# Python Seminar

March 16, 2017

# 1 Scientific Programming with Python

http://gdfa.ugr.es/python

## 1.1 Outline

- Introduction to Python
- Python for science, where to begin?
- Python language
- Scientific libraries

## 1.2 Introduction to Python

### 1.2.1 What is Python?

Python is a modern, general-purpose, object-oriented, high-level programming language.

General characteristics of Python:

- **clean and simple language:** Easy-to-read and intuitive code, easy-to-learn minimalistic syntax, maintainability scales well with size of projects.
- **expressive language:** Fewer lines of code, fewer bugs, easier to maintain.

Technical details:

- **dynamically typed:** No need to define the type of variables, function arguments or return types.
- **automatic memory management:** No need to explicitly allocate and deallocate memory for variables and data arrays. No memory leak bugs.
- **interpreted:** No need to compile the code. The Python interpreter reads and executes the python code directly.

### 1.2.2 Advantages:

- The main advantage is **ease of programming**, minimizing the time required to develop, debug and maintain the code.
- Well designed language that **encourage many good programming practices**:
- **Modular** and object-oriented programming, good system for packaging and re-use of code. This often results in more transparent, maintainable and bug-free code.

- **Documentation tightly integrated with the code**.
- A large standard library, and a **large collection of add-on packages**.
- Packaging of programs into **standard executables**, that **work on computers without Python** installed.

#### 1.2.3 Disadvantages:

- Since Python is an interpreted and dynamically typed programming language, **the execution of python code can be slow** compared to compiled statically typed programming languages, such as C and Fortran.
- Somewhat decentralized, with **different environment, packages and documentation spread out at different places**. Can make it harder to get started.

#### 1.2.4 What makes python suitable for scientific computing?

*Nature* **518**, 125–126 (05 February 2015) | doi:10.1038/518125a

- Python has a strong position in scientific computing
  - **Large community of users**, easy to find help and documentation.
- Extensive ecosystem of **scientific libraries**
  - NumPy: numerical Python ≈ MATLAB matrices and arrays
  - SciPy: scientific Python ≈ MATLAB toolboxes
  - pandas: extends NumPy
  - Matplotlib: graphics library
  - Sympy: symbolic mathematics library
- **Scientific (and non-scientific) development environments** available
  - spyder: MATLAB-like environment
  - Jupyter/IPython notebooks: environment for interactive and exploratory Python
  - Rodeo: new Python environment for data science
  - PyCharm: Python enviroment for developers
- **Great performance due to** close integration with time-tested and highly **optimized codes written in C and Fortran**
- Readily available and **suitable for use** on high-performance **computing clusters**
- **No license costs**, no unnecessary use of research budget

### 1.3 Python for science, where to begin?

#### 1.3.1 Why to choose Python 2?

- Python 3 is better, but **some non-widespread science modules are still not compatible**
- **Differences** between Python 2 and 3 **are relatively minor**
- Python 2 is **actively supported**. For example, Linux distributions and Macs are still using 2.x as default

TOOLBOX

# PICK UP PYTHON

*A powerful programming language with huge community support.*


ILLUSTRATION BY THE PROJECT TWINS

BY JEFFREY M. PERKEL

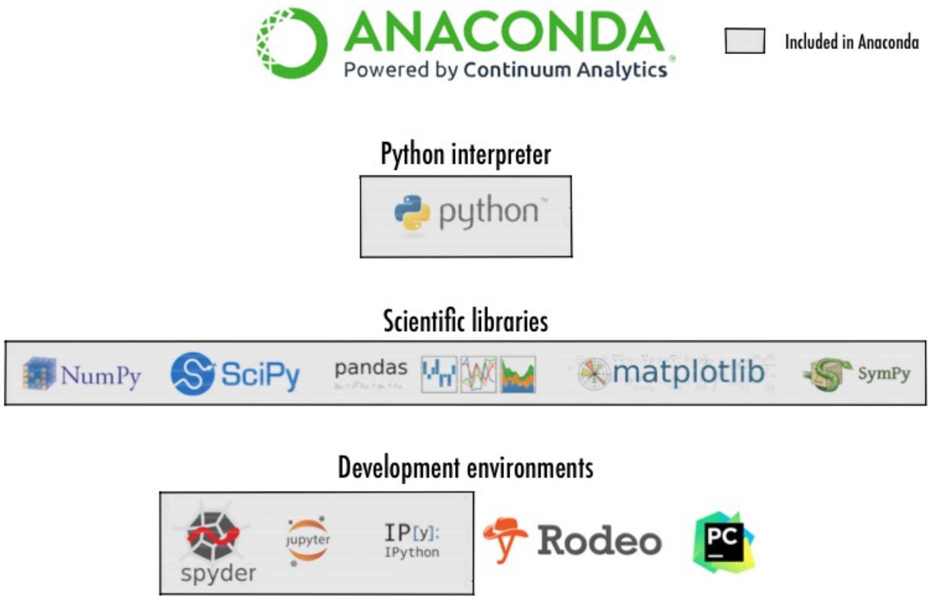
Last month, Adina Howe took up a post at Iowa State University in Ames. Officially, she is an assistant professor of agricultural and biosystems engineering. But she works not in the greenhouse, but in front of a keyboard. Howe is a programmer, and a key part of her job is as a 'data professor' — developing curricula to teach the next generation of graduates about the mechanics and importance of scientific programming.

Howe does not have a degree in computer science, nor does she have years of formal train-

Brown specializes in bioinformatics and uses computation to extract meaning from genomic data sets, and Howe had to get up to speed on the computational side. Brown's recommendation: learn Python.

Among the host of computer-programming languages that scientists might choose to pick up, Python, first released in 1991 by Dutch programmer Guido van Rossum, is an increasingly popular (and free) recommendation. It combines simple syntax, abundant online resources and a rich ecosystem of scientifically focused toolkits with a heavy emphasis on community.

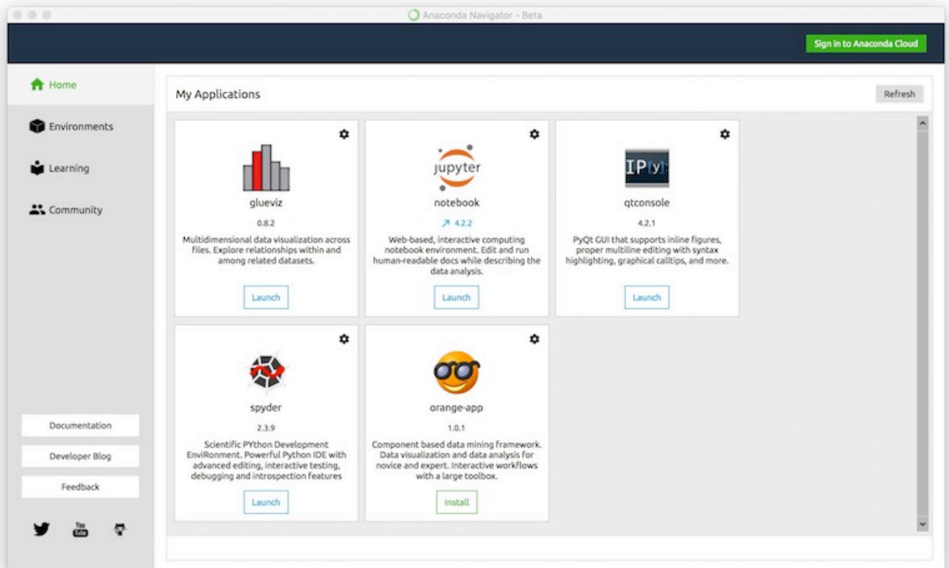
is becoming ever more crucial. Researchers who can write code in Python can deftly manage their data sets, and work much more efficiently on a whole host of research-related tasks — from crunching numbers to cleaning up, analysing and visualizing data. Whereas some programming languages, such as MATLAB and R, focus on mathematical and statistical operations, Python is a general-purpose language, along the lines of C and C++ (the languages in which much commercial software and operating systems are written). As such, it is perhaps more complicated, Brown says, but also more capable: it is amenable to everything from

