An Introduction to Python's Basics

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

2019

Contents

1 Preliminaries

Python Sources

- These notes 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 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 IDE (of course!)

Python Installation

- Best option: install Continuum's Anaconda
  - Either the full Anaconda suite or 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 inconsistencies
  - Install the latest Python 3.X (Python 2.7 will not be supported after 2019)
- To start the IPython/Jupyter tools, open the Anaconda command prompt and
  - Type `ipython` for the plain text IPython shell
  - Type `juyter qtconsole` for the GUI version of the IPython shell
  - Type `jupyter notebook` for the Jupyter Notebook environments

Jupyter Notebooks

- Browser based interface to develop and document code
- 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 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 `%reload` command

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:
  - 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 2.X: import module `module.py` with `import module as mod` and reload with `reload(mod)`
- Python 3.X: `import imp` and reload with `imp.reload(mod)`
- Much better: 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

2 First Things

Objects

- In Python everything is an object
  - If `o` is an object, typing `o.  + tab` lists its methods
  - Also `dir(o)`
- Two general data types:
  - Scalar (atomic??): `int, float, bool, str`
  - Containers: contain scalars and other containers
- Type checking:
  - Implicit type assignment and runtime checking
  - Explicit type checking with `isinstance(object, type)` or `type(object)is b`
- Special "value" `None` : absence of value
- Type casting possible

Variables and Expressions

- Variables: **names** of objects (no 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
- Often leading and trailing single _ and double __ are 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 more or less as in C, but there are clear cut differences
- There are not assignments but **bindings** between names and objects
  - Variable names **are not** synonyms of memory addresses where the variable values are stored
- Scope of bindings: (usually) the block in which the name appears
- 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)
```

Bindings and Identities

- The `id` function returns the **identity** of an object:
  - An (long) integer which is guaranteed to be unique and constant for this object during its lifetime
  - But not an actual 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
- 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
- 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 thinking over Python loops:

```
#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 is no longer straightforward
- More on iterables, `iterators` and `generators` later on

3 Strings

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` array

String Methods

- String methods: very useful tools for string handling
- `s.lower(), s.upper()` : returns lowercase or uppercase 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(sOld, sNew)` : returns a string with `sOld` replaced by `sNew`
  - `s.replace('', '')` trims all blank space in `s`

String Methods II

- `s.split(delim)` : returns a list of substrings separated by the given delimiter
  - `s.split()` splits `s` over any sequence of white space characters
- 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`

More on Strings

- Multiple line string literals possible ending each line with a backslash `\`
  - We can also put in multiple lines Python expressions inside parenthesis
  - We can also use `\` to span expressions on multiple lines
- **Raw strings**: literals preceded by `r`, as in `r'abc\edf\ghj'` that are not processed: `r'a\nb'` prints as `a\nb`
- Everything 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

- `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 formating 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` with the newline stripped

```
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')
    2
```

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

```
num = eval(input('enter an int: '))
    enter an int:   3+Enter

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

4 Functions

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 called by value and mutable by 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 it is rebound in the method, the outer scope will know nothing about it and no further outside changes are made

Python's Memory Model

- 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 >= 1.0` with precision `eps`:

```
def bisect_sqroot(x, eps):
    '''... docstring ...'''
    if x < 1:
        print("error: input %f < 1." % x); return None
    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 are 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, actually tuples)
- Values are bound to parameters either positionally or through the formal parameter names
- This is exploited using default values

Argument Default Values I

- Argument order may be changed if we use default values

```
def printName(firstN, lastN, reverse):
    #function's body: exercise

#callable as:
printN('Jose', 'Dorronsoro', False)
printN(lastN='Dorronsoro', firstN='Jose', reverse=False)
```

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

```
def printName(firstN, lastN, reverse=False):
    #...

#callable as:
printN('Jose', 'Dorronsoro')
printN('Jose', 'Dorronsoro', True)
```

Argument Default Values II

- 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:

```
printN('Dorronsoro', firstN='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 formst: reStructured text (reST) / Sphinx, which can be used to generate documentation automatically
  - Other frequent options: Google and Numpy formats
- More on Stack Abuse: Python Docstrings

Docstring Use

- Example

```
def bisect_sqroot(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
    """
    left= 1.; right = x; sqr = (left+right)/2
    ...
```

- `help(bisect_sqroot)` in shell displays arguments and docstring
- `bisect_sqroot(` in shell opens window with help
- `pydoc -w my_module` writes a file `my_module.html` with (among others) the docstring info
  - Watch out: it executes the file (and prints all garbage comments inside!!)

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 listFuncValues(n, f):
    l_func_vals =[f(i) for i in range(n)] #list comprehension
    return l_func_vals
```

