Introduction to

Falko Krause

Theoretical Biophysics

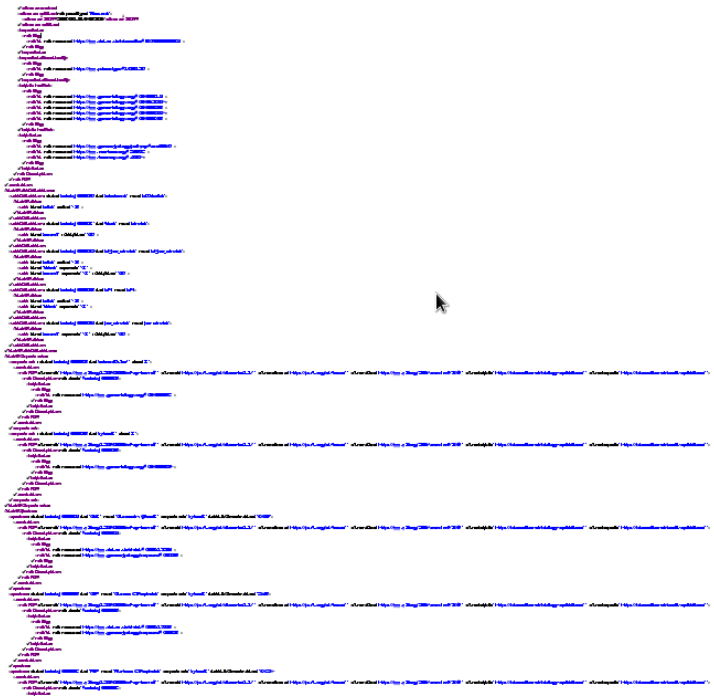
Are you ready?

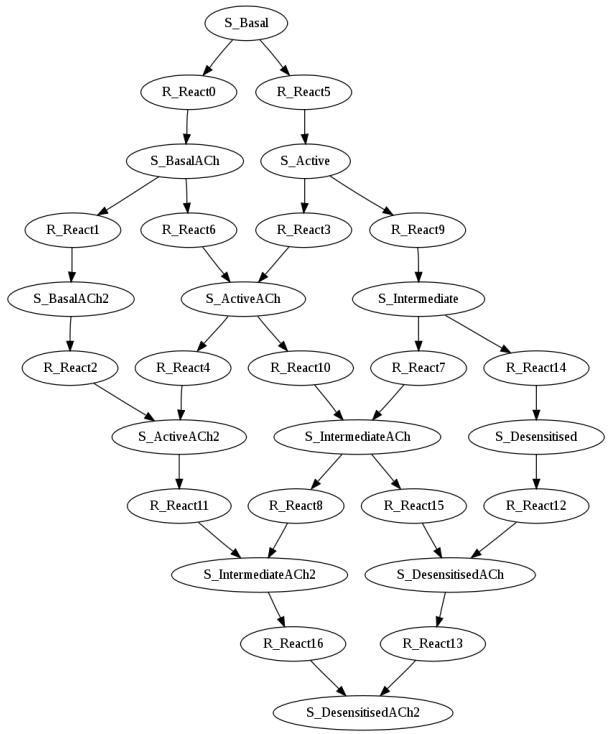
- Theoretical Biophysics - Bioinformatics
- Systems Biology - SBML
- Beginners - switching from Java / C++ / Pearl / PHP ...
- Basic Linux Knowledge
- 90 min + 90 min Exercise + 22 Page Tutorial
- http://docs.python.org/tutorial/

Can you do this?

HUMBOLDT-UNIVERSITÄT ZU BERLIN

SBML File → Graphical Representation

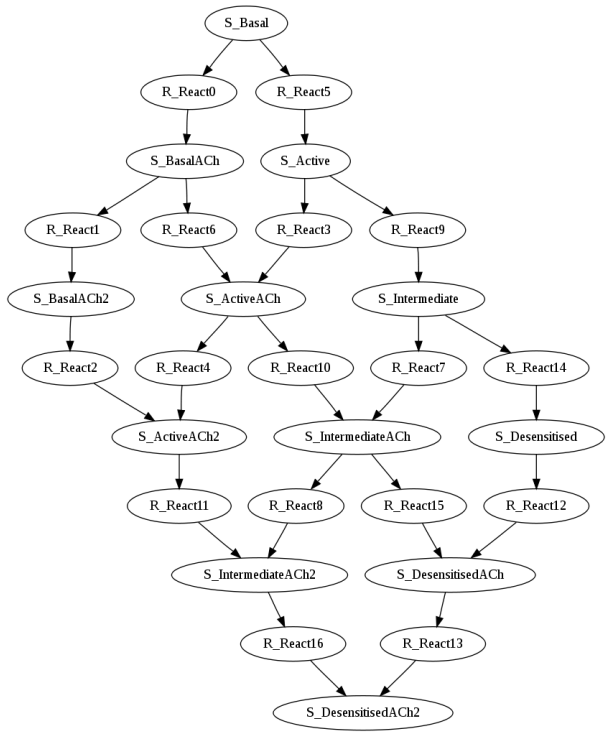




Can you do this?

HUMBOLDT-UNIVERSITÄT ZU BERLIN

SBML File → Graphical Representation



$<$ 100 lines fully documented source-code including a basic user interface

Software

The tutorial will make use of the software: Python, IPython. The software used is freely available.

Python

http://www.python.org/

IPython

http://ipython.scipy.org
for Windows please have a look at
http://ipython.scipy.org/moin/IpythonOnWindows

Interactive Mode

The command line interpreter (or Python interactive shell)

```
$ python
Python 2.5.2 (r252:60911, May 7 2008, 15:19:09)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Interactive Mode

HUMBOLDT-UNIVERSITÄT ZU BERLIN

The command line interpreter (or Python interactive shell)

```
$ python
Python 2.5.2 (r252:60911, May 7 2008, 15:19:09)
[GCC 4.2.3 (Ubuntu 4.2.3-2ubuntu7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

primary prompt >>>

secondary prompt ...

```
>>> myflag = 1
>>> if myflag:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

Two Important Concepts

Duck Typing

"*If it walks like a duck and quacks like a duck, I would call it a duck.*"

Two Important Concepts

Duck Typing

"If it walks like a duck and quacks like a duck, I would call it a duck."