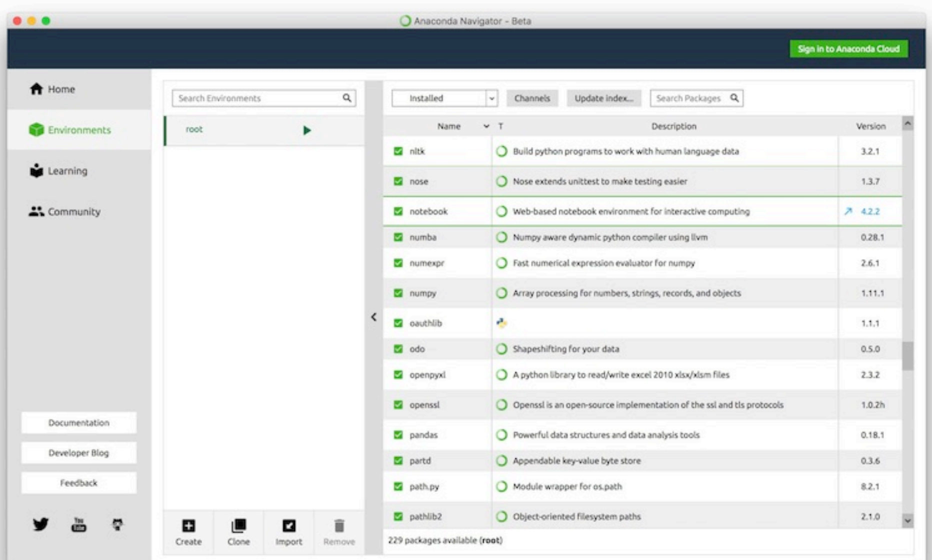


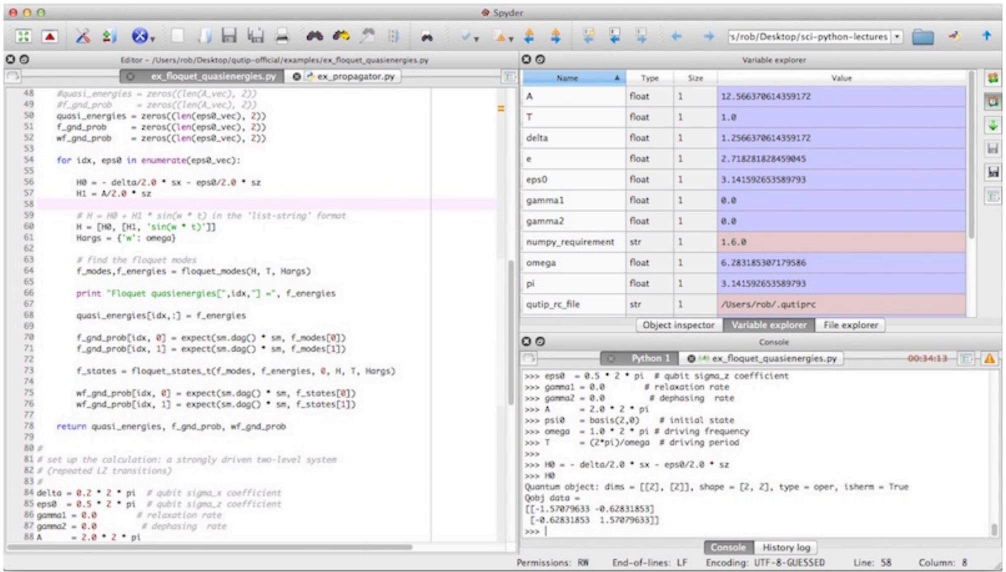

### 1.3.2 Scientific-oriented Python Distributions

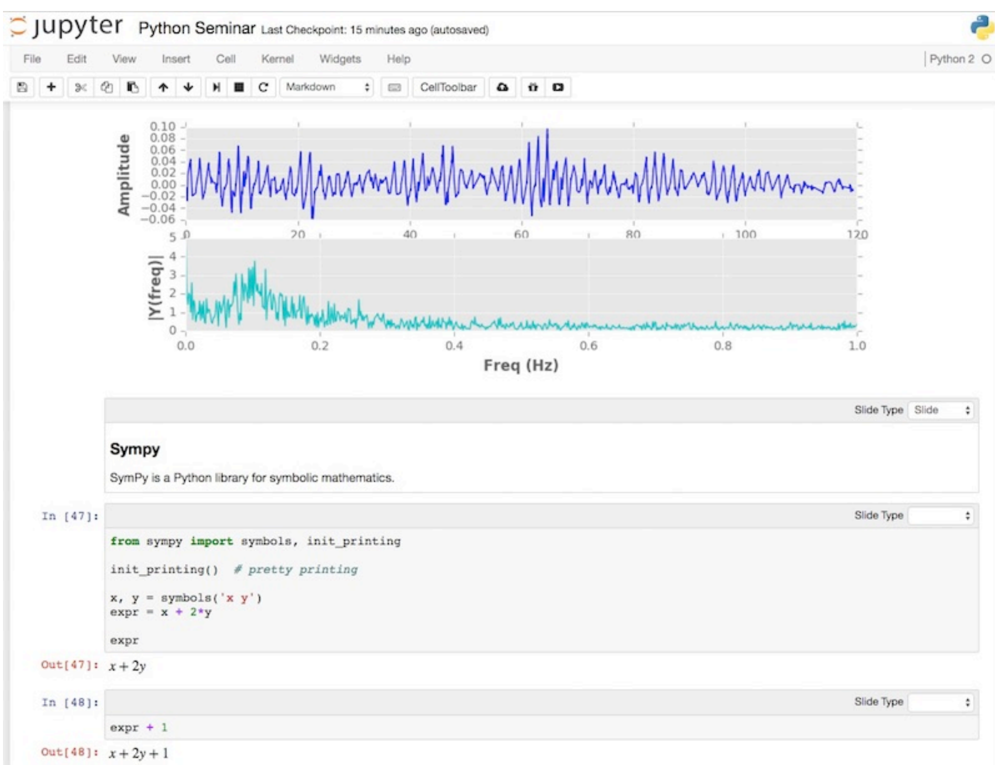
Provide a **Python interpreter** with commonly used **scientific libraries** in science like NumPy, SciPy, Pandas, matplotlib, etc. already installed. In the past, it was usually painful to build some of these packages. Also, include **development environments** with advanced editing, debugging and introspection features.

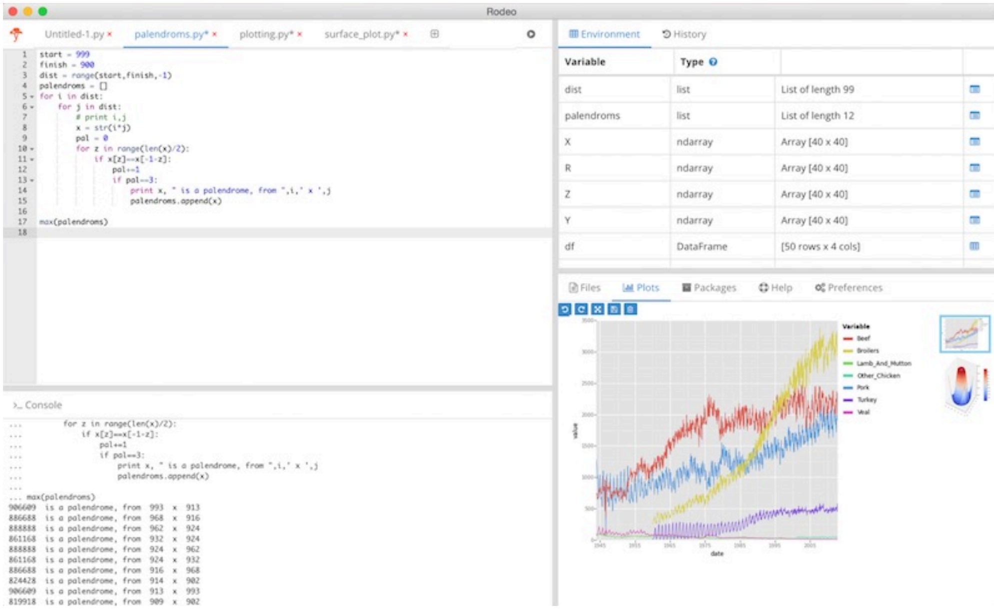
- **Anaconda**
  - Cross-platform
  - Supports Python 2 and 3
  - **Most widely adopted**
- Canopy
  - Cross-platform
  - Only supports Python 2
- Python(x,y)
  - Windows-only platform
  - Only support Python 2


Anaconda Navigator - Beta
Sign in to Anaconda Cloud
Home
Environments
Learning
Community
My Applications
Refresh
glueviz
0.8.2
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Launch
notebook
4.2.2
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch
qtconsole
4.2.1
PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more.
Launch
spyder
2.3.9
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch
orange-app
1.0.1
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
Install
Documentation
Developer Blog
Feedback


Anaconda Navigator - Beta
Sign in to Anaconda Cloud
Home
Environments
Learning
Community
Search Environments
root
Installed
Channels
Update index...
Search Packages
Name
T
Description
Version
nltk
Build python programs to work with human language data
3.2.1
nose
Nose extends unittest to make testing easier
1.3.7
notebook
Web-based notebook environment for interactive computing
4.2.2
numba
Numpy aware dynamic python compiler using llvm
0.28.1
numexpr
Fast numerical expression evaluator for numpy
2.6.1
numpy
Array processing for numbers, strings, records, and objects
1.11.1
oauthlib
1.1.1
odo
Shapeshifting for your data
0.5.0
openpyxl
A python library to read/write excel 2010 xlsx/xlsm files
2.3.2
openssl
Openssl is an open-source implementation of the ssl and tls protocols
1.0.2h
pandas
Powerful data structures and data analysis tools
0.18.1
partd
Appendable key-value byte store
0.3.6
path.py
Module wrapper for os.path
8.2.1
pathlib2
Object-oriented filesystem paths
2.1.0
Documentation
Developer Blog
Feedback
Create
Clone
Import
Remove
229 packages available (root)