Functions Inside Functions

- Functions can be defined inside other functions
- The outside function names are seen by the inside function

```
def f_outside(x):
    def f_inside():
        def f_inside2():
            return x*y
        return x+f_inside2()
    y = x*x
    return 2*f_inside()
```

- Allows easy (and messy?) way to pass “local global” variables to subsidiary functions

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([])`

5 Structured Types

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 and very useful set of 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**: 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**: 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 use as 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`
- 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)
```

- The application of `next` exhaust 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 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)))
```

- `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` and `enumerate`

- `zip` joins several lists of the same lenght 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]

for a, b in zip(l_1, l_2):
        print(a*b)

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 %2d es %4d" % (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

Gather and Scatter

- 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 **gathered** into a tuple and the keyword arguments into a dict that the function's body knows how to process
- The complement of gather are the **scatter** operators `*, **`
  - A single scatter `*` splits a list or tuple arguments into multiple arguments
  - A double scatter `**` splits a single dict argument into multiple keyword arguments

```
def do_something(*args, **kwargs):
    for arg in args:
        print(arg)
    for key in kwargs:
        print(key, '=', kwargs[key])

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 = []

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) )

print(l_2)
```

6 Files and Modules

Working with Files

- Files are used through a **file handle**:
  `fName = open('file', 'w')`
- A handle can be opened also with `'r', 'a'`
- 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 an extra 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 myModule as mm
```

- If the file `myModule.py` has been changed after its import, it has to be reloaded to update the previous binds:
  `reload(mm)`
  - `reload` performs syntactical checking
  - Automatic reload with the `autoreload` extension
- Module functions 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 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
  - `os.chdir(path)` changes the interpreter's active directory
  - `os.system(command)` execute the command in the string `command` in a sub-shell
- `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 `.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

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`

Redirecting Data Streams

- `sys.stdout` contains the current stdout stream

```
stdout = sys.stdout
f = open('out.log', 'w')
sys.stdout = f
print("something ...")
sys.stdout = stdout
f.close()
print("something ...")
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]
```

Passing Command Line Arguments

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

```
$ python myMod.py arg1 arg2 arg3
```

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

```
def main(args):
    if len(args) != 3:
        print "incorrect number of arguments ..."

    var1 = int(args[0])
    var2 = float(args[1])
    var3 = str(args[2])

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

- More complete parsing of command line arguments can be done with the `argparse` module

7 NumPy

The NumPy Library

- NumPy (Numerical Python): package for basic scientific computing and data analysis
- 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

- Array creation functions
  ```
  d = np.zeros( tuple )
  d = np.ones( tuple ) #also: np.empty, np.eye
  i_vals = np.arange(10, dtype=int)
  ```
- Or simply append things on a list and convert it: `a_l = 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
- 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, aMin, aMax)`
- 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 condition complying elements can also be selected:

```
x = np.random.normal(0., 1., 100)
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: `arrC = arrA + arrB`
  - 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 a default of 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, or converts a 1D array into a square matrix with zeros on the off-diagonal
- `trace, det, inv; T:` traspose
- `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 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

Scipy

- 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

8 Pandas

Pandas

- At first sight: NumPy + key indices + specialized tools
- Actually: register (index) – field (column) two–dimensional tables
- Reading and writing under different formats: CSV, text, Excel, SQL tables, HDF5 files
- Index based data alignment and handling of missing data
- Slicing, indexing and sub–seting of data sets
- Merging and joining of data
- (Some) Time series-functionality
- Basic data containers: series and dataframes

Pandas Series I

- **Series**: one-dimensional array of indexed data
- It can be created from a Numpy array: `s = pd.Series(np.arange(10))`
  - Other parameters: `dtype, name`
- Many attributes, many methods, often derived from NumPy:
  - Attributes: `shape, size, values, hasnans, ...`
  - Methods: `abs, max, min, argmax, argmin, sort_index, sort_values, ...`
  - Time series methods: `autocorr, corr, kurtosis, shift, ...`
  - `.values` returns the underlying array, `.index` return the index set
- Access to elements:
  - `iloc`: integer based `s.iloc[1]`
  - `loc`: purely **index** (or label) based `s.loc['ind01']`

Pandas Series II

- The series has a sequence of values (Numpy 1–dimensional arrays) and a sequence of **indices**
  - Indices are an object of type `pd.Index`
  - Indices can have immutable values of (essentially) any type

```
inds0010 =["ind%02d" % i for i in range(10)]
s = pd.Series(np.arange(10), index=inds0010)
```

  - Indices are accessed through the `index` attribute

- Indices can be seen as an ordered set with immutable elements and support standard set operations