```
1 myint = 1 # myint is now of type integer
2 mystring = 'free beer' # mystrting is now of type string
```

Two Important Concepts

Duck Typing

"If it walks like a duck and quacks like a duck, I would call it a duck."

```
1 myint = 1 # myint is now of type integer
2 mystring = 'free beer' # mystrting is now of type string
```

Indentation

Indentation determines the context of commands.

Two Important Concepts

Duck Typing

"If it walks like a duck and quacks like a duck, I would call it a duck."

```
1 myint = 1 # myint is now of type integer
2 mystring = 'free beer' # mystrting is now of type string
```

Indentation

Indentation determines the context of commands.

```
1 if flag==True:
2     print 'this is only printed if the flag is True'
3 print 'this is always printed'
```

Numbers

Integer

Use the Python interactive shell (or IPython) as a calculator

```
1 >>> 2+2
2 4
3 >>> (50-5*6)/4
4 5
```

Numbers

Integer

Use the Python interactive shell (or IPython) as a calculator

```
>>> 2+2
4
>>> (50-5*6)/4
5
```

Assign a number to a variable

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Numbers

Float

Floating point numbers (`float`).

```
1 >>> 3 * 3.75 / 1.5
2 7.5
```

Numbers

Float

Floating point numbers (`float`).

```
1 >>> 3 * 3.75 / 1.5
2 7.5
```

Convert an `int` into a `float`

```
1 >>> float(width)
2 20.0
```

This kind of type casting works for most datatypes in Python (!).

Strings

Single Quotes / Double Quotes


```
1 >>> 'spam eggs'
2 'spam eggs'
```

Single quotes do not interpret the contents

Strings

Single Quotes / Double Quotes


```
>>> 'spam eggs'
'spam eggs'
```

Single quotes do not interpret the contents
Double quotes do

```
>>> hello = "This is a rather long string containing\n\
... several lines of text just as you would do in C.\n\
...     Note that whitespace at the beginning of the line is\n\
...  significant."
>>>
>>> print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

Strings

Single Quotes / Double Quotes


```
>>> 'spam eggs'
'spam eggs'
```

Single quotes do not interpret the contents
Double quotes do

```
>>> hello = "This is a rather long string containing\n\
... several lines of text just as you would do in C.\n\
...     Note that whitespace at the beginning of the line is\n\
...  significant."
>>>
>>> print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is significant.
```

\ the same command continues on the next line

Strings

Triple Quotes

Multi line strings(''' or """)

```
>>> hello = '''This is a rather long string containing
... several lines of text just as you would do in C.
...     Note that whitespace at the beginning of the line is
... significant.'''
>>> print hello
This is a rather long string containing
several lines of text just as you would do in C.
    Note that whitespace at the beginning of the line is
significant.
```

Concatenatination

HUMBOLDT-UNIVERSITÄT ZU BERLIN

Concatenating strings

```
1 >>> word = 'Help' + 'A'
2 >>> word
3 'HelpA'
```

IPython will help you!

In IPython you can see all the string functions by tabbing them

```
In [1]: word = 'Help' + 'A'

In [2]: word
Out[2]: 'HelpA'

In [3]: word.<TAB>
In [2]: word.
word.__add__          word.__reduce_ex__   word.join
word.__class__        word.__repr__        word.ljust
word.__contains__     word.__rmod__        word.lower
...
word.__ne__           word.isspace         word.upper
word.__new__          word.istitle         word.zfill
word.__reduce__       word.isupper

In [4]: word.upper()
Out[4]: 'HELPA'

In [5]: wor<UP>
```

tuple

The most basic list type is the `tuple`.

```
1  >>> t = 12345, 54321, 'hello!'
2  >>> t
3  (12345, 54321, 'hello!')
4  >>> # Tuples may be nested:
5  ... u = t, (1, 2, 3, 4, 5)
6  >>> u
7  ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

list

A normal list is called `list`.

```
1 >>> l = ['spam', 'eggs', 100, 1234]
2 >>> l
3 ['spam', 'eggs', 100, 1234]
```

list

A normal list is called `list`.

```
1 >>> l = ['spam', 'eggs', 100, 1234]
2 >>> l
3 ['spam', 'eggs', 100, 1234]
```

This is the closest to what is known as “array” in other programming laguages.

set

A `set` is practical for finding members

```
1  >>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
2  >>> s = set(basket)               # create a set without duplicates
3  >>> s
4  set(['orange', 'pear', 'apple', 'banana'])
5  >>> 'orange' in s                 # fast membership testing
6  True
7  >>> 'crabgrass' in s
8  False
```

It will not store duplicate entries.

You can also use functions like `union`, `difference` etc. to create new `sets`.

dict

A dictionary (`dict`) contains key / value pairs

{ key:value , key:value }

```
1 >>> d = {'jannis': 4098, 'wolf': 4139}
2 >>> d['jannis']
3 4098
4 >>> d['guido'] = 4127
5 >>> d
6 {'wolf': 4139, 'guido': 4127, 'jannis': 4098}
```

dict

HUMBOLDT-UNIVERSITÄT ZU BERLIN

A dictionary (`dict`) contains key / value pairs

{ key:value , key:value }

```
1 >>> d = {'jannis': 4098, 'wolf': 4139}
2 >>> d['jannis']
3 4098
4 >>> d['guido'] = 4127
5 >>> d
6 {'wolf': 4139, 'guido': 4127, 'jannis': 4098}
```

The keys form a `set`

Sequence Keys

HUMBOLDT-UNIVERSITÄT ZU BERLIN

In `lists` and `tuples`, element positions are the “keys”.