Spyder
Editor - /Users/rob/Desktop/qutip-official/examples/ex_floquet_quasienergies.py
ex_floquet_quasienergies.py
ex_propagator.py
Variable explorer
Name
Type
Size
Value
Object inspector
Variable explorer
File explorer
Console
Python 1
Console
History log


jupyter
Python Seminar Last Checkpoint: 15 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Python 2
Markdown
CellToolbar
Amplitude
|Y(freq)|
Freq (Hz)
Slide Type
Slide
Sympy
SymPy is a Python library for symbolic mathematics.
In [47]:
from sympy import symbols, init_printing
init_printing() # pretty printing
x, y = symbols('x y')
expr = x + 2*y
expr
Out[47]: x + 2y
In [48]:
expr + 1
Out[48]: x + 2y + 1




### 1.3.3 Anaconda navigator

### 1.3.4 Anaconda navigator: installing new packages

### 1.3.5 spyder

### 1.3.6 IPython/Jupyter notebooks

### 1.3.7 Rodeo (need to be installed separately from Anaconda)

### 1.3.8 PyCharm (need to be installed separately from Anaconda)

| Editor | Learning curve | Users | Benefits |
|---|---|---|---|
| spyder | pretty short | Matlab and R background | mature, many features |
| rodeo | pretty short | Matlab and R background | modern, essential features |
| IPython/Jupyter | smooth | teachers | interactive |
| PyCharm | moderate | developers | code quality |

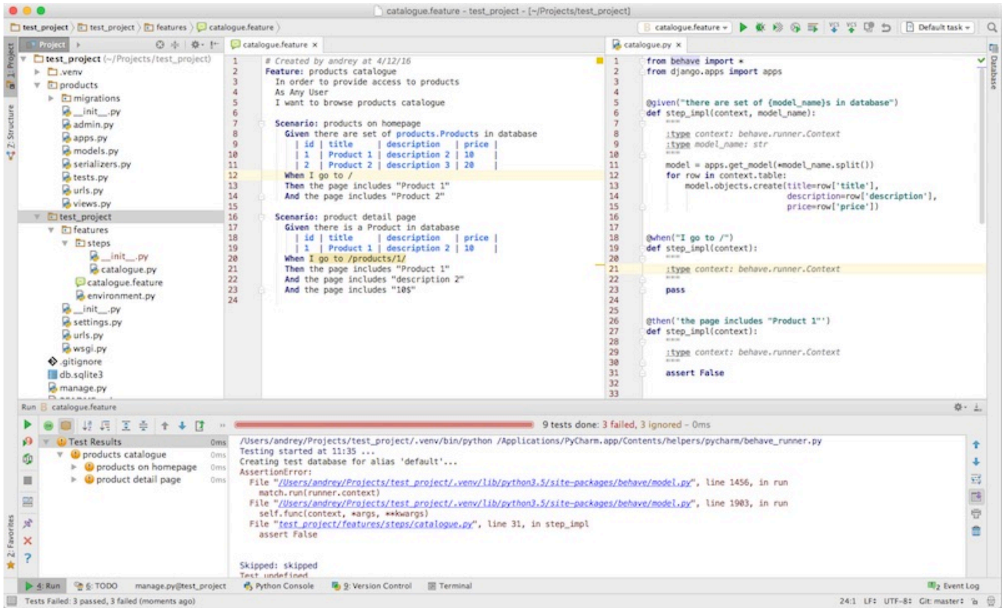
### 1.3.9 Where to look for help?

- **Official documentation**: `http://www.scipy.org/docs.html`
- Usually included in development environments as **contextual help**:
  - *spyder*: Ctrl+I (Windows) or Cmd+I (Mac)
  - *PyCharm*: F1 (Windows/Mac)
  - *Rodeo*: `?f` in the console
- **Be careful about code you get on the internet!**
- Dedicated **offline documentation browser** (*Python*, *LaTeX*, *C++*, *Java*, *Bootstrap*, *Bash*, ...):
  - Zeal (Windows/Linux): Free
  - Dash (Mac): Commercial
  - Velocity (Windows): Commercial



## 1.4 Python language

### 1.4.1 Using Python as a Calculator




### 1.4.2 Strings

In [4]:
```
prefix = 'Py'
word = prefix + 'thon'

# character in position 0
print word[0]

# characters from position 0 (included) to 6 (excluded)
print word[0:6]
```

```
P
Python
```

#### Note

- **0-based indexing**
- **half-open range indexing**: [a, b)
- **print** statement to get outputs
- **line comments**

### 1.4.3 Lists

In [5]:
```
# empty list
squares = []

# lists might contain items of different types
squares = ['cat', 4, 3.2]
```

```
# negative indices mean count backwards from end of sequence
print squares[-1]

# list concatenation
squares = squares + [81, 'dog']

# list functions
squares.remove(3.2)  # remove the first ocurrence
squares.append('horse')  # concatenation: same as +

print squares
```

```
3.2
['cat', 4, 81, 'dog', 'horse']
```

```
In [6]: a = ['a', 'b', 'c']
        n = [1, 2, 3]

        # it is possible to nest lists
        # (create lists containing other lists)
        x = [a, n]

        print x
        print x[0]
        print x[0][1]
```

```
[['a', 'b', 'c'], [1, 2, 3]]
['a', 'b', 'c']
b
```

### 1.4.4 Simple code: Fibonacci series

```
In [7]: a, b = 0, 1
        while a < 10:
            print a,
            # the sum of two elements defines the next
            a, b = b, a + b
```

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

#### Note

- **indentation level** of statements is significant
- **multiple assignment**

### 1.4.5 if Statements

```
In [8]: x = -4

        if x < 0:
            x = 0
            print 'Negative changed to zero'
        elif x == 0:
            print 'Zero'
        elif x == 1:
            print 'Single'
        else:
            print 'More'
```

```
Negative changed to zero
```

### 1.4.6 for Statements

```
In [9]: words = ['cat', 'window', 'defenestrate']

        for w in words:
            # len returns the number of items of an object.
            print w, len(w)
```

```
cat 3
window 6
defenestrate 12
```

#### Warning

Please **avoid Matlab-like for** statements

```
In [10]: for w in range(len(words)):
             print words[w], len(words[w])
```

```
cat 3
window 6
defenestrate 12
```

**range**(stop)

Built-in function to create lists containing arithmetic progressions.

```
In [11]: print range(10)
         print range(0, 10, 3)
         print range(0, -10, -1)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 3, 6, 9]
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

```
In [12]: for i in range(4):
             print 'cat',
```

```
cat cat cat cat
```

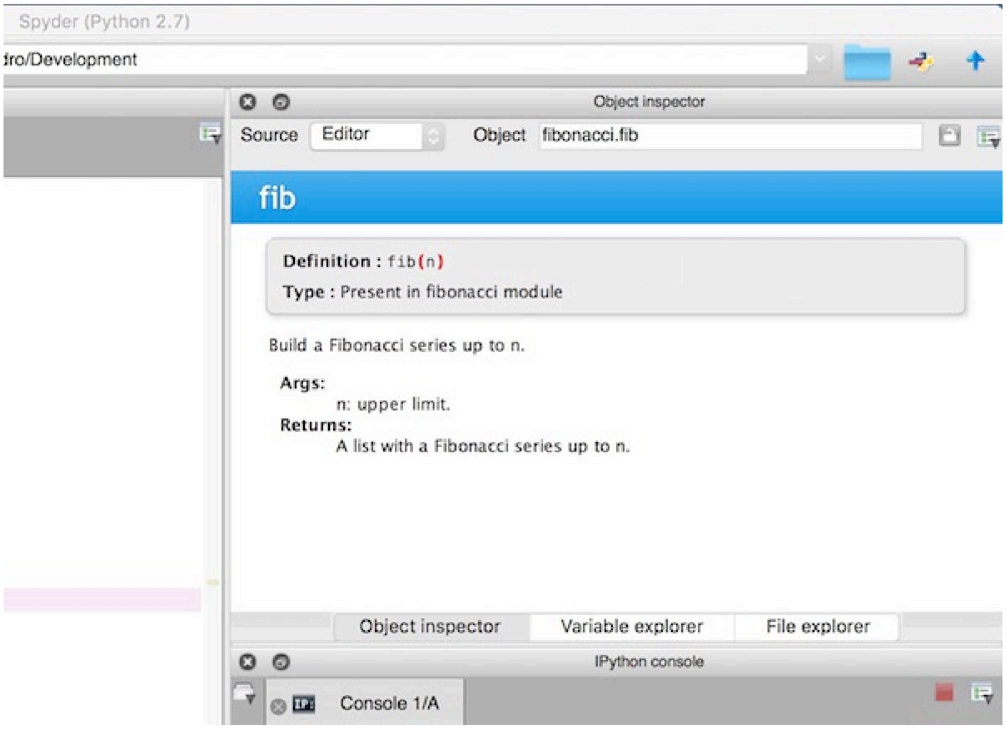
```
In [13]: words = ['cat', 'window', 'defenestrate']

         for i, w in enumerate(words):
             print i, w
```

```
0 cat
1 window
2 defenestrate
```

### 1.4.7 Functions

```
In [14]: def fib(n):
             """Build a Fibonacci series up to n.

