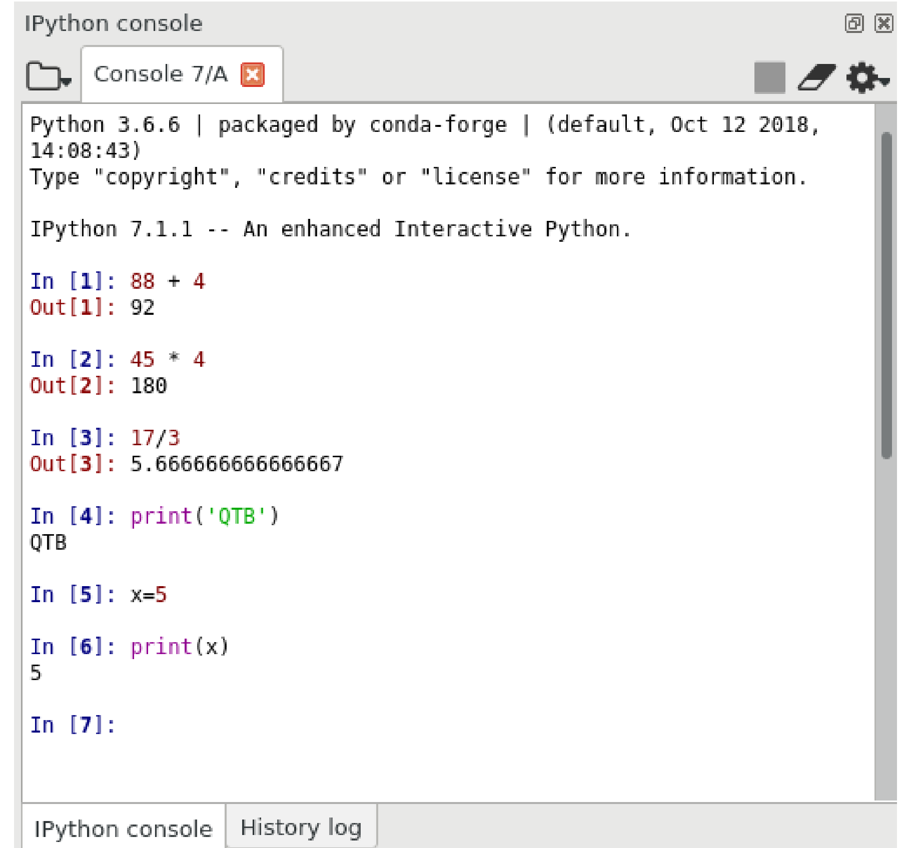
Console | History log | IPython console

Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 1 Column: 1 Memory: 7 %

Questions? Nima.Saadat@hhu.de

IPython Console

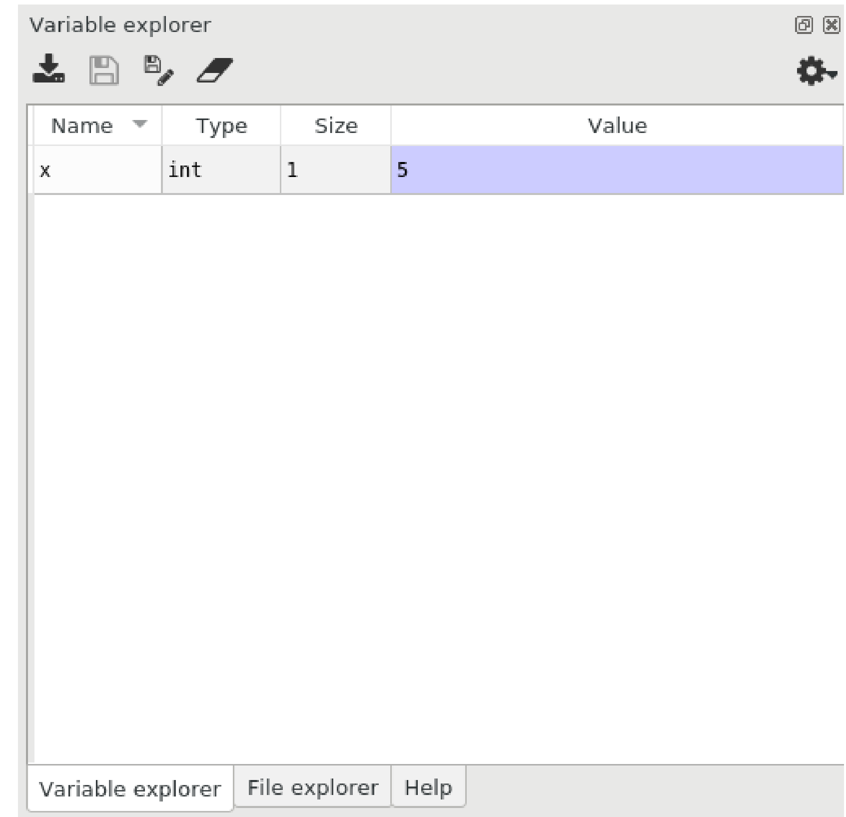
- “Interactive Mode”
- Executes your Code (Pass commands and instructions to the computer)
- “Remembers” the commands and definitions that you defined and delivered
- Essentially: The Console runs Code



Optional Tools

- A window displaying optional tools for making coding easier
- Bottom tabs show options
- Personal suggestion :

  Choose the “**Variable Explorer**”, because it is the most useful (for our course)



Script

- Many lines of code that work together can be passed to the interpreter / IPython Console at once
- Write your code in the script and hit the “Run” button
- Scripts are read from top to bottom

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 13 14:27:27 2018

@author: nima
"""


X=5
Y=10

Z= X*2 + Y*5 - X*Y

print (Z)
```

Name	Type	Size	Value
X	int	1	5
Y	int	1	10
Z	int	1	10

```
Python 3.6.6 | packaged by conda-forge | (default, Oct 12 2018,
14:08:43)
Type "copyright", "credits" or "license" for more information.