```
indA & indB  # intersection
indA | indB  # union
indA - indB  # difference
```

DataFrames I

- **DataFrames**: multi-dimensional array of indexed data
- Can be seen as a set of series with a common index
- Besides the `index` attribute, DataFrames have another attribute `columns`, an index holding the column labels
- DataFrames can be one dimensional:

```
dfs = pd.DataFrame(s, columns=['enteros'])
```

- They can be built from Numpy arrays:

```
idM = np.eye(10, 10)
df = pd.DataFrame(idM, index=inds0010)
```

DataFrames II

- They can also be read from a csv file with a first header row:

```
df = pd.read_csv(fName, sep=delim)
```

  returns a DataFrame with the headers as column titles

- Column names and indices can be defined while reading a file:

```
df = pd.read_csv(fName, sep=delim, index_col=0, names=l_col_names)
```

- They can also be read from a csv file with column names and indices:

```
df = pd.read_csv(fName, sep=delim, index_col=0, names=l_col_names)
```

- Each column in `df[colName]` contains a Series

Accesing and Indexing

- Accessing is also done by indices using `loc` or by integers using `iloc`

- Subset selection is done working on the `index` attribute, returning a boolean array

```
indices_altos = df.index > 'ind04'
df[indices_altos]
```

- `np.nonzero` returns the corresponding integer location values

```
int_indices_altos = np.nonzero(indices_altos)
df.iloc[int_indices_altos]
df.loc[ df.index[int_indices_altos] ]
```

- Very easy column selection and reordering through sublists of `df.columns`
- To add an index to an existing dataframe `df = pd.DataFrame(df.values, index=index, columns=df.columns)`

`concat`

- A way of thinking about dataframes is seeing them as a table with fields the column names and records the indexed rows
- `concat`: basic tool for concatenating pandas objects along a particular axis with optional set logic
- Basic use: to merge and align two series/dataframes according to their indices and the merge options used

```
df = pd.concat( objs, axis=0, join='outer', ...)
```

  - `objs`: sequence of series or dataframes
  - `axis`: along which concat takes place
  - `join`: `inner` for intersection, `outer` for union

- In this logic `concat` can be seen as a tool to perform joins on dataframes
- In particular, it makes very easy the merging and intersection of time–indexed data

9 Dates, Times and Datetimes

The `datetime` Module

- Contains classes for manipulating dates and times
- There are two kinds of date and time objects: *naive* and *aware*.
  - Naive objects basically contain plain date (the basic naive object) and time info

  - Aware objects get knowledge of possible time adjustments (such as time zones or summer/winter hours)

- Relevant types are `datetime.date, datetime.time, datetime.datetime`
- `class datetime.date:` a Gregorian calendar plain date, with attributes `year, month, day`
  - Basic constructor: `datetime.date(year, month, day)`

The `datetime` Module II

- `class datetime.time:` a plain time, with attributes `hour, minute, microsecond, second, tzinfo`
  - `tzinfo:` contains time zone information objects used by `datetime` and `time` to provide aware objects
- `class datetime.datetime:` a combination of a date and a time
  - Basic constructor: `datetime.datetime(year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None)`
  - Also `datetime.now(tz=None), datetime.utcnow()`
  - From formatted strings: `datetime.strptime(date_string, format)`
- `class datetime.timedelta:` expresses the difference between two date, time, or datetime instances

```
d = timedelta(days=3, hours=2, minutes=1)
```

`date` Attributes and Methods

- Instance attributes of an object `date` are `year, month, day`
- Some instance methods for changing `date` instances or getting information from them are
  - `date.replace(year, month, day)`
  - `date.isoweekday()` returns a number between Monday (1) and Sunday (7)
  - `date.isocalendar()` returns the ISO year, ISO week number and ISO weekday
- Some methods for formatting `date` instance information are
  - `date.isoformat()` returns a ISO format string `'YYYY-MM-DD'`
  - `date.strftime(format)` returns a string following an explicit format

```
from datetime import date
dir(date)
td = date.today()
td.isocalendar(); td.isoformat(); td.strftime('%d/%m/%Y')
```

`datetime` Attributes and Methods

- Instance attributes of an object `datetime` are those of `date` plus `hour, minute, second, microsecond, tzinfo`
- Several `time` methods extend to `datetime` instances:
  - Constructors `datetime.date(...), datetime.time(...)`
  - `datetime.replace(year, month, day, hour, minute, second, microsecond, tzinfo )`
  - `datetime.isoweekday(), datetime.isocalendar(), datetime.isoformat(), datetime .strftime(format)`

