DAMG 6210
Database Design

Nik Bear Brown
@NikBearBrown

Python for MySQL

Topics

Introduction to Python

Python

- Scripting Language
- Object-Oriented
- Portable
- Powerful
- Easy to learn and use
- Widely used for API’s
- Lots of scientific libraries

Python

- Python is a high-level programming language
- Open source and community driven
- “Batteries Included”
  - a standard distribution includes many modules
- Dynamically typed
- Source can be compiled or run just-in-time
- Scripting language
- Many mathematical and scientific libraries

Hello World

- Hello World

```
print "hello world"
```

- Prints hello world to standard out

Built-in Object Types

- Numbers - 3.1415, 1234, 999L, 3+4j
- Strings - 'spam', "guido's"
- Lists - [1, [2, 'three'], 4]
- Dictionaries - {'food':'spam', 'taste':'yum'}
- Tuples - (1, 'spam', 4, 'U')
- Files - text = open ('eggs', 'r'). read()

Variables

- No need to declare data type
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
if friendly: greeting = "hello world"
else: greeting = 12**2
print greeting
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

Numbers

- The usual suspects
  - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
  - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
    - In Python 2.2 and beyond, 2**100 does the same thing
  - 1j**2 -> (-1+0j)

Indentation and Blocks

- Python uses whitespace and indents to denote blocks of code
- Lines of code that begin a block end in a colon:
- Lines within the code block are indented at the same level
- To end a code block, remove the indentation
- You'll want blocks of code that run only when certain conditions are met

Grouping Indentation

In Python:

```
for i in range(20):
    if i%3 == 0:
        print i
        if i%5 == 0:
            print "Bingo!"
    print "---"
```

In C:

```
for (i = 0; i < 20; i++)
{
    if (i%3 == 0) {
        printf("%d\n", i);
        if (i%5 == 0) {
            printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

Conditional Branching

- if and else

```
if variable == condition:
        #do something based on v == c
else:
        #do something based on v != c
```

- elif allows for additional branching

```
if *condition*:
elif *another condition*:
…
else: #none of the above
```

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Looping with for

- For allows you to loop over a block of code a set number of times
- For is great for manipulating lists:

```
a = ['cat', 'window', 'defenestrate']
for x in a:
        print x, len(x)
```

Results:

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

Looping with while

```
while <boolean>:
    statement 1
    statement 2
    statement 3
```

break, continue

```
>>> for value in [3, 1, 4, 1, 5, 9, 2]:
...     print "Checking", value
...     if value > 8:
...         print "Exiting for loop"
...         break
...     elif value < 3:
...         print "Ignoring"
...         continue
...     print "The square is", value**2
...
```

```
Checking 3
The square is 9
Checking 1
Ignoring
Checking 4
The square is 16
Checking 1
Ignoring
Checking 5
The square is 25
Checking 9
Exiting for loop
>>>
```

Use “break” to stop the for loop

Use “continue” to stop processing the current item

Range()

- “range” creates a list of numbers in a specified range
- range([start,] stop[, step]) -> list of integers
- When step is given, it specifies the increment (or decrement).

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

How to get every second element in a list?