             Args:
                 n: upper limit.

             Returns:
                 A list with a Fibonacci series up to n.
             """
             f = []  # always initialize the returned value!

             a, b = 0, 1
             while a < n:
                 f.append(a)
                 # the sum of two elements defines the next
                 a, b = b, a + b

             return f

         # now call the function we just defined:
         print fib(1000)
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

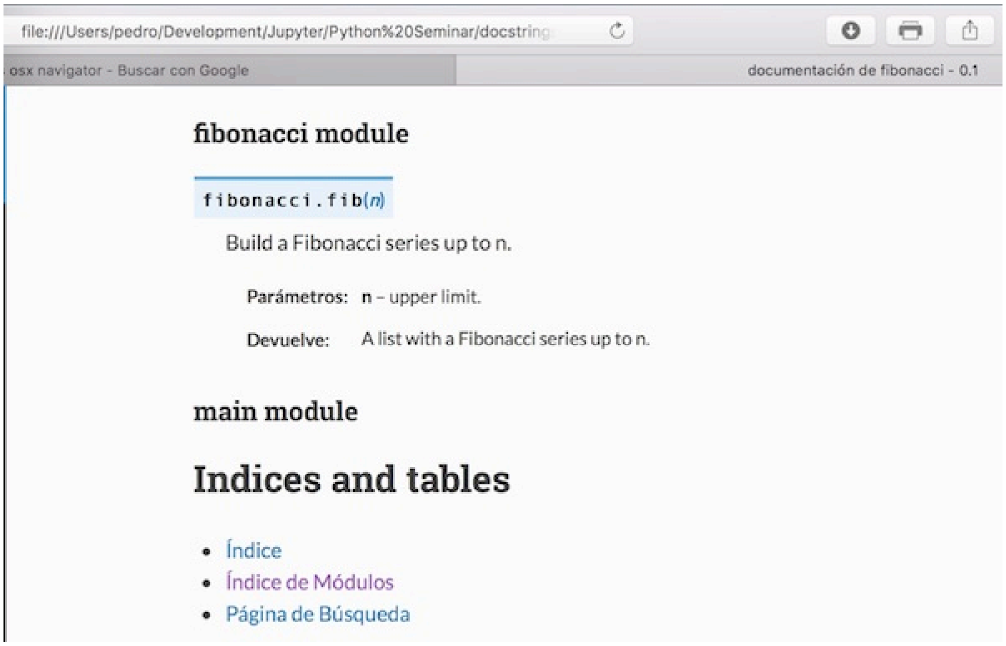


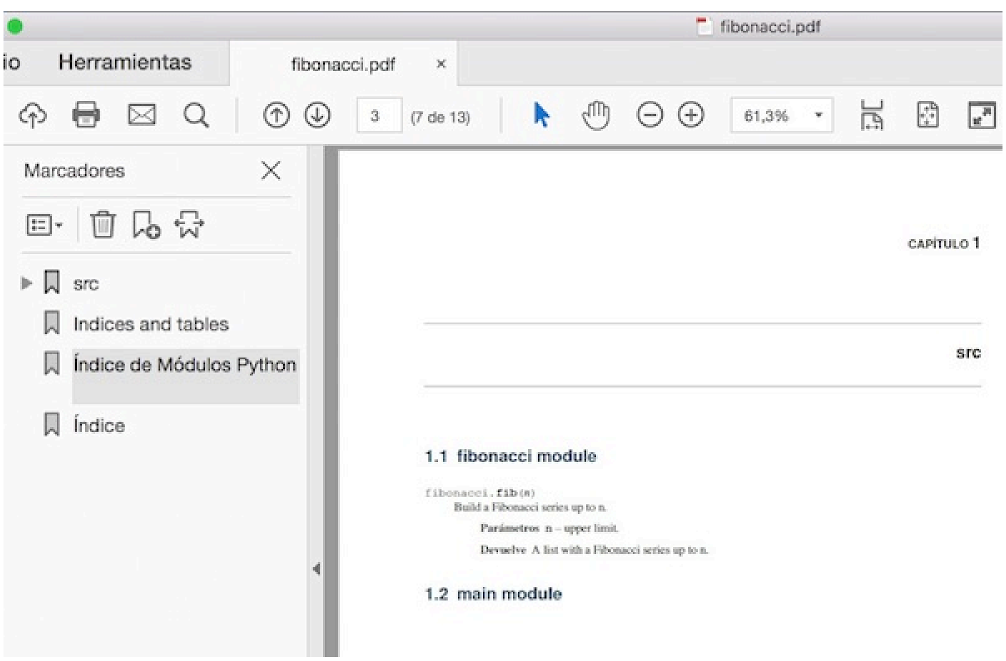

### 1.4.8 Functions: documentation strings (docstrings)

- Python documentation strings (docstrings) provide a convenient **way of associating documentation with Python functions** and modules.
- Docstrings can be written following **several styles**. We use Google Python Style Guide.
- An object's docsting is defined by including a **string constant as the first statement in the function's definition**.
- Unlike conventional source code comments **the docstring should describe what the function does, not how**.
- **All functions should have a docstring**.
- This allows to inspect these comments at run time, for instance as an **interactive help system**, or **export them as HTML, LaTeX, PDF** or other formats.

### 1.4.9 Functions: default argument values

```
In [15]: def fib(n, s=0):
             """Build a Fibonacci series up to n.

             Args:
                 n: upper limit.
                 s: lower limit. Default 0.

             Returns:
                 A list with a Fibonacci series up to n.
```


file:///Users/pedro/Development/Jupyter/Python%20Seminar/docstring
osx navigator - Buscar con Google
documentación de fibonacci - 0.1
fibonacci module
fibonacci.fib(n)
Build a Fibonacci series up to n.
Parámetros: n – upper limit.
Devuelve: A list with a Fibonacci series up to n.
main module
Indices and tables
Índice
Índice de Módulos
Página de Búsqueda


fibonacci.pdf
Herramientas
fibonacci.pdf
3
(7 de 13)
61,3%
Marcadores
src
Indices and tables
Índice de Módulos Python
Índice
CAPÍTULO 1
src
1.1 fibonacci module
1.2 main module

```
    """
    f = []  # always initialize the returned value!

    a, b = 0, 1
    while a < n:
        if a >= s:  # lower limit
            f.append(a)
        # the sum of two elements defines the next
        a, b = b, a + b

    return f

print fib(1000, 15)
print fib(1000, 0)
print fib(1000)
```

```
[21, 34, 55, 89, 144, 233, 377, 610, 987]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
```

### 1.4.10 Functions: keyword arguments

```
In [16]: print fib(1000, 15)  # positional arguments
         print fib(s=15, n=1000)  # keyword arguments
```

```
[21, 34, 55, 89, 144, 233, 377, 610, 987]
[21, 34, 55, 89, 144, 233, 377, 610, 987]
```

### 1.4.11 Functions: returning multiple values

```
In [17]: def fib(n, s=0):
             """Build a Fibonacci series up to n.

             Args:
                 n: upper limit.
                 s: lower limit. Default 0.

             Returns:
                 (f, l):

                 * ``f``: list with a Fibonacci series up to n.
                 * ``l``: length of Fibonacci series.
             """
             f = []  # always initialize return values!
             l = 0

             a, b = 0, 1
```

```
    while a < n:
        if a >= s:  # lower limit
            f.append(a)
        # the sum of two elements defines the next
        a, b = b, a + b
    l = len(f)  # number of elements

    return f, l

a, b = fib(1000)
print a
print b

c = fib(1000)
print c
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987]
17
([0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987], 17)
```

### 1.4.12 Functions: importing external functions

```
In [18]: import fibonacci  # without .py extension

         print fibonacci.fib(3)
```

```
[0, 1, 1, 2]
```

```
In [19]: from fibonacci import fib

         print fib(3)
```

```
[0, 1, 1, 2]
```

```
In [20]: import fibonacci as f  # alias

         print f.fib(3)
```

```
[0, 1, 1, 2]
```

#### Recommendation

The best way to import libraries is included in their official help

Some examples:

```
import math
import numpy as np
from scipy import linalg, optimize
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import sympy
```

#### 1.4.13 Functions: main

fibonacci.py

```
if __name__ == '__main__':
    print fib(1000)
```

A file `fibonacci.py` can be used in two ways.

- imported in another file: `import fibonacci`. In this case internal variable `__name__` is `fibonacci` (the name of the imported module), and `print fib(1000)` does not get executed
- executed directly: `python foo.py`. In this case internal variable `__name__` have a value `__main__`, and `print fib(1000)` does get executed

#### 1.4.14 Functions: modules and packages

**Modules** in Python are simply **Python files** with the .py extension, which implement **a set of functions**. Modules are imported from other modules using the `import` command.

**Packages** are **simply directories which contain a special file called `__init__.py`**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported. **Packages contain multiple modules and packages** themselves.

