

# Python's Very Basics

José R. Dorronsoro  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid  
28049 Madrid, Spain

# Outline

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Preliminaries

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Python Sources

- These notes and those at [github](#) for a short, self-contained introduction
- Many other basic sources; some examples:
  - J. Guttag's book, *Introduction to Computation and Programming Using Python* (MIT Press, 2013): chapters 1–5 and 7
    - Assumes Python as the first programming language (C programmers can read the above chapters fast)
    - Plus an introduction to data structures and algorithm analysis
  - The [Python tutorial](#)
  - [Google's minicourse](#)
    - A fast and good introduction to strings, lists, dicts and files that assumes some programming knowledge
    - Plus a good set of exercises on them

## More Advanced Sources

- For more experienced programmers: *Python Cookbook* (O'Reilly 2005)
- For Machine Learning/Data Science: W. McKinney's book, *Python for Data Analysis* (O'Reilly 2012)
  - Main goal: joint introduction to Python and data analysis
  - Good Python essentials summary in Appendix
- And all the documents in [python.org](https://python.org) as well as many web references (some below)
- As well as searches in stackoverflow
- As well as ...

## But Also ...

- Is Python an easy language? Well, the basics yes, but ...
- From Quora: Joshua Engel, on learning Java
  - Learning Java, the language, is the work of an afternoon for a C programmer.
  - Learning Java, the programming environment, with eighty gazillion libraries and dozens of important frameworks, is the work of a dozen lifetimes.
- Perhaps true also for the Python ecosystem?

# Working with Python

- Simplest initial mode: probably to work on a Jupyter notebook (more on them below)
  - Integrates a Python shell and an editor
  - Quite good for small programs; not so for larger ones
- Afterwards: for simple projects, combine a text editor with a Python shell in a edit(+copy)+execute+refine loop
  - Linux has a native shell but IPython is much better
  - One edits the program/module outside the shell, which can reload it automatically, and execute/check it
- For larger projects a Python-supporting IDE (of course!)

# Python Installation

- Best option: install it through Anaconda
  - Either the full Anaconda suite or, often better, the much simpler Miniconda
- Miniconda provides the standard library plus a set of common packages
- We can add further packages with `conda install xxx` , which handles **package dependencies**
- `pip install xxx` is another option but watch out for package dependency conflicts
  - Conflicts are frequent, as Python package development is very distributed and not much coordinated
  - Recommendation: stay updated but not too updated
- Because of this, better install the one (or two) before latest Python 3.X version
  - Python 2.7 is no longer supported: better avoid it (unless absolutely necessary)



# Working From Miniconda

- In Linux one can work right away from a console
- In Windows 10 it installs cmd and Powershell Anaconda prompts
  - The package `m2-base` adds a Linux-like command interface
- To start the IPython/Jupyter tools, open a Linux console or the Anaconda command prompt and
  - Type `jupyter notebook --notebook-dir="xxx"` for the Jupyter Notebook environments, with `"xxx"` as the root dir
  - Type `jupyter lab --notebook-dir="xxx"` for the Jupyter Lab environments (with more functionality/extensions)
  - Type `ipython` for the plain text IPython shell
  - Type `jupyter qtconsole` for the GUI version of the IPython shell

# Jupyter Lab Notebooks

- Browser based interface to develop and document code with tabs for individual notebooks
- Reasonable tool for beginner's Python programming
- Excellent tool for program– or work–flow documentation
- Cells for code, documentation, figures
- In code cells we can
  - Edit sentences or functions
  - Execute them with `Ctrl+Intro`
  - Debug, re–edit and re–execute until OK

# Jupyter Lab Notebooks II

- Text cells:
  - We mark them as Markdown cells with `Esc+m`
  - We can format text with Markdown syntax
  - They also admit formulas with LaTeX notation
- We can display figures from the Matplotlib module after executing the “magic” command `%matplotlib inline`
- Notebooks can be saved as such, as Python scripts, as plain html files or even converted to LaTeX using `nbconvert` (and then, say, to pdf)
- More in [The Jupyter notebook](#)

# Jupyter Qt Console I

- GUI interface with inline figures, multiline editing, syntax highlighting ...
- Can have several tabs opened with different kernels
- Tab completion suggests possible command completions (and also object attributes or function's help)
- Opens with `jupyter qtconsole` in an Anaconda shell
- Easiest use: edit a piece of code with an outside editor, copy-paste it and run it with Enter
- Alternatively, edit a `.py` script with a (non Windows) text editor and run it with magic command `%run`
- Much better: write code as functions in a `.py` module and automatically reload it with the `%load_ext`, `%autoreload` magic commands

# Jupyter Qt Console II

- Magic commands begin with `%`
  - Run modules: `%run module.py`
  - Load scripts for shell editing: `%load module.py` (OK for small files)
  - OS commands: `%pwd`, `%cd`, ...
  - `%quickref` gives a simple IPython cheat sheet
  - `%lsmagic`, `%magic` list available magic functions
- `%alias` prints a list of aliases to common Unix commands
- More in:  
A Qt Console for IPython

# Import and Reload

- Simple way to start programming in Python:
  - Write code adding functions in a `.py` module
  - Import the module into the shell (i.e. let the interpreter know about its functions)
  - Test functions and repeat cycle until OK

- Python 3.X: add the line `import imp` and then reload with

```
imp.reload(mod)
```

- Or much simpler: autoreload option in IPython:

```
%load_ext autoreload      #cargar extension autoreload  
%autoreload 2             #reload all
```

- Automatically reloads `mod` after editing
  - But watch out for syntactic errors: the module is not imported and the previous version used
- More in [A Qt Console for IPython](#)

# First Things

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Objects

- Two general data types:
  - Scalar: `int`, `float`, `complex`, `bool`, `str`
  - Containers: contain an arbitrary number of scalars or of other containers
- In Python everything is an object (and so are, for instance, ints)
  - If `o` is an object, typing `o.` + `tab` in a shell lists its methods
  - Also `dir(o)`
  - Try `dir(1)`
- Python variables are not strongly typed
  - Types are implicitly assigned and checked at runtime

```
a = 1; type(a)
a = 'a'; type(a)
```
  - Explicit type checking with `isinstance(object, type)` or `type(object) is xxx`
- Type casting possible
- Special “value” `None` : absence of value



