Machine Learning and Data Science with Python

Data Science International Summer School
Predeal
August 25, 2018

Razvan C. Bunescu
School of Electrical Engineering and Computer Science
http://ace.cs.ohio.edu/~razvan
bunescu@ohio.edu
Python Programming Stack (Today)

- **Python** = object-oriented, interpreted, scripting language.
  - imperative programming, with functional programming features.

- **NumPy** = package for powerful N-dimensional arrays:
  - sophisticated (broadcasting) functions.
  - useful linear algebra, Fourier transform, and random number capabilities.

- **SciPy** = package for numerical integration and optimization.

- **Matplotlib** = comprehensive 2D plotting library.
Python Programming Stack

• **Scikit-learn** = Python package with implementations of many ML algorithms and data pre-processing tools:

• **PyTorch** = Deep learning platform in Python:
  – Tensors, wrappers of Numpy n-dimensional array
    • Can be used on GPUs
  – Easy to define Neural Networks as computations graphs.
  – Automatic differentiation for computing gradients.
  – Many gradient-based algorithms for training.
Programming with Python

- Designed by Guido van Rossum in the early 1990s.
- Current development done by the Python Software Foundation.

- Python facilitates multiple programming paradigms:
  - imperative programming.
  - object oriented programming.
  - functional programming.

$\Rightarrow$ multi-paradigm programming language.
Python: Important Features

- Python = object-oriented interpreted “scripting” language:
  - **Object oriented:**
    - modules, classes, exceptions.
    - dynamically typed, automatic garbage collection.
  - **Interpreted**, interactive:
    - rapid edit-test-debug cycle.
  - **Extensible**:
    - can add new functionality by writing modules in C/C++.
  - **Standard library**:
    - extensive, with hundreds of modules for various services such as regular expressions, TCP/IP sessions, etc.
The Python Interpreter

Running the interpreter

[razvan@ohio ~]$ python3
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on
darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out the tutorial on the Internet at https://docs.python.org/3.6/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type "modules", "keywords", "symbols", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose name or summary contain a given string such as "spam", type "modules spam".

help>
The Python Interpreter: Keywords

help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

False      def     if     raise
None       del     import  return
True       elif     in     try
and        else     is     while
as         except   lambda  with
assert     finally  nonlocal
break      for      not
class      from     or
continue   global   pass
Lambda expressions (sometimes called lambda forms) have the same syntactic position as expressions. They are a shorthand to create anonymous functions; the expression "lambda arguments: expression" yields a function object. The unnamed object behaves like a function object defined with

```python
def <lambda>(arguments):
    return expression
```

See section Function definition* for the syntax of parameter lists. *Note that functions created with lambda expressions cannot contain statements or annotations.*
help> modules

commands:
execute shell commands via os.popen() and return status, output.

compiler:
package for parsing and compiling Python source code.

gzip:
functions that read and write gzipped files.

HTMLParser:
a parser for HTML and XHTML (defines a class HTMLParser).

math:
access to the mathematical functions defined by the C standard.

exceptions:
Python’s standard exception class hierarchy.
help> modules

**os:**
OS routines for Mac, NT, or Posix depending on what system we’re on.

**re:**
support for regular expressions (RE).

**string:**
a collection of string operations (most are no longer used).

**sys:**
access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter:

- `sys.argv`: command line arguments.
- `sys.stdin, sys.stdout, sys.stderr`: standard input, output, error file objects.

**threading:**
thread modules emulating a subset of Java’s threading model.
The Python Interpreter: Modules

**numpy**: NumPy.

**scipy**: SciPy: A scientific computing package for Python.

**matplotlib**: This is an object-oriented plotting library.

**sklearn**: Machine learning module for Python.

**torch**: PyTorch.

**conda**: OS-agnostic, system-level binary package manager.

**pandas**: a powerful data analysis and manipulation library for Python

**nltk**: Python library for Natural Language Processing
The Python Interpreter: Integer Precision

>>> def fibo(n):
...     a, b, i = 0, 1, 0
...     while i < n:
...         a, b, i = b, a+b, i+1
...     return a

>>> fibo(10)
55
>>> fibo(100)
354224848179261915075
>>> fibo(1000)
4346655768693745643568852767504062580256466051737178040248172908953655541
4905189040387984007925516929592259308032263477520968962323987332247116164
996440906533187938298969649928516003704476137795166849228875
Built-in Types: Basic

