Python 101

Alex Belianinov
Center for Nanophase Materials Sciences

Machine Learning for Materials Research
2018

If you are not new to Python…

- A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.
- Find the largest palindrome made from the product of two 3-digit numbers.

About

- Describe Python 3.5
- Use python for analysis
  - Spyder, statements, keywords, variables, operations, loops, lists, libraries, IO)
- Programming style and thought process
- Numpy, SciPy & Matplotlib

Course Details

- Download and install the following:

- Anaconda ***Python 3.5***
  - https://www.continuum.io/downloads
-

Spyder

- Spyder is an integrated development environment that runs on all major platforms
- Builtin console, IPython, and debugger
- On Windows: C:\Anaconda\Scripts>spyder.bat
- On Mac: $ spyder
- Some IDE alternatives
  - IDLE
  - PyCharm
  - PyScripter
  - Komodo Edit
  - Eclipse with Pydev
  - Wing IDE
  - IEP (Python3-based)

What is Python

- Interpreted high-level programming language
- Similar to Tcl, Ruby, Perl
- Guido van Rossum in 1990, named after Monty Python
  "…A need for a language to bridge C and the shell"
- Pure python: **http://www.python.org**
- Version 2.7 is most common (3.6(?) current)
- Cpython, Jython, IronPython, PyPy

Python is great for…

- Text processing/data processing Application scripting
- Systems administration/programming Internet programming
- Graphical user interfaces
- Testing
- Writing quick "throw-away" code
- Glue/Wrapper code

Python is NOT great for…

- Device drivers and low-level systems
- Computer graphics, visualization, and games
- Numerical algorithms (but see Numba)
- Comment : Python is still used in these application domains, but only as a high-level control language. Important computations are actually carried out in C, C++, Fortran, etc. For example, you would not implement matrix-multiplication in pure Python (without Numba or Cython)

Statements

- A Python program is a sequence of statements
- Each statement is terminated by a newline
- Statements are executed one after the other until the end of the file
- Comments are denoted by # and extend to the end of the line
- There are no block comments in Python (e.g., /* ... */).

Variables

- A variable is just a name for some value, CASE SENSITIVE
- Variable names follow same rules as C [A-Za-z_][A-Za-z0-9_]*
- You do not declare types (int, float, etc.)
  - Total = 30000 #int
  - Total = 30000.0 #float
  - Total = “30000” #string
- Keywords (cannot be variable names):

```
and, elif, if, print,
as, else, import, raise,
assert, except, in, return,
break, exec, is, try,
class, finally, lambda, while,
continue, for, not with, def, from,
or, yield,
del, global, pass
```

First functional program (Exercise 3)

- You decided to save up for a new 32,767$ car, by setting aside a dollar bill, and then doubling the amount each of the following days. How long does it take to save up, and how tall is the dollar bill stack? Each bill is ~100 um thick.

Looping

- The **while** statement executes a loop (terminated by a colon)
- Looping executes the indented statements underneath while the condition evaluates to True

```
46 while dollar_bills < total:
47     day = day + 1
48     dollar_bills = dollar_bills * 2
49 
50     print "Number of days", day
51     print "Number of dollar bills", dollar_bills
52     print "Stack Height", dollar_bills * thickness
```

- Indentation used to denote blocks of code
- Indentation must be consistent (4 spaces, or Tab)

Conditionals (Exercise 4)

- If - else

```
58 if a < b:
59     print "Yes"
60 else:
61     print "No"
62
```

- If – elif – else

```
58 if a == '4':
59     print "a = 4"
60 elif a == '3':
61     print "a = 3"
62 elif a == '2':
63     print "a = 2"
64 elif a == '1':
65     print "a = 1"
66 else:
67     print "Unknown"
```

- Booleans (and, or, not)

```
70 if b >= a and b <= c:
71     print "b is between a and c"
72 if not (b < a or b > c):
73     print "b is still between a and c"
```

- Relational Operators
  - <, >, <=, >=, ==, !=

Truth Values

- Evaluates as "True"
  - Non-zero numbers
  - Non-empty strings
  - Non-empty containers (lists, dicts, etc.)
- Evaluates as "False"
  - 0 (Zero)
  - Empty string or containers

Long Lines

- Sometime you will have long statements that would run off the screen that you'd like to break up
- Use the line continuation character (\)

```
125 if product == 'game' and style == 'RTS' \
126     and cost > 20 and age > 8 and rating > 4 \
127     and availability == True and number_of_reviews > 100:
128 
129     print 'Shut up and take my money!'
```