# Variables and Expressions

- Variables: **names** of objects (not synonyms of memory positions, as in C)
  - `a = 3` is **not an assignment** but a **binding** of `a` with the object `3`
- Python has a number of reserved words:  
`and, print, while, class, lambda, ...`
  - They correspond to types, operators or built in functions
- Leading and trailing single `_` and double `__` are often used for special meanings
  - Good discussion in [The Meaning of Underscores in Python](#)
- Expressions often work as in C
  - `+=, -=, *=` : OK
  - `++, --` do not exist
  - `a // b` : integer division (also `a/b` in Python 2.X if `a, b` integers)
  - `1 / 2 = 0` in Python 2.X, `0.5` in Python 3.X
  - `a**b` : power

# Variable Bindings

- Recall that variables in Python are in fact **names**
- At first sight they look more or less as in C, but there are clear cut differences
- Again, there are not assignments but **bindings** between names and objects
- Names in Python are very different from names in C
  - In Python variable names **are not** synonyms of memory addresses where the variable values are stored
- Global variables: defined elsewhere and identified as `global name`
- Same use (and same problems) as in C

# Scope Rules

- Python follows the LEGB scope Rule
- **L, Local**: names assigned in any way within a function and not declared global in that function
- **E, Enclosing function locals**: names in the local scope of any and all enclosing functions, from inner to outer
- **G, Global** (module): names assigned at the top-level of a module file, or declared global within the file
- **B, Built-in** (Python): names preassigned in the built-in names module

# Variables and Bindings Examples

- Sometimes things may not behave as expected:

```
a = []; b = a; a.append(1); b.append(2)
print (a); print (b)
```

```
a = 10; b = a; a+=1
print (a, b)
```

- Swapping variables is also much different than in C:

```
a, b = b, a
print (a, b)
```

Q: why/how does this work?

# Bindings and Identities

- The `id` function returns the **identity** of an object:
  - An (long) integer guaranteed to be unique and constant for this object during its lifetime (but **not a memory address**)
  - Two names binding to the same object (usually) result in the same id:

```
a = 'aaa'; b='aaa'  
print (id(a), id(b))
```

- But two names binding to the same integer beyond 256 will have different ids

```
a = 1000; b = 1000  
print (id(a), id(b))
```

- Using two different names for a mutable object means that changing one changes the other, but recall ...

```
a = []; b = a; a.append(1); b.append(2); print (id(a), id(b))  
a = 10; b = a; print (id(a), id(b))  
a+=1; print (id(a), id(b))
```

- Names can be destroyed using `del (name)` (kind of `free` in C)
  - Nice discussion on [Python Objects](#)

# Flow Control

- Code blocks are identified by their **indentation**:
  - Recommendation in [PEP 0008 – Style Guide for Python Code](#): 4 white spaces, no tabs
  - Results in highly structured code
  - But watch out for silly errors (as mixing blanks with tabs)

- Selection: `if condition:/elif condition:/else:`

- Iterations through `while` and `for`; no `do while` construction

- While iteration:

```
while condition:  
    code block
```

- For iteration:

```
for var in sequence:  
    code block
```

- `sequence` has to be an **iterable** object such as strings, and lists, tuples, files, ...

# Loop Control Statements

- `break` : the loop terminates and execution goes to the statement immediately following the loop
- `continue` : the remainder of the loop body is skipped and execution goes to checking the loop's condition
- `pass` : used when a statement is required but do not want any command or code to executed
  - For instance, to leave temporarily an empty code block

## More on `for`

- Try always to iterate over existing iterables and avoid C-style thinking over Python loops:

```
#on Python 3 do this only if needed
for i in range(1000000):
    print i
```

```
#never do this!!
for i in list(range(1000000)):
    print i
```

- `range(N)` defers the creation of the list element until it is needed
- The `while` and `for` equivalence in C does not translate to Python
- More on iterables, `iterators` and `generators` later on



# Strings

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Strings

- Alphanumerical characters between `'` or `"`: `a = 'aaa'`
- First **immutable** object: their individual elements cannot be changed
- Standard operators overload on strings:

```
str1+str2, int_*str_, str1 < str2
```

- `len(string)` returns its number of characters
- String elements accessible by indices: `a[0]`, `a[-1]`, `a[-2]`
- **Slicing** is used for substring access:

```
a[1:3], 'abc'[1:3], a[: -1]
```

- `sss[F:L]` extracts values of indices `F` to `L-1`
- **Extended slicing**: `sss[F:L:s]` extracts values of indices `F` to `L-1` by step `s`
  - `s[ : : -1]` inverts the `s` string (or any array)

# String Methods

- Very useful tools for string handling
- `s.lower()`, `s.upper()` : return lower or upper case versions of `s`
- `s.isalpha()`, `s.isdigit()` : tests if all the chars in `s` are of the corresponding type
- `s.find(string)` : searches for `string` and returns the first index where it begins or -1 if not found
- `s.replace(s_old, s_new)` : returns a string with `s_old` replaced by `s_new`
  - `s.replace(' ', '')` trims all blank space in `s`
- `s.split(delim)` : returns a list of substrings separated by the given delimiter
  - `s.split()` splits `s` over any sequence of white space characters

## String Methods II

- The `separator.join(sequence)` construct uses Python's `join` function to put together the `sequence` list of strings separated by the string `separator`

```
s = 'XYZ'.join( ['a', 'b', 'c', 'd'] )
```

- `join` is the “inverse” of `split` :
  - `s.split('XYZ')` splits `s` in its substrings delimited by `XYZ`
- Frequent use when replacing bash files with Python scripts:
  - Form a Unix command string `str_cmd`
  - Execute it with `os.system(str_cmd)`