```
from datetime import datetime
dt = datetime.strptime("21/11/06 16:30", "%d/%m/%y %H:%M")
dt.strftime("%A, %d. %B %Y %I:%M%p")
'The {1} is {0:%d}, the {2} is {0:%B}, the {3} is {0:%I:%M%p}.'.format
     (dt, "day", "month", "time")
```

Other Things

- Datetimes can be incremented using `timedelta`

  `dt.datetime.utcnow()+ dt.timedelta(days=3, hours=2, minutes=1)`
- Datetimes can be substracted, returning a `timedelta`

```
today = dt.datetime.utcnow()
tomorrow = today + dt.timedelta(days=1)
diff = tomorrow-today
diff.days
diff.seconds
```

10 Timing Python Programs

The `time` Module

- There are two kinds of execution time
  - **CPU or execution** time: how much CPU time is spent on executing a program
  - **Wall–clock (or elapsed or running)** time: total computer time to execute a program

- Wall–clock time is usually longer because other programs' execution influences it
- `time.time()` returns the time in seconds since the epoch (i.e., the point where the time starts)
  - `time.gmtime(0)` returns the epoch

The `time` Module II

- `time.clock()` returns in Unix the current processor time expressed in seconds
- In Windows returns "wall-clock time in seconds elapsed since the first call to this function ..."
- According to `15.3. time | Time access and conversions`

  "`clock()` is the function to use for benchmarking Python or timing algorithms"
- To time small bits of Python code we can use `timeit`

The `timeit` Module

- Function interface:

```
import timeit
timeit.timeit("'-'.join([str(n) for n in range(100)]) ", number=100)
```

  - Returns the time used for 100 repetitions of the statement
  - `timeit.repeat` adds a `repeat=N` argument and returns a list of execution times

```
l_t = timeit.repeat("'-'.join([str(n) for n in range(100)]) ", number
      =100, repeat=10)
min(l_t)
```

- It can also be called from the command line

```
python -m timeit "'-'.join([str(n) for n in range(100)])"
```

- More info in `26.6. timeit | Measure execution time of small code snippets`
- To time functions `timeit` uses a `setup` argument that adds code to be used for timing

`timeit` over Functions

- Typical use: use `setup` with an `import` statement to give `timeit` access to already defined functions

- Example: returns a list with the timings of the 10 executions

```
def f(x):
    return x*x

l_time = timeit.repeat(stmt="for x in range(10): f(x)",  \
    setup="from __main__ import f", repeat=10, number=10)
min(l_time)
```

`timeit` in IPython

- In IPython we can use the `%timeit` magic function

```
def fib2list(n):
    l_fib = [1, 1]
    p, q = 1, 1; i = 2
    while i <= n:
        p, q = q, p+q
        l_fib.append(q)
        i += 1
    return l_fib

%timeit for x in range(100): fib2list(x)
fib_times = %timeit -o -r 10 -n 10 for x in range(100): fib2list(x)
```

- `%timeit -o <statement>` returns a `TimeitResult` object with information about the `%timeit` run
- Interesting atributes are `best, worst, all_runs, loop, repeat`

Profiling

- Timers aim to measure precisely the time cost of code parts
- Profilers aim to measure how time is spent among several code parts
  - They balance accuracy and information on long programs
  - But are not for benchmarking
- Several profilers available in Python's standard library; cProfile is the most widely used
- Standard use through the method `run`

```
import cProfile, pstats

cProfile.run("retValue = myFunction(args)", "profile.log")
```

- On the command line it is used as

```
python -m cProfile [-o "profile.log"] [-s sort_order] myScript.py
```

  - `-s` specifies one of the `sort_stats()` sort values to sort the output by it

Profiler Information

- The profile information is usually written in a binary file, from which it can read, processed and printed
- Profile info is given for functions and organized in the columns

```
%{\ttfn \scriptsize
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
```

  - `tottime` is the total time spent in the function **without** considering time in the called functions
  - `cumtime` is the total time spent in the function including time in the called functions
  - `percall` is the quotient of `tottime`, `cumtime` by `ncalls`
  - `filename:lineno(function)` gives the module and line where the function is called

Analyzing Profile Information

- The `Stats` class is used to analyze profiler data
- A `Stats` object can be built as

```
pstats.Stats(profile_info, stream=sys.stdout)
```

  - `profile info` is either a list `*filenames` of profile output files or a `profile` object
  - Output will be printed to `stream`

- Important methods
  - `strip_dirs()` removes path information
  - `print_stats(*restrictions)` where `restrictions` usually take the form `num , str` : a number of lines and a substring that selects function names
  - `sort_stats(*keys)` that sorts the `Stats` object according to the list of key names

An Example I

- The use of the profiler and `Stats` is fairly uniform