- Not necessary for code in (), [], and {}

```
125 if product == 'game' and style == 'RTS' \
126     and cost > 20 and age > 8 and rating > 4 \
127     and availability == True and number_of_reviews > 100:
128 
129     print 'Shut up and take my money!'
```

Datatypes

- Primitive datatypes in Python are Numbers and Strings (characters & text)
- Numbers
  - Booleans
  - Integers
  - Floating point
  - Complex (imaginary numbers)
- Strings
  - Written in quotes (“”, ‘’)
  - Standard escape characters work (‘/n’)
  - Triple quotes captures all literal text enclosed

Numbers - Booleans

- Two values: True, False

```
131 a = True
132 b = False
133 c = 3
```

- Evaluated as integers with value 1,0

```
>>> c+a
4
```

- Although doing that in practice would be odd, and difficult for another person to interpret

Numbers - Integers

- Signed values of arbitrary size

```
135 a = 37
136 b = -2993929937277166273771284818122412 31
137 c = 0x7fa8 # Hexadecimal
138 d = 0o253 # Octal
139 e = 0b10001111 # Binary
```

- Two internal representations
  - int (small, less than 32 bits in size)
  - long (large values, arbitrary size)
- Sometimes see 'L' shown at the end of large values

```
>>> b
-2993929937277166273771284818122412 31L
>>>
```

Numbers – Integer Operation

`+`	Add
`-`	Subtract
`*`	Multiply
`/`	Divide
`//`	Floor divide
`%`	Modulo
`**`	Power
`<<`	Bit shift left
`>>`	Bit shift right
`&`	Bit-wise AND
`|`	Bit-wise OR
`^`	Bit-wise XOR
`~`	Bit-wise NOT
`abs(x)`	Absolute value
`pow(x,y[,z])`	Power with optional modulo (x**y)%z
`divmod(x,y)`	Division with remainder

Numbers – Integer Division

- Classic division (/) truncates
  - >>> 5/4
  - 1
  - >>>
- Floor division (//) truncates (same)
  - >>> 5/4
  - 1
  - >>>
- Future division converts to float
  - from __future__ import division
  - 5/4
  - 1.25
- In Python 3, / always produces a float
- In Python 2, / of integers produces another integer
- If truncation is intended, use //

Numbers – Floating point

- Use a decimal or exponential notation
  - 8.23
  - 2e7
  - -1.1343e-10
- Represented as double precision using the native CPU representation (IEEE 754)
  - 17 digits of precision
  - Exponent from -308 to 308
- Same as the C double

Numbers – Floating point Operators

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo (remainder)
**	Power
pow(x,y [,z])	Power modulo (x**y)%z
abs(x)	Absolute value
divmod(x,y)	Division with remainder

- And many more in math, and other libraries

```
125 import math
126
127 a = math.sqrt(x)
128 b = math.log10(x)
129 c = math.acos(x)
130 d = math.sin(x)
131
```

Numbers - Conversion

- Number types can be easily converted
  - a = int(x) #converts x to int
  - b = float(x) #converts x to float

- Will also work on strings containing numbers only
  - a = ”3.14159”
  - int(a) = 3
  - float(a) = 3.141598999…
  - Or optional integer bases
    - int("0xff",16) = 255

Strings, str()

- Immutable (read only, any operation makes a new string)
- An ordered sequence of bytes (characters)
- Stores 8-bit data (ASCII)
- May contain binary data, control characters, etc.
- Strings are frequently used for both text and for raw-data of any kind
- C inspired escape codes
  - '\n' Line feed
  - '\r' Carriage return
  - '\t' Tab
  - '\xhh' Hexadecimal value
  - '\”' Literal quote
  - '\\' Backslash

String representation

- Strings work like arrays s[n]
  - a = ”Hello world”
  - a[0] = “H”
  - a[3] = ”l”
  - a[-1] = “d”

- Can be sliced
  - a[:5] = “Hello”
  - a[6:] = “world”
  - a[3:8] = ”lo wo”
  - a[-5:] = “world”

- … and concatenated
  - a = "Hello" + "World"
  - b = "Say " + a

String Operators and Methods

- Length len()
  - s = ‘string’
  - len(s) = 6
- Membership test in
  - ‘e’ in s False
  - ‘ring’ in s True
- Replication
  - s*5
  - ”stringstringstringstringstring"