```
for i in range(0, len(data), 2):
    print data[i]
```

Modules/Libraries

- Modules are additional pieces of code that further extend Python’s functionality
- A module typically has a specific function
  - additional math functions, databases, network…
- Python comes with many useful modules

Operators

- Booleans: and or not < <= >= > == != <>
- Identity: is, is not
- Membership: in, not in
- Bitwise: | ^ & ~

No ++ -- +=, etc.

String Operators

- Concatenation: +
- Repeat: *
- Index: str[i]
- Slice: str[i:j]
- Length: len(str)
- String Formatting: "a %s parrot" % 'dead‘
- Iteration: for char in str
- Membership: ‘m’ in str

String Methods

- Assign a string to a variable

hi = “hello world”

- In this case “hw”
- hi.title()
- hi.upper()
- hi[0].isdigit()
- hi[0].islower()

Strings

- "hello"+"world" "helloworld" # concatenation
- "hello"*3 "hellohellohello" # repetition
- "hello"[0] "h" # indexing
- "hello"[-1] "o" # (from end)
- "hello"[1:4] "ell" # slicing
- len("hello") 5 # size
- "hello" < "jello"1 # comparison
- "e" in "hello" 1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"

String Methods

- The string held in your variable remains the same
- The method returns an altered string
- Changing the variable requires reassignment
  - hi = hi.upper()
  - hi now equals “HELLO WORLD”

Common Statements

- Assignment - curly, moe, larry = 'good', 'bad', 'ugly'
- Calls - stdout.write("spam, ham, toast\n")
- Print - print 'The Killer', joke
- If/elif/else - if "python" in text: print text
- For/else - for X in mylist: print X
- While/else - while 1: print 'hello'
- Break, Continue - while 1: if not line: break
- Try/except/finally - try: action() except: print 'action error'

Common Statements

- Raise - raise endSearch, location
- Import, From - import sys; from sys import stdin
- Def, Return - def f(a, b, c=1, d): return a+b+c+d
- Class - class subclass: staticData = []
- Global - function(): global X, Y; X = 'new'
- Del - del data[k]; del data [i:j]; del obj.attr
- Exec - yexec "import" + modName in gdict, ldict
- Assert - assert X > Y

Interactive “Shell”

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations: IDLE (GUI), python (command line)
- Type statements or expressions at prompt:

```
>>> print "Hello, world"
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Reference Semantics

- Assignment manipulates references
    - x = y **does not make a copy** of y
    - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Functions, Procedures

```
def name(arg1, arg2, ...):
    """documentation"""    # optional doc string
    statements

return              # from procedure
return expression        # from function
```

Example Function

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'
>>> gcd(12, 20)
4
```

Using Functions

- We use the **def** statement to start a block of code that will only be run when our function is called
- We want our functions to return values we can use in our geoprocessing
- The last statement in our function should be **return**.

Basic Function

- A simple function that adds 1 to value passed

-
  ```
  def addone(x):
          return x + 1
  ```

-
  ```
  print addone(2)
  >>> 3
  ```

-
  ```
  var = addone(3)
  print var
  >>> 4
  ```

Python Data Structures

- Lists (mutable sets of strings)
  - `var = [] # create list`
  - `var = [‘one’, 2, ‘three’, ‘banana’]`
- Tuples (immutable sets)
  - `var = (‘one’, 2, ‘three’, ‘banana’)`
- Dictionaries (associative arrays or ‘hashes’)
  - `var = {} # create dictionary`
  - `var = {‘lat’: 40.20547, ‘lon’: -74.76322}`
  - `var[‘lat’] = 40.2054`
- Each has its own set of methods

Lists

- Flexible arrays, *not* Lisp-like linked lists
    - a = [99, "bottles of beer", ["on", "the", "wall"]]
- Same operators as for strings
    - a+b, a*3, a[0], a[-1], a[1:], len(a)
- Item and slice assignment
    - a[0] = 98
    - a[1:2] = ["bottles", "of", "beer"]

      -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
    - del a[-1]   # -> [98, "bottles", "of", "beer"]

More Dictionary Ops

- Keys, values, items:
  - d.keys() -> ["duck", "back"]
  - d.values() -> ["duik", "rug"]
  - d.items() -> [("duck","duik"), ("back","rug")]
- Presence check:
  - d.has_key("duck") -> 1; d.has_key("spam") -> 0
- Values of any type; keys almost any
  - {"name":"Guido", "age":43, ("hello","world"):1, 42:"yes", "flag": ["red","white","blue"]}

More List Operations