```
1 >>> t = 12345, 54321, 'hello!'
2 >>>t[0]
3 12345
```

Sequence Keys

In `lists` and `tuples`, element positions are the "keys".

```
1 >>> t = 12345, 54321, 'hello!'
2 >>>t[0]
3 12345
```

Extracting subsequences: *list[start:end]*

```
1 >>> word = 'WOOT this Python lesson is awesome'
2 >>> word.split()
3 ['WOOT', 'this', 'Python', 'lesson', 'is', 'awesome']
4 >>> word[10:17]+word.split()[4]+word[-7:]
5 'Python is awesome'
```

Empty values	start of the list or end of the list
Negative values	subtracted from the length of the list (−1 the last element of the list)

Other Important Datatypes

`bool` Values: `True`, `False`

Other Important Datatypes

`bool`	Values: `True`, `False`	
	`[]`	`False`
	`['a','b']`	`True`
	0	`False`
	all other	`True`

Other Important Datatypes

`bool`	Values: `True`, `False`	
	`[]`	`False`
	`['a','b']`	`True`
	`0`	`False`
	all other	`True`
`None`	Value: `None`	
	frequently used to represent the absence of a value	

while Statements

HUMBOLDT-UNIVERSITÄT ZU BERLIN


```
1 >>> a, b = 0, 1 #multiple assignment
2 >>> while b < 1000:
3 ...     print b, # , prevents new line
4 ...     a, b = b, a+b
5 ...
6 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

If the boolean (`bool`) statement is `True`
The indented code below the `wile` is executed

if Statements

Chaninging if statements

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

for Statements

HUMBOLDT-UNIVERSITÄT ZU BERLIN

Looping through lists str,list,tuple,set

```
>>> # Measure some strings:
... a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
```

The range() Function

Generating lists of numbers

```
1 >>> range(10)
2 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3 >>> range(5, 10)
4 [5, 6, 7, 8, 9]
5 >>> range(0, 10, 3)
6 [0, 3, 6, 9]
7 >>> range(-10, -100, -30)
8 [-10, -40, -70]
```

The range() Function

Generating lists of numbers

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

list indices

```
>>> a = ['Mary', 'had', 'a', 'little']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
```

break and continue Statements, and else Clauses on Loops


```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break # break out of the smallest enclosing loop
...     else:# executed when the loop terminates through exhaustion (or ...
 ...when the condition becomes false for while)
...         if n==3:
...             continue # skip to the next iteration of the loop
...         print n, 'is a prime number'
...
2 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

pass Statements

It will do nothing.

```
>>> x = int(raw_input('Please enter an integer: '))
Please enter an integer: 10
>>> if x < 0:
...     pass
... elif x == 42:
...     pass #TODO must fill the answer to life, the universe, and ...
 ...everything here later
... else:
...     print 'More'
...
More
```

Function Definition Syntax


```
>>> def sagMiau(who):
...     return who+' sagt Miauuuuu'
...
>>> print sagMiau('Jannis')
Jannis sagt Miauuuuu
```

Default Argument Values and Keyword Arguments


```
>>> def sagKompliment(who,person='Falko',antwort='Oh danke'):
...     return who+' sagt: '+person+' du hast die Haare schoen.\\n'+...
 ...person+' sagt: '+antwort
...
>>> print sagKompliment('Jannis','Wolf')
Jannis sagt: Wolf du hast die Haare schoen.
Wolf sagt: Oh danke
>>> print sagKompliment('Timo',antwort='Verarschen kann ich mich selber...
 ...')
Timo sagt: Falko du hast die Haare schoen.
Falko sagt: Verarschen kann ich mich selber
```

Default Argument Values and Keyword Arguments


```
>>> def sagKompliment(who,person='Falko',antwort='Oh danke'):
...     return who+' sagt: '+person+' du hast die Haare schoen.\\n'+...
 ...person+' sagt: '+antwort
...
>>> print sagKompliment('Jannis','Wolf')
Jannis sagt: Wolf du hast die Haare schoen.
Wolf sagt: Oh danke
>>> print sagKompliment('Timo',antwort='Verarschen kann ich mich selber...
 ...')
Timo sagt: Falko du hast die Haare schoen.
Falko sagt: Verarschen kann ich mich selber
```

This is very useful for functions that have many arguments with default values of which you only need to use a few.

Function as Datatype

Functions are not very different than other datatypes.

```
10 >>> kmplmnt=sagKompliment
11 >>> print kmplmnt('Falko')
12 Falko sagt: Falko du hast die Haare schoen.
13 Falko sagt: Oh danke
```

Documentation Strings

Python's built in method of documenting source-code.

```
>>> def my_function():
...     '''Do nothing, but document it.
...
...     No, really, it doesnt do anything.
...     '''
...     pass
...
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesnt do anything.
```

Documentation Strings

IPython


```
In [1]: def my_function():
   ...:     ''' the same here '''
   ...:     pass
   ...:

In [2]: my_function?
Type:           function
Base Class:     <type 'function'>
String Form:    <function my_function at 0x83f4f7c>
Namespace:      Interactive
File:           /home/me/<ipython console>
Definition:     my_function()
Docstring:
    the same here

In [3]: str?
Type:           type
Base Class:     <type 'type'>
String Form:    <type 'str'>
...
```

List Comprehensions

Manipulate a list on the fly

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
```

Writing a Module

A file containing Python source-code is called a module

`fibo.py`:

```
'''
Fibonacci numbers module
'''

def fib(n):   # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Importing a Module

To use `fibo.py` we can `import` *modulename* (without the .py extension)

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Executing Modules as Scripts

To execute the module with

```
$ python fibo.py <arguments>
```

We add

```
1 if __name__ == '__main__':
2     import sys
3     fib(int(sys.argv[1]))
```

Now we can run

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Executing Modules as Scripts

Make the file directly executable by adding (as the first line)