## More on Strings

- Multiple line string literals possible ending each line with a backslash \
  - We can also use \ to span expressions on multiple lines
  - We can also put in multiple lines Python expressions inside parenthesis
- **Raw strings:** literals preceded by `r`, as in `r'abc\edf\ghj'` that are not processed: `r'a\nb'` prints as `a\nb`
- Everything is Unicode in Python 3.X
- The `string` library contains several useful string constants:
  - `string.ascii_letters` : the concatenation of the `ascii_lowercase` and `ascii_uppercase` constants
  - `string.digits` , `string.hexdigits` , `string.octdigits`
  - `string.punctuation`
  - `string.whitespace`

# String Examples

- Just to play with them

- `s = 'abc'; s+s; 10*s, len(10*s)`
- `(3*s)[1:6]; (3*s)[: -1]; (3*s)[ : : -1]`
- `(3*s).replace('a', 'A')`
- `s = ';' .join( ['a', 'b', 'c', 'd'] ); s.split(';')`
- `s = '1 2 3 4 5'; s.split(' '); s.split()`
- `import string; string.digits; string.whitespace`

# Printing (Old Style)

- Python's `print` can be made to work like C's `printf()` using the `%` format operator
- To do so one defines a string to be printed where
  - Inside the string `%d`, `%f`, `%g`, `%s` ... are used to define formats
  - At the right `%` precedes a tuple with the values to be printed

- Example:

```
a=3, b=3.1416, c='abcdefgh'
text = "int: %d float: %f string: %s" % (a, b, c)
print (text)
```

- Format delimiters of the form `%[flags][width][.precision]type` can be used to define the number of characters `width` and of decimal digits `precision`
  - Typical flag: `0` for 0-padded numerical values

# Pythonic Printing: `format` Method

- Apply the `format` method to a string mixing text and formatting code
- The format contains one or more format codes (fields to be replaced) embedded in constant text
- The format codes are surrounded by `{ }`
- Inside `{ }` one has a positional parameter, plus `:`, plus a format string

```
"Second argument: {1:3d}, first one: {0:7.2f}".format(47.42,11)
```

```
"Art: {a:5d}, Price: {p:8.2f}".format(a=453, p=59.058)
```

```
"various precisions: {0:6.2f} or {0:6.3f}".format(1.4148)
```

- More in [Python3 Tutorial: Formatted Output - Python Course](#)



# Basic Console Input/Output

- `input([prompt])` prints the optional string `prompt` on the shell console and returns a string after `Enter` dropping the newline char `'n'`

```
num = input("Enter a 3 to continue .....\\n")
      Enter a 3 to continue .....      3+Enter

print(4*num, 4*int(num))
      3333 12
```

- `eval(expression)` processes the string `expression`

```
x = 1
eval('x+1')
```

- `input` and `eval` can be used jointly to process console inputs

```
#val will get the value of an expression
val = eval(input('enter an expression: '))
      enter an expression:  4*'s' + Enter

print(val)
      ssss
```

# Functions

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Functions

- Definition

```
def name(parameters):  
    function body
```

- Function call: expression with value the returned value or `None`
- Call by value or by reference? In fact none of them
  - In C the terms value or reference correspond to variables as synonyms of memory addresses
  - In Python immutable objects are usually called by (a kind of) value and mutable by (a kind of) reference (but watch out!)
- Python uses **call by object** or **call by object reference**: if you pass a mutable object into a function/method:
  - It gets a reference to that same object and can be mutated with effects in the outside scope
  - But if the object's name is rebound in the function, the outer scope will know nothing about it and no further outside changes are made

# Python's Function Call Handling

- In C we have the **heap** and the **stack**
- In Python we have (global) **objects** and **frames**
- Frames are essentially dynamic blocks of pointers to objects
- There is a global frame for global objects (data, functions and so on)
- When called, each function creates its own dynamic frame (with its local variables)
- Good (recursive) visualization of frame and object evolution in the [Python Tutor](#) web page

## An Example

- Bisection search for square root (from Guttag, p. 28):
- The following Python code yields approximate values to  $\sqrt{x}$  for a given  $x \geq 1.0$  with precision  $\text{eps}$ :

```
def bisect_sqrt(x, eps=1.e-3):  
    """... docstring ..."""  
    assert x >= 1., "error: input {0:f} < 1.".format(x)  
  
    left = 1.; right = x; sqr = (left+right)/2  
    while abs (sqr**2 - x) > eps:  
        if sqr**2 < x:  
            left = sqr  
        else:  
            right = sqr  
        sqr = (left+right)/2  
    return sqr
```

- Exercise: change things to get a function `cube_root(x, eps)` that approximates the cubic root of  $x \geq 1$

# Calling Functions

- When a function is called
  - ① The function's frame and **namespace** are created
  - ② If needed, parameter expressions are evaluated and parameter names are bound to their results
  - ③ The function body is executed (and more names may be added to the name space) until a return is reached
  - ④ The return value is bound according to the function call expression and the namespace is (usually) destroyed
- Multiple returned values are possible (well, in fact, no: they are actually tuples)
- Values are bound to parameters either positionally or through the formal parameter names
- This is exploited using default values

# Argument Default Values

- Argument order may be changed if we use default values

```
def print_name(first_name, last_name, reverse):  
    #function's body: exercise  
  
#callable as:  
print_name('Jose', 'Dorrnsoro', False)  
print_name(last_name='Dorrnsoro', first_name='Jose', reverse=False)
```

- Default values are defined in the form `arg=value`

```
def print_name(first_name, last_name, reverse=False):  
    #...  
  
#callable as:  
print_name('Jose', 'Dorrnsoro')  
print_name('Jose', 'Dorrnsoro', True)
```

# Passing Arguments

- In more detail: when a function is called,
  - The **positional arguments** are actually packed up into a **tuple** (`args`)
  - The **keyword arguments** are packed up into a **dict** (`kwargs`) with the variable names as keys
- Tuples are ordered and immutable, so we **cannot move** positional arguments around
- Dicts are not ordered and their objects are accessed through their keys; thus we **can move** `kwargs` around
- But cannot use a non keyword argument after a keyword one:  

```
print_name('Dorronsoro', first_name='Jose', False) #error
```
- More on tuples and dicts below