- **integers**, with unlimited precision – int().
  - decimal, octal and hex literals.
- **floating point numbers**, implemented using double in C – float().
- **complex numbers**, real and imaginary as double in C – complex().
  - 10+5j, 1j
  - z.real, z.imag
- **boolean values**, True and False – bool().
  - False is: None, False, 0, 0L, 0.0, 0j, ‘’, (), [], {},
  - user defined class defines methods nonzero() or len().
- **strings** – str(), class, function, …
  - “Hello world”, ‘Hello world’
Built-in Types: Composite

- **lists:**
  - [], [1, 1, 2, 3], [1, “hello”, 2+5j]

- **tuples:**
  - (), (1,), (1, 1, 2, 3), (1, “hello, 2+5j)

- **dictionaries:**
  - {“john”: 12, “elaine”: 24}

- **sets:**
  - {1, 2, 3}

- **files**
Integers

```python
>>> int
<class 'int'>
>>> 1024
1024
>>> int(1024)
1024
>>> repr(1024)
'1024'
>>> eval('1024')
1024
>>> str(1111)
'1111'
>>> int('1111')
1111
```

```python
>>> a = 100
>>> b = 200
>>> a + 1, b + 1  # this is a tuple
(101, 201)
>>> print(a, b + 1)
100 201
>>> int(3.6), abs(-10), 11%3, 11//3, 11/3, 2**3
(3, 10, 2, 3, 3.6666666666666665, 8)
>>> int('1110',2), int('170',8), int('40',16)
(14, 120, 64)
>>> [170, 0170, 0x40]  # this is a list
[170, 120, 64]
>>> float(8), 2**3, pow(2,3)
(8.0, 8, 8)
```
Booleans

```python
>>> bool
<class 'bool'>

[bool(0), bool(0.0), bool(0j), bool([]), bool(), bool({}), bool(‘’), bool(None)]
[False, False, False, False, False, False, False, False]

[bool(1), bool(1.0), bool(1j), bool([1]), bool((1,)), bool({1:’one’}), bool(‘1’)]
[True, True, True, True, True, True, True]

>>> str(True), repr(True)
(‘True’, ‘True’)

>>> True and True, True and False, False and False
(True, False, False)

>>> True or True, True or False, False or False
(True, True, False)

>>> not True, not False
(False, True)
```
Floating Point

>>> float
<class 'float'>
>>> 3.14, 3.1400000000000001
(3.14, 3.14)
>>> repr(3.1400000000000001)
3.14
>>> 3.14/2, 3.14//2
(1.5, 1.0)
>>> 1.99999999999999999999
2.0
>>> import math
>>> math.pi, math.e
(3.1415926535897931, 2.7182818284590451)
>>> help('math')

------------- Python -------------

error = 0
for i in range(10):
    error -= 0.1
if sum == 0:
    print('Converged!')

------------- C++ -------------

float sum = 0.0;
for (int i = 0; i < 10; i++)
    sum += 0.1;
cout.precision(17);
cout << sum << endl;
⇒ 1.0000001192092896

http://docs.python.org/3/tutorial/introduction.html#numbers

ML and Data Science with Python
IEEE 754 Standard

https://en.wikipedia.org/wiki/IEEE_754_revision

\[ \text{value} = (-1)^{\text{sign}} \times 2^{(e-127)} \times \left( 1 + \sum_{i=1}^{23} b_{23-i} 2^{-i} \right) \]
Strings (Immutable)

- Immutable Sequences of Unicode Characters:

```python
>>> str
<class 'str'>
>>> s = "object oriented"
>>> len(s), s.find('ct'), s.split()
(15, 4, ['object', 'oriented'])
>>> str.upper('it'), 'it'.upper()
('IT', 'IT')
>>> s[0], s[1], s[4:6], s[7:]
('o', 'b', 'ct', 'oriented')
>>> '', ''
>>> s[7:100]
>>> "functional {0} lang".format('programming')
'functional programming'
>>> "object oriented " + "programming"
'object oriented programming'
>>> 'orient' in 'object oriented', 'orient' in 'Object Oriented'
(True, False)
```