```
#!/usr/bin/env python
```

And setting the file as executable

```
$ chmod +x fibo.py
$ mv fibo.py fibo
$ ./fibo 50
1 1 2 3 5 8 13 21 34
```

If move to `/usr/bin` it can be executed from any location the filesystem

Formatted Strings

A string containing %<*someletter*> (or %<*number*><*someletter*>) followed by a % and as many variables/values (in a `tuple`) as % signs

```
>>> b = 'hello'
>>> a = '!'
>>> c = 'world'
>>> print '%s %s %s'%(b,c,a)
hello world !
>>> print '%20s'%b
               hello
>>> print '%-20s%s'%(b,a)
hello                !
>>> x = 1.23456789
>>> print '%e | %f | %g' % (x, x, x)
1.234568e+00 | 1.234568 | 1.23457
>>> print '%4d'%10
  10
>>> print '%.4d'%10
0010
```

Reading a File

'r' read, 'w' write, 'rw' read and write, 'a' append (like write, but append to the end of the file)

```
1 >>> f=open('/etc/issue', 'r')
2 >>> f.read()
3 'Ubuntu 8.10 \\n \\l\n\n'
4 >>> f.close()
```

Reading a File

'r' read, 'w' write, 'rw' read and write, 'a' append (like write, but append to the end of the file)

```
1 >>> f=open('/etc/issue', 'r')
2 >>> f.read()
3 'Ubuntu 8.10 \\n \\l\n\n'
4 >>> f.close()
```

`read`	read the whole file into a string
`readline`	read the file line by line
`readlines`	read the file into a `list`

The pickle Module

"Serialization is the process of saving an object onto a storage medium [...] such as a file" (Wikipedia).

```
1  >>> import pickle
2  >>> x=[('man',1),(2,'this is getting'),{True:'so very',False:'...
    ...complicated'}]
3  >>> f1=open('test.picklefile','w')
4  >>> pickle.dump(x, f1)
5  >>> f1.close()
6  >>> f2=open('test.picklefile','r')
7  >>> x = pickle.load(f2)
8  >>> x
9  [('man', 1), (2, 'this is getting'), {False: 'complicated', True: 'so ...
    ...very'}]
```

Syntax Errors


```
>>> while True print 'Hello world'
  File ''<stdin>'', line 1, in ?
    while True print 'Hello world'
                   ^
SyntaxError: invalid syntax
```

Error at the keyword print: a colon (':') is missing before it

Exceptions

Some exceptions you could encounter

```
1  >>> 10 * (1/0)
2  Taceback (most recent call last):
3    File ''<stdin>'', line 1, in ?
4  ZeroDivisionError: integer division or modulo by zero
5  >>> 4 + spam*3
6  Traceback (most recent call last):
7    File ''<stdin>'', line 1, in ?
8  NameError: name 'spam' is not defined
9  >>> '2' + 2
10 Traceback (most recent call last):
11   File ''<stdin>'', line 1, in ?
12 TypeError: cannot concatenate 'str' and 'int' objects
```

Exceptions

Some exceptions you could encounter

```
1  >>> 10 * (1/0)
2  Taceback (most recent call last):
3    File ''<stdin>'', line 1, in ?
4  ZeroDivisionError: integer division or modulo by zero
5  >>> 4 + spam*3
6  Traceback (most recent call last):
7    File ''<stdin>'', line 1, in ?
8  NameError: name 'spam' is not defined
9  >>> '2' + 2
10 Traceback (most recent call last):
11   File ''<stdin>'', line 1, in ?
12 TypeError: cannot concatenate 'str' and 'int' objects
```

A Python script the script will terminate on unhandled exceptions.

Handling Exceptions

exception between in the `try` / `except` statements are caught

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print 'Oops! That was no valid number. Try again...'
...     else:
...         print 'good boy!'
...
```

optional `else` executes commands in case no exception is raised

Raising Exceptions

You can `raise` exceptions

```
1 >>> raise Exception('spam', 'eggs')
2 Traceback (most recent call last):
3   File ''<stdin>'', line 1, in <module>
4 Exception: ('spam', 'eggs')
```

Raising Exceptions

You can `raise` exceptions

```
1 >>> raise Exception('spam', 'eggs')
2 Traceback (most recent call last):
3   File ''<stdin>'', line 1, in <module>
4 Exception: ('spam', 'eggs')
```

Each exception is a class that inherits from the `Exception` base class

Class Definition Syntax

Classes are the essential concept of object-oriented programming.

```
1 >>> class MyLameClass:
2 ...     pass
```

Class Definition Syntax

Classes are the essential concept of object-oriented programming.

```
1 >>> class MyLameClass:
2 ...     pass
```

This was too easy, right?

Class Definition Syntax

Classes are the essential concept of object-oriented programming.

```
>>> class MyLameClass:
...     pass
```

This was too easy, right?

```
>>> class Animal:
...     ''' This is an animal ''' # documentation string
...     nana='nana'
...     def __init__(self,number_of_legs): # optional constructor
...         self.legs=number_of_legs # instance variable legs
...     def saySomething(self): # method
...         print 'I am an Animal, I have %s legs' % self.legs
...
```

Class Objects

Attribute reference: *obj.name*

```
9  >>> Animal.nana
10 'nana'
```

Class Objects

Attribute reference: *obj.name*

```
9  >>> Animal.nana
10 'nana'
```

Class instantiation uses the function notation

```
11 >>> my_pet = Animal(4)
```

new instance assigned to the local variable `my_pet`.

Class Objects

Attribute reference: *obj.name*

```
9  >>> Animal.nana
10 'nana'
```

Class instantiation uses the function notation

```
11 >>> my_pet = Animal(4)
```

new instance assigned to the local variable `my_pet`.

```
12 >>> my_pet.saySomething()
13 I am an Animal, I have 4 legs
```

Naming Conventions

There are two styles of writing strings in source-code that I like.

