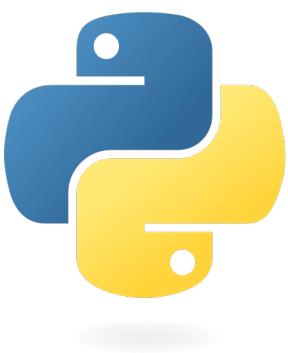


python™

A course by Johan van de Koppel
September 2018
Royal Netherlands Institute of Sea Research

Royal NIOZ is part of NWO-I, in cooperation with Utrecht University

Aims of this course

- Not a course in programming (OK maybe a little)
- Not an alternative to R (OK it could be, but a bad one)
- But: A course to explain Python to Matlab users
  => Knowledge of simulation in Matlab is presumed (but not totally)

Why Python in stead of Matlab?

Matlab is:

- Better organized
- Better documented
- Faster (slightly)
- A licensing headache

Python :

- Free for anyone
- Free on any computer
- Free to develop further
- More versatile

Disclaimer

- I am by no means a expert in Python!
- I am learning it as much as you are
- Lets learn together!
    - Assignments

Today's agenda

- What is Python
- A history of Python
- The Development Cycle
- Basic Syntax

About Python

- A scripting language, or script “interpreter”
  (a program, called the “console”, reads your code and executes it)
- Object-oriented (i.e. a modern programming language)
- Indentation for used de separate programming blocks
- Computer-Agnostic (runs on anything that runs python)

Differences between a programming and a scripting language

Programming language

- a program is compiled from the code (in machine instructions)
- A "program" in general, is a sequence of instructions written so that a computer can perform certain task.

Scripting language

- a script is interpreted
- a "script" is code written in a scripting language: a code that is executed by some software application (the - python console).

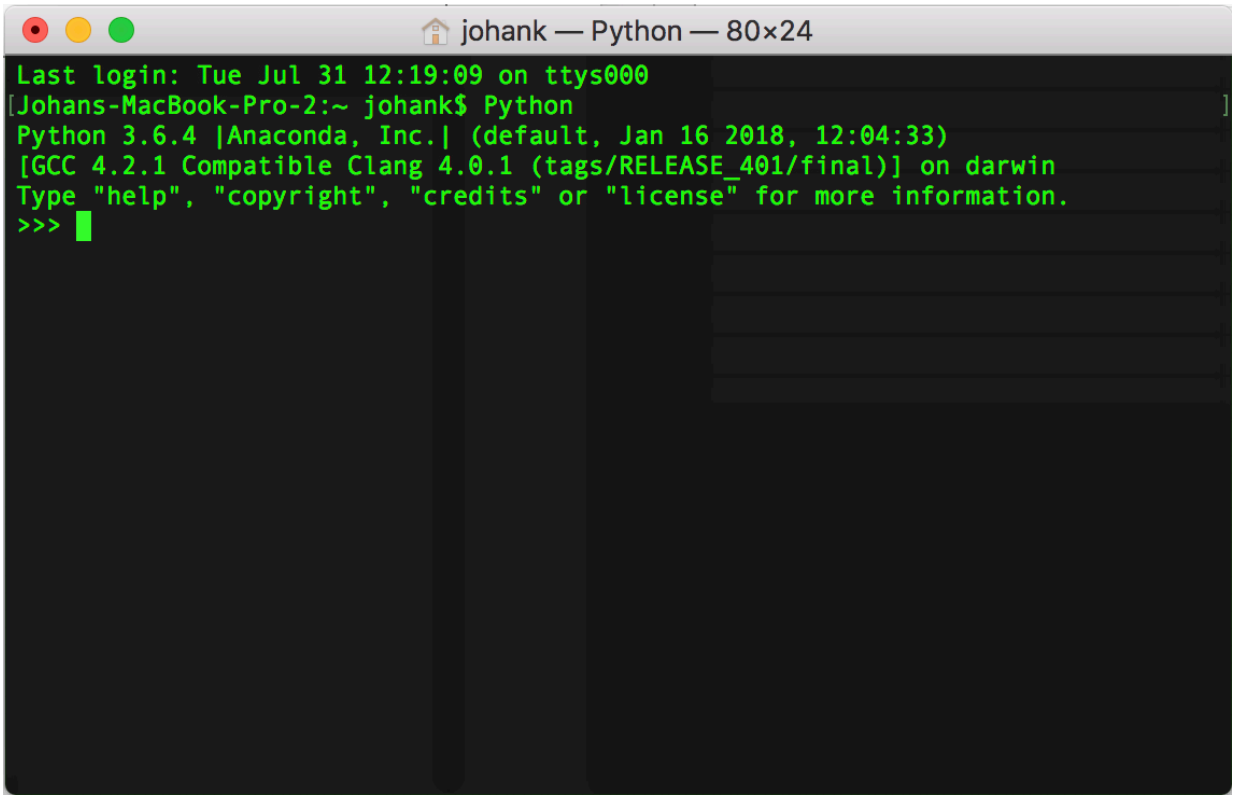
A history of Python

- Invented in the Netherlands, in the late 80s, by Guido van Rossum
- Name comes from 'Monty Python's Flying Circus', a famous 60s-70s TV show