String Operators and Methods

- Strings have "methods" that perform various operations with the string data.

```
s.endswith(suffix)      # Check if string ends with suffix
s.find(t)               # First occurrence of t in s
s.index(t)              # First occurrence of t in s
s.isalpha()             # Check if characters are alphabetic
s.isdigit()             # Check if characters are numeric
s.islower()             # Check if characters are lower-case
s.isupper()             # Check if characters are upper-case
s.join(slist)           # Joins lists using s as delimiter
s.lower()               # Convert to lower case
s.replace(old,new)      # Replace text
s.rfind(t)              # Search for t from end of string
s.rindex(t)             # Search for t from end of string
s.split([delim])        # Split string into list of substrings
s.startswith(prefix)    # Check if string starts with prefix
s.strip()               # Strip leading/trailing space
s.upper()               # convert to upper case
```

- dir() is your friend!

Lists

- Array of arbitrary values
  - names = [”Alex", ”Dan", ”Gilad"]
  - nums = [39, 38, 42, 65, 111]
- Can contain mixed data types
  - items = [”Hemingway", 37, 1.5]
- Adding new items (append, insert)
  - items.append("that") # Adds at end
  - items.insert(2,"this") # Inserts in middle
- Concatenation : s + t
  - s = [1,2,3]
  - t = ['a','b']
  - s + t → [1,2,3,'a','b']

Lists

- Lists are indexed by integers (starting at 0, like str)
  - names = [”Alex", ”Dan", ”Gilad"]
  - names[0] → ”Alex"
  - names[1] → ”Dan"
  - names[2] → ”Gilad
- Negative indices are from the end
  - names[-1] → ”Gilad"
- Changing one of the items
  - name[1] = ”Ichiro"

List Operations

- Length len()
  - s = [”Alex", ”Dan", ”Gilad”, ”Ichiro”]
  - len(s) = 4
- Membership test in
  - “Ichiro” in s True
  - “Elvis” in s False
- Replication
  - s = [1, 2, 3]
  - s*3 = [1, 2, 3, 1, 2, 3, 1, 2, 3]

List item removal

- Removing an item
  - s = [”Alex", ”Dan", ”Gilad”, ”Ichiro”]
  - s.remove(“Alex”)
- Deleting an item by index
  - del s[0]
- Removal results in items moving down to fill the space vacated (i.e., no "holes").

Iterating with Lists

- Iterating over the list contents
  - for name in names:
    - # use name
    - ...
- Similar to a 'foreach' statement from other programming languages
- Use range to iterate over an index
  - for idx in range(10):
    - print idx #prints 0-9 values

List Sorting

- Lists can be sorted "in-place" (sort method)
  - s = [10,1,7,3]
  - s.sort() # s = [1,3,7,10]
- Sorting in reverse order
  - s.sort(reverse=True) # s = [10,7,3,1]
- Sorting works with any ordered type
  - s = ["foo","bar","spam"]
  - s.sort() # s = ["bar","foo","spam"]

Math with Lists

- Caution: pure Python Lists weren't designed for "math”
  - >>> nums = [1,2,3,4,5]
  - >>> nums * 2
  - [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
  - >>> nums + [10,11,12,13,14]
  - [1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
  - >>>
- They don't represent vectors/matrices
- Not the same as in MATLAB, Octave, IDL, etc.
- There are some add-ons for this (e.g., NumPy)

File IO

- Opening a file
  - f = open("foo.txt","r") # Open for reading
  - g = open("bar.txt","w") # Open for writing
- To read data
  - data = f.read([maxbytes])
- To write text to a file
  - g.write("some text")
- To close when you're done
  - f.close()

Reading File

- Reading an entire file all at once as a string
  - f = open(filename,"r")
  - data = f.read()
  - f.close()

- Reading an entire text-file line-by-line
  - f = open(filename,"r")
  - for line in f:
  - # Process the line
  - f.close()

Reading File Data

- End-of-file indicated by an empty string
  - data = f.read(nbytes)
  - if data == '':
  - # No data read. EOF
  - ...