#### 1.4.15 Functions: passing arguments by assignment

**Arguments are passed by assignment in Python**. Since assignment just creates references to objects, it depends on the **mutability** of the arguments **if they will be altered or not inside functions**.

**Common immutable type:**

- numbers: `int`, `float`, `complex`
- immutable sequences: `str` (strings), `tuple`

**Common mutable type (almost everything else):**

- mutable sequences: `list`
- mapping type: `dict`
- classes: `ndarray` (numpy arrays), `Series` (pandas one-dimensional array), `DataFrame` (pandas 2-dimensional array)

The function `deepcopy(x)` from module `copy` is available when it is needed to make a copy of a mutable argument to avoid its modification inside a function:

### 1.4.16 Procedures: functions without a return value

A procedure is a sub-routine that does not return a value, but does have side-effects. This could be writing to a file, printing to the screen, or modifying the value of its input.

Therefore, in Python, there is not difference between function and procedures, except **a procedure does not contain a return statement**.

```
def print_cat():
    for i in range(4):
        print 'cat',
```

```
In [21]: import copy

         nums = [1, 2, 3]

         def add_zero_w_copy(l):
             l_tmp = copy.deepcopy(l)
             l_tmp.append(0)

         def add_zero_wo_copy(l):
             l.append(0)

         add_zero_w_copy(nums)
         print nums

         add_zero_wo_copy(nums)
         print nums
```

```
[1, 2, 3]
[1, 2, 3, 0]
```

### 1.4.17 Code Style

- Style Guide for Python Code: **PEP8**.
- Use **only English (ASCII) characters for variables, functions and files**. It is possible to use non-English characters in strings and comments by adding the following line at the beginning of each file: `# -*- coding: utf-8 -*-`.
- Name your **variables, functions and files** consistently: the convention is to use **lower_case_with_underscores**.
- We all use **single-quoted strings** to be consistent. Nevertheless, single-quoted strings and double-quoted strings are the same. PEP does not make a recommendation for this, **except for function documentation** where tripe-quote strings should be used.

- **Constants** should be written in **ALL_CAPITAL_LETTERS** with underscores separating words
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`
- To avoid conflicts with Python keywords, simple add a single trailing_underscore: **abs_**

### 1.4.18 PEP8 exceptions:

**Long lines** It is very conservative and requires limiting lines to 79 characters. We use **all lines to a maximum of 119 characters**. This is the default behaviour in *PyCharm*.

**Disable checks in one line** Skip validation in one lines by adding following comment: `# nopep8`

### 1.4.19 `datetime` data type

The `datetime` module supplies classes for **manipulating dates and times**. **Avoid converting dates or times** to int (`datenum` or similar).

```
In [22]: from datetime import datetime, date, time
         # Using datetime.combine()
         d = date(2005, 7, 14)
         t = time(12, 30)
         dt1 = datetime.combine(d, t)

         print dt1
         print dt1.year
```

```
2005-07-14 12:30:00
2005
```

```
In [23]: from datetime import timedelta

         dt2 = dt1 + timedelta(hours=5)

         print dt2
```

```
2005-07-14 17:30:00
```

**timedelta**(`[days[, seconds[, microseconds[, milliseconds[, minutes[, hours[, weeks]]]]]]]`)

> All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative.

### 1.4.20 `boolean` data type

`boolean` values are the **two constant objects `False` and `True`**. In numeric contexts (for example when used as the argument to an arithmetic operator), they behave like the integers 0 and 1, respectively.

Nevertheless, other values can also be considered false or true: * the following values are considered false: 0, `''`, `[]`, `()`, `{}`, `None` * all other values are considered true, so objects of many types are always true

### 1.4.21 Recommended preferences settings for *spyder*

#### Plots on a separate window

- `IPython console` -> `Graphics` -> `Graphics backend` -> `Automatic`.

  It is necessary to restart *spyder* (or at least *IPython kernel*) to take affect.

#### Activate PEP8 checking

- `Preferences` -> `Editor` -> `Code Instropection/Analysis` -> `Analysis` -> `Style analysis (pep8)`

#### Modify the maximum line length:

**Step 1**

- `Preferences` -> `Editor` -> `Show vertical line after 119 characters`

**Step 2**

- Create a file:

| | Windows | Mac |
|---|---|---|
| file name | `.pep8` | `pep8` |
| folder | user folder (usually `C:\Users\<username>`) | `~/.config` (usually `/Users/<username>`) |

With the following content:

```
[pep8]
max-line-length = 119
```

### 1.4.22 More on `list`

The list data type has some more methods. Here are all of the methods of list objects:

**append**`(x)` Add an item to the end of the list; equivalent to a[len(a):] = [x].

**extend**`(L)` Extend the list by appending all the items in the given list; equivalent to a[len(a):] = L.

**insert**`(i, x)` Insert an item at a given position. The first argument is the index of the element before which to insert, so a.insert(0, x) inserts at the front of the list, and a.insert(len(a), x) is equivalent to a.append(x).

**remove**`(x)` Remove the first item from the list whose value is x. It is an error if there is no such item.

**pop**`([i])` Remove the item at the given position in the list, and return it. If no index is specified, a.pop() removes and returns the last item in the list. (The square brackets around the i in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

**index**(x) Return the index in the list of the first item whose value is x. It is an error if there is no such item.

**count**(x) Return the number of times x appears in the list.

**sort**(cmp=None, key=None, reverse=False) Sort the items of the list in place (the arguments can be used for sort customization, see sorted() for their explanation).

**reverse**() Reverse the elements of the list, in place.

### 1.4.23 List comprehensions

**List comprehensions provide a concise way to create lists**. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
In [24]: squares = []
         for x in range(10):
             squares.append(x**2)

         print squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```
In [25]: squares = []
         squares = [x**2 for x in range(10)]

         print squares

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more for or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it.

### 1.4.24 Lambda expressions

**Small anonymous functions** can be created with the lambda keyword. To create a lambda function first write keyword `lambda` followed by one of more arguments separated by comma, followed by colon sign (`:`), followed by a single line expression. Note that lambda function cannot contain more than one expression.

```
In [26]: print map(lambda x: x**2, range(10))

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**map**(function, iterable, ...) > Apply function to every item of iterable and return a list of the results.

### 1.4.25 Dictionaries

A dictionary is a data type which allows to **store data just like a `list`**, but instead of using only numbers to get the data **it is possible to use strings** or other data types **as the index**. This is very useful for storing and organizing data. Note that **dictionaries are unordered** key-value-pairs.

```
In [27]: tel = {'jack': 4098, 'sape': 4139}

         tel['guido'] = 4127
         print tel

         print tel['jack']
```

```
{'sape': 4139, 'jack': 4098, 'guido': 4127}
4098
```

#### Note

`OrderedDict` is available if you need a ordered dictionary.

```
In [28]: from collections import OrderedDict

         ordered_tel = OrderedDict([('jack', 4098),('sape', 4139),
                                    ('guido', 4127)])
         print ordered_tel
```

```
OrderedDict([('jack', 4098), ('sape', 4139), ('guido', 4127)])
```

### 1.4.26 Sets

A set object is an **unordered collection of distinct objects**.

```
In [29]: s = set([1, 0, 2, 2, 3])

         print s
```

```
set([0, 1, 2, 3])
```

### 1.4.27 One line `if` statement

```
<expression1> if <condition> else <expression2>
```

```
In [30]: age = 15
         # Conditions are evaluated from left to right
         print('kid' if age < 18 else 'adult')
```

```
kid
```

Programming languages derived from C usually have following syntax:
`<condition> ? <expression1> : <expression2>`
The creator of Python, Guido van Rossum, rejected it as non-Pythonic, since it is hard to understand for people not used to C.

### 1.4.28 Logging

Logging is a means of **tracking events that happen when some software runs**. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the level or severity.

The logging is **better than printing** because:

- It is easy to put a timestamp in each message, which is very handy.
- You can have different levels of urgency for messages, and filter out less urgent messages.
- When you want to later find/remove log messages, you will not get them confused for real `print()` calls.
- If you just print to a log file, it is easy to leave the log function calls in and just ignore them when you do not need them. You do not have to constantly pull out `print()` calls.

To print log messages to the screen:

```
import logging
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info('added %s and %s to get %s' % (x, y, z))
```

To write log messages to a file:

```
import logging
logging.basicConfig(filename='log_filename.txt',
                    level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')
logging.info('added %s and %s to get %s' % (x, y, z))
```

The different levels of logging, from highest urgency to lowest urgency, are:

```
logging.critical('This is a critical message.')
logging.error('This is an error message.')
logging.warning('This is a warning message.')
logging.info('This is an informative message.')
logging.debug('This is a low-level debug message.')
```

The `level` argument in `logging.basicConfig` call sets the minimum log level of messages it actually logs.

## 1.5 Scientific libraries

### 1.5.1 NumPy

NumPy's main object is the **homogeneous multidimensional array (`ndarray`)**. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called axes. The number of axes is rank.