[http://docs.python.org/3/tutorial/introduction.html#strings](http://docs.python.org/3/tutorial/introduction.html#strings)
List (Mutable)

```python
>>> []  #empty list
[]
>>> x = [3.14, "hello", True]
[3.14, 'hello', True]
>>> x + [10, [], len(x)]
[3.14, 'hello', True, 10, [], 3]
>>> x.append(0.0)
>>> x.reverse()
>>> x
[0.0, True, 'hello', 3.14]
>>> x.sort()
>>> x
[0.0, True, 'hello', 3.14]
>>> x * 2
[0.0, True, 'hello', 3.14, 0.0, True, 'hello', 3.14]
```

```python
>>> help('list')
```

ML and Data Science with Python

22
Tuple (Immutable)

>>> tuple
<class 'tuple'>
>>> ()  # empty tuple
()
>>> (10)  # not a one element tuple!
10
>>> (10,)  # a one element tuple
(10,)
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 2
(1, 2, 1, 2)
>>> x = (0, 1, 'x', 'y')

>>> x[0], x[1], x[:-1]
(0, 1, (0, 1, 'x'))
>>> y = (13, 20, -13, -5, 0)
>>> temp = list(y)
>>> temp.sort()
>>> y = tuple(temp)
>>> y
(-13, -5, 0, 13, 20)

>>> help('tuple')

Immutable types are hashable!
Set (Mutable) & Frozenset (Immutable)

>>> set, frozenset
(<class 'set'>, <class 'frozenset'>)

>>> set()  # empty set
set()

>>> type(set())
<class 'set'>

>>> {}  # not an empty set!
{}

>>> type({})
<class 'dict'>

>>> s = {1, 3, 3, 2}

>>> s
{1, 2, 3}

>>> frozenset({1, 3, 3, 2})
frozenset({1, 2, 3})

>>> 3 in s, 4 in s
(True, False)

>>> s = set('yellow')

>>> t = set('hello')

>>> s
{'e', 'o', 'l', 'w', 'y'}

>>> t
{'h', 'e', 'l', 'o'}

>>> s – t, s | t
({'w', 'y'}, {'h', 'e', 'o', 'l', 'w', 'y'})

>>> s & t, s ^ t
({'e', 'l', 'o'}, {'h', 'w', 'y'})

>>> {1, 3} <= {1, 2, 3, 2}
True

>>> help(set) >>> help(frozenset)
Mutable Operations on Sets

```python
>>> s = set(['abba', 'dada', 'lola', 'bama'])
>>> s
{'abba', 'bama', 'dada', 'lola'}
>>> s |= {'bama', 'lily'}
>>> s
{'abba', 'bama', 'dada', 'lily', 'lola'}
>>> s
−
= {'lola', 'abba', 'mama'}
>>> s
{'bama', 'dada', 'lily'}
>>> s &= {'lily', 'dodo', 'bama'}
>>> s
{'bama', 'lily'}
>>> s = {[1]}
TypeError: unhashable type: ‘list’
```

How can we prove the actual set object changes and not a new one is created?

Hint: are `s -= t` and `s = s − t` equivalent for sets? How about strings?
Dictionaries (Mutable)

```python
>>> dict
<class 'dict'>
>>> {}
{}  # empty dictionary

>>> d = {'john':23, 'alex':25, 'bill':99}

>>> d['alex']
25

>>> d['alex'] = 26

>>> del d['john']

>>> d['tammy'] = 35

>>> d
{'alex':26, 'bill':99, 'tammy':35}

>>> d.keys() | ['alex', 'john']  # set ops.
{'alex', 'bill', 'john', 'tammy'}

>>> d.items()   # this is a view
dict_items[('alex',26), ('bill',99), ('tammy',35)]

>>> d.keys()   # this is a view
dict_keys['alex', 'bill', 'tammy']

>>> d.values()   # this is a view
dict_values[26, 99, 35]

>>> for x, y in d.items():
    ... print (x, y)

alex 26
bill 99
tammy 35

>>> for key in d:
    ... print(key, end = '  
alex  bill  tammy'

>>> d['mary'] = 10

>>> d.keys()

>>> d['mary'] = 10

>>> d.keys()

```

ML and Data Science with Python
Dictionaries (Mutable)

>>> dict
<class 'dict'>

>>> d.items()    # this is a view
dict_items[('alex',26), ('bill',99), ('tammy',35)]

>>> {}    # empty dictionary
{}

dict_keys[‘alex’, ‘bill’, ‘tammy’]

>>> d = {'john':23, 'alex':25, 'bill':99}

>>> d.values()    # this is a view
>>> d[‘alex’]
dict_values[26, 99, 35]

25

>>> for x, y in d.items():
... 