- Example: Reading a file in fixed-size chunks
  - f = open(filename,"r")
  - while True:
  - chunk = f.read(chunksize)
  - if chunk == '':
  - break
  - # Process the chunk
  - ...
  - f.close()

Writing File Data

- Writing string data
  - f = open("outfile","w")
  - f.write("Hello World\n")
  - ...
  - f.close()

- Redirecting the print statement
  - f = open("outfile","w")
  - print >>f, "Hello World"
  - ...
  - f.close()

Easy File Management

- Files should be properly closed when done
  - f = open(filename,"r") # Use the file f
  - ...
  - f.close()

- In modern Python (2.6 or newer), use "with"
- This automatically closes the file when control leaves the indented code block
  - with open(filename,"r") as f:
  - # Use the file f
  - ...
  - statements

Simple Functions

- Use functions for code you want to reuse
  - def sumcount(n):
  - '''Returns the sum of the first n integers'''
  - total = (n + 1) * n / 2
  - return total

- Calling a function
  - a = sumcount(100)

- A function is just a series of statements that perform some task and return a result

Libraries

- Python comes with a large standard library
- Library modules accessed using import
  - import math
  - x = math.sqrt(10)
  - import urllib
  - u = urllib.urlopen("http://www.python.org/index.html")
  - data = u.read()

dir() Function

- dir(module) returns all names in a library
  - Useful for exploring library contents

```
>>> import sys
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__pac
kage__', '__stderr__', '__stdin__', '__stdout__', '_clear_type_cach
e', '_current_frames', '_getframe', '_mercurial', 'api_version', 'a
rgv', 'builtin_module_names', 'byteorder', 'call_tracing', 'callsta
ts', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear'
, 'exc_info', 'exc_traceback', 'exc_type', 'exc_value', 'excepthook
', 'exec_prefix', 'executable', 'exit', 'exitfunc', 'flags', 'float
_info', 'float_repr_style', 'getcheckinterval', 'getdefaultencoding
', 'getdlopenflags', 'getfilesystemencoding', 'getprofile', 'getrec
ursionlimit', 'getrefcount', 'getsizeof', 'gettrace', 'hexversion',
'last_traceback', 'last_type', 'last_value', 'long_info', 'maxint',
'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hook
s', 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'py3
kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile', 'set
recursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversi
on', 'version', 'version_info', 'warnoptions']
```

Back to the Palindrome problem

- A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is $9009 = 91 \times 99$.
- Find the largest palindrome made from the product of two 3-digit numbers.

Python 102

- In this section, we look at how Python programmers represent and work with data
- Common programming idioms
- How to (not) shoot yourself in the foot
- Few last basics to cover

Data Structures

- Real world applications have more complex data

- Typically objects with multiple parts
  - Descriptors (string)
  - Data (float)
  - Metadata (strings, integers)

Tuples

- Immutable!

- A collection of values grouped together
  - tup = (’Mayfield Av', 35, 37830.1)

- Sometimes the () are omitted in syntax
  - tup = ’Mayfield Av', 35, 37830.1

- Special cases (0-tuple, 1-tuple)
  - emt = () # An empty tuple
  - e = (Hemingway',) # A 1-item tuple

Tuple Comments

- Are they just a read-only list?
- Tuples are most often used for a *single record* consisting of multiple parts (think a row record in a database)
- Whereas Lists are usually a collection of distinct terms (typically all of the same type)
- Tuples are focused on packing and unpacking data, not storing distinct items in a list
- The tuple is then easy to pass around to other parts of a program as a single object

Tuple Comments

- To use the tuple elsewhere, you typically unpack its parts into variables
- Unpacking values from a tuple
  - address = (4095, 'Powder Mill Rd', 'Beltsville', 'MD', 2070.5)
  - num, road, city, state, zipcode = adress
- Note: Again, the () syntax is sometimes omitted

Dictionaries

- A hash table or associative array
- A collection of values indexed by "keys”
- The keys serve as field names
- Example:
  - dic = {'number':4095, 'road':'Powder Mill Rd', 'city':'Beltsville', 'state':'MD', 'zip':20705.1111}
- Dictionaries are useful
  - When there are many different values
  - The values will be modified/manipulated
- You also get better code clarity
  - dic[‘number’] vs dic[0]

Dictionaries

- The order of items (keys) in a dictionary is arbitrary
- Can not have duplicate keys ☹
  - But! Value pairs can be lists
- If you want them in a particular order, you have to create that order

Containers

- Programs often have to work many objects
  - Numerical scientific data
  - Mixed hyperspectral marked up data
- Three choices:
  - Lists (ordered data)
  - Dictionaries (unordered data)
  - Sets (unordered collection)

Lists as a container

- Use a list when the order of data matters
- Lists can hold any kind of object
- Example: A list of tuples
  - censor_data = [('Beltville', 148709, 67),
    ('Muscatine', 34087, 187),
  - ('Oak Ridge', 62492, 246)]

