



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



NWO



Utrecht University



rijksuniversiteit
groningen

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

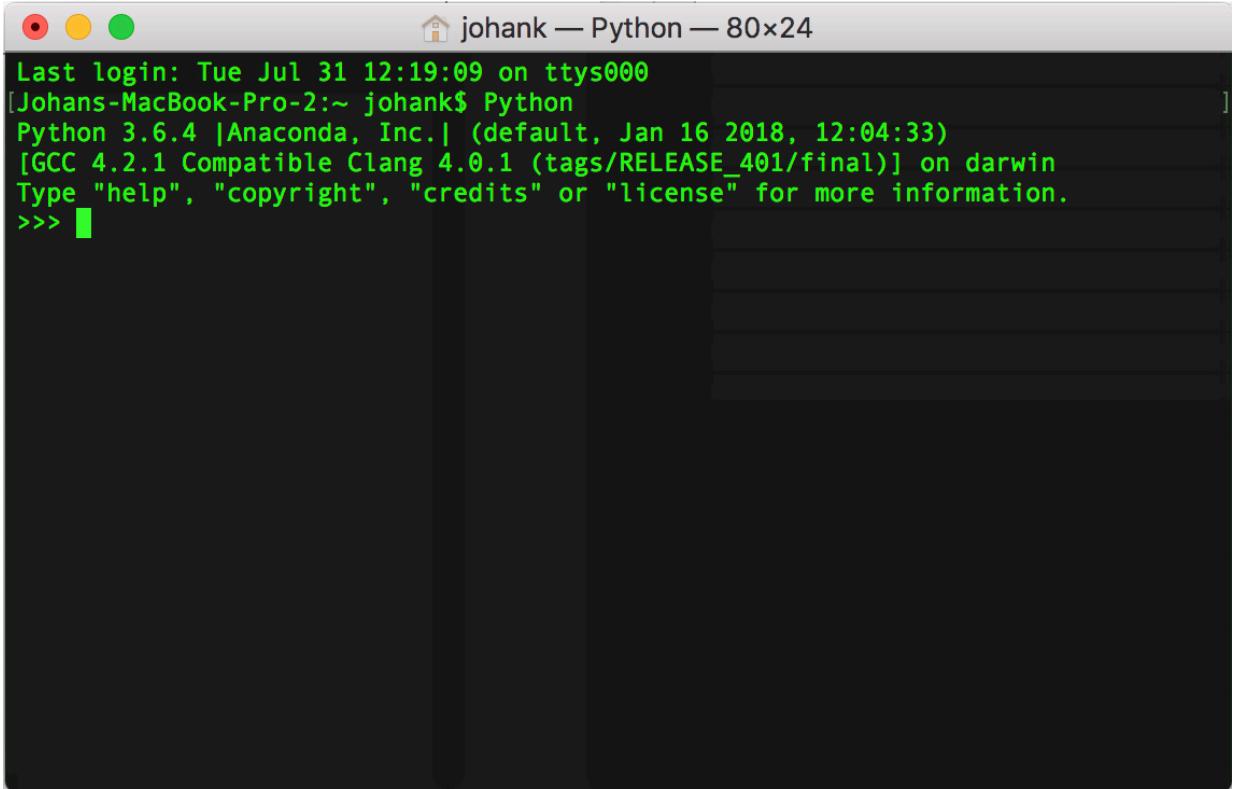


- Invented in the Netherlands, in the late 80s, by Guido van Rossum
 - <https://www.python.org/psf-history/>

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.