>>> d[‘alex’] = 26

>>> … print (x, y)

>>> del d[‘john’]

>>> d[‘tammy’] = 35

>>> d
‘bill’ 99

>>> d[‘mary’] = 10

>>> d[1, 2, 3, 4] = 4

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'

Immutable types are hashable, mutable types are not:

>>> d = dict(abba=1, dada=2)

>>> d[frozenset({1, 2, 3})] = 3

>>> d
{‘abba’:1, ‘dada’:2, frozenset({1, 2, 3}):3}

>>> d[{1, 2, 3, 4}] = 4

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'

>>> for key in d:
... 

>>> … print(key, end = ‘ ‘)

‘alex’ ‘bill’ ‘tammy’

>>> d.keys() | 

>>> for key in d:
... 

>>> d[‘mary’] = 10

>>> d.keys()
dict_keys[‘alex’, ‘bill’, ‘tammy’, ‘mary’]
Files

>>> file
<type ‘file’>

>>> output = open('/tmp/output.txt', 'w') # open tmp file for writing
>>> input = open('/etc/passwd', 'r') # open Unix passwd file for reading
>>> s = input.read() # read entire file into string s
>>> line = input.readline() # read next line
>>> lines = input.readlines() # read entire file into list of line strings

>>> output.write(s) # write string S into output file
>>> output.write(lines) # write all lines in ‘lines’

>>> output.close()
>>> input.close()

>>> from urllib.request import urlopen
>>> html = urlopen('https://datascience.ase.ro')
>>> lines = [line.rstrip() for line in html.readlines()]

ML and Data Science with Python
Statements & Functions

• Assignment Statements
• Compound Statements
• Control Flow:
  – Conditionals
  – Loops
• Functions:
  – Defining Functions
  – Lambda Expressions
  – Documentation Strings
  – Generator Functions
Assignment Forms

- **Basic form:**
  - $x = 1$
  - $y = ‘John’$

- **Tuple positional assignment:**
  - $x, y = 1, ‘John’$
  - $x == 1, b == ‘John’ => (True, True)$

- **List positional assignment:**
  - $[x, y] = [1, ‘John’]$

- **Multiple assignment:**
  - $x = y = 10$
Compound Statements

- Python does not use block markers (e.g. ‘begin .. end’ or ‘{ … }’) to denote a compound statements.
  - Need to indent statements at the same level in order to place them in the same block.
  - Can be annoying, until you get used to it; intelligent editors can help.
  - Example:

```python
if n == 0:
    return 1
else:
    return n * fact(n - 1)
```
Conditional Statements

if <bool_expr_1>:
    <block_1>
elif <bool_expr_2>:
    <block_2>
...
else:
    <block_n>

• There can be zero or more `elif` branches.
• The `else` branch is optional.
• “Elif“ is short for “else if” and helps in reducing indentation.
Conditional Statements: Example

```python
code
name = 'John'
...
if name == 'Mary':
    sex = 'female'
elif name == 'John':
    sex = 'male'
elif name == 'Alex':
    sex = 'unisex'
elif name == 'Paris':
    sex = 'unisex'
else:
    sex = 'unknown'
```
Conditional Statements

• There is no C-like ‘switch’ statement in Python.
• Same behavior can be achieved using:
  – if … elif … elif sequences.
  – dictionaries:
    name = ‘John’
    dict = {‘Mary’:‘female’, ‘John’: ‘male’,
             ‘Alex’:‘unisex’, ‘Paris’:‘unisex’}
    if name in dict:
        print(dict[name])
    else:
        print(‘unknown’)
While Loops

```python
x = 'university'
while x:
    print(x, end = ' ')
    x = x[1:]
```

```python
a, b = 1, 1
while b <= 23:
    print(a, end = ' ')
    a, b = b, a + b
```
For Loops

sum = 0
for x in [1, 2, 3, 4]
    sum += x
print(sum)

D = {1:'a', 2:'b', 3:'c', 4:'d'}
for x, y in D.items():
    print(x, y)
Names and Scopes

• Static scoping rules.
  – if $x$ is a variable name that is only read, its variable is found in the closest enclosing scope that contains a defining write.
  – a variable $x$ that is written is assumed to be local, unless it is explicitly imported.
  – use \texttt{global} and \texttt{nonlocal} keywords to override these rules.