Two versions, 2.7 and 3.6 (latest)

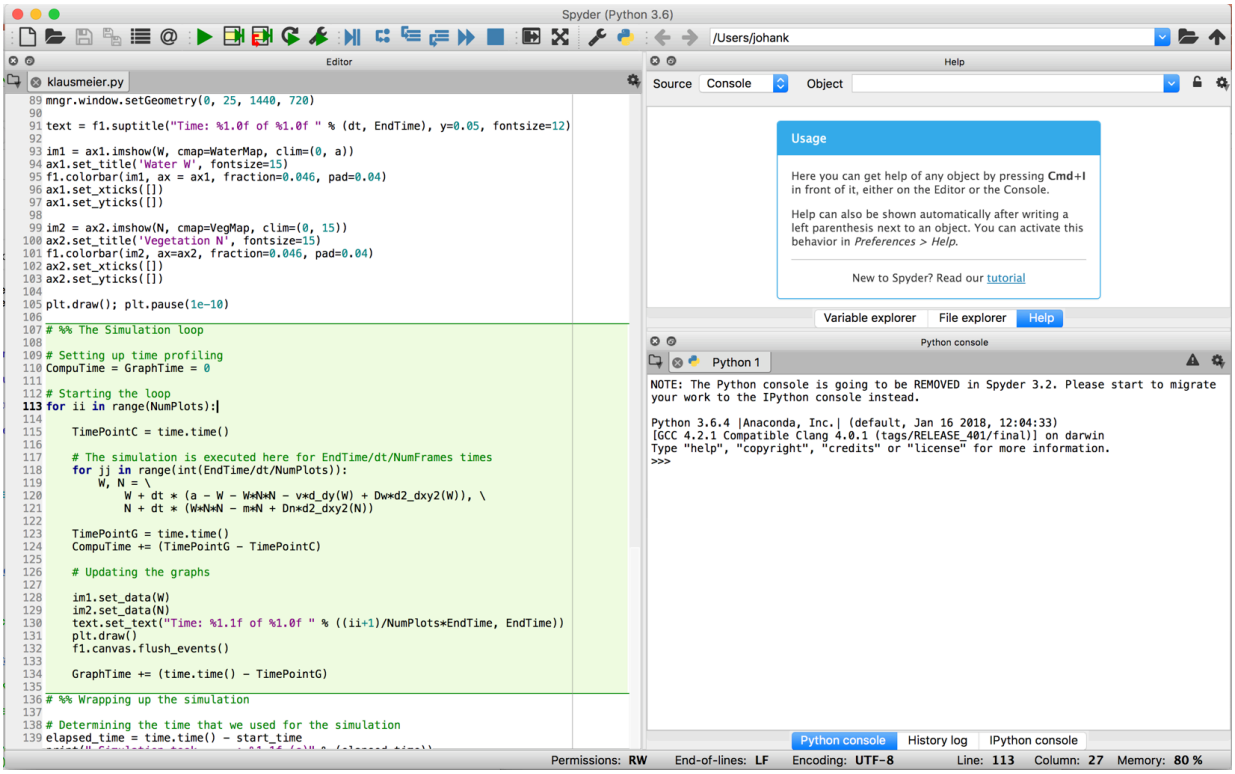
- As any good scientist should, Guido figured that he didn't design Python perfectly (and future proof).
- In 2008, Python 3.0, a major, backwards-incompatible version, was released.
  - V 2.7: print "Hello World!"
  - V 3.x: print("Hello World!")
- Many didn't want this, because old code libraries were useless.