```
import cProfile, pstats
import smocd05 as cd
cProfile.run("a, g, k, d, c, n = cd.smo_all(X, y, C=10, \
              kernel='rbf', gammaRBF=None, kMax=100, epsKKT=0.001, \
              update_CD=False, update_LU=1, logName=None, \
              fl_compare=False)",
              "profile.log")

s = pstats.Stats("profile.log") #get Stats info
s.strip_dirs(); s.sort_stats('cumtime')
s.print_stats('smocd05', 10) #print at most 10 lines of the module
```

An Example II

- The output is

```
      70850 function calls (70750 primitive calls) in 0.413 seconds

Ordered by: cumulative time
List reduced from 120 to 8 due to restriction <'smocd05'>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.011    0.011    0.413    0.413 smocd05.py:242(smo_all)
   100    0.000    0.000    0.368    0.004 smocd05.py:32(kernel_update)
   100    0.006    0.000    0.367    0.004 smocd05.py:47(gaussian_kernel_update)
   100    0.011    0.000    0.016    0.000 smocd05.py:170(delta_KKT)
   100    0.010    0.000    0.014    0.000 smocd05.py:130(select_LU)
   100    0.002    0.000    0.002    0.000 smocd05.py:187(clip_step)
   100    0.002    0.000    0.002    0.000 smocd05.py:222(step)
     2    0.000    0.000    0.000    0.000 smocd05.py:210(svm_dual_cost)
```

Summing Things Up I

- In general, interpreted languages with dynamical types are slower than languages with static types

- Thus, Python is slow, but:
  - This is bad for programs that execute entirely on the CPU, such as heavy numerical computations (but `numpy` is extremely efficient)
  - This is no that crucial for disk or web applications
  - And Python programs are shorter and way easier to write than those of many other languages

- An interesting (and not too hard) discussion in Why Python is Slow: Looking Under the Hood

Summing Things Up II

- In any case, one can
  - Use a profiler to identify a program's bottlenecks
  - Measure accurately the time these bottlenecks require
  - Find ways/tools to make code segments faster if needed

- A such example is Cython ("Python with C data types")
  - It compiles very much Python–like programs files into C code
  - Useful links: Language Basics, Cython for NumPy users

11 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)` puts the value of the `int 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 hadled 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.show(), plot.close()` displays and closes 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 ≤ 10, 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()
```

12 Classes

Classes

- Essential motivation: abstract data types as tools to focus globally on data objects and not only locally on functions
- Advantages
  - Better program design
  - Reduced development time
  - Easier reusability

  provided we spend the necessary time and ingenuity setting abstract data types right
- Classes are the standard tool for this

Defining Classes

- Definition

```
class name_class([object]):
    statements
```

- Naming conventions in PEP 0008 – Style Guide for Python Code
- `object` makes the class a subclass of the general Python `object` (default in Python 3.X)
- The class will inherit all the properties of objects
  - Thus we can bind variables to a class, put it on a list or dict, ...
  - It may not be strictly needed but most often won't hurt either
- Statements are usually class variables and method definitions

An Example: Nodes and Linked Lists

- We define first the linked list (LL) nodes that will have `info, next` "fields"

```
class Node(object):
    def __init__(self, info=None, next=None):
        self.info = info
        self.next  = next

    def __lt__(self, other):
        return str(self.info) < str(other.info)

    def __str__(self):
        return str(self.info)
```

- The method `__init__` applies when the class is **instantiated**, i.e., when a new object of the class is built:

```
node = Node(1)
```

Nodes for Linked Lists II

- Although `__init__` has two parameters, `self` refers to the instance itself
  - Thus we pass only one argument to `__init__`
- `self` is often the name of the first parameter of a method
- It is taken to refer to object itself
- But we could use any other name

Nodes for Linked Lists III

- The method `__str__` is applied when we apply the `print` function to an object or when we apply the `str` function
  - It should thus return a string
  - This is straightforward for simple objects
  - For more complicated ones it will require composing a string out of the object's information
- The method `__lt__` overloads Python's `<` operator
  - It will depend on the node's content, which could be anything in principle
  - We opt for the fairly general string comparison
- We can similarly overload other operators: `le, eq, ne, ...`
- There are many other special method names `__xx__` to customize standard operations

  - More on The Python Language Reference: Data model

The Linked List Class

- We now define the `MyList` class

```
class MyList(object):
    def __init__(self):
        self.length = 0
        self.head   = None

    def __str__(self):
        s = ""
        if self.head != None:
            node = self.head
            while node:
                s += str(node.info) + ' -> '
                node = node.next
        return s[ : -4] #remove the last " -> "

    def add_ini(self, info):
        node = Node(info)
        node.next = self.head
        self.head = node
        self.length += 1

    def remove_ini(self):
        if self.head != None:
            node_temp = self.head
            self.head = node_temp.next
            del(node_temp)
            self.length -= 1
```