```
>>> a = range(5)          # [0,1,2,3,4]
>>> a.append(5)           # [0,1,2,3,4,5]
>>> a.pop()               # [0,1,2,3,4]
5
>>> a.insert(0, 42)            # [42,0,1,2,3,4]
>>> a.pop(0)              # [0,1,2,3,4]
5.5
>>> a.reverse()           # [4,3,2,1,0]
>>> a.sort()              # [0,1,2,3,4]
```

Dictionaries

- Hash tables, "associative arrays"
  - d = {"duck": "eend", "water": "water"}
- Lookup:
  - d["duck"] -> "eend"
  - d["back"] # raises KeyError exception
- Delete, insert, overwrite:
  - del d["water"] # {"duck": "eend", "back": "rug"}
  - d["back"] = "rug" # {"duck": "eend", "back": "rug"}
  - d["duck"] = "duik" # {"duck": "duik", "back": "rug"}

Lists

- Think of a list as a stack of cards, on which your information is written
- The information stays in the order you place it in until you modify that order
- Methods return a string or subset of the list or modify the list to add or remove components
- Written as var[*index*], index refers to order within set (think card number, starting at 0)
- You can step through lists as part of a loop

List Methods

- Adding to the List
  - var[*n*] = *object*
    - replaces *n* with *object*
  - var.append(*object*)
    - adds *object* to the end of the list
- Removing from the List
  - var[*n*] = []
    - empties contents of card, but preserves order
  - var.remove(*n*)
    - removes card at *n*
  - var.pop(*n*)
    - removes *n* and returns its value

Tuples

- Like a list, tuples are iterable arrays of objects
- Tuples are immutable –
  once created, unchangeable
- To add or remove items, you must redeclare
- Example uses of tuples
  - County Names
  - Land Use Codes
  - Ordered set of functions

Dictionaries

- Dictionaries are sets of key & value pairs
- Allows you to identify values by a descriptive name instead of order in a list
- Keys are unordered unless explicitly sorted
- Keys are unique:
  - var[‘item’] = “apple”
  - var[‘item’] = “banana”
  - print var[‘item’] prints just banana

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing

Tuples

- key = (lastname, firstname)
- point = x, y, z   # parentheses optional
- x, y, z = point   # unpack
- lastname = key[0]
- singleton = (1,)   # trailing comma!!!
- empty = ()   # parentheses!
- tuples vs. lists; tuples immutable

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables
- Importing modules:
  - import re; print re.match("[a-z]+", s)
  - from re import match; print match("[a-z]+", s)
- Import with rename:
  - import re as regex
  - from re import match as m
  - Before Python 2.0:
    - import re; regex = re; del re

Packages

- Collection of modules in directory
- Must have __init__.py file
- May contain subpackages
- Import syntax:
  - from P.Q.M import foo; print foo()
  - from P.Q import M; print M.foo()
  - import P.Q.M; print P.Q.M.foo()
  - import P.Q.M as M; print M.foo()   # new

Catching Exceptions

```
def foo(x):
    return 1/x

def bar(x):
    try:
        print foo(x)
    except ZeroDivisionError, message:
        print "Can’t divide by zero:", message

bar(0)
```

Try-finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()  # always executed
print "OK"  # executed on success only
```

Raising Exceptions

- raise IndexError
- raise IndexError("k out of range")
- raise IndexError, "k out of range"
- try:
  ```
      something
  except: # catch everything
      print "Oops"
      raise  # reraise
  ```

More on Exceptions

- User-defined exceptions
  - subclass Exception or any other standard exception
- Old Python: exceptions can be strings
  - WATCH OUT: compared by object identity, not ==
- Last caught exception info:
  - sys.exc_info() == (exc_type, exc_value, exc_traceback)
- Last uncaught exception (traceback printed):
  - sys.last_type, sys.last_value, sys.last_traceback
- Printing exceptions: traceback module

File Objects

- f = open(*filename*[, *mode*[, *buffersize*]])
  - mode can be "r", "w", "a" (like C stdio); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - buffersize: 0=unbuffered; 1=line-buffered; buffered
- methods:
  - read([*nbytes*]), readline()*,* readlines()
  - write(*string*), writelines(*list*)
  - seek(*pos*[, *how*]), tell()
  - flush(), close()
  - fileno()

Standard Library

- Core:
  - os, sys, string, getopt, StringIO, struct, pickle, ...
- Regular expressions:
  - re module; Perl-5 style patterns and matching rules
- Internet:
  - socket, rfc822, httplib, htmllib, ftplib, smtplib, ...
- Miscellaneous:
  - pdb (debugger), profile+pstats
  - Tkinter (Tcl/Tk interface), audio, *dbm, ...