Barebone Python



Matlab's Integrated Development Environment

Spyder




```
mngr.window.setGeometry(0, 25, 1440, 720)

text = f1.suptitle("Time: %1.0f of %1.0f " % (dt, EndTime), y=0.05, fontsize=12)

im1 = ax1.imshow(W, cmap=WaterMap, clim=(0, a))
ax1.set_title('Water W', fontsize=15)
f1.colorbar(im1, ax = ax1, fraction=0.046, pad=0.04)
ax1.set_xticks([])
ax1.set_yticks([])

im2 = ax2.imshow(N, cmap=VegMap, clim=(0, 15))
ax2.set_title('Vegetation N', fontsize=15)
f1.colorbar(im2, ax=ax2, fraction=0.046, pad=0.04)
ax2.set_xticks([])
ax2.set_yticks([])

plt.draw(); plt.pause(1e-10)

# %% The Simulation loop

# Setting up time profiling
CompuTime = GraphTime = 0

# Starting the loop
for ii in range(NumPlots):

    TimePointC = time.time()

    # The simulation is executed here for EndTime/dt/NumFrames times
    for jj in range(int(EndTime/dt/NumPlots)):
        W, N = \
             W + dt * (a - W - W*N*N - v*d_dy(W) + Dw*d2_dxy2(W)), \
             N + dt * (W*N*N - m*N + Dn*d2_dxy2(N))

    TimePointG = time.time()
    CompuTime += (TimePointG - TimePointC)

    # Updating the graphs

    im1.set_data(W)
    im2.set_data(N)
    text.set_text("Time: %1.1f of %1.0f " % ((ii+1)/NumPlots*EndTime, EndTime))
    plt.draw()
    f1.canvas.flush_events()

    GraphTime += (time.time() - TimePointG)

# %% Wrapping up the simulation

# Determining the time that we used for the simulation
elapsed_time = time.time() - start_time
```

Integrated development environments (IDEs)

Text-editors with code coloring

- Vim (very basic)
- Atom

Basic IDEs

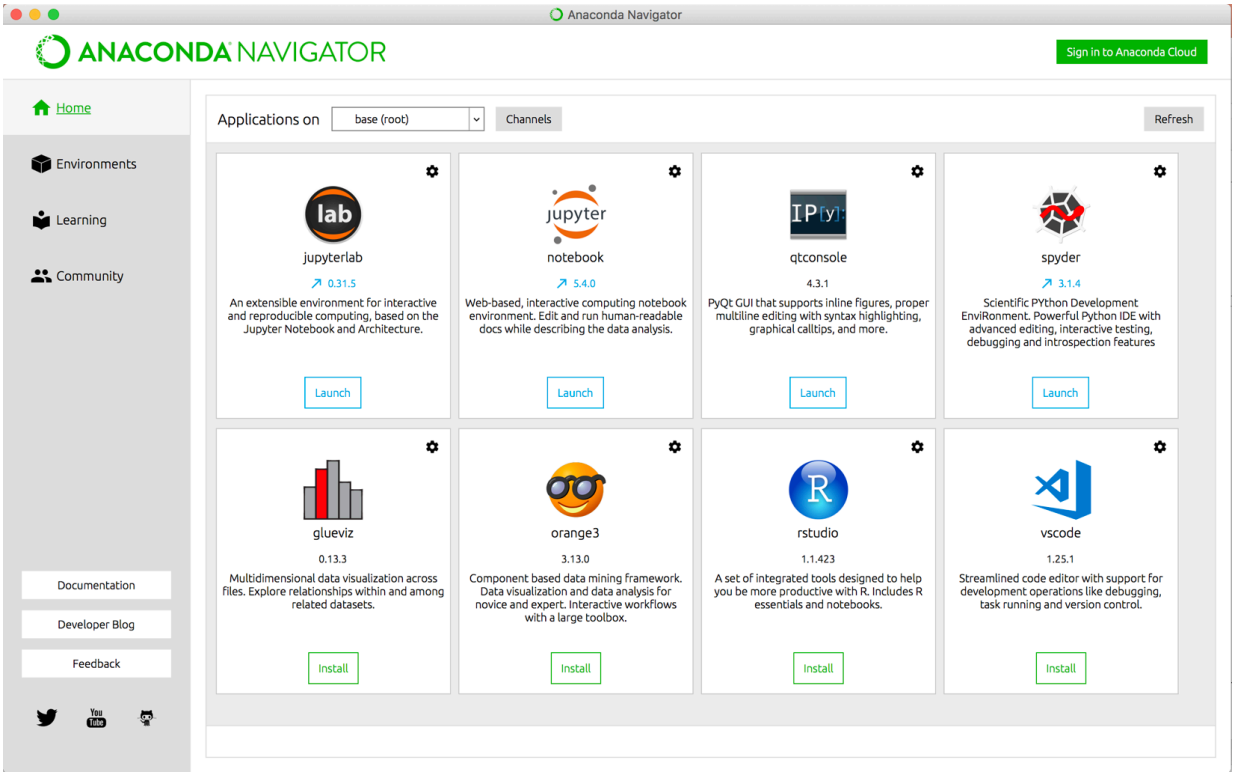
Environments

- You can run multiple versions of Python next to each other
- A single copy is called an Environment
- Each Python Environment with packages can sit in a folder
- You can tell the computer to “activate” another version
- Environment/Package management system: pip or conda
- To install a package, type in terminal:
  `conda install --name myenv numpy`
  “myenv” is just a name that you give to your environment