IPython 7.1.1 -- An enhanced Interactive Python.

In [1]: runfile('/home/nima/exampleXY.py', wdir='/home/nima')
10

In [2]:
```

Before we begin: Some tips for Python beginners

- Computers only do what we tell them to do
  - If “it doesn’t work”, it is our fault – not the computers
- A major skill is to efficiently use Google while coding
- The chances that the problems you may encounter have already been solved on StackOverflow are extremely high

Lets begin – with comments

- Comments are lines of text in code that are ignored by the interpreter
- Comments are used in code to
  - Describe in words what a certain part of the code is used for
  - Deactivate parts of your code without deleting it
- One line comments begin with an hash #
- Comments over multiple lines start and end with three quotation marks ''' or " " " (decide on one of both)

```
1 #This is a one line comment
2
3 '''
4 This is a comment
5 that needs more
6 than one line
7 '''
```

Don’t use non-ASCII characters in neither your code nor the comments

Data types in general

- Many data types in Python
  - Integers and floats
  - Strings
  - Lists
  - Dictionaries
  - Etc.

```
In [1]: type(42)
Out[1]: int
```

- Find out what datatype you are dealing with using **type( )**
- **In Python, dots are used as decimal placements and commas are used for separating elements**

Data types – Integers and floats

- Integers: Numbers without decimal placements
  - Numbers without points and/or decimals or redefine numbers to integers with **int( )**
- Floats: Numbers with decimal placements
  - Define numbers with points or decimals or redefine numbers to floats with **float( )**
- Floats for calculation, integers for indexing

```
In [1]: type(42)
Out[1]: int

In [2]: type(42.5)
Out[2]: float

In [3]: type(42.)
Out[3]: float

In [4]: int(42.5)
Out[4]: 42

In [5]: float(42)
Out[5]: 42.0

In [7]: 23/5
Out[7]: 4.6

In [8]: int(23/5)
Out[8]: 4
```

Defining Variables

- Define variable with letters and numbers, as well as underlines
- First character must be a letter (beware capital and lowercase!)
- The console will remember the name and content (unless you redefine it or restart the console)
- Tip: Define variable names rather too detailed than too abbreviated
  - At some point we forget the difference between X, X1, X2 [...] X42

```
In [1]: Var1=4

In [2]: Var2=2

In [3]: type(Var1)
Out[3]: int

In [4]: Var3=Var1/Var2

In [5]: Var3
Out[5]: 2.0

In [6]: type(Var3)
Out[6]: float
```

Datatypes – Strings

- Strings contain are chains of any character
- Define Strings with quotation marks ‘ or “ in the beginning and the end if your string
- Strings can be concatenated with plus characters (+)

```
In [1]: type('fourtytwo')
Out[1]: str

In [2]: type('42.5')
Out[2]: str

In [3]: string1='4'

In [4]: string2='2'

In [5]: string42=string1+string2

In [6]: string42
Out[6]: '42'
```

Introduction to Objects (!!!)

- Almost every data type is an object (part of a class) that contains useful built-in functions
- Access the built-in functions with a dot after the object name
- Functions are used with opened & closed brackets at the end ( )
  - If the function needs arguments, they are provided inside of the brackets
- Use **dir( )** and the abbreviation for the data type to see all built-in functions
- Use documentation to understand and use functions

```
In [1]: String1='Fourtytwo'

In [2]: String1.upper()
Out[2]: 'FOURTYTWO'

In [3]: String1.count('t')
Out[3]: 2
```

Datatypes – Lists and indexing I (!!!)

- The (for us) most important data type is lists or arrays
- Lists are enumerations of objects that can be any data type
- Define with square brackets [ ]
- The objects inside a list have indices and can be accessed by their index

```
list1 = [42,24,'fourty','two']
```

Index : 0 1 2 3

Index counting starts at 0 in Python

Datatypes – Lists and indexing II (!!!)

- Access elements of an list with the list name and the index in square brackets
- The index -1 returns the last element
- Get the absolute length of a list with **len( )**
- Lists include many very useful built-in functions

```
In [1]: test_list = [42, 24, 2, 4]

In [2]: test_list[0]
Out[2]: 42

In [3]: test_list[-1]
Out[3]: 4

In [4]: len(test_list)
Out[4]: 4

In [5]: test_list.sort()

In [6]: test_list
Out[6]: [2, 4, 24, 42]

In [7]: test_list.append(4200)

In [8]: test_list
Out[8]: [2, 4, 24, 42, 4200]
```

Exercises I – Use the Documentation

1. Define a list containing the age of all your family members
2. Sort this list from youngest to the oldest
3. Delete the youngest person from the list
4. Add number 27 to your list
5. Change the second number in the list to 14
6. Revert the order of the list
7. Create a new list containing only the first two elements of the old list
8. Concatenate both lists into another new list

Avoid “hardcoding”! Your code should work with ANY list of ages

Dictionaries

- Dictionaries are another type of container for elements
- In dictionaries each element (Value) has a “Key”
- Keys = Strings!
- Define dictionaries with { }
- Add elements with a string (Key), acolon and a value
- Separate entries with commas
- Access elements in dictionaries with the right “Key” in [ ]

```
In [1]: test_dict = {'one':1, 'fourtytwo':42}

In [2]: test_dict['fourtytwo']
Out[2]: 42
```

Python functions

- Python itself includes many useful functions like:

  print( ) displays the element in the console

  len( ) returns the absolute length of a list

  max( ) returns the biggest number element from a list or the longest string from a list

  type( ) returns the data type of an element

  range( ) creates a list of integers from zero to the provided integer

- For more, search in the documentation!