The Linked List Class II

- The code is more or less self–explaining
- In `__str__` we compose and return a string with the list information
- We do not overload the `<` operator
  - What will happen then when we write `ll_1 < ll_2`?
- There is no way to reach the last node without traversing the entire list
  - Exercise 1: write a method that returns the last node traversing the list
  - Exercise 2: change the definition adding an attribute `tail` that "points" to the last node

More on Classes

- Classes can be **inherited** from previously defined ones

```
class DerivedClassName(BaseClassName):
    #class BaseClassName is defined in the same module
    <statements>

class DerivedClassName(modname.BaseClassName):
    #class BaseClassName is defined in the module modname
    <statements>
```

- **Class variables** are attributes and methods shared by all instances of a class
  - Examples: `add_ini, remove_ini` for the `MyList` class
- **Instance variables** are data unique to each instance
  - Examples: `self.head` for the `MyList` class, `self.next` for the `Node` class
- Instance (or more generally) data attributes override method attributes with the same name
- Instance variables override class variables with the same name

Public or Private?

- Neither: private variables don't exist in Python but usually some conventions are followed
- A name preceded by _ should be considered as non-public and, hence, not used to accede to a class attribute
  - Good discussion on underscores in The Meaning of Underscores in Python
- **name mangling** is used to avoid coincident name conflicts:
  - It could be an implementation–specific element that could change without notice
  - An identifier such as `__xxx` with two leading underscores is replaced with `_classname__xxx` where `classname` is the current class name after stripping leading underscores
  - Can be called as `classname._classname__xxx` but it takes willingness

Getters and Setters

- Python objects can have attributes and methods without being actually classes
  - For instance, `lists`: `append, pop, ...`
- The built–in functions `getattr, setattr` can also be used to handle them
- `getattr(object, "attribute")` is equivalent to `object.attribute`
  - It returns the value of the named attribute of `object`
  - Can be used as `getattr(object, name, default)` that returns a default value if the named attribute does not exist
  - When the named attribute does not exist, it raises an `AttributeError`
- `setattr(object, "name", value)` is equivalent to `object.name = value`

13 Exception Handling

Managing Exceptions

- Functions should be organized in `try/except` blocks
  - Generally used for error handling
  - But also for control flow: example from Guttag, p. 86:

```
def read_val(val_type, in_msg, err_msg):
    """val_type: type to cast the string return of raw_input"""
    while True:      #inf loop to be broken by return
        val = raw_input(in_msg+' ') #raw_input returns a string
        try:
            val = val_type(val)   #error unless val is an int
            return val           #arrive here if no exception
        except ValueError:
            print(val, err_msg)
```

  - Use as `read_val(int, 'input an int', 'not an int')`
- The statements in the `except` block specify how to handle **exceptions**: anomalous circumstances found during code execution

Managing Exceptions II

- `except` can have associated a tuple with possible exceptions
  - If we use `except (ValueError, TypeError):` we can handle both types of error
  - If we only use `except:` the exception block will be entered no matter what error has appeared and we will get a long error message possibly with some backtracking
- We can use `else:` for code that will be executed only if the try: block succeeds
- The code after `finally:` executes always, even if an exception happened
- Exceptions are also handled if they occur in functions called in the try clause

Defining Exceptions

- Python has a number of predefined exceptions, actually defined as classes derived from the base class `exception BaseException`
  - They have associated an information string in `Exception.message`
  - We can get the concrete exception name with `type(Exception)`
- We can define our own exceptions inherited from the base class `Exception`

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
```

- Good (defensive) programming practice requires pre–detection of possible exception appearances and their appropriate handling

Standard Exceptions

- `ZeroDivisionError, OverflowError`
- `ValueError`: a built-in operation or function receives a right type but inappropriate value argument
- `TypeError`: an operation or function is applied to a wrong type object
- `OSError`: raised when a function returns a system-related error
- `NameError`: a local or global name is not found
- `IndexError`: subscript is out of range
- `KeyError`: a dictionary key is not found in the set of existing keys
- `EOFError`: `input()` or `raw_input()` reaches EOF without reading any data
- `IOError`: I/O-related failure, such as “file not found” or “disk full”
- `RuntimeError`: error that doesn’t fall in any of the other categories

General Exception Handling

- The general exception `Exception` catches all built-in, non-system-exiting exceptions
- Exceptions not so catched are `KeyboardInterrupt` and `SystemExit`
  - Catching them could make it very difficult to exit a script
- `Exception` can be used for the (almost) lowest level exception handling as in

```
try:
   <statements>
except Exception as e:
   #sys.exit('don't know what's going on!')
   print(e.message)
   print(type(e))
   print(sys.exc_info())
```

- `sys_exc.info` returns a tuple with type, value and track info about the most recent exception caught
  - track info contains the call stack at the point where the exception

A Crude Example

- We can have several `except:` statements