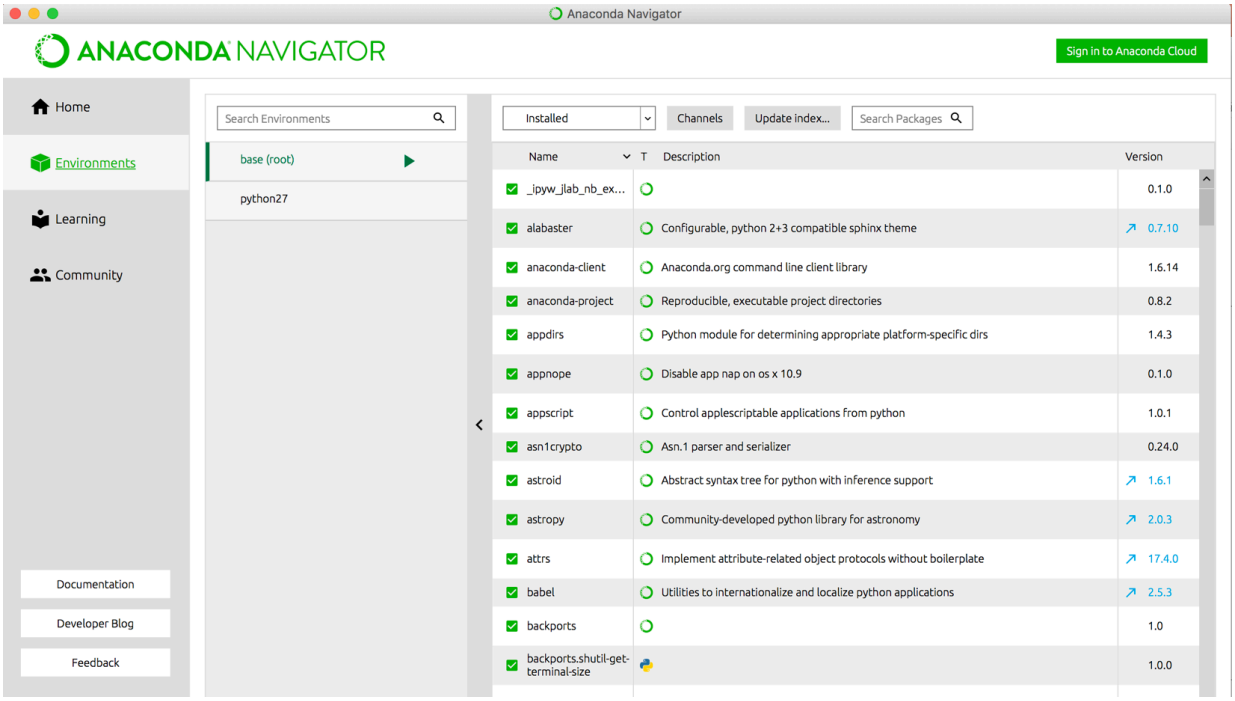
Anaconda

- Anaconda is a package management system with an attached IDE (Spyder) and Notebook system (Jupyter)
- It comes with either Python 2.7 or 3.6, but you can install both.
- In this course, we will be using Anaconda 5.2 with Python 3.6
- https://www.anaconda.com/download (https://www.anaconda.com/download)
- We will use this in this course (it is the most convenient one I know)