```
In [1]: test_list = [4, 2, 42]

In [2]: print(test_list)
[4, 2, 42]

In [3]: len(test_list)
Out[3]: 3

In [4]: max(test_list)
Out[4]: 42

In [5]: type(test_list)
Out[5]: list

In [5]:

In [6]: range_list = range(10)
   ...:

In [7]: list(range_list)
Out[7]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Operators in Python

- Assignment: **=**
- Arithmetic: **+**, **-**, *****, **/**, ******, **%**
- Comparison: **<**, **>**, **<=**, **>=**
- Logical: **and**, **in**, **or**, **not**
- Increasing/Decreasing: **+=**, **-=**

If statements

- Logic statements
- If a condition is satisfied, then a piece of code is executed
- Examples for conditions:
  - If a number or length is equal / bigger than / smaller than sth.
  - If an element is the same as another element
  - If an element is inside of a list of elements
  - Etc.

How to formulate If statements and Indentations

- Begin with **if** command
- Define a condition and finish with a colon **:**
  - In this case, our condition is a comparison
  - Comparing for equality is done with **==**
- In the line after the colon, place a indentation
  - All indented code below is executed if the condition is satisfied (indent with tab key)
- After indentation, write a code that is only executed when defined conditions are satisfied
- The **else** statement is constructed similarly and is executed if the defined condition is not satisfied
  - optional, depending on your code

```
11 x = 5
12
13 if x == 5:
14     print(x)
15 else:
16     print('Not 5')
```

Small Exercise for if statements and indentations

1. Define two variables and an empty list
2. Define an if statement:
   - When the sum of the variables is greater or equal to 200, then add that number to your list
   - When the sum is not greater or equal to 200, then print a message

For loops I

```
11 loop_list = [10,20,30,40,50]
12
13 for i in loop_list:
14     result = 100 + i
15     print(result)
```

```
110
120
130
140
150
```

- The length of the corresponding list defines how long a For-loops lasts
- In every cycle, temporary variable i is assigned to the next element inside the corresponding list

For loops II

```
11 loop_list = [10,20,30,40,50]
12
13 for i in loop_list:
14     result = 100 + i
15     print(result)
```

```
for i in loop_list:
    result = 100 + i
    print(result)
```

- Begin with a **for** command
- Define the name of the loop-variable
- Continue with an **in** command
- Define the list which defines the values of ***i***
- Finish with a colon
- All lines below the colon that include code that should be looped need to be indented

While loops

- Looping as long as a condition is satisfied
- In this example:
  - The variable counter is 0
  - loops as long as counter is smaller than 10
  - Inside the loop, the counter needs to be increased
    - Otherwise: Infinite loop

```
10 counter=0
11
12 while counter<10:
13     print('loop')
14     counter += 1
```

Exercises II

1. Find all numbers dividable by three between 0 and 100

Hint 1: Use modulo **%**

Hint 2: Use combination of **if statement** and **for loop**

2. Change the code so the results will be stored in a list

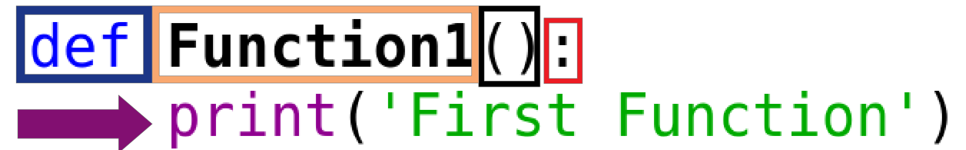
Hint 3: Declare empty lists outside the loop

3. Select only even numbers from the new list

Functions I – Without Arguments

- Functions are used to define a piece of code that can be used multiple times without redefining it
- Begin with a **def** statement
- Continue with the desired function name
- After the name, insert brackets that may include arguments
- End with a colon
- The lines under the colon need to be indented
  - Only indented code is performed inside the function
- To use a previously defined function, write the function name and brackets with or without arguments

```python
def Function1():
    print('First Function')
```




```
In [16]: Function1()
First Function
```

Functions II – Defining with Args

- Functions become useful when used with arguments
- Provide arguments inside the brackets
- Arguments can be used inside the function
- The **return** statement allows for the assignment of function results to variables

```
def Function2(x):
    result = x * 25
    return result
```

- When using defined functions with arguments, write function name with brackets, and the desired argument inside the brackets
- If the function ends with **return**, it is possible to assign the result to a variable

```
In [20]: Function2(2)
Out[20]: 50

In [21]: Z = Function2(10)

In [22]: print(Z)
250
```

Exercises III

1. Define two functions

1.1 One that takes a list of numbers as argument and returns the sum of all (**my_list=[2, 3, 4]** should return **9**)

1.2 One that takes a list of numbers as argument and returns the product of all

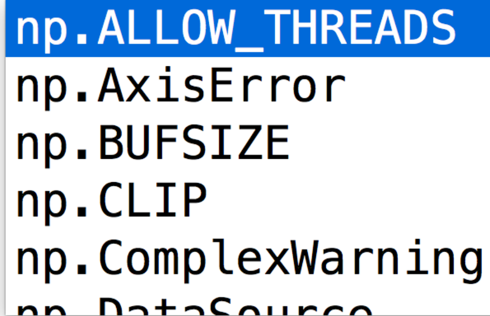
2. Define a function that checks whether an element occurs in a list