# Docstrings

- Given by a string contained between two triple quotes (`'''docstring'''` , `"""docstring"""` ) right after the `def` sentence
- Standard content:
  - A one line description of the function.
  - A full description after an empty line
  - A description of its parameters, returns and their types
- Standard parameter format: reStructured text (reST) / Sphinx, which can be used to generate documentation automatically
  - Other frequent options: Google and Numpy formats
- **Very important:** every function must have them
- More on [Stack Abuse: Python Docstrings](#)

# Docstring Use

- Example

```
def bisect_sqrt(x, eps):  
    """Computes a square root of x by the bisection method.  
  
    Returns an approximation to the square root of x up to a  
        precision eps  
  
    Args:  
        x (float): the number whose square root we want  
        eps (float): the precision wanted  
  
    Returns:  
        sqr (float): the approximate square root  
    """  
    ...
```

- `help(bisect_sqrt)` in shell displays arguments and docstring
- `bisect_sqrt(` in shell opens window with help
- `pdoc --html -o . my_module.py` writes a file `my_module.html` with (among others) the docstring info
  - Installed through the `pdoc3` module

# Functions as Function Arguments

- In Python functions are **first class objects**: they can be used as any other object (say, a float or a list)
- They can appear in expressions
- They can be list objects
- They can be function arguments:

```
def square(n):  
    return n**2  
  
def list_func_values(n, f):  
    l_func_vals = [f(i) for i in range(n)] #list comprehension  
    return l_func_vals  
  
list_func_values(10, square)
```

# Functions Inside Functions

- Functions can be defined inside other functions
- **Decorators** exploit this to dynamically add new functionalities to previous functions

```
def log_it(my_func):  
    def logging(*args, **kw):  
        print("..... executing %s ....." % my_func.__name__)  
        result = my_func(*args, **kwargs)  
        return result  
    return logging  
  
@log_it  
def add(*l):  
    return sum(l)  
  
>>> add(1, 2, 3, 4)
```

- Used for adding timers, loggings and so on without writing extra boilerplate code

# Euclid Meets Python

- Python tries to build a kind of programming culture: [PEP 0008 – Style Guide for Python Code](#), [The Elements of Python Style](#)
- The [Zen of Python](#) contains short design guiding principles
- Pythonic code follows this one:  
*There should be one — and preferably only one — obvious way to do it*
- An example (?): Euclid's algorithm

```
def mcd(x, y):  
    while(y):  
        x, y = y, x % y  
    return x
```

- By the way: in Python (almost) everything is `True` except 0 and "empty" things: `[]`, `""`, `set()`

# True or False?

- But `and` and `or` have some quirks
  - `x and y` returns `y` if `x` is true, and `x` if not
  - `x or y` returns `x` if true, and `y` if not
  - Apply `bool()` to get `True`, `False`

- Examples:

```
10 and [] , [] or 0.
```

- Sometimes quite useful: assume we want their first character when two strings coincide

```
c = 'abc'; s = 'efg'  
c == s and s[0]
```

```
c = 'abc'; s = 'abc'  
c == s and s[0]
```

- To simplify things (?) Python has the functions `all()`, `any()`
  - `all(xx)` returns `True` if there are no `False` elements in iterable `xx`
  - `any(xx)` returns `True` if there is at least a `True` element
- To have some fun, check `all([])`, `any([])`

# Structured Types

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Structured Types

- Python has five structured types: strings, tuples, lists, dicts and sets
- Recall that **strings** are ordered sequences of chars, each accessible through an index
- They are immutable
- They have a large set of very useful methods
- Strings can be concatenated, indexed and sliced, and we can find their length through `len`
- `str(object)` transforms `object` into a string with results depending on what the object is
- More generally `type(object)` transforms when possible `object` into a another of type `type` with results depending on how the object is defined/programmed



# Tuples

- **Tuples: immutable** ordered sequences of values possibly of different types accessible through an index
- Examples

```
a = ('a', 1, 'b', 2); b = 'a', 1, 'b', 2  
a == b
```

- **Empty tuple:** `tup = ( )` ; **one element tuple:** `tup = ( 'a', )`
- Tuples are **immutable**: their individual elements cannot be changed
- Tuples can be concatenated, indexed and sliced, and we can find their length through `len`
- Apparent multiple returns in functions are actually handled as tuples
- Tuples are the immutable cousins of lists

# Lists

- **List: mutable** ordered sequences of values possibly of different types, each accessible through an index
- Perhaps the most used structured type in Python
- Lists can be concatenated ( + ), indexed and sliced
- Empty list: `l = [ ]`
- `len(l)` returns the number of objects
- Implemented as dynamic arrays
  - Adding or removing items at the end is fast
  - Not so in other positions
  - Efficient data structure for stacks (but not so for queues)

# List Methods

- Some list methods: `l.append(object)`, `l.count(object)`, `l.sort()`, `l.reverse()`, `l.remove(object)`, `l.insert(index, object)`, `l.pop(index)`
- Some of them such as `sort()`, `reverse()` are in place and return `None`
  - `sorted(l)` returns a sorted version of `l`
- `l.index(object)` returns the index where `object` is or raises an exception
  - To just check whether `elem` is in `l`, simply use `if elem in l:`
- The function `tuple` changes (freezes) a list into a tuple
- The function `list` changes (thaws) a tuple into a list
- List **comprehension** is an efficient way to generate particular lists

```
oddN = [2*n+1 for n in range(10)]
```

  - Also works for dicts and sets

# Lists as Function Arguments I

- Python allows to use lists to pass arguments to a function
  - Thus you can build a list `l` with the arguments of a (long) call
  - And pass it to the function as `*l`
  - `*l` **unpacks** the existing list `l` into its elements (more later on)
- Example:

```
def some_f(arg0int, arg1float, arg2string, arg3tuple):  
    print (str(arg0int)+str(arg1float)+  
           arg2string+str(arg3tuple))  
  
#callable as:  
l = [1, 3.14, 'abcd', ('a', 'b', 'c', 'd')]  
some_f(*l)
```