Anaconda - included applications



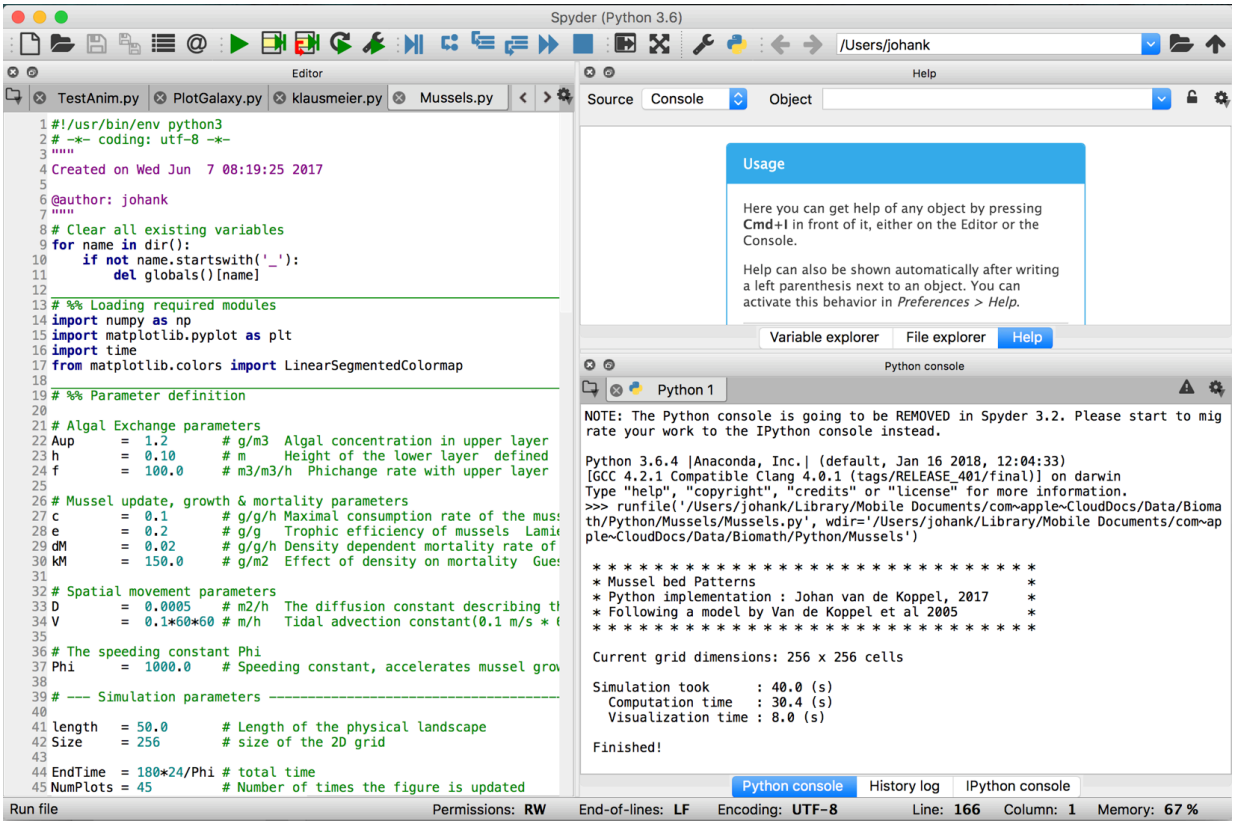
Anaconda - Package environment



Python-console versus iPython-console

- Python comes with a barebone console for running single commands or entire scripts
- iPython (with Jupyter) is a notebook based console with follows the principle: set of commands – printed result, and does so within a notebook document. It is similar to Mathematica & Latech.
- iPython can not run animated simulations. It first simulates, and then produces a result (figure or movie).

Python executes from the console



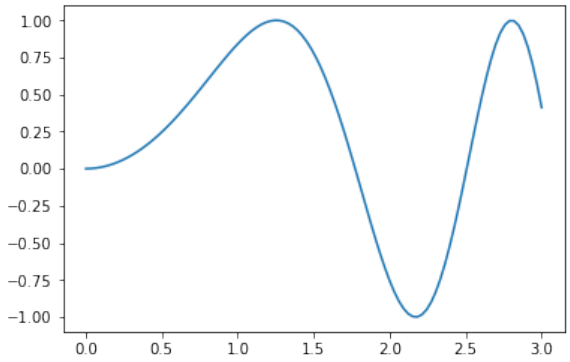
Python executes from the console

iPython runs in a Jupyter notebook

First you give a piece of code in the notebook, and then iPython interprets it and gives the result.

In [1]:
```
# Importing the packages
import numpy as np
from matplotlib import pyplot as plt
```

In [2]:
```
# Calculating and plotting y=sin(x^2)
x = np.linspace(0, 3, 100)
y = np.sin(x*x)
f=plt.plot(x,y)
```



Packages

- Most code functions sit in Packages or Modules
- Often used packages for numerical computing
  - numpy – Numerical operations on array operation
  - scipy – Many mathematical functions
  - Matplotlib – plotting functions

Importing a package

You can import packages containing collections of functions, also called modules, into your code, often allowing you to do a specialized set of operations. One example is the NumPy package (from Numerical Python), that allows you to define arrays and do all sorts of array operations. There are a number of ways to load a package. One way is to load it as a so-called "object", which puts the functions that are important in a enclosed block (i.e. object):

In [1]:
```
import numpy as np
A = np.sqrt(16)
print(A)
```

```
4.0
```

You can also just load all the functions into the common "namespace", so that they can be used directly. A disadvantage is that separate packages can use the same name, creating a conflict.