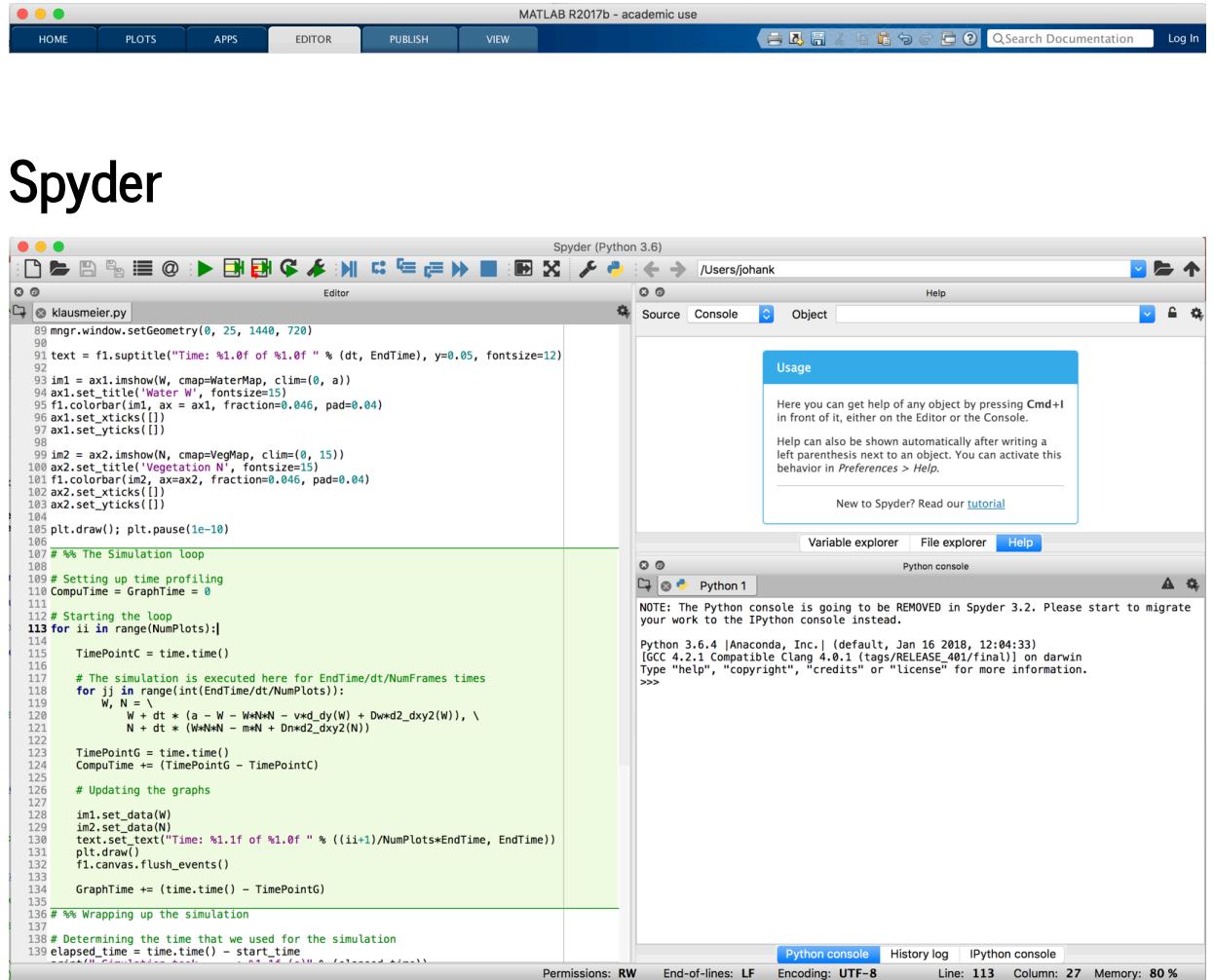
Barebone Python



The screenshot shows a terminal window with the title bar "johank — Python — 80x24". The window contains the following text:

```
Last login: Tue Jul 31 12:19:09 on ttys000
[Johans-MacBook-Pro-2:~ johank$ Python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 12:04:33)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ]
```

Matlab's Integrated Development Environment



Integrated development environments (IDEs)