Standard Library

- https://docs.python.org/2/library/index.html

The Python Standard Library

- While *The Python Language Reference* describes the exact syntax and semantics of the Python language, this library reference manual describes the standard library that is distributed with Python. It also describes some of the optional components that are commonly included in Python distributions

Classes

```
class name:
    "documentation"
    statements
```

-or-

```
class name(base1, base2, ...):
    ...
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

May also be *class variable* assignments

Example Class

```
class Stack:
    "A well-known data structure…"
    def __init__(self):           # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)   # the sky is the limit
    def pop(self):
        x = self.items[-1]        # what happens if it’s empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0   # Boolean result
```

Using Classes

- To create an instance, simply call the class object:

```
x = Stack()  # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()   # -> 1
x.push(1)                    # [1]
x.empty()   # -> 0
x.push("hello")                    # [1, "hello"]
x.pop()             # -> "hello"        # [1]
```

- To inspect instance variables, use dot notation:

```
x.items             # -> [1]
```

Subclassing

```
class FancyStack(Stack):
    "stack with added ability to inspect inferior stack items"

    def peek(self, n):
        "peek(0) returns top; peek(-1) returns item below that; etc."
        size = len(self.items)
        assert 0 <= n < size                    # test precondition
        return self.items[size-1-n]
```

Subclassing (2)

```
class LimitedStack(FancyStack):
    "fancy stack with limit on stack size"

    def __init__(self, limit):
        self.limit = limit
        FancyStack.__init__(self)        # base class constructor

    def push(self, x):
        assert len(self.items) < self.limit
        FancyStack.push(self, x)         # "super" method call
```

Class / Instance Variables