In [2]:
```
from numpy import *
A = sqrt(16)
print(A)
```

```
4.0
```

Installing a package

- Very few packages are installed with standard Python
- Anaconda does install a lot of packages at install (numpy, scipy, matplotlib, etc)
- You can install more from within the Anaconda Navigator
- If you want an odd package, you need to install it via the Terminal (Mac OS, Linux) or Command Prompt (Windows)
- for this, type in google: conda install "package name" for explanation
- Or go to: https://anaconda.org (https://anaconda.org)

Variables and types

In many programming languages, you define *variables*, and then work with these variables to get things done. There are various *types* of variables, and these so-called types we explain here:

In [2]:
```
a = 2          # a is of type integer: discrete numbers used for counting
b = 1.3        # b is of type float, meaning it can contain continuous variables
c = "Hallo"    # c is of type string, containing a letter or sentence
d = True       # d is of type Boolean, either True or False, used for testing
e = [0, 2]     # e is a list, containing numbers. It is NOT an array or matrix
f = [1, "no"] # f is also a list
```

Operations with variables

In [3]:
```
a*b
```

Out[3]:
```
2.6
```

In [4]:
```
print(c + " John")
```

```
Hallo John
```

In [5]:
```
c + " " + str(a)
```

Out[5]:
```
'Hallo 2'
```

In [6]:
```
c*3
```

Out[6]:
```
'HalloHalloHallo'
```

In [12]:
```
(c+" ")*3
```

Out[12]:
```
'Hallo Hallo Hallo '
```

Oddities with strings

In [13]:
```
word = 'NIOZ'
```

In [14]:
```
word[2]
```

Out[14]:
```
'O'
```

In [18]:
```
word[-1]
```

Out[18]:
```
'Z'
```

In [22]:
```
word[0:2]
```

Out[22]:
```
'NI'
```

In [25]:
```
word[1:]
```

Out[25]:
```
'IOZ'
```

In [29]:
```
word[0:3]+'O'
```

Out[29]:
```
'NIOO'
```

Lists

A list of comma-separated *items* between square brackets. Lists might have items of different types!

In [33]:
```
squares = [1, 4, 9, 16, 25]
```

In [38]:
```
Items = [1, 3.1, 'String', True]
```

In [40]:
```
squares + [3, 5, 2, 5, -1]
```

Out[40]:
```
[1, 4, 9, 16, 25, 3, 5, 2, 5, 3, 5, 2, 5, -1]
```

In [39]:
```
Items*2
```

Out[39]:
```
[1, 3.1, 'String', True, 1, 3.1, 'String', True]
```

Arrays, vectors and matrices

There is no build-in support for arrays in Python. This ability comes from a package called `Numpy`. Numpy allows you to define vectors (1D), matrices (2D) and larger arrays and calculate with them. Also, it contains a hugh collection of computational functions.

`Numpy` has to be loaded as a package, and then an array can be defined:

In [5]:
```
import numpy as np
A=np.zeros([3,3])
A
```

Out[5]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Or, you can assign the values from a list

In [8]:
```
A=np.array([[1,2,3],[4,5,6],[7,8,9]])
A
```

Out[8]:
```
array([[1, 2, 3],
       [4, 5, 6],
```

With these, you can compute:

In [10]:
```
A*2
```

Out[10]:
```
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

Or use in a spatial model:

In [11]:
```
dA_dt=(1-A/10.0)*A
A+dA_dt
```

Out[11]:
```
array([[1.9, 3.6, 5.1],
       [6.4, 7.5, 8.4],
       [9.1, 9.6, 9.9]])
```

For more, look at the Numpy website: http://www.numpy.org (http://www.numpy.org)

Classes - Object orientation in Python

- Classes provide a means of bundling data and methods (function, procedure).
- Creating a new class creates a new type of object, allowing new instances of that type to be made.

In [1]:
```
class Mussel:
    x=0
    y=0
```

In [2]:
```
M=Mussel()
print('The mussels is at location %1.1f,%1.1f' % (M.x,M.y) )
```

```
The mussels is at location 0.0,0.0
```

In [3]:
```
import numpy as np
class Mussel:
    def __init__(self):   # the is a 'method' called when a Mussel is defined
        self.x=np.random.rand(1)
        self.y=np.random.rand(1)
```

In [4]:
```
M=Mussel()
print('The mussels is at location %1.2f,%1.2f' % (M.x,M.y) )
```

```
The mussels is at location 0.44,0.23
```

In [5]:
```
import numpy as np
class Mussel:
    def __init__(self):   # this is a 'method' called when a Mussel is defined
        self.x=np.random.rand(1)
        self.y=np.random.rand(1)
    def distance(self):   # this is a callible 'method'
        return np.sqrt(self.x**2+self.y**2)
```

In [6]:
```
M=Mussel()
print('The mussels is at location %1.2f,%1.2f, which is at %1.2f cm van the orig
in' % (M.x,M.y,M.distance()) )
```