Text-editors with code coloring

- Vim (very basic)
- Atom

Basic IDEs

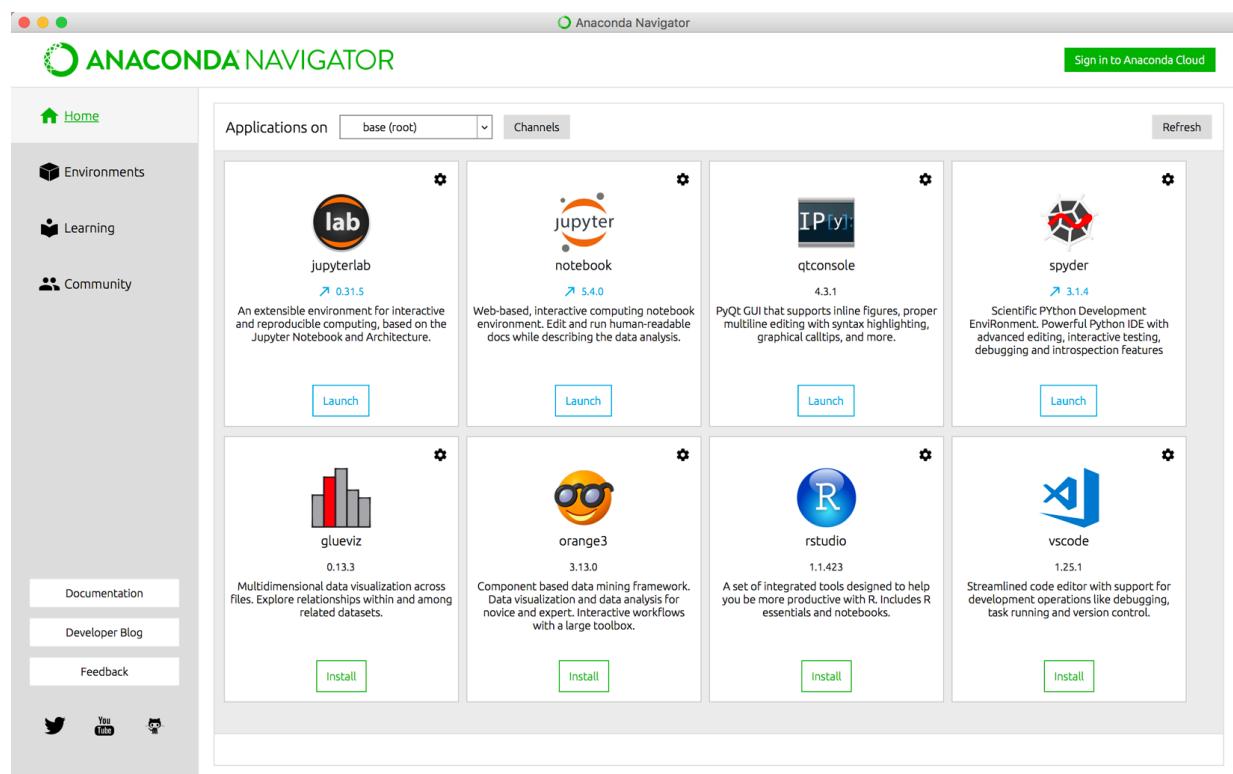
Environments

- 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

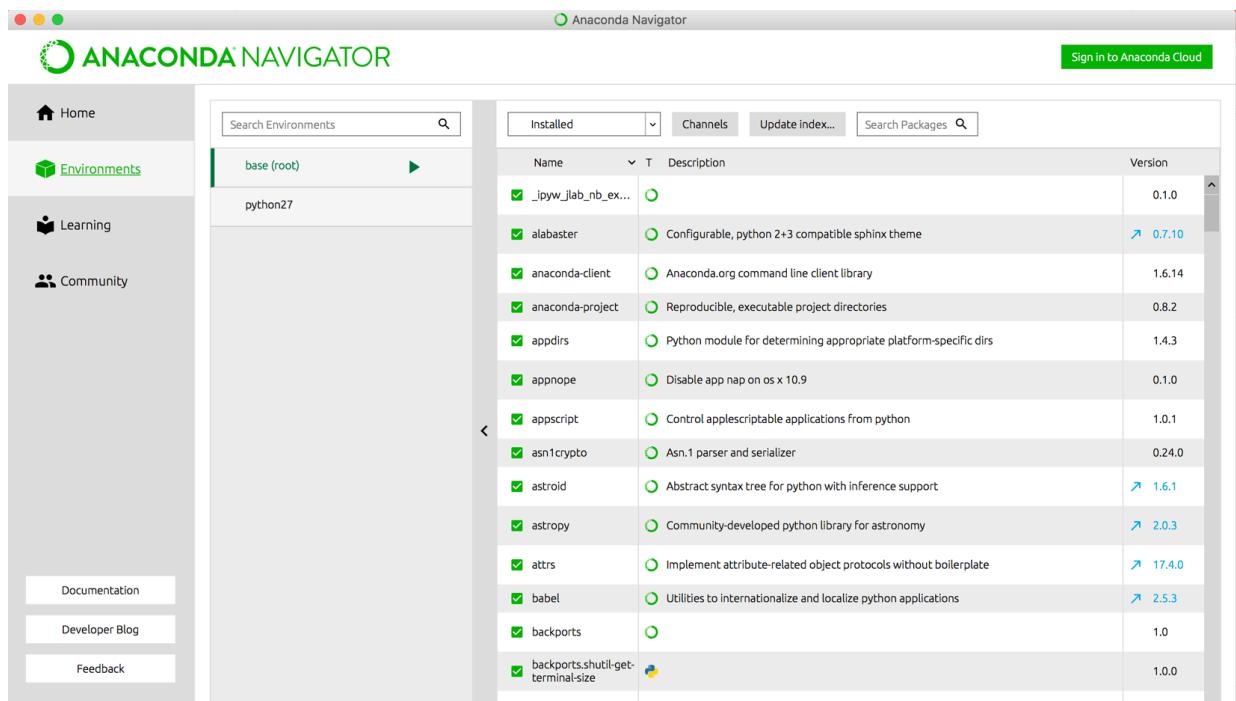
Anaconda

- 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

The screenshot shows the Spyder Python IDE interface. The top menu bar includes 'File', 'Edit', 'Cell', 'Run', 'Kernel', 'Help', and 'About'. The toolbar contains icons for file operations like Open, Save, Run, and Help. The left sidebar lists open files: 'TestAnim.py', 'PlotGalaxy.py', 'klausmeier.py', and 'Mussels.py'. The main area has tabs for 'Source' (selected), 'Console', and 'Object'. A 'Help' panel is open, showing the 'Usage' tab which provides information on getting help for objects. Below the help panel is a 'Variable explorer', 'File explorer', and 'Help' tab. The central workspace shows Python code for a simulation, including parameters for algae and mussels, spatial movement, and simulation time. The 'Python console' tab at the bottom displays the output of running the script, including a note about its removal in Spyder 3.2, the Python version (3.6.4), and the results of the run. The status bar at the bottom shows file permissions, encoding, line and column numbers, and memory usage.

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Created on Wed Jun 7 08:19:25 2017
5
6 @author: johank
7 """
8 # Clear all existing variables
9 for name in dir():
10     if not name.startswith('_'):