```
In [31]: import numpy as np

         # defining arrays and matrices
         Z = np.array([1, 3, 4])

         A = np.array([[1, 1],
                       [0, 1]])
         B = np.array([[2, 0],
                       [3, 4]])

In [32]: # selecting elements
         print A[0, :]

         # elementwise product with * operator!
         print A * B

         # matrix product
         print np.dot(A, B)
```

```
[1 1]
[[2 0]
 [0 4]]
[[5 4]
 [3 4]]
```

```
In [33]: from numpy.linalg import solve, inv  # linear algebra

         a = np.linspace(-np.pi, np.pi, 10)
         print a

         a = np.array([[1, 2, 3], [3, 4, 6.7], [5, 9.0, 5]])
         print a

         b =  np.array([3, 2, 1])
         print solve(a, b)  # solve the equation ax = b
```

```
[-3.14159265 -2.44346095 -1.74532925 -1.04719755 -0.34906585  0.34906585
  1.04719755  1.74532925  2.44346095  3.14159265]
[[ 1.   2.   3. ]
 [ 3.   4.   6.7]
 [ 5.   9.   5. ]]
```

```
[-4.83050847  2.13559322  1.18644068]
```

```
In [34]: print inv(a)
```

```
[[-2.27683616  0.96045198  0.07909605]
 [ 1.04519774 -0.56497175  0.1299435 ]
 [ 0.39548023  0.05649718 -0.11299435]]
```

```
In [35]: print a.transpose()
```

```
[[ 1.   3.   5. ]
 [ 2.   4.   9. ]
 [ 3.   6.7  5. ]]
```

**Warning**

> The **transpose of a 1D array is still a 1D array**. If you want to turn your 1D vector into a 2D array and then transpose it, just slice it with `np.newaxis`.

```
In [36]: print b
         print b.transpose()
         print b[:, np.newaxis]
```

```
[3 2 1]
[3 2 1]
[[3]
 [2]
 [1]]
```

**ndim** the number of axes (dimensions) of the array. In the Python world, the number of dimensions is referred to as rank.

**shape** the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns, shape will be (n, m). The length of the shape tuple is therefore the rank, or number of dimensions, ndim.

**size** the total number of elements of the array. This is equal to the product of the elements of shape.

**dtype** an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

**Warning**

> When operating and **manipulating arrays**, their data is **sometimes copied into a new array and sometimes not**. For example, simple assignments make no copy of array objects or of their data.

**Vectorization** Numpy arrays enable you to express batch operations on data without writing any for loops. This is usually called **vectorization**:

- vectorized code is more concise and easier to read
- fewer lines of code generally means fewer bugs
- the code more closely resembles standard mathematical notation

But:

sometimes it's difficult to move away from the **for-loop** school of thought

### 1.5.2 Pandas

Pandas is a newer package built on top of NumPy and pandas objects are **valid arguments to most NumPy functions**:

- fast and efficient **Series (1-dimensional) and DataFrame (2-dimensional) heterogeneous objects** for data manipulation with integrated indexing
- tools for **reading and writing data from different formats**: CSV and text files, Microsoft Excel, SQL databases, HDF5...
- intelligent **label-based slicing**
- **time series-functionality**
- integrated **handling of missing data**

```
In [37]: import pandas as pd

         # ignore the following commands
         # just for the slides
         pd.set_option("display.max_rows", 10)
         pd.set_option("display.max_columns", 5)

         simar = pd.read_table('WANA_2006008_Algeciras.txt',
                               delim_whitespace=True,
                               parse_dates= {'date' : [0,1,2,3]},
                               index_col='date', skiprows=70)

         simar
```

```
Out[37]:                      Hm0  Tm02  ...   VelV   DirV
         date                            ...
         1996-01-14 03:00:00  0.5   2.2  ...    4.5  176.0
         1996-01-14 06:00:00  0.5   2.3  ...    4.3  193.0
         1996-01-14 09:00:00  0.4   2.3  ...    4.3  193.0
         1996-01-14 12:00:00  0.7   2.6  ...    8.7  118.0
         1996-01-14 15:00:00  0.9   3.0  ...    8.7  118.0
         ...                  ...   ...  ...    ...    ...
         1996-12-31 09:00:00  2.5   4.4  ...   17.1  241.0
         1996-12-31 12:00:00  2.0   4.1  ...   15.4  263.0
         1996-12-31 15:00:00  2.0   4.1  ...   15.4  263.0
```

```
1996-12-31 18:00:00  1.4   3.6  ...    12.4  263.0
1996-12-31 21:00:00  1.4   3.5  ...    12.4  263.0

[2823 rows x 14 columns]
```

**read_table**(. . .)

Read general delimited file into DataFrame.

- `delim_whitespace`: boolean, default False. Specifies whether or not whitespace (e.g. ' ' or ' ') will be used as the sep.
- `parse_dates`: boolean or list of ints or names or list of lists or dict, default False boolean. dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'
- `index_col`: int or sequence or False, default None. Column to use as the row labels of the DataFrame.
- `skiprows`: list-like or integer, default None. Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file
- `header`: int or list of ints, default 'infer'. Row number(s) to use as the column names, and the start of the data. Default behavior is as if set to 0 if no names passed, otherwise None.

```
In [38]: simar['Hm0']  # selecting a single column

Out[38]: date
         1996-01-14 03:00:00    0.5
         1996-01-14 06:00:00    0.5
         1996-01-14 09:00:00    0.4
         1996-01-14 12:00:00    0.7
         1996-01-14 15:00:00    0.9
                               ...
         1996-12-31 09:00:00    2.5
         1996-12-31 12:00:00    2.0
         1996-12-31 15:00:00    2.0
         1996-12-31 18:00:00    1.4
         1996-12-31 21:00:00    1.4
         Name: Hm0, dtype: float64

In [39]: simar[['Hm0', 'Tp']]  # selecting several columns using a list

Out[39]:                      Hm0   Tp
         date
         1996-01-14 03:00:00  0.5  2.7
         1996-01-14 06:00:00  0.5  2.9
         1996-01-14 09:00:00  0.4  2.9
         1996-01-14 12:00:00  0.7  3.2
         1996-01-14 15:00:00  0.9  3.9
         ...                  ...  ...
         1996-12-31 09:00:00  2.5  5.7
         1996-12-31 12:00:00  2.0  5.2
```

```
        1996-12-31 15:00:00  2.0  5.2
        1996-12-31 18:00:00  1.4  4.7
        1996-12-31 21:00:00  1.4  4.7

        [2823 rows x 2 columns]
```

```
In [40]: simar.iloc[0:3]  # selecting rows by position
```

```
Out[40]:                     Hm0  Tm02  ...    VelV   DirV
        date                            ...
        1996-01-14 03:00:00  0.5   2.2  ...     4.5  176.0
        1996-01-14 06:00:00  0.5   2.3  ...     4.3  193.0
        1996-01-14 09:00:00  0.4   2.3  ...     4.3  193.0

        [3 rows x 14 columns]
```

```
In [41]: print simar.loc['1996-01-14 03:00:00']  # selecting rows by label
```

```
Hm0          0.5
Tm02         2.2
Tp           2.7
DirM       185.0
Hm0_V        0.4
           ...
Hm0_F2       0.0
Tm02_F2      0.0
DirM_F2      0.0
VelV         4.5
DirV       176.0
Name: 1996-01-14 03:00:00, dtype: float64
```

```
In [42]: # selecting columns and rows

        print simar.loc['1996-01-14 03:00:00', 'Hm0']  # selection by label
        print simar.iloc[0, 0]   # selection by position
        print simar.ix[0, 'Hm0']  # mixed integer and label based selection
```

```
0.5
0.5
0.5
```

```
In [43]: simar.iloc[:,0]
```

```
Out[43]: date
        1996-01-14 03:00:00    0.5
        1996-01-14 06:00:00    0.5
        1996-01-14 09:00:00    0.4
```

```
        1996-01-14 12:00:00    0.7
        1996-01-14 15:00:00    0.9
                              ...
        1996-12-31 09:00:00    2.5
        1996-12-31 12:00:00    2.0
        1996-12-31 15:00:00    2.0
        1996-12-31 18:00:00    1.4
        1996-12-31 21:00:00    1.4
        Name: Hm0, dtype: float64

In [44]: simar.describe()

Out[44]:                Hm0          Tm02     ...            VelV         DirV
         count  2823.000000  2823.000000     ...     2823.000000  2823.000000
         mean      1.206412     3.432164     ...        9.565604   169.971661
         std       0.729701     0.880544     ...        3.607439    92.598314
         min       0.100000     1.300000     ...        0.000000     0.000000
         25%       0.700000     2.800000     ...        6.800000    80.000000
         50%       1.000000     3.300000     ...        9.600000   191.000000
         75%       1.600000     4.000000     ...       12.000000   260.000000
         max       5.200000     7.400000     ...       20.700000   360.000000

         [8 rows x 14 columns]

In [45]: simar['Hm0'].value_counts()  # histogram

Out[45]: 0.7    246
         0.5    195
         0.6    192
         1.0    189
         0.8    185
               ...
         3.9      4
         4.0      3
         5.2      2
         3.7      2
         4.2      1
         Name: Hm0, dtype: int64

In [46]: simar.dropna(how='all')