• Example:
Functions

def mul(x, y):
    return x * y

mul(2, 5) => ?
mul(math.pi, 2.0) => ?
mul([1, 2, 3], 2) => ?
mul(('a', 'b'), 3) => ?
mul('ou', 5) => ?
Parameter Correspondence

def f(a, b, c): print(a, b, c)
f(1, 2, 3) => 1 2 3
f(b = 1, c = 2, a = 3) => 3 1 2

def f(*args): print(args)
f("one argument") => ('one argument')
f(1, 2, 3) => (1, 2, 3)

def f(**args): print(args)
f(a=2, b=4, c=8) => { 'a':2, 'b':4, 'c':8}
Lambda Expressions

- **Scheme:**
  ```scheme
  >(define (make-adder (num)
    (lambda (x)
      (+ x num)))
  ```

- **Python:**
  ```python
  >>> def make_adder(num):
  ...    return lambda x: x + num
  ...  
  >>> f = make_adder(10)
  >>> f(9)
  19
  ```
Lambda Expressions

```python
>>> formula = lambda x, y: x*x + x*y + y*y
>>> formula
<function <lambda> at 0x2b3f213ac230>
>>> apply(formula, (2,3))
19
>>> list(map(lambda x: 2*x, [1, 2, 3]))
[2, 4, 6]
>>> list(filter(lambda x: x>0, [1, -1, 2, -2, 3, 4, -3, -4]))
[1, 2, 3, 4]
>>> from functools import reduce
>>> reduce(lambda x,y:x*y, [1, 2, 3, 4, 5])
120
>>> def fact(n): return reduce (lambda x, y: x*y, range(1, n+1))
>>> fact(5)
120
```
Iterators

- **An iterator** is an object representing a stream of data:
  - to get the next element in the stream:
    - call `__next__()` method.
    - pass the iterator to the built-in function `next()`.
  - to create an iterator:
    - call `iter(collection)`.
    - some functions create iterators/iterables instead of collections:
      - `map()`, `filter()`, `zip()`, ...
      - `range()`, `dict.keys()`, `dict.items()`, ...
    - **why create iterators/iterables instead of collections?**

- **Examples:**
Iterators

```python
for x in range(5):
    print(x)

Internally, this is implemented as:

it = iter(range(5))
while True:
    try:
        x = next(it)
        print(x)
    except StopIteration:
        break
```

an iterable (provides the `__iter__()` method)
A Convenient Shortcut to Building Iterators: Generator Functions/Expressions

def squares(n):
    for i in range(n):
        yield i * i

>>> for i in squares(5):
...    print(i, end = ' ')
0 1 4 9 16

>>> s = squares(5)
>>> next(s) => 0
>>> next(s) => 1
>>> next(s) => 4

Equivalent generator expression:

```python
>>> (i * i for i in range(n))
```
Generator Functions

def fib():  # generate Fibonacci series
    a, b = 0, 1
    while 1:
        yield b
        a, b = b, a+b

>>> it = fib()
>>> next(it) => 1
>>> next(it) => 1
>>> next(it) => 2
List/Set/Dict Comprehensions

- Mapping operations over sequences is a very common task in Python coding:
  - Introduce a new language feature called *list comprehensions*.

```python
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 25]

>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]

>>> [x+y for x in [1, 2, 3] for y in [-1,0,1]]
[0,1,2,1,2,3,2,3,4]

>>> [(x,y) for x in range(5) if x%2 == 0 for y in range(5) if y%2==1]
[(0,1), (0,3), (2,1), (2,3), (4,1), (4,3)]
```
List/Set/Dict Comprehensions

```
['expression for target_1 in iterable_1 [if condition]
 for target_2 in iterable_2 [if condition]
 ...
 for target_n in iterable_n [if condition] ']
```

```python
>>> [line[:-1] for line in open('myfile')]
>>> {x for x in 'ohio' if x not in 'hawaii'}
>>> {x:2*x for x in 'ohio'}
>>> {x:y for x in [1, 2] for y in [3, 4]}
```
Errors & Exceptions