```
class Connection:
    verbose = 0                    # class variable
    def __init__(self, host):
        self.host = host           # instance variable
    def debug(self, v):
        self.verbose = v           # make instance variable!
    def connect(self):
        if self.verbose:           # class or instance variable?
            print "connecting to", self.host
```

Instance Variable Rules

- On use via instance (self.x), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (self.x = ...):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

Python 2.6 or 2.7

- Python 2.6 or 2.7

- Not 3+

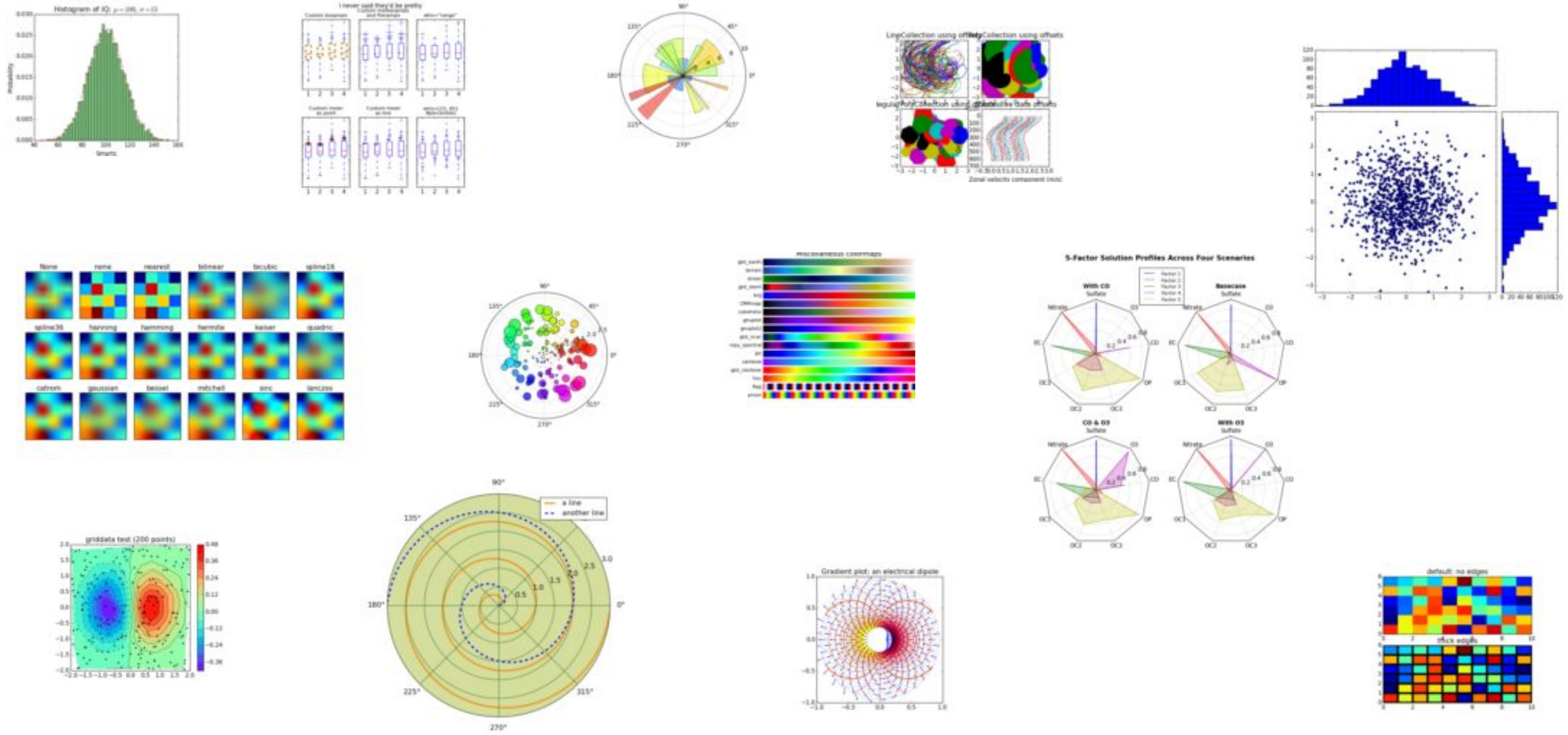
NumPy

NumPy is the fundamental package for scientific computing with Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

SciPy

SciPy (pronounced “Sigh Pie”) is a computing environment and open source ecosystem of software for the Python programming language used by scientists, analysts and engineers doing scientific