CamelCase

writing compound words or phrases in which the words are joined without spaces and are capitalized within the compound: ThisIsCamelCase

snake_case

writing compound words or phrases in which the words are joined with and underscore: this_is_snake_case

Naming Conventions

There are two styles of writing strings in source-code that I like.

CamelCase
: writing compound words or phrases in which the words are joined without spaces and are capitalized within the compound: ThisIsCamelCase

snake_case
: writing compound words or phrases in which the words are joined with and underscore: this_is_snake_case

In our project:

variables	snake_case
functions	camelCase
classes	CamelCase

Inheritance

Inheritance, a key feature of object-orientation

```
14 >>> class Cat(Animal):
15 ...     '''This is the animal cat'''
16 ...     def __init__(self):
17 ...         '''cats always have 4 legs, this is initialized in this ...
    ...function'''
18 ...             Animal.__init__(self,4)
19 ...     def petTheCat(self):
20 ...             print ''purrrrrr''
21 ...
22 >>> snuggles=Cat()
23 >>> snuggles.saySomething()
24 I am an Animal, I have 4 legs
25 >>> snuggles.petTheCat()
26 purrrrrr
27 >>>
```

Inheritance

HUMBOLDT-UNIVERSITÄT ZU BERLIN

Inheritance, a key feature of object-orientation

```
14 >>> class Cat(Animal):
15 ...     '''This is the animal cat'''
16 ...     def __init__(self):
17 ...         '''cats always have 4 legs, this is initialized in this ...
    ...function'''
18 ...             Animal.__init__(self,4)
19 ...     def petTheCat(self):
20 ...             print ''purrrrrr''
21 ...
22 >>> snuggles=Cat()
23 >>> snuggles.saySomething()
24 I am an Animal, I have 4 legs
25 >>> snuggles.petTheCat()
26 purrrrrr
27 >>>
```

multiple inheritance

```
class DerivedClassName(Base1, Base2, Base3)
```

Custom Exceptions

Creating a custom exception.

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError, 'oops!'
Traceback (most recent call last):
  File ''<stdin>'', line 1, in ?
__main__.MyError: 'oops!'
```

Additional Software

Writing a Sophisticated Bioinformatics Application: We need some extra tools - freely available and run on Linux, Windows and Os X.

libSBML

http://sbml.org/Software/libSBML (don't forget to install the Python bindings)

Graphviz

http://www.graphviz.org/

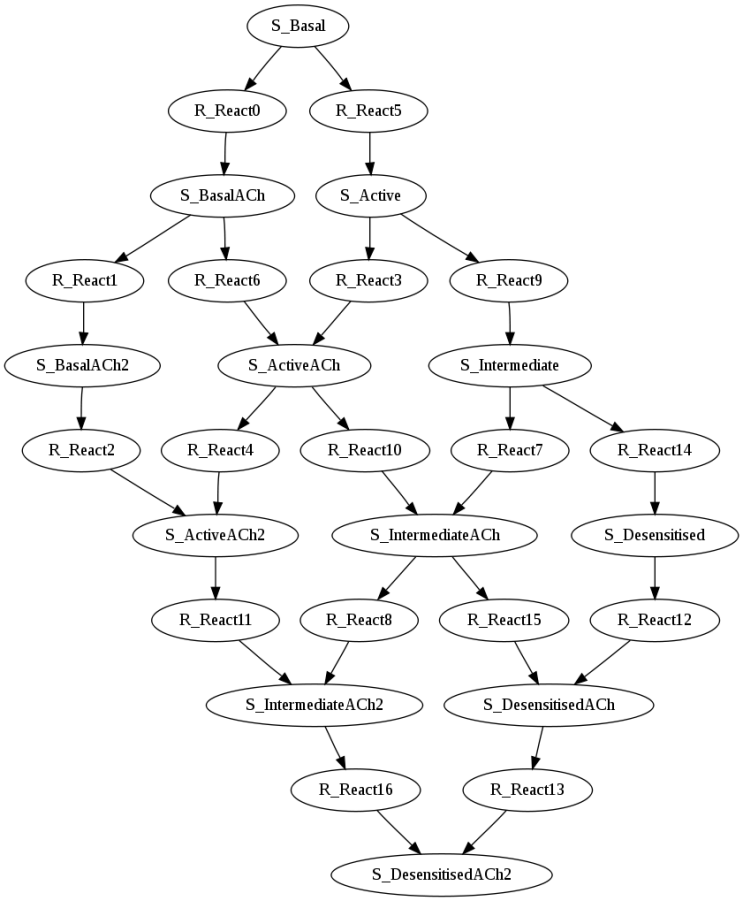
Epydoc

http://epydoc.sourceforge.net/

The Result

HUMBOLDT-UNIVERSITÄT ZU BERLIN

What our software will be able to create



The Code

Part 1

sbml_graph.py:

```
1  #!/usr/bin/env python
2  '''
3  B{SBML Graph Representation}
4  this module generates a graphical representations of SBML models
5  '''

7  import os,sys,optparse,libsbml

9  dot_path= '/usr/bin/dot' # configure the path to graphviz dot executable...
     ... here

11 class SBMLGraph():
12   '''this class enables you to create graph representations of SBML ...
       ...models'''
```

The Code

Part 2

HUMBOLDT-UNIVERSITÄT ZU BERLIN


```
13  def __init__(self,sbml_file_name):
14    '''
15    check if the sbml file exists
16    if it exists generate a graph representation
17    if not return an error message to the use and exit
18    @param sbml_file_name: path to the sbml file
19    @type sbml_file_name: str
20    '''
21    self.graph_dot=''
22    self.in_file_path=sbml_file_name
23    if not os.path.exists(self.in_file_path):
24      print 'The file %s was not found' % self.in_file_path
25      sys.exit(1)
26    else:
27      document = libsbml.readSBMLFromString(open(self.in_file_path,'r').....
        ...read())
28      model= document.getModel()
29      self.graph_dot=self.generateGraph(model)
```

The Code

Part 3