3. Define a function that takes a list of words and returns the length of the longest one

Reminder: Objects

- Save or define the object in a variable
- To access the functions and properties of an object, write a **dot** after the name

```
In [1]: import numpy as np

In [2]: np.
```

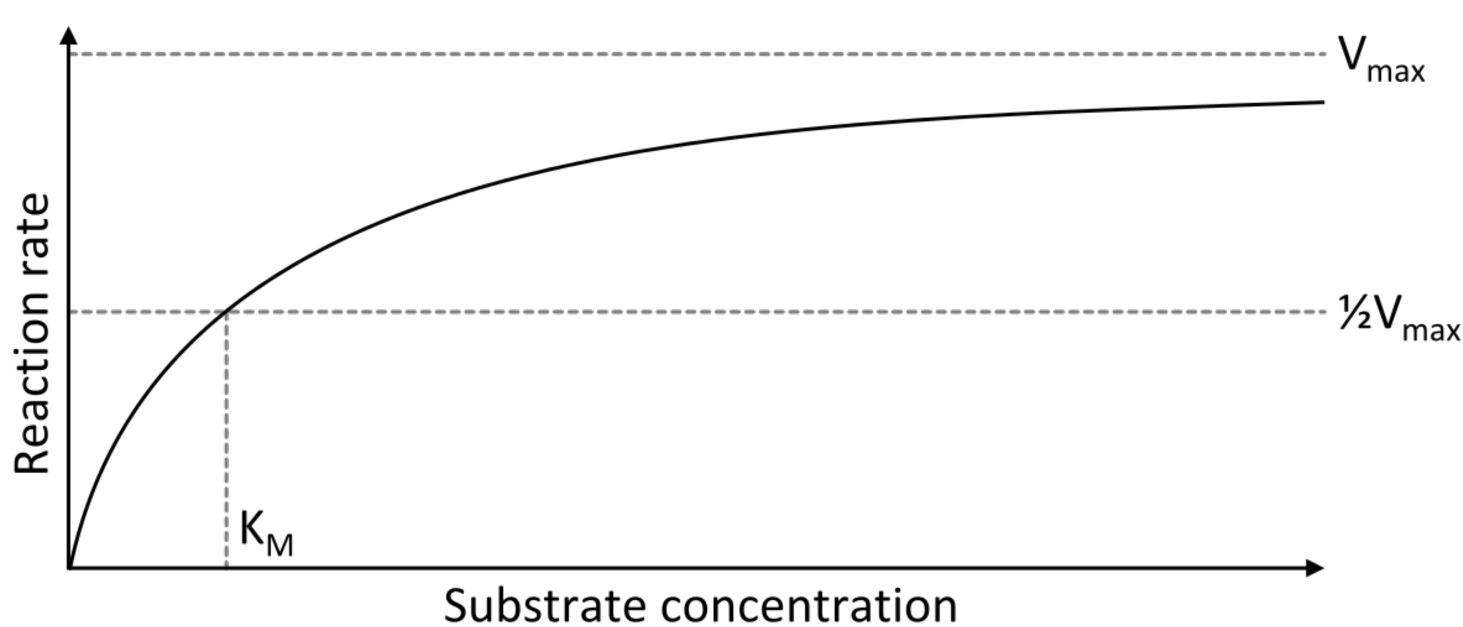


Importing useful packages

- Python packages can be imported
- Objects that include many preprogrammed objects and functions
  - Access the properties of an object with a dot!
- Very helpful functions that do not need to be self defined
- Important example: **NumPy**

```
In [1]: import numpy as np

In [2]: np.linspace(0.,5.,10)
Out[2]:
array([0.        , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
       2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.        ])

In [3]: [1,2,3]+[1,2,3]
Out[3]: [1, 2, 3, 1, 2, 3]

In [4]: np.array([1,2,3])+np.array([1,2,3])
Out[4]: array([2, 4, 6])
```

Michaelis Menten Kinetics



$$V_0 = \frac{V_{max}[S]}{(K_M + [S])}$$

First Program – Michaelis Menten I

- Define a function that
  - Calculates the reaction rate of a Michaelis Menten reaction
  - Arguments: Substrate concentration (mmol), $\mathbf{V_{max}}$(mmol/h) and **Km**(mmol)

- Calculate the reaction rate for Substrate concentrations between 0 and 50 mmol ($\mathbf{V_{max}}$ of 0.2 mmol/s and **Km** of 1.5 mmol)
  - Store the solutions in a list!

Visualize with Matplotlib

- Visualize your results as graphs (and more)
- Import **matplotlib.pyplot**
- Function **plot( )** plots two lists of same length
  - First argument is the list for X axis values
  - Second argument is the list for Y axis values
- Function **show( )** displays the graph

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

x=np.linspace(0.,100.,100)
y=np.linspace(0.,10.,100)

plt.plot(x,y)
plt.show()
```

First Program – Michaelis Menten II

- Visualize the results from the previous task
  - Look into the documentation to label the x and y axis
- Increase/Decrease the $\mathbf{V_{max}}$ and **Km** values and calculate again
  - Visualize and compare the results in one plot
  - Look into the documentation to label different graphs in one plot

Differential equations and programming

- Instead of solving ODEs analytically we solve them numerically
- Step by step solving using time steps and initial conditions
- Takes a long time to do by hand – program computers to perform simulations
- Write programs that calculate ODEs over time

- Examples: Simplified bacterial growth,
  Lotka Volterra

$$\frac{\mathrm{d}X}{\mathrm{d}t} = \mu X$$

Import the necessary packages

- The modelbase package includes all necessary function for the implementation of ODE models
- The numpy package includes diverse functions from vector and matrix calculations to list generation
- The matplotlib package includes useful functions for visualization of graphs

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

1. Parameter definition and model initialisation

- Define the model parameters in a dictionary (here **p**)
- Define a variable that will include the model object (here **m**)
- Define the model object with an instantiation of a modelbase Model
  - The instantiation needs the parameter dictionary as an argument