11         del globals()[name]
12
13 # %% Loading required modules
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import time
17 from matplotlib.colors import LinearSegmentedColormap
18
19 # %% Parameter definition
20
21 # Algal Exchange parameters
22 AUp = 1.2      # g/m3 Algal concentration in upper layer
23 h = 0.1        # m Height of the lower layer defined
24 f = 100.0      # m3/m3/h Phichange rate with upper layer
25
26 # Mussel update, growth & mortality parameters
27 c = 0.1        # g/g/h Maximal consumption rate of the mussel
28 e = 0.2        # g/g Trophic efficiency of mussels Lameijer
29 dM = 0.02       # g/g/h Density dependent mortality rate of
30 kM = 150.0      # g/m2 Effect of density on mortality Guesdon
31
32 # Spatial movement parameters
33 D = 0.0005     # m2/h The diffusion constant describing the
34 V = 0.1*60*60 # m/h Tidal advection constant(0.1 m/s * 60)
35
36 # The speeding constant Phi
37 Phi = 1000.0   # Speeding constant, accelerates mussel growth
38
39 # ---- Simulation parameters -----
40
41 length = 50.0    # Length of the physical landscape
42 size = 256        # size of the 2D grid
43
44 endTime = 180*24/Phi # total time
45 numPlots = 45      # Number of times the figure is updated
```

Run file Permissions: RW End-of-lines: LF Encoding: UTF-8 Line: 166 Column: 1 Memory: 67%

Python executes from the console

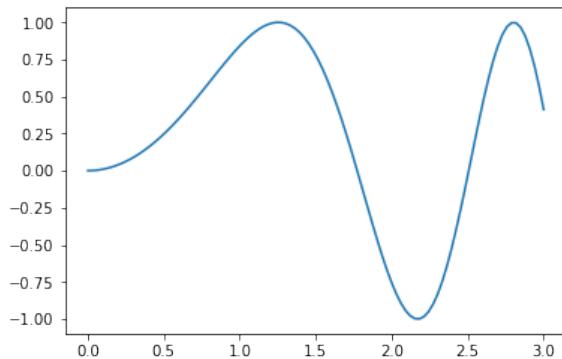


iPython runs in a Jupyter notebook

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")
```

```
Haloo John
```