```
import sys
import traceback
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print("I/O error({0}): {1}".format(e.errno, e.strerror))
except ValueError:
    print("Could not convert data to an integer.")
except: #wildcard exception:we don't know what's going on!!!
    text = traceback.format_exc()
    sys.exit(text)
```

- `traceback.format_exc()` returns a string with info on the concrete exception
- `sys.exit` raises the `SystemExit` exception that causes the Python interpreter to exit

The `assert` Statement

- The syntax for assert is:

```
assert Expression[, ArgumentExpression]
```

- When Python encounters an `assert`
  - It evaluates the accompanying expression, hopefully true
  - If it is false, an `AssertionError` exception is raised (that we have to decide how to handle) and `ArgumentExpression` is printed
- An example:

```
def kelvin_2_celsius(temp_kelvin):
    assert (temp_kelvin >= 0), "colder than absolute zero"
    return (temp_kelvin - 273)

print(kelvin_2_celsius(273))
print(kelvin_2_celsius(-1))
```

The `with` Statement

- Encapsulates common `try...except...finally` constructions for convenient reuse

```
with open('workfile', 'r') as f:
    read_data = f.read()
    do something with data
f.closed
```

- `with` wraps the execution of its suite block with methods defined by a **context manager**
  - It defines the runtime context to be established when executing `with`
  - It handles the entry into, and the exit from the code block
- When used with more than one item, the suite statements are processed as if multiple `with` statements were nested

```
with A() as a, B() as b:
    suite
#equivalent to
with A() as a:
    with B() as b:
        suite
```

14 The Sklearn Library

The `scikit- learn` Library

- `scikit--learn` or just Sklearn is becoming the standard basic library for Machine Learning in Python
- From their web page scikit-learn Machine Learning in Python:
  - Simple and efficient tools for data mining and data analysis
  - Accessible to everybody, and reusable in various contexts
  - Built on NumPy, SciPy, and matplotlib
  - Open source, commercially usable - BSD license
- Contains most of the main algorithms for
  - Supervised learning in classification, regression
  - Model selection: grid search, cross validation
  - Clustering
  - Preprocessing, feature selection, dimensionality reduction

Model Building in Sklearn

- Model building follows the define–fit–predict cycle
  - Define: select a model

    ```
    from sklearn import linear_model
    lr_m = linear_model.LinearRegression()
    ```

  - Fit: build a numpy training data matrix `x_tr` with shape `(NTr, d)` and an `NTr`–dimensional training target vector `y_tr` and train the model

```
lr_m.fit(x_tr, y_tr)
```

  - Predict: build a numpy test data matrix `x_ts` with shape `(NTs, d)` and apply the model to get a prediction numpy vector `y_ts_pred`

```
y_ts_pred = lr_m.predict(x_ts)
```

- If we have a target test vector `y_ts` we can use several metrics to compare `y_ts` with `y_ts_pred`

```
from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_ts, y_ts_pred)
```

Fitting Curves to Values

- Assume we have two arrays

```
x = np.array([i for i in range(100, 201)])
y = 1.0 + .5 * x**2 + np.random.normal(0., 200., len(x))
```

- To fit a linear model $ax^2 + b$ to the `y` elements:

```
from sklearn.linear_model import LinearRegression

lr_m =  LinearRegression()

x = x.reshape(-1, 1)
lr_m.fit(x**2, y)
```

- And then we plot the model predictions against the `y` values:

```
y_pred = lr_m.predict(x**2)

_ = plt.plot(x, y, '*r', x, y_pred, 'b')
```

An Example: Boston Housing

- Patterns: several real estate–related variables of Boston areas
- Target: median house values in the area
- We load the data into the IPython shell as a **dataframe** using the `pandas` module

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

f_h = 'housing.csv'
df_h = pd.read_csv(f_h, sep=',')
l_vars_targ = df_h.columns
l_vars_targ
```

Scatterplots

- We first do some basic plotting

```
targ =  l_vars_targ[-1]
var = 'LSTAT'              #'CHAS','ZN', 'B'

plt.figure( figsize = (18, 5) )
plt.subplot(1, 3, 1); plt.title('Boxplot %s' % var)
_ = plt.boxplot(df_h[var])
plt.subplot(1, 3, 2); plt.title('Hist %s' % var)
_ = plt.hist(df_h[var])
plt.subplot(1, 3, 3); plt.title('%s vs %s' % (var, targ) )
_ = plt.plot(df_h[var], df_h[targ], '.')
```