```
17 p = {'g':0.6}
18
19 m = modelbase.Model(p)
```

2. Define the variables (compounds/species)

- Define a list containing the names of the model variables as strings
- Execute the modelbase function ***set_cpds()*** to set the variables into the model object

```
22 species = ['B']
23
24 m.set_cpds(species)
```

3. Define the rate equations for reactions

- Define a function that takes the parameter dictionary as first argument and the involved variables as following arguments
- The function should return the rate
- The parameters inside the functions will be called from a modelbase parameter object
  - Therefore access the parameters via the dictionary name and a dot

```
28 def Growth(p,B):
29     return p.g * B
```

4. Set the rates/reactions

- Every ODE consists of rate equations
  - In simple bacterial growth, there is only one ODE including one rate equation
- Define a rate to the modelbase object by using the function ***set_rate*** and providing the arguments of
  - name of the rate as a string
  - corresponding Python function
  - species involved in the rate equation as strings

```
32 m.set_rate('Growth',Growth,'B')
```

5. Set the stoichiometry

- Rate equations affect one or more variables (e.g. due to conversion).
  - Their effect is defined by the rate itself and the stoichiometric coefficient of the rate/reaction
- Define the stoichiometric coefficients of a rate/reaction with the modelbase function **set_stoichiometry()**
  - Provide the rate/reaction name as a string, and a dictionary of species names with the corresponding stoichiometric coefficients

```
34 m.set_stoichiometry('Growth', {'B':1})
```

6. Instantiate a simulation object

- Use the model object that is now ready to initiate a new object which can perform simulations
- Define a new variable name (here s) and instantiate a Simulator object by providing the model object as an argument

```
37 s = modelbase.Simulator(m)
```

7. Define the Time and the initial conditions

- In order to perform a ODE simulation, we need the time to integrate over, as well as the starting points of our simulation (initial values or initial conditions)
- Define the time in a variable (here **T**) as a list of time points (suggested: ***np.linspace()***)
- In the case of only one ODE, define a initial value variable

```
40 T = np.linspace(0.,10.,100)
41 initial_values = 1.
```

8. Perform a simulation!

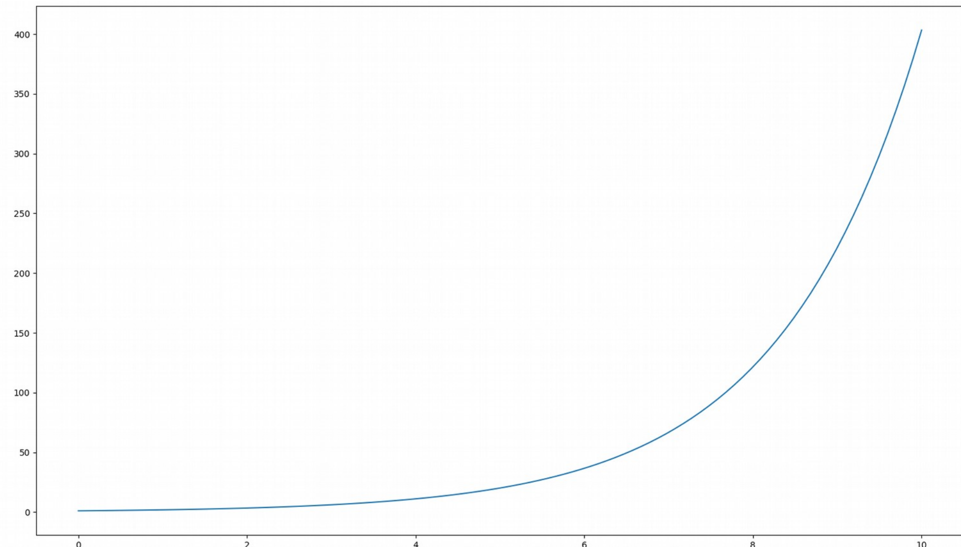
- To perform a simulation over time, execute the ***timeCourse()*** function of the simulator object with the time span and the initial conditions as arguments

```
44 s.timeCourse(T, initial_values)
```

9. Accessing and plotting the results

- The results of the simulation are a list of species quantities for each time point in the time list
- The simulation results are captured in the simulator object
  - can be accessed by the **getY()** function
- The results can now be plotted against the time

```
47 results = s.getY()
48
49 plt.plot(T,results)
50
51 plt.show()
```



Example 2: Lotka Volterra

- Initiate a parameter dictionary and a list of species names
- Instantiate the model object and set the species names with ***set_cpd()***

```
62 species = ['R', 'F']
63
64 p = par={'g1':15.2, 'd1':1.3,
65          'g2':0.05, 'd2':0.4}
66
67 m = modelbase.Model(p)
68
69 m.set_cpds(species)
```

Rate equations and stoichiometric coefficients

- The rate that increases prey population is dependent on prey and increases prey (stoichiometry is positive)