## Lists as Function Arguments II

- Putting `*args` (or `*xxx`) as the last item the argument list of a function `fff` allows `fff` to accept an arbitrary number of positional arguments

```
def my_sum(*args):  
    return sum(args)
```

```
my_sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
my_sum(*l)
```

- Putting `**kwargs` as the same effect with a list of keyword arguments passed as `keyword:value`
- More on this later on and also in [Control Flow](#) section of The Python Tutorial

# Iterators

- Iterators are objects that support two methods:
  - `__iter__` that returns the iterator object itself; used in `for` and `in` statements
  - `__next__` or `next()` that returns the next value from the iterator or raises the `StopIteration` exception if there are no more items to return
- They are usually built as `new = iter(old)` where `old` must be another iterator or a sequence (e.g. a list)

```
l = [1, 2, 3]
iter_l = iter(l)
next(iter_l)
for o in iter_l:
    print(o)
```

```
next(iter_l) #exception
```

- The repeated application of `next` exhausts the iterator

# Generators

- Generators are “lazy” iterators created using a function with a `yield` keyword

```
def counter(low, high):  
    while low <= high:  
        yield low  
        low += 1
```

```
counter(1, 5)
```

```
for l in counter(1,5):  
    print(l)
```

- The function remembers its state in its last execution and starts from it in a new call
- Generators are lazy in the sense that values are generated just when they are needed
- Generators can be created with a variant of list comprehension replacing `[ ]` with `yield` with parentheses

```
def counter(low, high):  
    return (yield(x) for x in range(low, high+1))
```

## filter, map and reduce

- `filter(function, sequence)` returns a sequence (i.e., list, tuple) with the items from `sequence` for which `function(item)` is true

```
par = lambda x: x % 2 == 0
list(filter(par, (1, 4, 9, 16, 25)))
```

- `lambda` is used to define inline simple functions
- `map(function, sequence)` calls `function(item)` for each item in `sequence` and returns a list with the values

```
cube = lambda x: x**3
list(map(cube, filter(par, range(1, 11))))
```

- `reduce(function, sequence)`
  - Calls the **binary** function `function` on the first two items of the `sequence`, then on the result and the next item, and so on
  - Returns the single value finally computed

```
from functools import reduce
prod = lambda a, b: a*b
fact = lambda n: reduce(prod, range(1, n+1))
fact(5)
```



- `zip` joins several lists of the same length in a single list of tuples made of the elements on each list

```
l_1 = range(10)
l_2 = [i*i for i in l_1]
#bien
for a, b in zip(l_1, l_2):
    print(a * b)
#peor
for l in zip(l_1, l_2):
    print(l[0] * l[1])
```

- `enumerate` allows to iterate on a list and its indices:

```
l_2 = [i*i for i in l_1]

for i, sq in enumerate(l_2):
    print("el cuadrado de {0:2d} es {1:4d}".format(i, sq))
```

# Dictionaries

- **dict**: built in implementation of ADT dictionary
- Can be seen as unordered lists with elements of the form  
key:value
  - Elements are **accessed by key values** and not indices
- Empty dict: `d = { }`
- Adding elements: `d.update({ 'a':'alpha' })` , `d['a']='alpha'`
- The `keys()` method returns a list with the (unordered) key values
- The `values()` method returns a list with the `dict` values
- The `items()` method returns a list of key–value tuples
- We can iterate on the keys of a dict `d`: `for k in d:`
- The statement `k in d` returns `True` if the key `k` is in the dict `d`

## args and kwargs Revisited

- We can define functions with an arbitrary number of positional and keyword arguments using `*args` and `**kwargs`
- In the following definition

```
def do_something(*args, **kwargs):  
    # whatever ...
```

Python assumes that `do_something` will get a first set with a variable number of arguments and then a set with a variable number of keyword arguments

- If we call it as

```
do_something(pa1, pa2, pa3, kw1=kwa1, kw2=kwa2)
```

the tuple `(pa1, pa2, pa3)` and the dict `{'kw1':kwa1, 'kw2':kwa2}` are passed to the function's body

- Typical uses:
  - Writing higher order functions that pass arbitrary values to inside functions
  - Understanding others' code

# Packing and Unpacking

- \* **unpacks** the values of an iterable object:

```
print([1, 2, 3, 4])  
#[1, 2, 3, 4]  
print(*[1, 2, 3, 4])  
#1, 2, 3, 4
```

In the second call the four values are passed to `print` as separate argument

- We can also use \* to **pack** values into a list

```
*l_1, = 1, 2, 3, 4  
#[1, 2, 3, 4]; watch out for the ','
```

- \*\* **unpacks** the key-value pairs of a dict

```
num_dict = {'a': 1, 'b': 2, 'c': 3}  
num_dict_2 = {'d': 4, 'e': 5, 'f': 6}  
new_dict = {**num_dict, **num_dict_2}  
# {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}
```

# Packing Function Arguments

- Python functions can take a variable number of arguments
- Parameter names `*args` and `**kwargs` are used to define functions which can receive a variable number of positional or keyword arguments
- When called, the positional arguments are packed into a tuple and the keyword arguments into a dict that the function's body knows how to process

```
def do_something(*args, **kwargs):  
    for arg in args:  
        print(arg)  
    for key in kwargs:  
        print(key, '=', kwargs[key])  
  
#equivalent calls  
do_something(1, 2, 3, a=11, b=22)  
do_something(*(1, 2, 3), **{'a':11, 'b':22})
```

## More on dicts

- To be searched efficiently, dicts are under the hood hash tables
  - If not, dict searches might require to examine the entire dict, with an  $O(N)$  cost, with  $N$  the number of items in the dict
- Pairs `key:value` are placed in buckets determined by `hash(key)`, with the number of pairs in a bucket being small
  - This guarantees  $O(1)$  search costs
- To be eligible for a key, an object must support the `__hash__` and `__cmp__` or `__eq__` methods
  - Tuples can be dict keys, as they are immutable
  - But lists cannot, as they are mutable and cannot be hashed