Out[46]:                      Hm0  Tm02  ...   VelV   DirV
         date                            ...
         1996-01-14 03:00:00  0.5   2.2  ...    4.5  176.0
         1996-01-14 06:00:00  0.5   2.3  ...    4.3  193.0
         1996-01-14 09:00:00  0.4   2.3  ...    4.3  193.0
         1996-01-14 12:00:00  0.7   2.6  ...    8.7  118.0
         1996-01-14 15:00:00  0.9   3.0  ...    8.7  118.0
         ...                  ...   ...  ...    ...    ...
```

```
1996-12-31 09:00:00  2.5   4.4  ...    17.1  241.0
1996-12-31 12:00:00  2.0   4.1  ...    15.4  263.0
1996-12-31 15:00:00  2.0   4.1  ...    15.4  263.0
1996-12-31 18:00:00  1.4   3.6  ...    12.4  263.0
1996-12-31 21:00:00  1.4   3.5  ...    12.4  263.0

[2823 rows x 14 columns]
```

**dropna**(axis=0, how='any', thresh=None, subset=None, inplace=False)

Return object with labels on given axis omitted where alternately any or all of the data are missing * how: {'any', 'all'}. any: if any NA values are present, drop that label. all: if all values are NA, drop that label * axis: {0 or 'index', 1 or 'columns'}, or tuple/list thereof. Pass tuple or list to drop on multiple axes

```
In [47]: # selecting with complex criteria

         simar[(simar['Hm0'] == 0.5) & (simar['VelV'] == 4.5)]

Out[47]:                      Hm0  Tm02  ...    VelV   DirV
         date                            ...
         1996-01-14 03:00:00  0.5   2.2  ...     4.5  176.0
         1996-08-30 06:00:00  0.5   2.4  ...     4.5  195.0
         1996-08-30 09:00:00  0.5   2.4  ...     4.5  195.0
         1996-10-23 18:00:00  0.5   2.8  ...     4.5   98.0
         1996-10-23 21:00:00  0.5   2.6  ...     4.5   98.0

         [5 rows x 14 columns]

In [48]: simar[(simar['Hm0'] == 0.5) | (simar['VelV'] == 4.5)]

Out[48]:                      Hm0  Tm02  ...    VelV   DirV
         date                            ...
         1996-01-14 03:00:00  0.5   2.2  ...     4.5  176.0
         1996-01-14 06:00:00  0.5   2.3  ...     4.3  193.0
         1996-01-19 21:00:00  0.5   3.7  ...     3.6  251.0
         1996-01-26 21:00:00  0.5   2.6  ...     5.4  178.0
         1996-02-02 12:00:00  0.4   2.2  ...     4.5  243.0
         ...                  ...   ...  ...     ...    ...
         1996-12-07 15:00:00  0.5   2.4  ...     6.1  207.0
         1996-12-08 00:00:00  0.5   2.2  ...     6.5  225.0
         1996-12-15 00:00:00  0.5   2.4  ...     5.8  258.0
         1996-12-16 03:00:00  0.5   2.6  ...     4.0   59.0
         1996-12-26 15:00:00  0.5   2.3  ...     6.8   77.0

         [205 rows x 14 columns]
```

**Warning**

It is necessary to use ***boolean vectors*** to perform this kind of operations to filter the data. The operators are: **| for `or`, & for `and`, and ~ for `not`**. These **must be grouped by using parentheses**.

Otherwise, you will get the following error message: `ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()`.

In recent versions, it is possible to use `query` to create this kind of selection criteria.

```
In [49]: simar.query('Hm0 == 0.5 and VelV == 4.5')

Out[49]:                     Hm0  Tm02  ...    VelV   DirV
         date                           ...
         1996-01-14 03:00:00  0.5   2.2  ...     4.5  176.0
         1996-08-30 06:00:00  0.5   2.4  ...     4.5  195.0
         1996-08-30 09:00:00  0.5   2.4  ...     4.5  195.0
         1996-10-23 18:00:00  0.5   2.8  ...     4.5   98.0
         1996-10-23 21:00:00  0.5   2.6  ...     4.5   98.0

         [5 rows x 14 columns]
```

### 1.5.3 SciPy

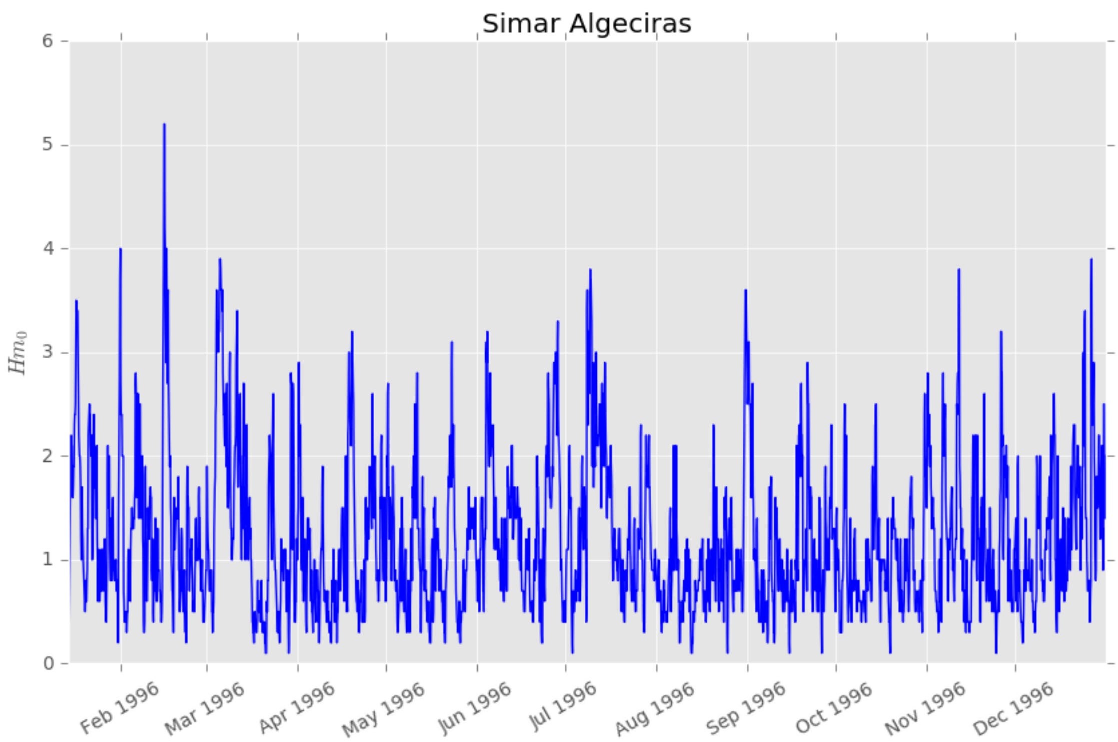
SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python.

- Clustering algorithms (`scipy.cluster`)
- Physical and mathematical constants (`scipy.constants`)
- Fast Fourier Transform routines (`scipy.fftpack`)
- Integration and ordinary differential equation solvers (`scipy.integrate`)
- Interpolation and smoothing splines (`scipy.interpolate`)
- Input and Output (`scipy.io`)
- Linear algebra (`scipy.linalg`)
- N-dimensional image processing (`scipy.ndimage`)
- Orthogonal distance regression (`scipy.odr`)
- Optimization and root-finding routines (`scipy.optimize`)
- Signal processing (`scipy.signal`)
- Sparse matrices and associated routines (`scipy.sparse`)
- Spatial data structures and algorithms (`scipy.spatial`)
- Special functions (`scipy.special`)
- Statistical distributions and functions (`scipy.stats`)
- C/C++ integration (`scipy.weave`)

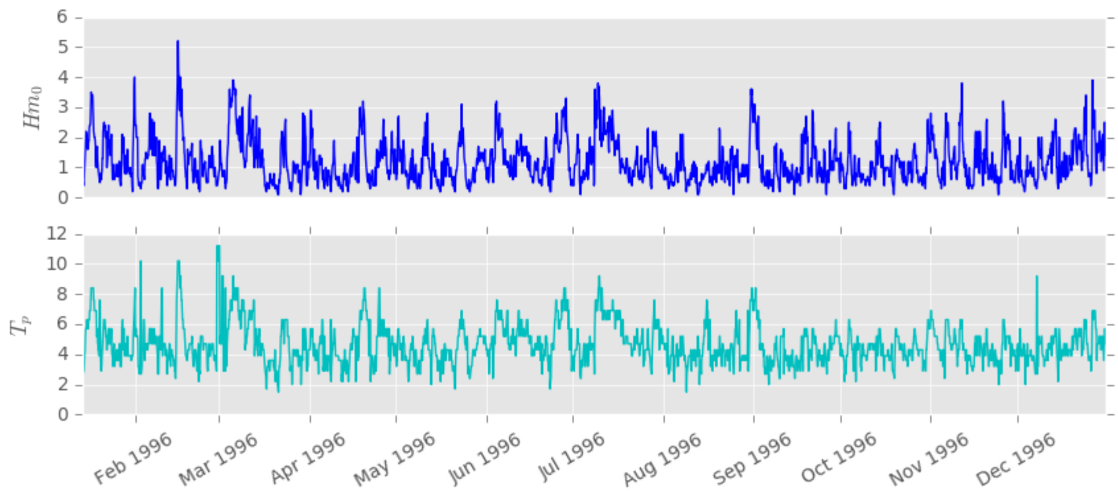
### 1.5.4 matplotlib

`matplotlib` is a library for making plots in Python. The main component of `matplotlib` is `pylab` which allow the user to create plots with code quite similar to MATLAB figure generating code. `matplotlib` has its origins in emulating the `MATLABő` graphics commands.

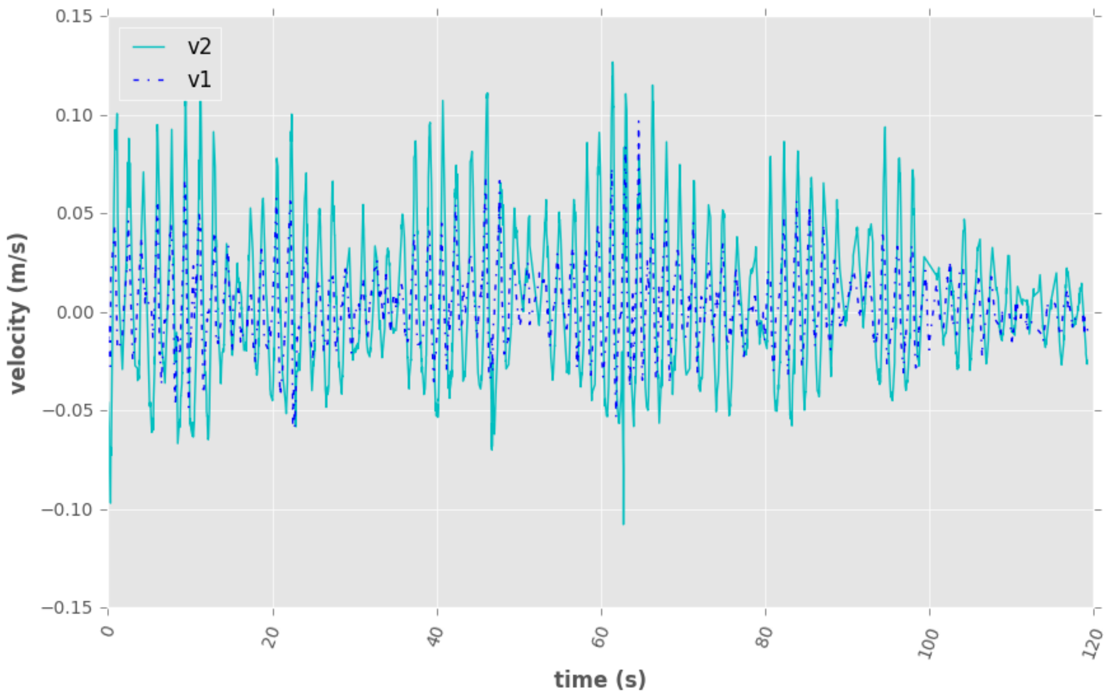
In [50]:
```
# ignore the following command
# just for the slides
%matplotlib inline