Feature and Target Correlation

- Next we analyze feature–target correlations

```
c = (df_h[l_vars_targ]).corr()
c_vals = c.values[ -1, : -1]

#get inverse index sort of feat-target corrs
c_decr_ind =  abs(c_vals).argsort()[ : : -1]

#print names and corr values
print("Features and feature-target correlations:\n%10s\t%10s" %
      ('feature', 'corr'))
for n, v in zip(l_vars_targ[c_decr_ind], list(c_vals[c_decr_ind])):
    print("%10s\t%10.3f" % (n, v))
```

- Fancier graphics can also be obtained with modules as `seaborn`

Visualizing the Correlation Matrix

- An image is worth 1,000 words

```
#the correlation matrix as an image
plt.figure( figsize=(9, 9) )

n_ticks = len(l_vars_targ)
plt.xticks(range(n_ticks), l_vars_targ,   rotation='vertical')
plt.yticks(range(n_ticks), l_vars_targ)

_ = plt.colorbar( plt.imshow(c, interpolation='nearest',
                             vmin=-1., vmax=1.,
                             cmap=plt.get_cmap('bwr')) )
```

Model Computation

- Usually we must scale first the data matrix:

```
import sklearn.preprocessing as sk_pp

scaler_x = sk_pp.StandardScaler()
x_sc = scaler_x.fit_transform( df_h[ l_vars_targ[ : -1] ] )
y = df_h[targ] #intercept will be non zero
```

- We then compute the linear regression model using Sklearn's cycle define–fit–predict

```
from sklearn.linear_model import LinearRegression

lin_m =  LinearRegression()
lin_m.fit(x_sc, y)

print("coefficients:\n", lin_m.coef_)
print("intercept:\n", lin_m.intercept_)

y_pred = lin_m.predict(x_sc).clip(0, np.inf) #negative values impossible

_ = plt.plot(y, y_pred, '.', y, y) #so that we get a diagonal line
```

Coefficient Relevance

- We first print the intercept and plot the coefficients

```
print("linear model intercept: %f" % lin_m.intercept_)

plt.title('Linear Regression coefs')
plt.xlabel('feature'); plt.ylabel('coef')
plt.xticks(range(n_ticks), l_vars_targ[ : -1], rotation='vertical')

_ = plt.plot(lin_m.coef_)
```

- We sort the coefficients by absolute value

```
coef_decr_ind =  abs( lin_m.coef_ ).argsort()[ : : -1]

print("Features and linear model coefficients:\n%10s\t%10s" %
      ('feature', 'coeff'))
for n, v in zip(l_vars_targ[ : -1][coef_decr_ind],
                list(lin_m.coef_[coef_decr_ind]) ):
    print("%10s\t%10.3f" % (n,v))
```

Residuals and Plots

- First we plot the residuals

```
res = y-y_pred

plt.title('Real Values vs Residuals')
plt.xlabel('target'); plt.ylabel('residual')
_ = plt.plot(y, res, '.', y, y-y) #so that we get a y=0 line
```

- And then the residuals' histogram

```
plt.title("Residuals' histogram")
plt.xlabel('residual'); plt.ylabel('frequencies')
_ = plt.hist(res, bins=31)
```

And Now to Notebooks

- The preceding computations can be placed in a Notebook for reuse, remembering and documentation
- We recall some Notebook basics
  - They have cells for code, documentation, figures
  - Notebooks can be saved as such, as Python files or as plain html files
  - They can also be converted to LaTeX using `nbconvert` (and then, say, to pdf)
- Important: they are most useful as information and communication tools
  - They should mostly contain info, comments/conclusions, pictures, tables
  - Their visible code should be small: get things as functions into a module, import it and call the functions as needed

Jupyter Notebooks

- Before IPython Notebooks
- Core languages supported: Julia, Python, R
- Browser based interface to develop and document code
- Reasonable tool for beginner's Python programming
- Excellent tool for program– or work–flow documentation
- Cells for code, documentation, figures
- Code cells:
  - Edit sentences or functions
  - Execute them with `Ctrl+Intro`
  - Debug, re–edit and re–execute until OK

Jupyter Notebooks II

- Text cells:
  - Marked as Markdown cells with `Esc+m`
  - Can format text with Markdown syntax
  - Also admit formulas with LaTeX notation
- Also header–only cells

- Can display figures from the Matplotlib module executing `%matplotlib inline`
- The notebook server can be started with the command `jupyter notebook` from an Anaconda shell
  - A file browser opens on the command's directory
  - We can open an existing notebook or start a new one
- Notebooks can be saved as such, also as plain html files or converted to LaTeX using `nbconvert` (and then, say, to pdf)
- More on Jupyter Notebooks in The Jupyter notebook
- Final exercise: move the above housing code to a Notebook