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

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

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

```
Out[6]: 'HalooHalloHallo'
```

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

```
Out[12]: 'Haloo Haloo Haloo '
```

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):      # this 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 callable '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 origin' % (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. Algorithms 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

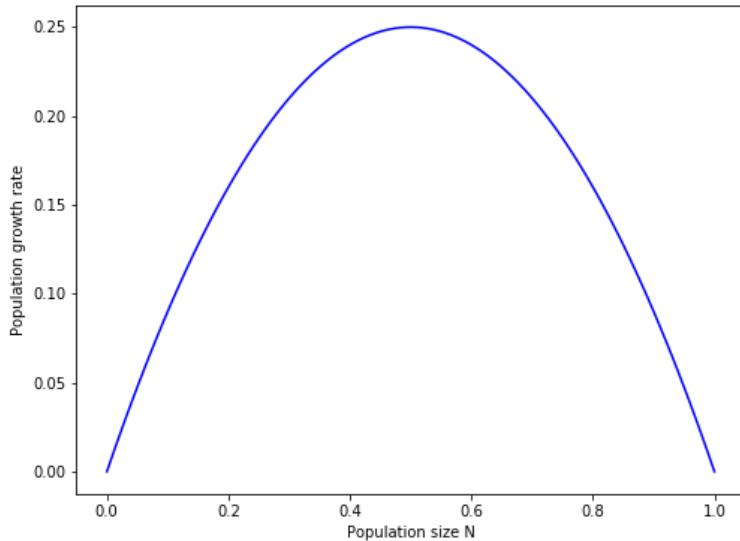
```
In [65]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

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')
```



[home](#) | [examples](#) | [tutorials](#) | [pyplot](#) | [docs](#) »

Fork me on GitHub

[modules](#) | [index](#)

Gallery

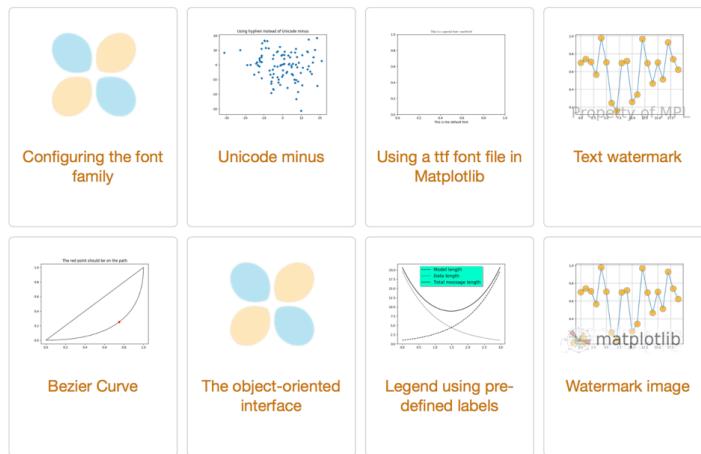
This gallery contains examples of the many things you can do with Matplotlib. Click on any image to see the full image and source code.

For longer tutorials, see our [tutorials page](#). You can also find [external resources](#) and a [FAQ](#) in our [user guide](#).

Matplotlib API

These examples use the Matplotlib api rather than the pylab/pyplot procedural state machine. For robust, production level scripts, or for applications or web application servers, we recommend you use the Matplotlib API directly as it gives you the maximum control over your figures, axes and plotting commands.

The example `agg_oo.py` is the simplest example of using the Agg backend which is readily ported to other output formats. This example is a good starting point if you are a web application developer. Many of the other examples in this directory use `matplotlib.pyplot` just to create the figure and show calls, and use the API for everything else. This is a good solution for production quality scripts. For full fledged GUI applications, see the `user_interfaces` examples.



Quick search
<input type="text"/> Go
Table Of Contents
Gallery
Matplotlib API
Pyplot
Subplots, axes and figures
Color
Statistics
Lines, bars and markers
Images, contours and fields
Shapes and collections
Text, labels and annotations
Pie and polar charts
Style sheets
Showcase
Animation
Axes Grid
Axis Artist
Event Handling
Front Page
Miscellaneous
3D plotting
Our Favorite Recipes
Scales
Specialty Plots
Ticks and spines
Units
Embedding Matplotlib in graphical user interfaces
Userdemo
Widgets
Related Topics

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'
```