# Sets

- **set**: collection of different elements
- Initialization: `s = set()`
- Some methods:
  - `add, pop` : adds an object, removes and returns an object
  - `remove, clear` : removes an object, removes all objects
  - Membership: `in, not in`
  - `union, intersection, difference, symmetric_difference`
  - `issubset, issuperset`
- `len(s)` : number of objects in `s`
- `set(iterable)` : builds a set with the **unique** objects in the iterable

# Removing Duplicates

- A usual task is to remove duplicate elements in a list
- Doing it a la C:

```
l_1 = [1, 2, 3, 1, 2, 3]
l_2 = []
#to avoid
for item in l_1:
    if item not in l_2:
        l_2.append(item)

print(l_2)
```

- The Pythonic way:

```
l_1 = [1, 2, 3, 1, 2, 3]
l_2 = list( set(l_1) )
#much better
print(l_2)
```



# Files and Modules

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# Working with Files

- Files are used through a **file handle**:

```
fName = open('file', 'w')
```

- A handle can be opened also with 'r', 'a', 'b', 't', ...

- Once the handle `fName` is defined, we can then use:

```
fName.read(size) # to read the next size bytes
fName.read() # to return a string with the entire file
fName.readline() # to return a string with the next line
# to return string list with each of the file lines:
fName.readlines()
fName.write(string)
#to write the strings in the list S as file lines:
fName.writelines(S)
fName.close()
```

- In Python a file is a sequence of lines; thus we can loop through a file

```
fName = open('file', 'r')
for line in fName:
    print line[:-1] #-1 avoids the final line break
```

## seek and tell

- The `seek(offset)` method resets the file's current position at `offset`
- Positions are computed in terms of bytes since the file begins
  - Essentially the number of the file's ANSI characters, including `'\n'`

```
#examplefile.txt: file with 5 lines with five characters
f = open('examplefile.txt', 'r'); c = 0
for l in f:
    c += len(l)
    print(c)

f.seek(19)
for l in f:
    print(l[:-1])
```

- `f.seek(0)` rewinds the file `f`
- The `tell()` method returns the file's current position

```
f.seek(0); chunk = 20
while len(f.read(chunk)) == chunk:
    print( f.tell() )
print ( "file has %d characters" % f.tell() )
```

# Modules

- Files `*.py` containing statements, function definitions, global variables, etc.
- The `import` statement binds a module within the scope where the import occurs

```
import my_module as mm
```

- If the file `my_module.py` has been changed after its import, it has to be reloaded to update the previous binds:

```
reload(mm)
```

- `reload` performs syntactical checking
  - Easier automatic reload in Ipython with the `autoreload` extension
- Module functions (or variables) are used through object (dot) notation: `mm.funcname( ... )`

# Using Modules

- Example (from Guttag, p. 52):

```
#module circle.py
pi = 3.1416

def area(radius):
    return pi*(radius**2)

#using circle.py
import circle #or reload(circle)
pi = 2
print pi
print circle.pi
print circle.area(1)
```

# Module Variables

- Python modules can be run by `python module.py [arg_1, ...]`
- Or, better, by `module.py [arg_1, ...]` if the first line in `module.py` is the Python shebang `#!/usr/bin/env python`
- When the Python interpreter reads a source file, it defines some special variables and executes its (executable) code
  - If `xxx.py` is directly run from the Python interpreter, the special `__name__` variable is set to `'__main__'`
  - If `xxx.py` is being imported from another module, `__name__` is set to `'xxx'`
- Usually the following elements appear in a module to be run as a standalone program:

```
def main(...args...):  
    #main's body  
  
if __name__ == "__main__":  
    main(...args...)  
else:  
    #lo que sea
```

# Important Modules

- There are Python modules for almost everything: see for instance [UsefulModules](#)
- Modules that are often imported are
  - `sys`, `os` for OS-related tasks (see next)
  - `math` for standard math operations
  - `matplotlib` for plotting (to be seen later on)
  - `numpy` for linear algebra (to be seen later on), `scipy` for scientific computing
  - `pandas` for index-field computing with tables (to be seen later on)
  - `sklearn` for machine learning (to be seen later on)
  - `statsmodels` for statistics

# The `os` and `sys` Modules

- `os` provides interfaces to operating system dependent functionality, such as, for instance
  - `os.chdir(path)` changes the interpreter's active directory
  - `os.system(command)` executes the command in the string `command` in a subshell
- `sys` provides access to some interpreter variables and to functions that interact strongly with the interpreter.
- `sys.path` is a list of strings that specifies the search path for modules
  - Add new dirs using `sys.path.append()`
- `sys.argv` is a list containing command-line arguments
  - Thus `len(sys.argv)` gives the number of command-line arguments
  - `sys.argv[0]` is the script name



# An Example: Redirecting Data Streams

- `sys.stdout` contains the current stdout stream

```
stdout = sys.stdout
f = open('out.log', 'w')
sys.stdout = f
#some code ...
sys.stdout = stdout
f.close()
#more code ...
os.listdir('.')
```

- The file methods apply to the standard data streams:

```
sys.stdout.write("Hello world!\n")
sys.stdout.write("Enter value\n")
sys.stdin.readline()[:-1]
```

# The `pickle` and `gzip` Modules

- `pickle` provides methods to **serialize** Python data structures, i.e., to transform them into a format that can be stored in a file
- `pickle.dump(obj, file, protocol=None)` pickles the object `obj` and saves it into an open `file`
- `pickle.load(file)` reads a pickled object representation from the open `file`
- The `pickle` methods can be used with files compressed with methods from the `gzip` module
- `gzip.open(filename, mode='rb', compresslevel=9)` opens a gzip-compressed file and returns a file object
  - The `mode` can be any combination of `r`, `w`, `a` and `b`, `t`