```
30  def generateGraph(self,model):
31    '''
32    @param model: libsbml model instance
33    @type model: libsbml.Model
34    @return: graph representation as string in dot format
35    @rtype: str
36    '''
37    #generate a dictionary of all species in the sbml file
38    id2libsbml_obj={}
39    for species in list(model.getListOfSpecies()):
40      id2libsbml_obj[species.getId()]=species
```

The Code

Part 4

HUMBOLDT-UNIVERSITÄT ZU BERLIN


```
41      out='digraph sbmlgraph {'
42      #go through all reactions
43      for reaction in list(model.getListOfReactions()):

45        for i in range(reaction.getNumReactants()):
46          reactant_name= id2libsbml_obj[reaction.getReactant(i).getSpecies...
              ...()].getName() or reaction.getReactant(i).getSpecies()
47          out+= 'S_%s -> R_%s' % (reactant_name,reaction.getName() or ...
              ...reaction.getId())

49        for i in range(reaction.getNumProducts()):
50          product_name= id2libsbml_obj[reaction.getProduct(i).getSpecies()...
              ...].getName() or reaction.getProduct(i).getSpecies()
51          out += 'R_%s -> S_%s' % (reaction.getName() or reaction.getId(),...
              ...product_name)
52      return out +'}'
```

The Code

Part 5

HUMBOLDT-UNIVERSITÄT ZU BERLIN


```
53  def writeImage(self,format='svg',filename=''):
54    '''
55    write the graph image to the hard disk
56    @param format: output image format
57    @type format: str
58    @param filename: filename of image
59    @type filename: str
60    '''
61    if not filename:
62      filename = os.path.splitext(os.path.basename(self.in_file_path))...
        ...[0]+'.'+format

64    open('temp.dot','w').write(self.graph_dot)
65    os.system('%s temp.dot -T%s -o %s'%(dot_path,format,filename))
66    os.remove('temp.dot')
```

The Code

Part 6

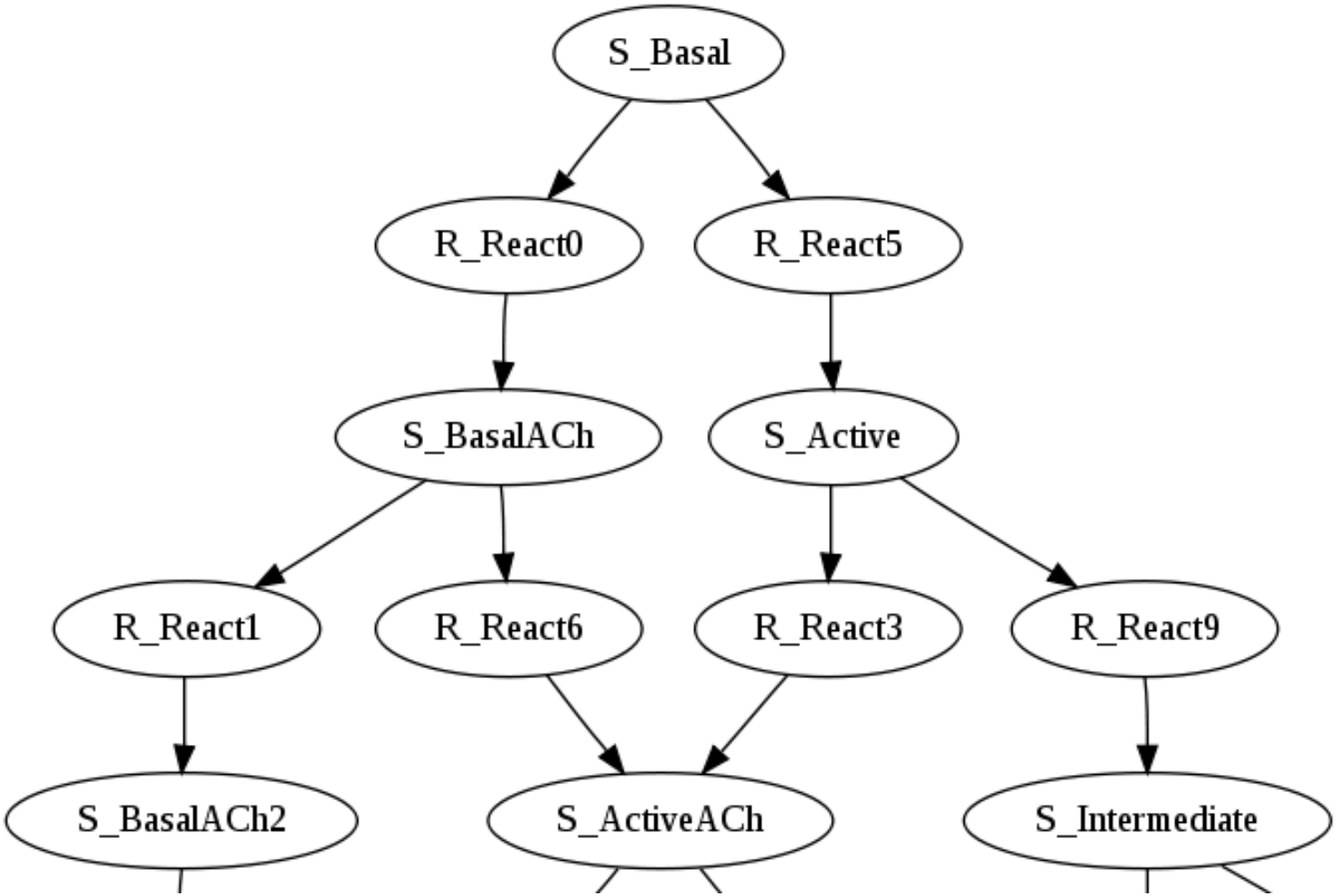

```
67  if __name__ == '__main__':