computing and technical computing. SciPy also refers to a specific open source library / Python package of algorithms and mathematical tools that form a core element of the SciPy environment for technical computing. The SciPy environment includes the NumPy and SciPy libraries, along with an expanding set of additional scientific computing libraries like IPython, Matplotlib, pandas and SymPy. It has similar users to other applications such as MATLAB, GNU Octave, and Scilab.

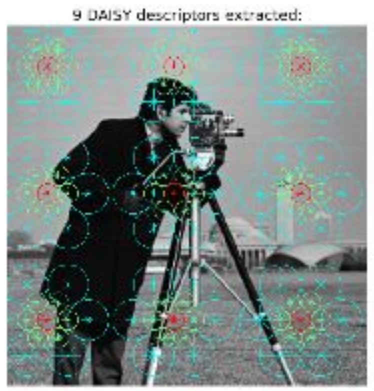
matplotlib

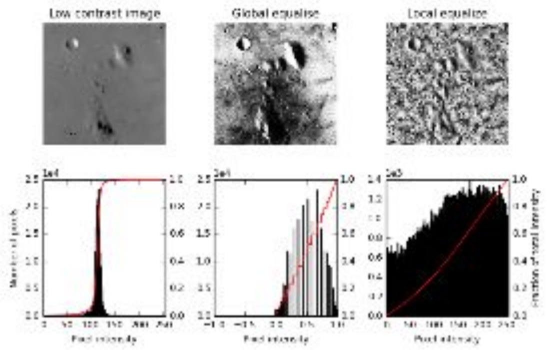


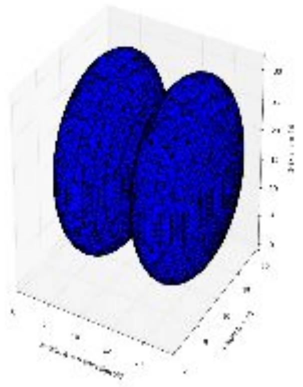
Histogram of IQ
I never said they'd be pretty
LineCollection using offsets
PolyCollection using offsets
Miscellaneous colormaps
5-Factor Solution Profiles Across Four Scenarios
griddata test (200 points)
a line
another line
Gradient plot: an electrical dipole
default: no edges
thick edges

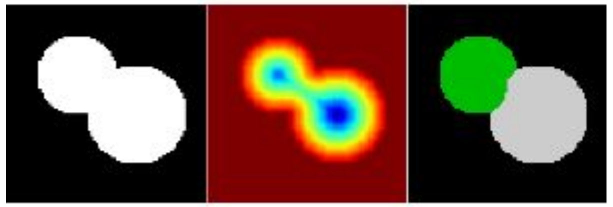
scikit-image