Dictionaries as a container

- Dictionaries are useful if you want fast random lookups (by key name)
- Example: A dictionary of measurement runs
  - measurements = {'HPLC':97.123
  - 'GC':98.4
  - 'LC':97.1}
- Easy to test existence
  - if *key* in measurements:
    - #Do stuff
    - Else:
    - #Do other stuff

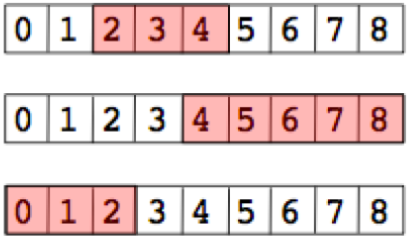
Sets

- Sets
  - techniques = set(['SEM', 'TEM', 'STM', 'STEM'])
- Holds collection of unordered items
- No duplicates, BUT supports common set ops
  - techniques | techniques2 #union
  - techniques & techniques2 #intersection
  - techniques - techniques2 #difference
- Useful for membership tests

Slicing preview (more in NumPy)

- Slicing operator s[start:end]

```
a = [0,1,2,3,4,5,6,7,8]

a[2:5]  ⟶  [2,3,4]
a[-5:]  ⟶  [4,5,6,7,8]
a[:3]   ⟶  [0,1,2]
```

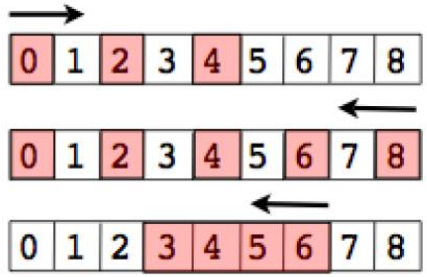


- Indices must be integers
- Slides do not include end value
- If indices are omitted, they default to the beginning or end of the list

Slicing with steps

- Extended slicing: s[start:end:step]

```
a = [0,1,2,3,4,5,6,7,8]
a[0:5:2] ⟶ [0,2,4]
a[::-2]  ⟶ [8,6,4,2,0]
a[6:2:-1]⟶ [6,5,4,3]
```



- **step** indicates stride and direction
- **end** index is not included in result

- *Go easy* for code clarity

A bit on iteration

- var = [0, 1, 2, 3, 4, 5]
- for i in var:
  - print I

- Each time through the loop, a new value is placed into an iteration variable
- Overwrites the previous value (if any)
- After the loop finishes, the variable has the value from the last iteration of the loop

range, xrange, and enumerate

- range([start,] end [,step])
  - x = range(100) # x = [0,1,...,99]
  - y = range(10,20) # y = [10,11,...,19]
  - z = range(10,50,2) # z = [10,12,...,48]
- range() creates a list of integers
- If you are only looping, use xrange() instead. It computes its values on demand instead of creating a list
- enumerate() provides a loop counter value
  - names = ["Elwood","Jake","Curtis"]
  - for i,name in enumerate(names):
  - # Loops with i = 0, name = 'Elwood'
  - # i = 1, name = 'Jake'
  - # i = 2, name = 'Curtis'

Break & Continue

- Break, exits out of the loop and continues to next statement

```
for name in namelist:
    if name == username:
        break
    ...
    ...
statements
```

- On the other hand, Continue will skip to next iteration

```
for line in lines:
    if line = '':          # Skip empty lines
        continue
    # More statements
    ...
```

Zip()

- Combines multiple sequences into tuples

```
505 names = ['alex', 'dan', 'gilad']
506 colors = ['green', 'blue', 'red']
507 for idx in range(len(names)):
508     print names[idx], colors[idx]
509
510 print zip(names, colors)
511
512 for name, color in zip(names, colors):
513     print name, color
```

- One use, looping over two sequences

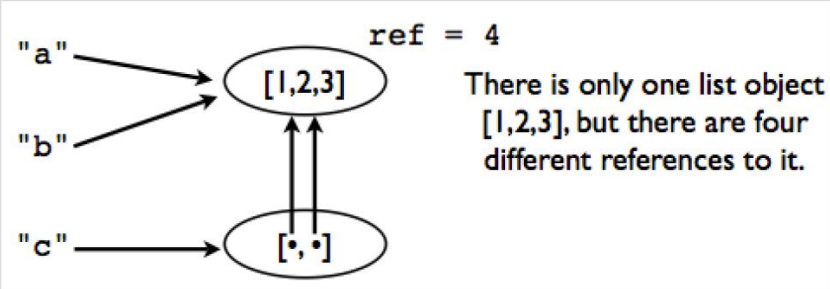
- Another use: making dictionaries

Zip()

- zip() always stops with shortest sequence
- You may combine as many sequences as needed
  - a = [1,2,3,4,5,6]
  - b = [’alex','dan']
  - x = zip(a,b) # x = [(1,’alex'),(2,’dan')]
  - a = [1, 2, 3, 4, 5, 6]
  - b = ['a', 'b', 'c']
  - c = [10, 20, 30]
  - x = zip(a,b,c) # x = [(1,'a',10),(2,'b',20),...]