69    parser = optparse.OptionParser()
70    parser.add_option('-i', '--infile', dest='infile',\
71      help='Input: an SBML file')
72    parser.add_option('-o', '--outfile', dest='outfile', default='',\
73      help='specify a out filename, this is optional')
74    parser.add_option('-f', '--imageformat', dest='format', default='',\
75      help='output formats are: svg, png, ps, eps, tiff, bmp')
76    (options,args) = parser.parse_args()

78    if not options.infile:
79      print 'No input file specified'
80      parser.print_help()
81      sys.exit()
82    else:
83      graph=SBMLGraph(options.infile)
84      graph.writeImage(filename=options.outfile,format=options.format)
```

Executing the Script


```
$ ./sbml_graph.py -h
Usage: sbml_graph.py [options]
Options:
  -h, --help            show this help message and exit
  -i INFILE, --infile=INFILE
                        Input: an SBML file
  -o OUTFILE, --outfile=OUTFILE
                        specify a out filename, this is optional
  -f FORMAT, --imageformat=FORMAT
                        output formats are: svg, png, ps, eps, tiff, bmp
$ ./sbml_graph.py -i nofile.xml -f png
The file nofile.xml was not found
$ wget http://www.ebi.ac.uk/biomodels/models-main/publ/BIOMD0000000001....
  ...xml
...
$ ls
BIOMD0000000001.xml sbml_graph.py
$ ./sbml_graph.py -i BIOMD0000000001.xml -f png
$ ls
BIOMD0000000001.png BIOMD0000000001.xml sbml_graph.py
```

The Result

HUMBOLDT-UNIVERSITÄT ZU BERLIN



Documentation

Creation

Autogenerating a good-looking source-code documentation with Epydoc.

```
20 $ epydoc sbml_graph.py
```

Documentation

Screenshot

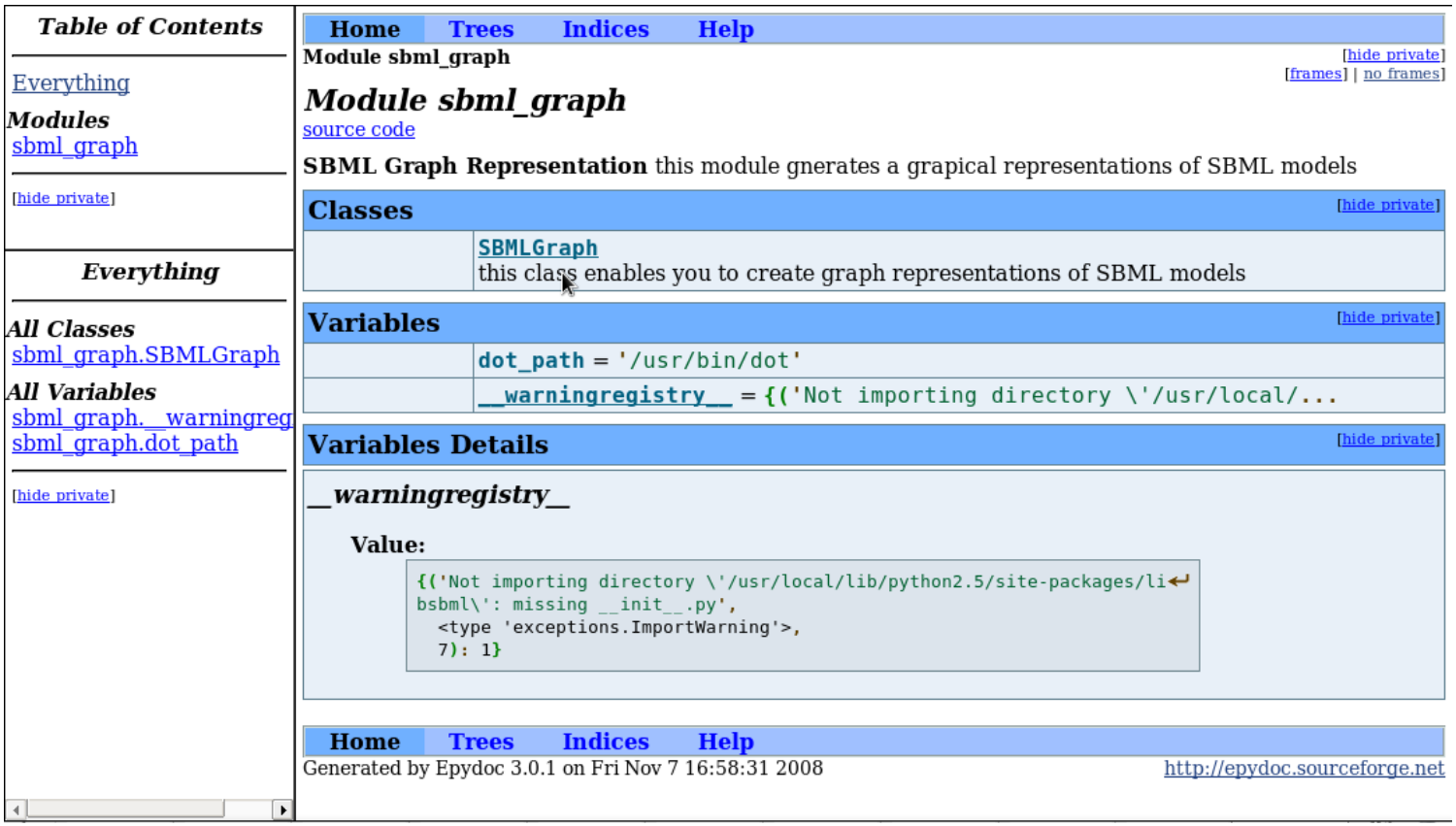


Table of Contents

Everything

Modules
sbml_graph

[hide private]

Everything

All Classes
sbml_graph.SBMLGraph

All Variables
sbml_graph.__warningreg
sbml_graph.dot_path

[hide private]

Home **Trees** **Indices** **Help**

Module sbml_graph [hide private]
[frames] | no frames]

Module sbml_graph

source code

SBML Graph Representation this module gnerates a grapical representations of SBML models

Classes [hide private]

	SBMLGraph this class enables you to create graph representations of SBML models

Variables [hide private]

	dot_path = '/usr/bin/dot'
	__warningregistry__ = {('Not importing directory \'/usr/local/...

Variables Details [hide private]

__warningregistry__

Value:

```
{('Not importing directory \'/usr/local/lib/python2.5/site-packages/li
bsbml\': missing __init__.py',
  <type 'exceptions.ImportWarning'>,
  7): 1}
```

Home **Trees** **Indices** **Help**

Generated by Epydoc 3.0.1 on Fri Nov 7 16:58:31 2008 http://epydoc.sourceforge.net

Documentation

Screenshot

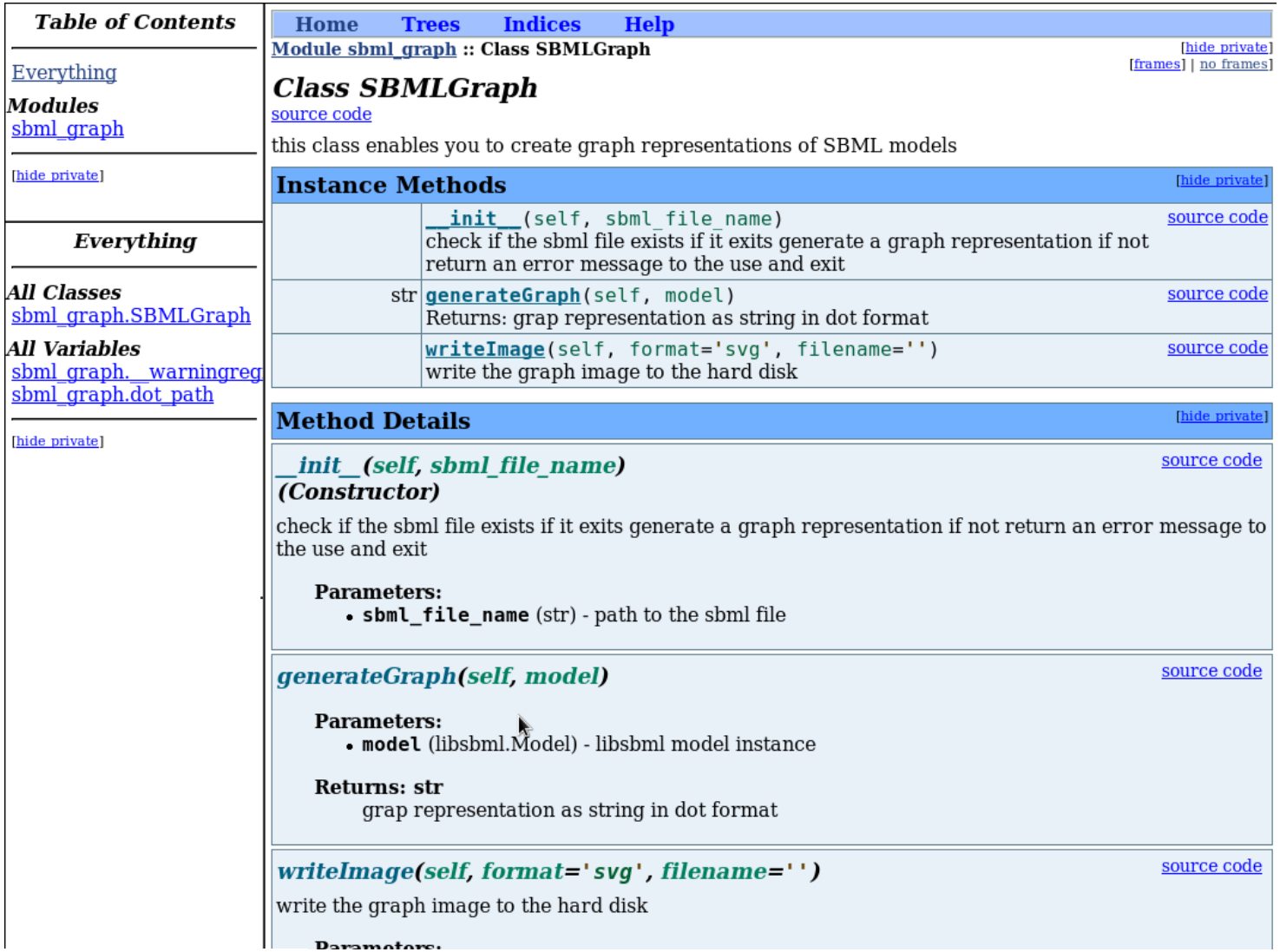


Table of Contents

Everything

Modules
sbml_graph

[hide private]

Everything

All Classes
sbml_graph.SBMLGraph

All Variables
sbml_graph.__warningreg
sbml_graph.dot_path

[hide private]

Home Trees Indices Help

Module sbml_graph :: Class SBMLGraph [hide private] [frames] | no frames]

Class SBMLGraph

source code

this class enables you to create graph representations of SBML models

Instance Methods [hide private]

	`__init__(self, sbml_file_name)` check if the sbml file exists if it exits generate a graph representation if not return an error message to the use and exit	source code
str	`generateGraph(self, model)` Returns: grap representation as string in dot format	source code
	`writeImage(self, format='svg', filename='')` write the graph image to the hard disk	source code

Method Details [hide private]

__init__(self, sbml_file_name) source code
(Constructor)

check if the sbml file exists if it exits generate a graph representation if not return an error message to the use and exit

Parameters:
- **sbml_file_name** (str) - path to the sbml file

generateGraph(self, model) source code

Parameters:
- **model** (libsbml.Model) - libsbml model instance

Returns: str
grap representation as string in dot format

writeImage(self, format='svg', filename='') source code

write the graph image to the hard disk

Parameters:
- **filename** (str) - filename of image