```
def Rabbit_G(p,R):
    return p.g1 * R
m.set_rate('Rabbit_G',Rabbit_G,'R')
m.set_stoichiometry('Rabbit_G', {'R':1})
```

- The rate that decreases prey population is dependent on prey and predators and decreases prey (stoichiometry is negative)

```
def Rabbit_D(p,R,F):
    return p.d1 * R*F
m.set_rate('Rabbit_D',Rabbit_D,'R','F')
m.set_stoichiometry('Rabbit_D', {'R':-1})
```

Rate equations and stoichiometric coefficients

- The rate that increases predator population is dependent on prey and predators and increases predator # (stoichiometry is positive)

```
def Fox_G(p,R,F):
    return p.g2 * F*R
m.set_rate('Fox_G',Fox_G,'R','F')
m.set_stoichiometry('Fox_G', {'F':1})
```

- The rate that decreases predator population is dependent on predators and decreases predator # (stoichiometry is negative)

```
def Fox_D(p,F):
    return p.d2 * F
m.set_rate('Fox_D',Fox_D,'F')
m.set_stoichiometry('Fox_D', {'F':-1})
```

List of initial values

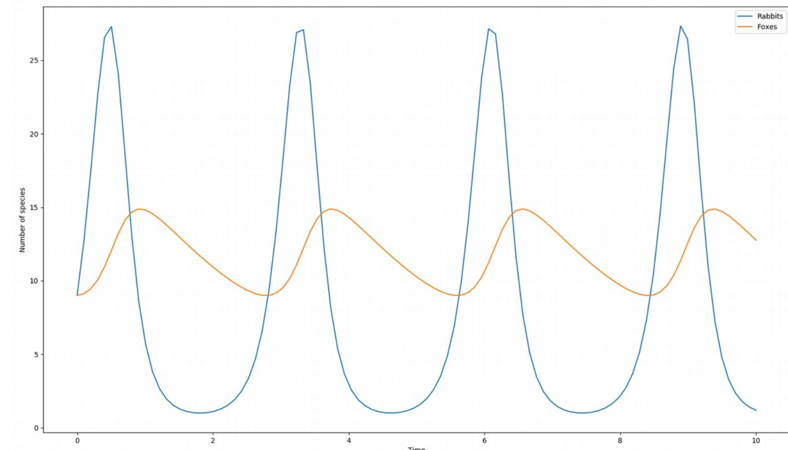
- More than one ODE:
  - Provide **initial_values** as a list
  - Same order as in species definition!

```
 95 s = modelbase.Simulator(m)
 96
 97 T = np.linspace(0.,10.,100)
 98
 99 initial_values = [9.,9.]
100
101 s.timeCourse(T, initial_values)
```

Visualization of results

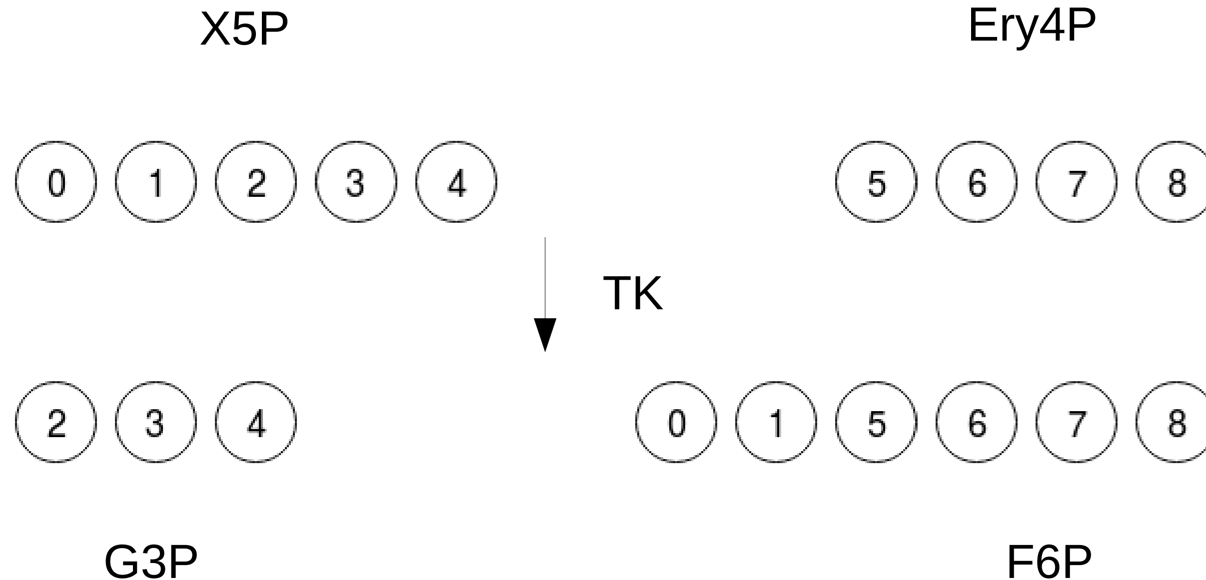
- The result **s.getY()** is now a list of lists containing the corresponding species quantities for every time point
- The list of results for one species can be accessed with **[:,i]** after the result-list name and with the corresponding index (**i**)
- **plt.xlabel()** and **plt.ylabel()** are functions that take a string and that label the axes of a plot

```
104 results = s.getY()
105
106 plt.plot(T,results[:,0],label='Rabbits')
107
108 plt.plot(T,results[:,1],label='Foxes')
109
110 plt.legend(loc='best')
111 plt.xlabel('Time')
112 plt.ylabel('Number of species')
113 plt.show()
```



Carbon Label Models

- Implementation of carbon label models



Carbon Label Models

- Construction of isotope-label specific models
- Assign **LabelModel()** class
- Setting parameters