- scikit-image is a collection of algorithms for image processing. http://scikit-image.org/docs/dev/auto_examples/



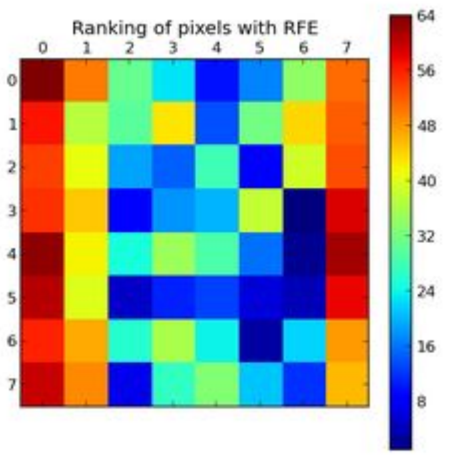


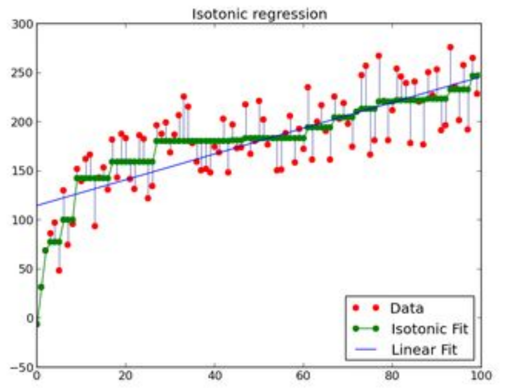


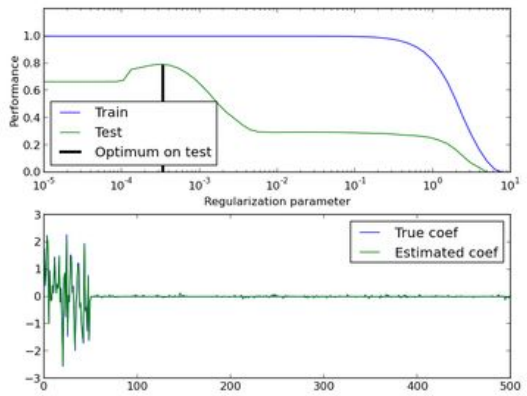


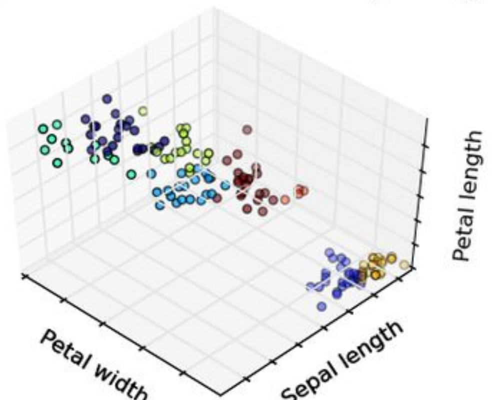
scikit-learn

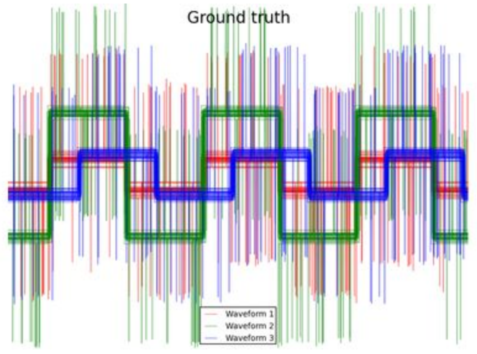
- scikit-learn Machine Learning in Python. Simple and efficient tools for data mining and data analysis. http://scikit-learn.org/stable/auto_examples/





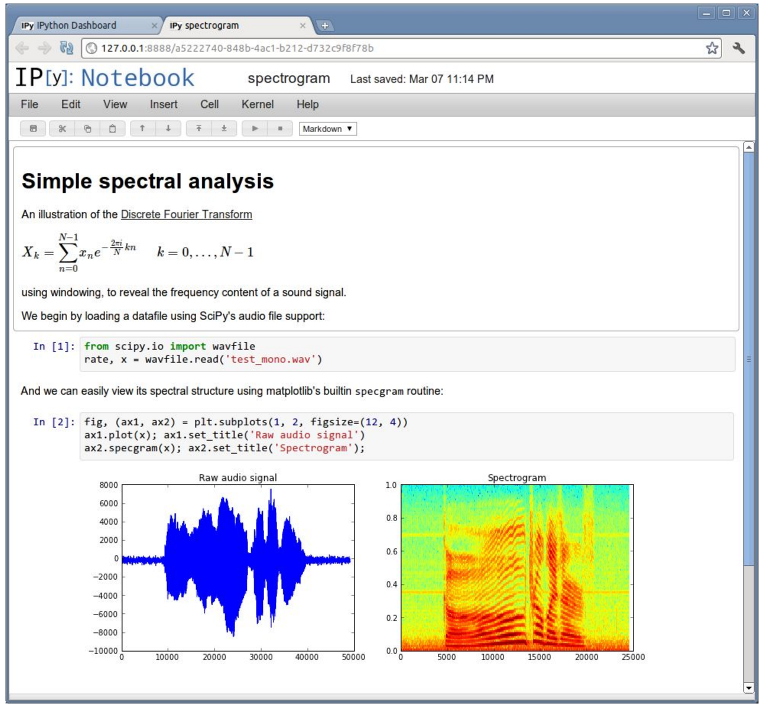






IPython

- The IPython Notebook is a web-based interactive computational environment where you can combine code execution, text, mathematics, plots and rich media into a single document:



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \dots, N-1$$

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a datafile using SciPy's audio file support:

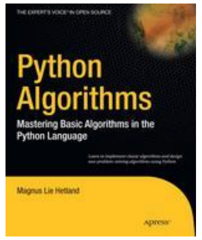
```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