# Passing Command Line Arguments

- The following gives a basic way of passing command line arguments to a module `myMod`

```
$ ./my_mod.py arg1 arg2 arg3
```

provided we define a `main` function more or less as follows:

```
#!/usr/bin/env python
# coding: utf-8
def main(args):
    if len(args) != 2:
        print "incorrect number of arguments ..."

    var1 = int(args[0])
    var2 = float(args[1])

if __name__ == '__main__':
    main(sys.argv[1:])
```

- The **shebang** `#!/usr/bin/env python` tells bash to use the Python interpreter to process the containing file
- More complete parsing of command line arguments can be done with the `argparse` module

# NumPy

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# The NumPy Library

- NumPy (Numerical Python): package for basic scientific computing and data analysis
  - Very efficient C programming under the hood
  - Different and better than Python standard arrays
- Importing: `import numpy as np`
- Using: `xxx = np.yyy(zzz)`
- (Bad) Alternative: `from numpy import *`
  - Then we can write `xxx = yyy(zzz)`
  - And end up with insidious problems
  - Better not to use this to avoid potential naming conflicts
- Array: basic NumPy data structure

# NumPy Arrays

- Can have elements of any type
- Building arrays:

```
d = np.array([[1,2,3], [4,5,6]], dtype=float)
```

- First array methods:

`xx.shape`, `xx.size` : dimensions of the array `xx` and overall size  
`xx.astype(type)` : type change

- Have to distinguish arrays from lists (or dicts or tuples):

```
d1 = [[1,2,3], [4,5,6]] #list of lists
d1.shape #error
d = np.array(d1)
d.shape # (2,3)
d.dtype # int
```

- But many basic things are done in just the same way

# Working With Arrays

- Special array creation functions

```
d = np.zeros( tuple )
d = np.ones( tuple ) #also: np.empty, np.eye
i_vals = np.arange(10, dtype=int)
lin_vals = np.linspace(start=0, stop=100, num=101)
```

- Or simply append things on a list and convert it: `a_1 = np.array(l)`

- NumPy data types

- `intX`, `uintX`: signed and unsigned X=8,16,32,64-bit integer types
- `floatX`: X=16,32,64,128-bit floating point types
- **different** from those of Python

- Also `complex`, `boolean`, `str`, `unicode`, ...

- Special `float` values: `numpy.inf`, `numpy.nan` (not a number)

- Warning: cannot use equality to test NaN

## Working With Arrays II

- We can clip elements in arrays: `clip(a, a_min, a_max)`
- Arrays can be reshaped as long as the overall size remains constant

```
v0 = np.random.rand(365*24)
v1 = v0.reshape(365, 24)
```

```
v0.shape
v1.shape
v1.flatten().shape
```

- Arrays can be stacked along different axes

```
x0 = np.random.normal(-1., 1., 1000); x0.shape
x = x0.reshape(1000, 1); x.shape
y = np.random.normal( 1., 1., 1000).reshape(1000, 1)
```

```
z = np.hstack((x, y)); z.shape
v = np.vstack((x, y)); v.shape
```

```
p = np.concatenate((x, y), axis=1)
q = np.concatenate((x, y), axis=0)
```



# Array Input and Output

- `np.loadtxt` loads text matrices/tables into arrays

```
#csv file in array.txt  
arr = np.loadtxt('array.txt', dtype='str', delimiter=',')
```

- Default values for `dtype` and `delimiter` are `float` and `whitespace` respectively

- `np.savetxt` writes an array to a delimited text file

```
x = y = z = np.arange(0.0,5.0,1.0)  
np.savetxt('xyz.tex', (x,y,z), delimiter='&')
```

- `np.load` : loads arrays in binary uncompressed/compressed formats `.npy`, `.npz`

- `np.save`, `np.savez` : save arrays in formats `.npy`, `.npz`

```
np.save('xyz.npy', (x,y,z))  
np.savez('xyz.npz', (x,y,z))  
%ls xyz*
```

# Index Handling in NumPy

- Conditions on array values can be captured as boolean arrays:

```
x = np.random.normal(0., 1., 100)

ind_pos = x >= 0.; ind_neg = x < 0.
num_pos = ind_pos.sum() #; num_neg = ind_neg.sum();

np.logical_and(ind_pos, ind_neg)
np.logical_or(ind_pos, ind_neg)
```

- And also as index values (returning tuples):

```
ind_values_pos = np.nonzero(ind_pos)
ind_values_neg = np.nonzero(ind_neg)
```

- The elements complying a condition can also be selected:

```
x = np.random.normal(0., 1., 100)
#select from x the elements with square >= 1.
np.select([x**2 >= 1.], [x])
```

- Alternatively `np.where` returns arrays of indices of condition complying elements

```
np.where(x**2 >= 1)
```

# Array Operations and Ufuncs

- Basic array operations: usually elementwise
  - Arithmetic operations overload when working with equal size arrays: `arr_c = arr_a + arr_b`
  - Scalar operations work (more or less) as expected:  
`1/arr , arr**0.5`
- Unary and binary **universal functions**: also perform elementwise operations
- Unary: `np.sqrt(arr)`, `np.exp(arr)`, ...
- Also logs, trigonometric functions, `ceil`, `floor`, ...
- Binary: `add`, ..., `divide`, `max`, `min`, `mod`, ...
- More in [Universal functions \(ufunc\)](#)

# Mathematical and Statistical Methods

- More or less all to be expected: `sum`, `mean`, `std`, `var`, `min`, `max`, ...
- Most can be called either as methods or as functions:

```
x.mean(); np.mean(x)
```

- Can take an axis as argument, indicating along which axis the operation is to be done

```
x = np.random.rand(10); y = np.random.rand(10)
z = np.array([x,y])
np.shape(z)
z.mean(axis=0)
z.mean(1)
```

- If no axis passed, the function is computed over the **flattened** array
- More in [Mathematical functions](#) and [Statistics](#)

# Histograms

- Histograms:

```
hist, binEdges = np.histogram(a, bins=10, range=None, density=False)
```

- Computes an histogram from `a` with 10 bins and automatic ranges (`a.min()`, `a.max()`)