```
m = modelbase.LabelModel()
```

Carbon Label Models

- Adding compounds and respective numbers of carbon atoms
- Definition of forward and backward reaction

```
m.add_base_cpd('X5P',5)
m.add_base_cpd('Ery4P',4)
m.add_base_cpd('F6P',6)
m.add_base_cpd('G3P',3)
```

Name

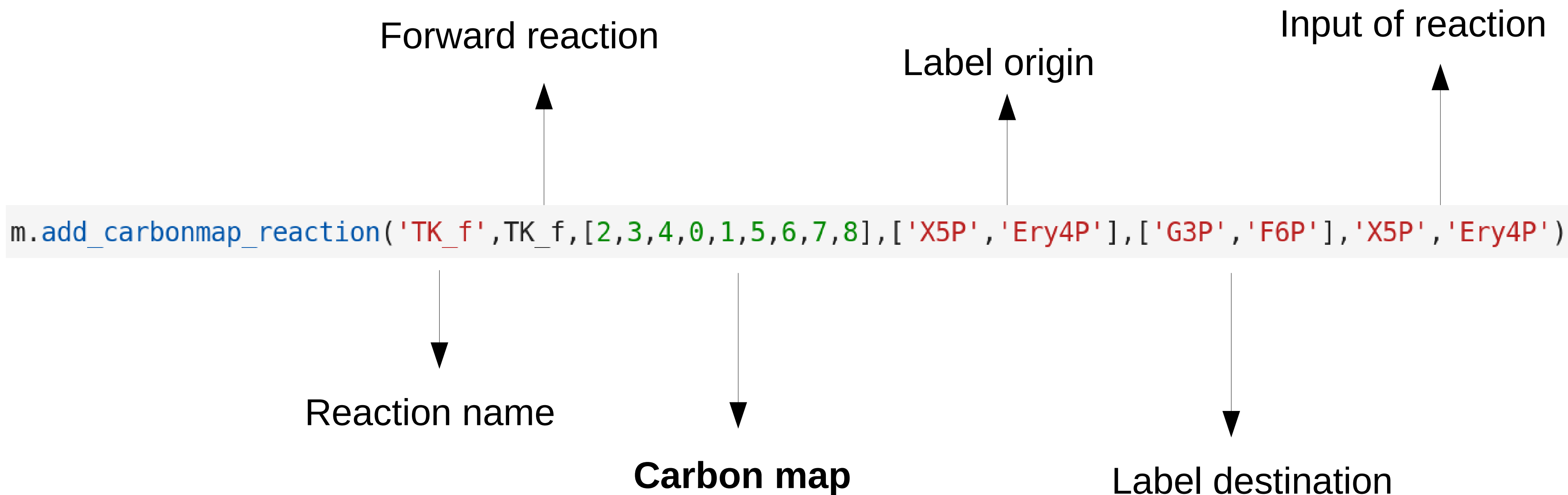
Number of carbon atoms

```
def TK_f(p,y,z):
    return p.k_TK_f*y*z
```

```
def TK_r(p,y,z):
    return (p.TK_K/p.k_TK_f)*y*z
```

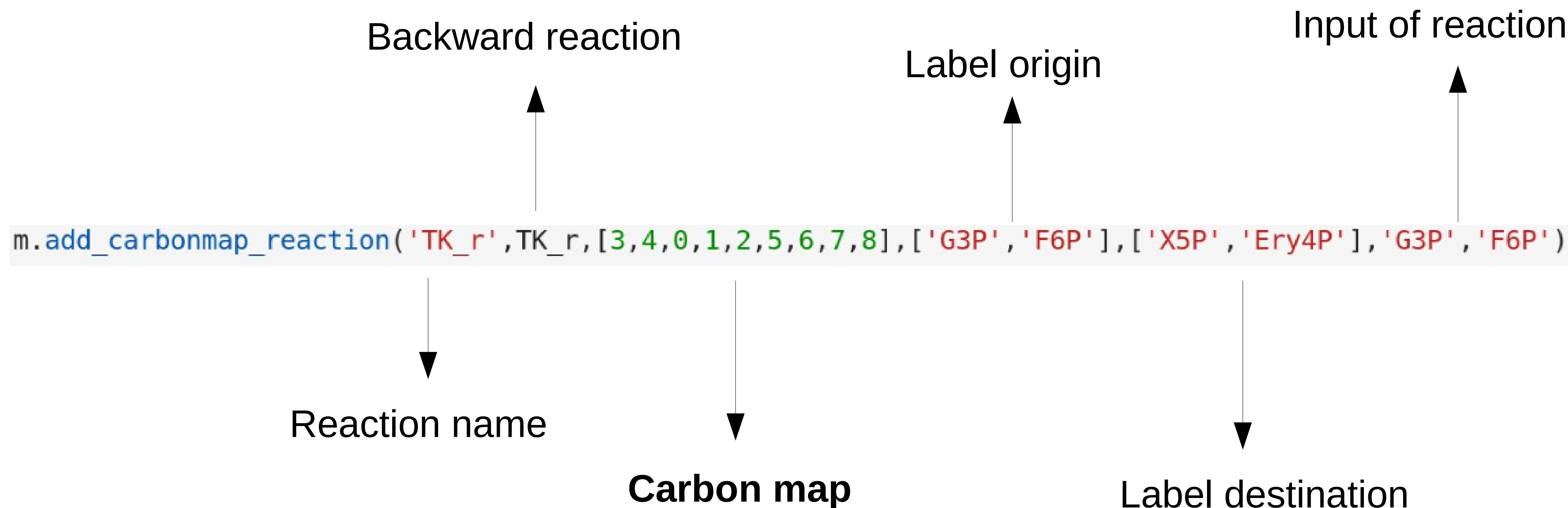
Carbon Label Models

- Add carbonmap reaction of forward reaction



Carbon Label Models

- Add carbonmap reaction of backward reaction



Carbon Label Models

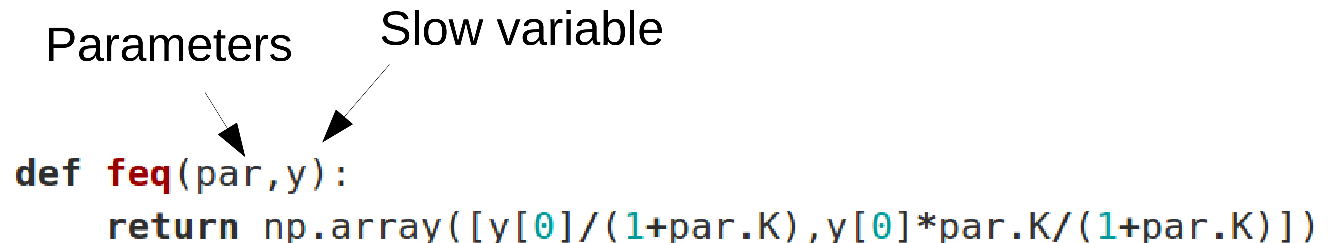
- Add initial concentrations and label positions
- Simulation over time