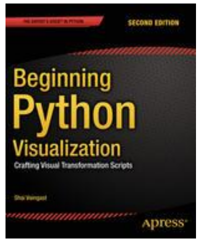
```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

Python Books (http://link.springer.com/)

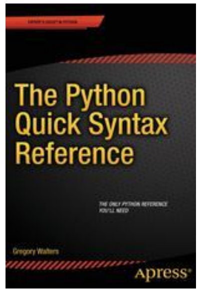
Python Algorithms: Mastering Basic Algorithms in the Python Language
Magnus Lie Hetland *(2010)*
http://link.springer.com/book/10.1007/978-1-4302-3238-4



Beginning Python Visualization
Crafting Visual Transformation Scripts
Shai Vaingast *(2014)*
http://link.springer.com/book/10.1007/978-1-4842-0052-0

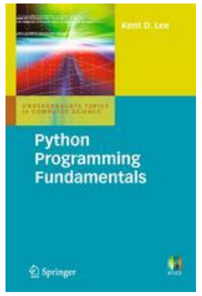

SECOND EDITION
Beginning Python Visualization
Crafting Visual Transformation Scripts
Apress

The Python Quick Syntax Reference
Gregory Walters *(2014)*
http://link.springer.com/book/10.1007/978-1-4302-6479-8

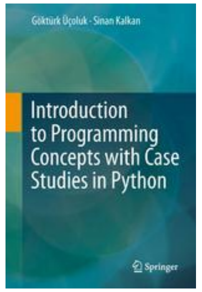


Python Books (http://link.springer.com/)

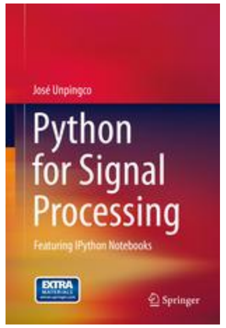
Python Programming Fundamentals
Kent D. Lee in *Undergraduate Topics in Computer Science (2011)*
http://link.springer.com/book/10.1007/978-1-84996-537-8



Introduction to Programming Concepts with Case Studies in Python
Göktürk Üçoluk, Sinan Kalkan *(2012)*
http://link.springer.com/book/10.1007/978-3-7091-1343-1

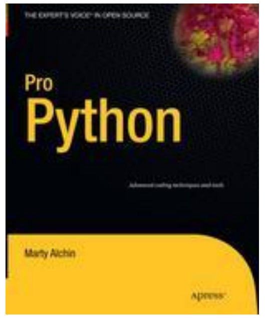

Göktürk Üçoluk · Sinan Kalkan
Introduction to Programming Concepts with Case Studies in Python
Springer

Python for Signal Processing
Featuring IPython Notebooks
José Unpingco *(2014)*
http://link.springer.com/book/10.1007/978-3-319-01342-8

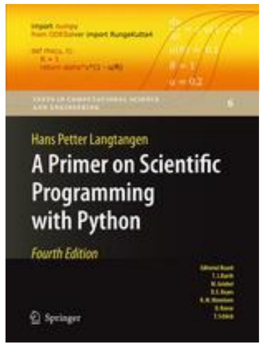


Python Books (http://link.springer.com/)

Pro Python
Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham… *(2010)*



A Primer on Scientific Programming with Python
Hans Petter Langtangen in *Texts in Computational Science and Engineering* *(2014)*



Python Tutorials

Things to read through

- "Dive into Python" (Chapters 2 to 4)
  http://diveintopython.org/
- Python 101 – Beginning Python
  http://www.rexx.com/~dkuhlman/python_101/python_101.html

Things to refer to

- The Official Python Tutorial
  http://www.python.org/doc/current/tut/tut.html
- The Python Quick Reference
  http://rgruet.free.fr/PQR2.3.html

YouTube Python Tutorials

Python Fundamentals Training – Classes
http://www.youtube.com/watch?v=rKzZEtxIX14

Python 2.7 Tutorial Derek Banas·
http://www.youtube.com/watch?v=UQi-L-_chcc

Python Programming Tutorial - thenewboston

http://www.youtube.com/watch?v=4Mf0h3HphEA

Google Python Class
http://www.youtube.com/watch?v=tKTZoB2Vjuk

codecademy.com python

codecademy.com

http://www.codecademy.com/tracks/python

YouTube Python Tutorials

Image analysis in python with scipy and scikit image tutorials

Part 1 http://www.youtube.com/watch?v=MP-MTiCETYg

Part 2 http://www.youtube.com/watch?v=SE7h0IWD93Y

Part 3 http://www.youtube.com/watch?v=Yxpnvc4RHy4

Scientific Python distributions

- Python 2.6 or 2.7
  https://www.python.org/download/releases/2.7.6/
- NumPy
  http://www.numpy.org/
- SciPy
  http://www.scipy.org/
- matplotlib
  http://matplotlib.org/
- scikit-learn http://scikit-learn.org
- scikit-image http://scikit-image.org

Scientific Python distributions

- For most users, especially on Windows and Mac, the easiest way to install the packages of the SciPy stack is to download one of these Python distributions, which includes all the key packages:
- Anaconda: A free distribution for the SciPy stack. Supports Linux, Windows and Mac.
- Enthought Canopy: The free and commercial versions include the core SciPy stack packages. Supports Linux, Windows and Mac.
- Python(x,y): A free distribution including the SciPy stack, based around the Spyder IDE. Windows only.
- WinPython: A free distribution including the SciPy stack. Windows only.
- Pyzo: A free distribution based on Python 3 (see *Note on Python 3*) with the IEP editor. Supports Linux and Windows.
- Algorete Loopy: A free, community oriented distribution for the SciPy stack maintained by researches at Dartmouth College. Loopy supports both Python 2 and 3 on Linux, Windows and Mac OSX. The distribution is derived from Anaconda with additional packages (e.g. Space Physics, BioInformatics).