```
The mussels is at location 0.38,0.41, which is at 0.56 cm van the origin
```

Many Python types are actually a class with all sorts of build in methods

In [7]:
```
S='This a really really really long string'
```

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

Out[8]:
```
str
```

In [9]:
```
S.__len__()
```

Out[9]:
```
39
```

In [10]:
```
S.upper()
```

Out[10]:
```
'THIS A REALLY REALLY REALLY LONG STRING'
```

To know more about the Methods associated with `str` and other standard types, see: https://docs.python.org/3/library/stdtypes.html (https://docs.python.org/3/library/stdtypes.html)

Condition testing: the `if` statement

In [35]:
```
x = 5.1 # Just assign a number
```

In [39]:
```
if x < 0:
    x = 0
    print('Negative changed to zero.')
elif x == 0:
    print('Zero is nothing ...')
elif x == 1:
    print('Yes one is enough!')
else:
    print('More than one!')
```

```
More than one!
```

Loops and conditions

Loop are an important part of computing. Algoritms are repeated on new data, or repeated (iterated) on the same data.
In classical computing, a loop repeated with increasing some counter (here a MATLAB example):

```
for i=1:5
    disp(i);
end
```

In Python:

In [11]:
```
for i in range(5):
    print(i)
```

```
0
```

In Python, a loop will iterate a **list**, *any list!*

In [12]:
```
for i in [2,4.5,'Hallo', True]:
    print( type(i) )
```

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
```

A nested loop over a unstructured list

Imagine a nested database of names, with 3 female and 4 male names.

In [75]:
```
Department=[['Yvette','Olga', 'Renee'],['Daan','Geert', 'Roland','Leo']]
```

You can print all using a nested loop:

In [76]:
```
for Group in Department:
    print(Group)
```

```
['Yvette', 'Olga', 'Renee']
['Daan', 'Geert', 'Roland', 'Leo']
```

In [74]:
```
for Group in Department:
    for Names in Group:
        print(Names)
```

```
Yvette
Olga
Renee
Daan
Geert
Roland
Leo
```

So what if you want the item and a counter at the same time?

In [23]:
```
for i,Group in enumerate(Department):
    print('Group ' + str(i))
    for j,Names in enumerate(Group):
        print(' ' + str(j) + ' - ' + Names)
```

```
Group 0
 0 - Yvette
 1 - Olga
 2 - Renee
Group 1
 0 - Daan
 1 - Geert
 2 - Roland
 3 - Leo
```

A conditional loop

Sometimes you don't want a loop to run a specific number of times, but rather until a condition is met. For this, there is the `while` loop:

In [27]:
```
i=0
while i<5:
    print(i)
    i=i+1
```

```
0
1
2
3
4
```

`break` and `continue` statements

`break` stops the loop entirely.

In [32]:
```
for i in range(5):
    if i==3:
        break
    print(i)
```

```
0
1
2
```

`continue` skips the current iteration and goes to the next.

In [34]:
```
for i in range(5):
    if i==3:
        continue
    print(i)
```

```
0
1
2
4
```

Functions

In Python, you can define your own procedures (do not return a value) and functions (returns a value):

In [44]:
```
def Fibonacci(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while a < n:
        print(a),
        a, b = b, a+b
```

In [52]:
```
Fibonacci(20)
```

```
0
1
1
2
3
5
8
13
```

In [48]:
```
def dN_dt(N):           # A function for the logistic growth equation
    return (1-N)*N
```

In [50]:
```
dN_dt(0.6)
```

Out[50]:
```
0.24
```

In [54]:
```
def Product(X,Y):           # A function for the logistic growth equation
    return X*Y

10 + Product(4,3)
```

Out[54]:
```
22
```

Plotting

If you want to plot something in Python, you first have to load a plotting library (or package). The most commonly used library is `Matplotlib', which contains many MATLAB like plotting functions. So first, we have to load Matplotlib

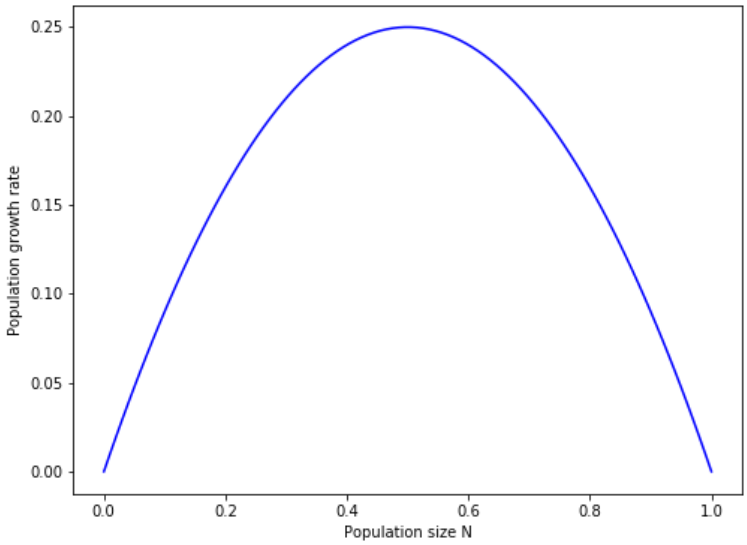
In [65]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