```
y0d = {'G3P': 1e-6,
       'X5P': 1e-6,
       'Ery4P': 1e-6,
       'F6P': 1e-6}

y0 = m.set_initconc_cpd_labelpos(y0d,labelpos={'X5P':3})
s = modelbase.Simulator(m)
T = np.linspace(0,200,100)
s.timeCourse(T,y0)
```

Fast equilibrium calculations (Algebraic Modules)

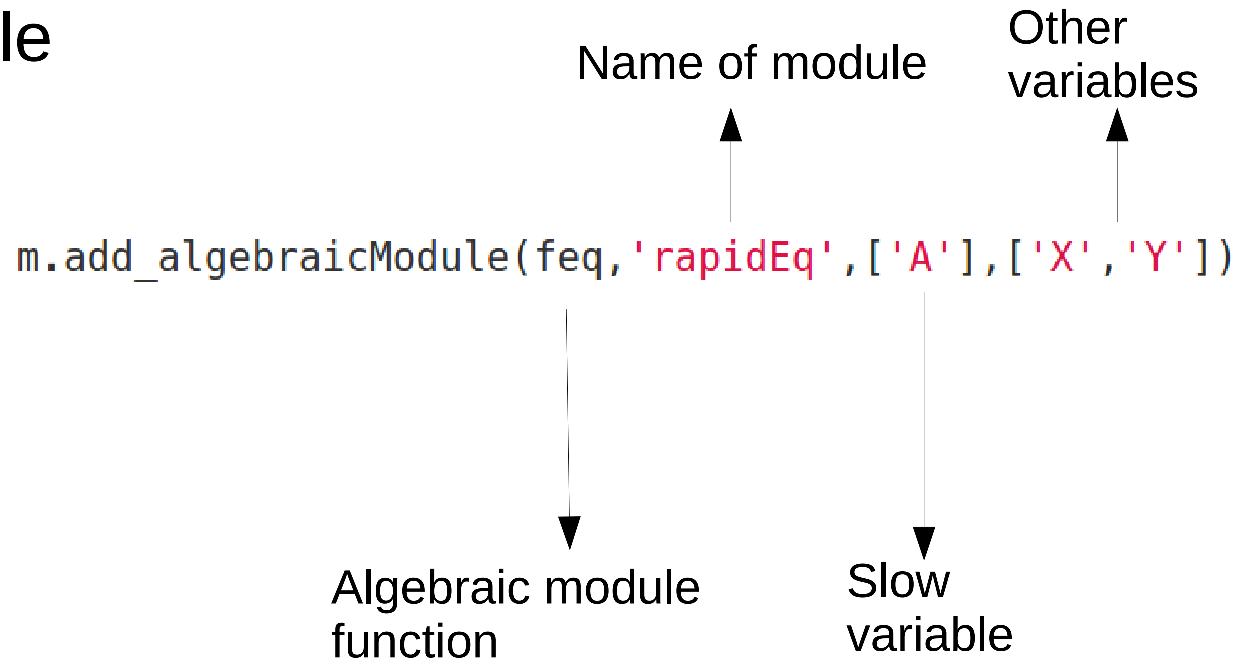
- Define the “slow variable” and parameters
- Define equilibrium function for algebraic module

```
cl = ['A']
p = {'v0':1,'k2':0.1, 'K':5}

m = modelbase.Model(p)

m.set_cpds(cl)
```



Fast equilibrium calculations (Algebraic Modules)

- Adding the algebraic module to the model



Fast equilibrium calculations (Algebraic Modules)

- Introducing influx and outflux reactions for slow variable
- Simulation over time

```
# constant influx to the pool A
m.set_rate('v0',lambda p:p.v0)
m.set_stoichiometry('v0',{'A':1})

# mass-action outflux from the pool A
# note that rate expression depends on variable Y!
def v2(p,y):
    return p.k2*y

m.set_rate('v2',v2,'Y')
m.set_stoichiometry('v2',{'A':-1})

s = modelbase.Simulator(m)
s.timeCourse(np.linspace(0,100,1000),np.zeros(1))
```