The Very Basics of Python

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contents

1 Preliminaries

Sources

- These Basic Notes: quick introduction to the very basics
- Notes in https://github.com/joseDorronsoro/Notes-on-Python: large extension of the Basic Notes
- 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

Sources II

- 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
- For more experienced programmers: *Python Cookbook* (O'Reilly 2005)
- And all the documents in python.org as well as many web references (some below)
- As well as searches in stackoverflow
- As well as ...

Working with Python

- Basic initial mode: a loop of edit(+copy)+execute+refine
- Basic tools: shells, text editor, notebooks
- Basic Python shell: IPython
- Simple shell + editor:
  - QT Console of IPython + notepad++: edit `xxx.py` and `run xxx.py` (or simply copy+paste+Enter)
- IPython Notebook: workflow documentation + small programming if needed
- Installations:
  - Get a basic Python (native in Linux) and add other packages
  - Much better: install a Python distribution from Anaconda

Windows Installation and Running

- Strongly recommended: install Continuum's Anaconda
  - Have to choose between 2.X (to be unsupported after 20220) y 3.X (recommended): can create **environments** and switch among them
- Open the Anaconda command prompt and
  - Type `ipython` for the plain IPython shell
  - Type `ipython qtconsole` for the IPython GUI shell (recommended here)
  - Type `ipython notebook` for the IPython Notebook environments
  - Bonus: Spyder IDE
- Install a new package `xxx` through `conda install xxx` (or `pip install xxx`)
- Similar set up in Linux

IPython 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
- Code cells:
  - Edit sentences or functions
  - Execute them with `Ctrl+Intro`
  - Debug, re–edit and re–execute until OK

IPython 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 QT Console too)
- 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

2 First Things

Objects

- In Python everything is an object
  - If `o` is an object, typing `o.` + `tab` lists its methods
  - Also `dir(o)`
- Types: scalar (atomic??), non scalar (structured??)
  - Implicit type assignment, partial type checking at runtime

  - Explicit type checking with `isinstance(a, type)` , where `type = int, float, ...`

- Scalar object types: `int` (plus `long` in Python 2.X), `float`, `bool`
  - In Python 3.X, the `long` type has been dropped completely with `ints` of arbitrary lenght (details in `longobject.c`)
- `None` : absence of value
- Type casting possible

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

More on Strings

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

More on Strings II

- Multiple line string literals possible ending each line with a backslash `\`
- We can also put expressions inside parenthesis or use `\` to span multiple lines on expressions
- **Raw strings**: literals preceded by `r`, as in `r'abc\edf\ghj'` that are not processed: `r'a\nb'` prints as `a\nb`
- Unicode strings preceded by `u` in Python 2.X; everything Unicode in Python 3.X
  - From What's New In Python 3.0: *Everything you thought you knew about binary data and Unicode has changed.*

More on Strings III

- 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`
- 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`

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
- 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, ...`
- Uses leading and trailing single `_` and double `__` for special meanings
  - Good discussion in The Meaning of Underscores in Python

Variables and Bindings

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

Printing Strings (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

Flow Control

- Indentation extremely important: identifies code blocks
  - 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`

- Watch out for C thinking over Python loops:

```
for i in range(10):
    print i

#don't do it on Python 2.X! watch out for memory
for i in range(1000000):
    print i
```

- `xrange(N)` defers the creation of the list element until it is needed
  - Only in Python 2.X; in Python 3 `range` is in fact `xrange`
- The `while` and `for` equivalence is no longer straightforward
- More on `iterators` and `generators` later on

3 Variables and Functions

Variables and Scope

- 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

In More Detail ...

- 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

Functions

- Definition

```
def name(parameters):
    """ doc strings """
    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

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

An Exercise

- 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 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 `cubeRoot(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

- Contain function documentation ideally in the form of function **assumptions** and **guarantees**, and other info we think important/helpful: type contract, description, ...
- **Assumptions**: conditions to be met by the caller of the function
- **Guarantees**: conditions to be met by the function when called according to its assumptions
- Introduced between two sets of `"""` right after the definition and before the body
- In other parts of the function's body `"""..."""` contains multi line comments (or comments out code)
- But many approaches possible (for instance, Google's docstring style)

Docstring Use

- Example (from Guttag, p. 42; too formal?)

```
def sqroot(x, eps):
    """
    Assumes x int or float >=1, eps a positive float.
    Returns a float root that approximates the square root of x
    up to a precision eps
    """
    left= 1.; right = x; sqr = (left+right)/2
    ...
```

- `help(sqroot)` in shell displays arguments and docstring
- `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
```

4 Structured Types

Structured Types

- Python has five structured types: strings (plus unicode in Python 2.X), 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 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)

Lists II

- 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** enables to apply an operation to all the values of a list

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

  - Also works for dicts and sets

`zip` and `enumerate`

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

```
L1 = range(10) #Python 2: L1 = list(range(10) in Python 3
L2 = [i*i for i in L1]

for l1, l2 in zip(L1, L2):
        print (l1*l2)

for l in zip(L1, L2):
        print (l[0]*l[1])
```

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

```
L2 = [i*i for i in L1]

for i, sq in enumerate(L2):
    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
- Elements defined/accessed through keys: `d['a'] = 'alpha'`
- 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 doSomething(*args, **kwargs):
      # whatever ...
  ```

  Python assumes that `doSomething` 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

  ```
  doSomething(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

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)
```

5 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
```

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 `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`

  ```
  $ 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

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`

6 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

7 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 $\leq$ 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()
```

8 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

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))
```