import matplotlib.pyplot as plt

plt.figure(1, figsize=(10, 6))
plt.plot(simar.index, simar['Hm0'], 'b')
plt.xticks(rotation=30)
plt.title('Simar Algeciras')
plt.ylabel('$Hm_0$')

plt.savefig('wana.png')  # save to file
plt.show()  # display on screen
```




In [51]:
```
plt.style.use('ggplot')  # pre-defined styles

plt.figure(2, figsize=(10, 6))
plt.plot(simar.index, simar['Hm0'], 'b')
plt.xticks(rotation=30)
plt.title('Simar Algeciras')
plt.ylabel('$Hm_0$')

plt.show()
```




In [52]:
```
plt.figure(3, figsize=(10, 6))

plt.subplot(311)
plt.plot(simar.index, simar['Hm0'], 'b')
plt.ylabel('$Hm_0$')
plt.xticks([])

plt.subplot(312)
plt.plot(simar.index, simar['Tp'], 'c')
plt.ylabel('$T_p$')
plt.xticks(rotation=30)

plt.show()
```




**Fourier Transform (full code)**

```
In [53]: import pandas as pd
         import numpy as np
         import matplotlib.pyplot as plt

         # Input data
         df = pd.read_csv('T130_6_1_2.csv', sep=',',skiprows=2,
                          header=None, error_bad_lines=False, na_values='',
                          skipinitialspace=True)

         df
```

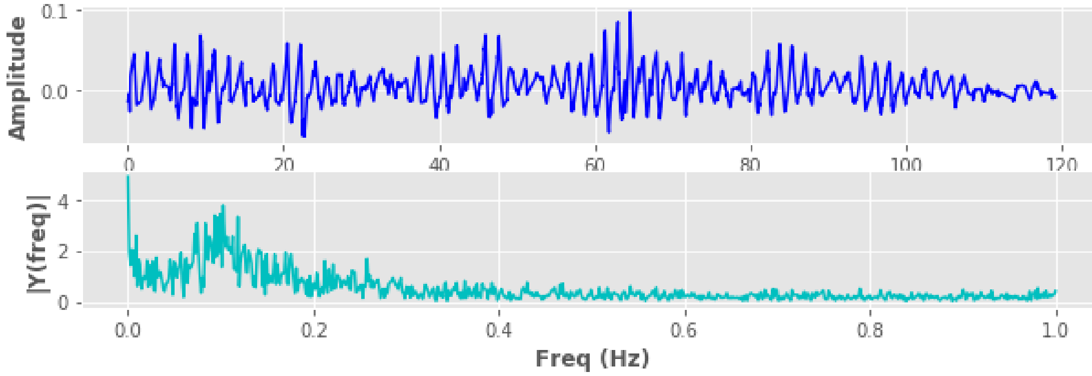
Out[53]:

| | 0 | 1 | ... | 8 | 9 |
|---|---|---|---|---|---|
| 0 | 0.019507 | -0.015088 | ... | NaN | 908.778442 |
| 1 | 0.204670 | -0.005019 | ... | NaN | NaN |
| 2 | 0.205357 | -0.005533 | ... | NaN | NaN |
| 3 | 0.208304 | -0.007504 | ... | NaN | NaN |
| 4 | 0.278389 | -0.027514 | ... | NaN | NaN |
| ... | ... | ... | ... | .. | ... |
| 1669 | NaN | NaN | ... | NaN | NaN |
| 1670 | NaN | NaN | ... | NaN | NaN |
| 1671 | NaN | NaN | ... | NaN | NaN |
| 1672 | NaN | NaN | ... | NaN | NaN |
| 1673 | NaN | NaN | ... | NaN | NaN |

```
[1674 rows x 10 columns]
```

In [54]:
```python
# One-dimensional discrete Fourier Transform
y = np.fft.fft(df[1].dropna())
n = len(y)
y = y[range(int(n/2))]
t = np.linspace(0, 1, int(n/2))  # Frecuency generation

plt.style.use('ggplot')

#  Signal plot
plt.figure(4, figsize=(10, 6))
plt.plot(df[5], df[6], '-c', label='v2')
plt.plot(df[0], df[1], '-.b', label='v1')
plt.xlabel('time (s)', weight='bold')
plt.ylabel('velocity (m/s)', weight='bold')
plt.legend(loc=2)
plt.xticks(rotation=70)

#  Signal and spectral amplitude plots
plt.figure(5, figsize=(10, 8))

plt.subplot(511)
plt.plot(df[0], df[1], 'b')
plt.xlabel('Time', weight='bold')
plt.ylabel('Amplitude', weight='bold')

plt.subplot(512)
plt.plot(t, abs(y), 'c')
plt.xlabel('Freq (Hz)', weight='bold')
plt.ylabel('|Y(freq)|', weight='bold')

plt.show()
```

$$x + 2y + 1$$




### 1.5.5 Sympy

SymPy is a Python library for symbolic mathematics.

In [55]:
```
from sympy import symbols, init_printing

init_printing()  # pretty printing

x, y = symbols('x y')
expr = x + 2*y

expr
```

Out[55]:

$$x + 2y$$

In [56]:
```
expr + 1
```

Out[56]:

$$x + 2y + 1$$

Derivative of $sin(x)e^x$

In [57]:
```
from sympy import diff, sin, exp

diff(sin(x)*exp(x), x)
```

Out[57]:

$$e^x \sin(x) + e^x \cos(x)$$

Compute $\int (e^x \sin(x) + e^x \cos(x))\, dx$

In [58]:
```
from sympy import integrate, cos

integrate(exp(x) * sin(x) + exp(x) * cos(x), x)
```

Out[58]:

$$e^x \sin(x)$$

Compute $\int_{-\infty}^{\infty} \sin(x^2)\,dx$

In [59]:
```
from sympy import oo

integrate(sin(x**2), (x, -oo, oo))
```

Out[59]:

$$\frac{\sqrt{2}\sqrt{\pi}}{2}$$