- If `bins` is a sequence, it defines the bin edges, allowing for non-uniform bins
- If a range tuple is provided, values of `a` outside that range are ignored
- If `density=False` the histogram will contain the number of samples in each bin
- If `density=True` the histogram will contain the normalized number of samples in each bin

- Returns

- An array `hist` with the values of the histogram
- A `float` array `bin_edges` with the `length(hist)+1` bin edges

# Linear Algebra in NumPy

- The submodule `numpy.linalg` contains the most used linear algebra functions
- `dot` : general matrix multiplication
  - Infix version operator: `@`
- `diag` : returns the diagonal of a square matrix as a 1D array (as `diagonal` ), or converts a 1D array into a square matrix with zeros on the off-diagonal
- `trace`, `det`, `inv`; `T`: `transpose`
- `eig` : compute the eigenvalues and eigenvectors of a square matrix
- `solve` : solve the linear system  $Ax = b$  for  $x$ , where  $A$  is a square matrix

## And Much, Much More ...

- The submodule `numpy.random` contains a lot of very useful random tools
- And there is also `numpy.polynomial` , all sorts of math functions, set functionality, ...
- Support for sparse matrices
- Support for **masked** arrays: automatic handling of exceptional values
- One can also define and work with **structured** arrays that store and handle general structured values
- Has 24 built in data-types but more can be defined
- Details in [Numpy manual contents](#)

- Numerical and scientific modules on top of NumPy
  - Integration and Interpolation
  - Linear Algebra and Sparse Eigenvalue Problems
  - Optimization
  - Fourier Transforms and Signal Processing
  - Statistics
  - And more
- SciPy stack: NumPy + Pandas + SciPy + Matplotlib + Simpy + IPython



# matplotlib.pyplot

- 1 Preliminaries
- 2 First Things
- 3 Strings
- 4 Functions
- 5 Structured Types
- 6 Files and Modules
- 7 NumPy
- 8 Matplotlib and Pyplot

# The `matplotlib` Library

- `matplotlib` is a 2D plotting library to generate plots, histograms, power spectra, bar charts, error charts, scatterplots, etc
- Resources available:
  - [Gallery](#): with first simple examples and source code
  - [Matplotlib Examples](#) with more sophisticated examples
  - [Plotting commands summary](#)
- The `pyplot` submodule combines standard plotting with functions to plot histograms, autocorrelation functions, error bars, ...
- Import: `import matplotlib.pyplot as plt`
- Online plot is possible in IPython's qtconsole or notebooks with magic command `%matplotlib inline`

# Basic plotting

- Basic plot: `plt.plot(x, y, str)`
  - `x`, `y` are arrays or sequences
  - If any is two dimensional, columns are plotted individually
- The string `str` controls color and style with many options available
  - `'b-'` : solid blue line (solid line is the default)
  - `'g--'` : dashed green line
  - `'r-.'` : red dash-dot line
  - `'y:.'` : yellow dotted line
- There can be several array-sequence groups:

```
plt.plot(x1, y1, 'g:', x2, y2, 'g-')
```

# Basic pyplot commands

- **Title:** `plt.title(str)`
- **Axis labels:** `plt.xlabel('variable %d' % v)`
- **Axis limits:** `plt.xlim(xmin, xmax)`, `plt.ylim(ymin, ymax)`
- **Legends:** `plt.legend(handles, labels, loc)` assigns the strings in `labels` to the lines in `handles` and draws them in a position according to `loc`
  - `loc` values: 0–best, 1–upper right, ...
  - `handles` and `labels` can be handled implicitly if defined elsewhere:

```
_ = plt.hist(var[indP].values, bins=11, alpha=0.5, label='P')
_ = plt.hist(var[indN].values, bins=11, alpha=0.5, label='N',
             color='r')
plt.legend(loc='best')
```
- `plt.xticks`, `plt.yticks` show x, y axes ticks:

```
plt.xticks(range(len(l_ticks)), l_ticks, rotation=90)
```
- `plt.show()`, `plot.close()` display, close a plot

## Basic pyplot commands II

- Bar plots: `plt.bar(left, height, width=0.8, ...)` makes a bar plot with rectangles with left sides `left`, heights `height` and widths `width`
- Histogram plots: `plt.hist(x, bins, range, ...)` works similarly to `np.histogram` with analogous first arguments
  - Returns arrays `hist, bin_edges` as `np.histogram`
- Saving plots:  
`plt.savefig(fname, dpi=None, orientation='portrait', format=None)`
  - `format` is one of the file extensions supported:  
`pdf, png, ps, eps, ...`
  - Can be inferred from the extension in `fname`

## figure and subplot

- `plt.figure(num=None, figsize=None, dpi=None, ...)` creates a figure referenced as `num` with width and height in inches determined by the tuple in `figsize`
  - Basic use: `plt.figure(figsize=(XX, YY))`
- `subplot` is used to create a subplot within a figure and to refer to that particular subplot
- Typical use: `subplot(nrows, ncols, plot_number)`
  - The figure is notionally split in a grid with `nrows * ncols` subaxes
  - `plot_number` identifies the current plot in that grid **starting from 1**
  - If `nrows, ncols, plot_number` are  $\leq 9$ , a 3-digit version can be used: `subplot(311)`
- `plt.plot` implicitly creates a `subplot(111)`
- More sophisticated subplot location can be obtained using `plt.axes()`

# An Example

```
d = { 'x': np.random.rand(100), 'y': np.random.rand(100) }

plt.figure( figsize = (12, 5) )
plt.subplot(1, 2, 1)
plt.title("Hist %s" % 'x')
plt.xlabel("%s" % 'x')
plt.ylabel("abs. frequencies")
_ = plt.hist(d['x'])

plt.subplot(1, 2, 2)
plt.title("%s vs %s" % ('x', 'y') )
plt.xlabel("Values")
plt.ylabel("abs. frequencies")
_ = plt.hist(d['x'], bins=11, alpha=0.5, label='x')
_ = plt.hist(d['y'], bins=11, alpha=0.5, label='y', color='r')
plt.legend(loc='best')
plt.show()
```