• Syntax errors:
  while True print ‘True’
  File "<stdin>", line 1
    while True print ‘True’
      ^
  SyntaxError: invalid syntax

• Exceptions: errors detected during execution:
  1 / 0
  Traceback (most recent call last):
    File "<stdin>", line 1, in ?
  ZeroDivisionError: integer division or modulo by zero
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
    f.close()
**Modules**

- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
- A **module** is a file containing Python definitions and statements.
- The file name is the module name with the suffix `.py` appended.
- Within a module, the module's name (as a string) is available as the value of the global variable `__name__`
# Fibonacci numbers module

def fib(n):
    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

>>> import fibo
>>> fibo.fib(100)
1 1 2 3 5 8 13 21 34 55 89
Readings

• Reading assignment:
  – https://docs.python.org/3/tutorial
    • https://docs.python.org/3/tutorial/classes.html#iterators
    • https://docs.python.org/3/tutorial/classes.html#generators
  – https://docs.python.org/3/library/functions
  – https://docs.python.org/3/library/stdtypes
  – https://docs.python.org/3/whatsnew/3.0.html
Linear Algebra in Python:

```python
import numpy as np
```

- `np.array()`
  - indexing, slices.
- `ndarray.shape, .size, .ndim, .dtype, .T`
- `np.zeros(), np.ones(), np.arange(), np.eye()`
  - `dtype` parameter.
  - `tuple` (shape) parameter.
- `np.reshape(), np.ravel()`
  - also np.ndarray.
- `np.amax(), np.maximum(), np.sum(), np.mean,() np.std()`
  - `axis` parameter, also np.ndarray
- `np.stack(), np.[hv]stack(), np.column_stack(), np.split()```
- `np.exp(), np.log()`,

ML and Data Science with Python
NumPy: Broadcasting

• Broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
• The smaller array is “broadcast” across the larger array so that they have compatible shapes, subject to broadcasting rules:
  – NumPy compares their shapes element-wise.
  – It starts with the trailing dimensions, and works its way forward.
  – Two dimensions are compatible when:
    • they are equal, or one of them is 1.

  ▪ https://docs.scipy.org/doc/numpy-dev/user/basics.broadcasting.html
Other Numpy Functions

- np.dot(), np.vdot()
  - also np.ndarray.
- np.outer(), np.inner()

- import numpy.random as random:
  - randn(), randint(), uniform()

- import numpy.linalg as la:
  - la.norm(), la.det(), la.matrix_rank(), np.trace()
  - la.eig(), la.svd()
  - la.qr(), la.cholesky()

- https://docs.scipy.org/doc/numpy/reference/routines.linalg.html
Optimization and Statistics in Python:

```python
import scipy
```

- **scipy.optimize:**
  - `scipy.optimize.fmin_l_bfgs_b()
    ```python
    w, _, _ = fmin_l_bfgs_b(lr_cost_grad, w,
    args = (X_data, y_data),
    maxiter = 100, disp = 1)
    ```
  - `scipy.optimize.fmin_cg()
  - `scipy.minimize`

[https://docs.scipy.org/doc/scipy-0.10.1/reference/tutorial/optimize.html](https://docs.scipy.org/doc/scipy-0.10.1/reference/tutorial/optimize.html)
Optimization and Statistics in Python:

from scipy import stats

• Probability distributions:
  – Continuous:
    • stats.norm, stats.chi2, stats.cauchy, stats.beta, …
    • stats.dirichlet, stats.multivariate_normal, …
  – Discrete:
    • stats.bernoulli, stats.binom, stats.poisson, …

• Statistical functions:
  – stats.moment, stats.normaltest, stats.variation, stats.zscore, …

• Statistical tests:
  – (ANOVA) f_oneway, pearsonr, spearmanr, kendalltau, ttest,
    chisquare, mannwhitneyu, (Wilcoxon’s) ranksum,
    wasserstein_distance, …
Creating Graphs and Plots in Python:

```python
import matplotlib.pyplot as plt
```