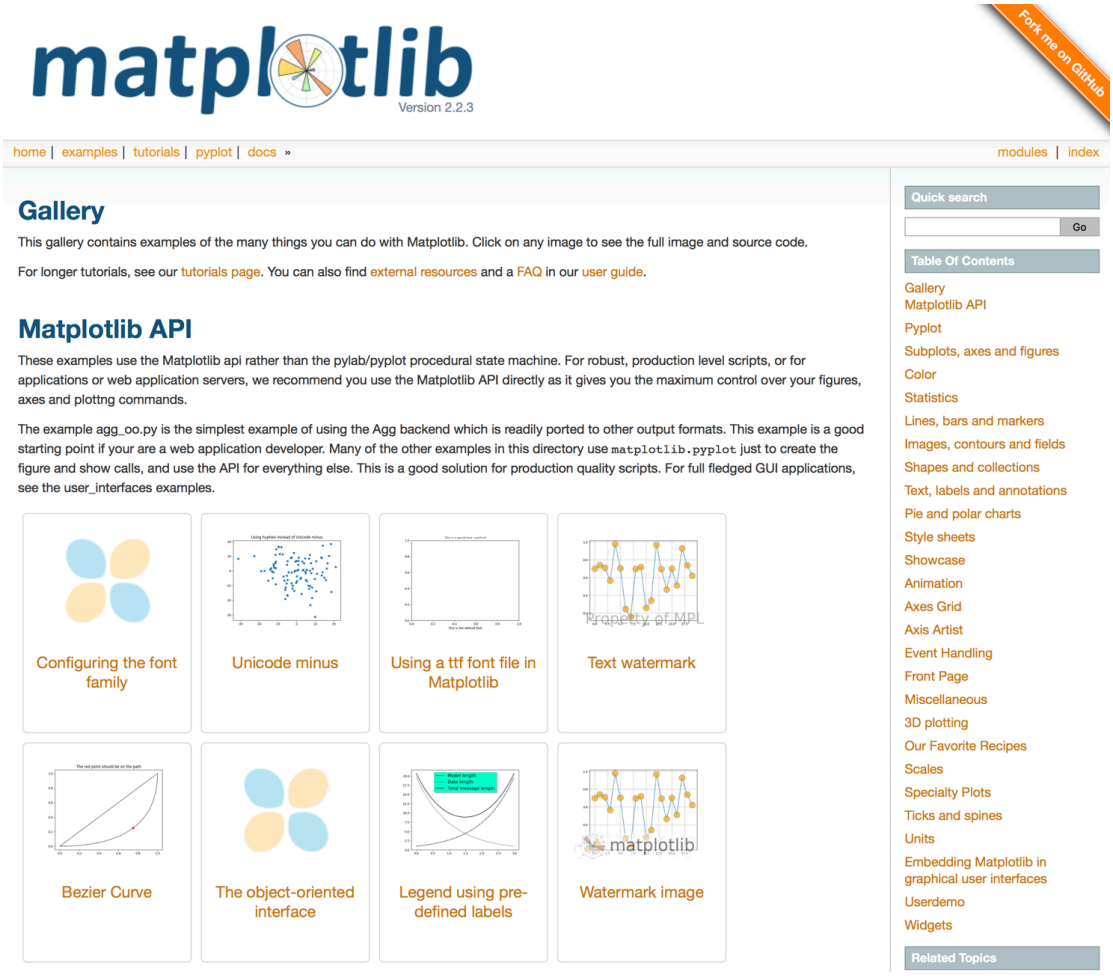
Define some variables to plot:

In [67]:
```
N=np.linspace(0,1,100)  # a series of numbers from 0 to 1 in 100 steps
dN=dN_dt(N)
```

Then open a figure:

In [71]:
```
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
f=ax.plot(N,dN,'b-')
f=ax.set_xlabel('Population size N')
f=ax.set_ylabel('Population growth rate')
```





This course

1 Introduction to programming in Python and iPython
2 ODE models
3 Spatial models
4 IBM models

Practical for today

Test Spyder

- Install Anaconda
- Open Spyder
- Choose the Python console
- Run a small program
- Run a program in the Python console and in the iPython console

Test Jupyter

- Activate Jupyter
- Add a line of code
- Add a markdown line
- Add a simple figure

In [13]:
```
pwd
```

Out[13]:
```
'/Users/johank/Simulations/Jupyter/Python_Course'
```