Object details

- Many operations in Python are related to "assigning" or "storing" values
- A caution: assignment operations never make a copy of the value being assigned
- All assignments are merely reference copies
- Consider
  - a = [1, 2, 4]
  - b = a
  - c = [a, b]



Object details

- Now lets say a.append(1231)
- Check a, b & c…
  - Reassigning a value never overwrites the memory used by the previous value
  - Variables are names, not memory locations
  - This is one of the reasons why the primitive data types (int, float, string) are immutable
- Use the "is" operator to check if two values are exactly the same in memory
  - a is b
  - True

Identity and References

- Use the sys module to get a reference count
  - import sys
  - sys.getrefcount(a)
  - Note: The result is always one more than the actual reference count (an additional reference is created by the call to getrefcount())
- If you need to make a ‘deep’ copy use the copy module
  - >>> a = [2,3,[100,101],4]
  - >>> import copy
  - >>> b = copy.deepcopy(a)
  - >>> a[2].append(102)
  - >>> b[2]
  - [100,101]

A (brief) word on Modules & Libraries

- Modules are objects when you import a module, the module acts as an object you can manipulate
- Assign to variables, place in lists, rename, etc.
  - import math
  - m = math # Assign to a variable
  - x = m.sqrt(2) # Access through the variable
- You can even store new things in it
  - math.twopi = 2*math.pi

A (brief) word on Modules & Libraries

- from <MODULE> import <FUNCTION> lifts selected symbols out of a module and puts them into local scope
- Allows parts of a module to be used without having to type the module prefix
- If library functions are used frequently, this makes them run faster (one less lookup)
- from <MODULE> import * takes all symbols from a module and places them into local scope
- Makes it very difficult to understand someone else's code if you need to locate the original definition of a library function

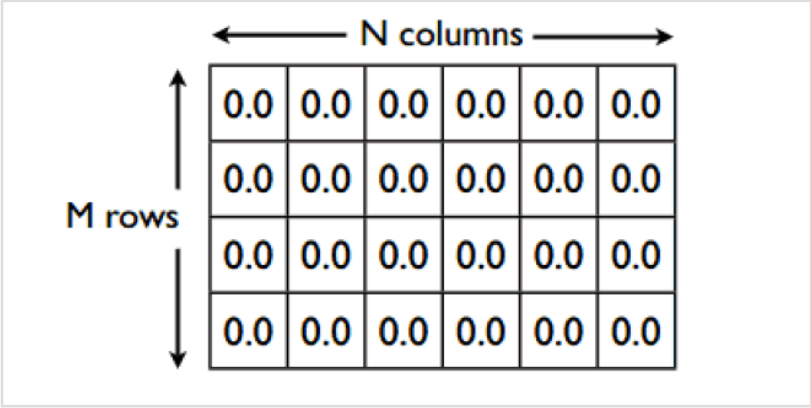
NumPy

NumPy

- NumPy is a library that provides multi-dimensional arrays, tables and matrices for Python and fast routines for array operations (C, ATLAS, MKL)
- NumPy is used for
  - Image and signal processing
  - Linear algebra
  - Data transformation and query
  - Time series analysis
  - Statistical analysis

NumPy Arrays

- A collection of values like arrays in C and Fortran
- Arrays have a shape (dimensions)



- You can create arrays from python lists
- Creating NumPy arrays from lists is not very efficient, native python data types are slow
- Often read and write directly from files instead
- Or use some other utility, zeros(), diag(), ones(), arange()

NumPy Array Creation

- Evenly spaced values on an interval arange([start,] end, [,step])

```
In [6]: np.arange(0, 5)
Out[6]: array([0, 1, 2, 3, 4])

In [7]: np.arange(0, 5, 0.5)
Out[7]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
4.5])

In [8]: np.arange(10, 0, -1)
Out[8]: array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

- arange allows fractional and negative steps
- Values can be equidistant (linear scale)
  - Or non linear np.logspace(0, 3, 4)
- Note: end-points by default included (use num=N+1 for Nsegments)

NumPy Array Creation

- Constant diagonal value

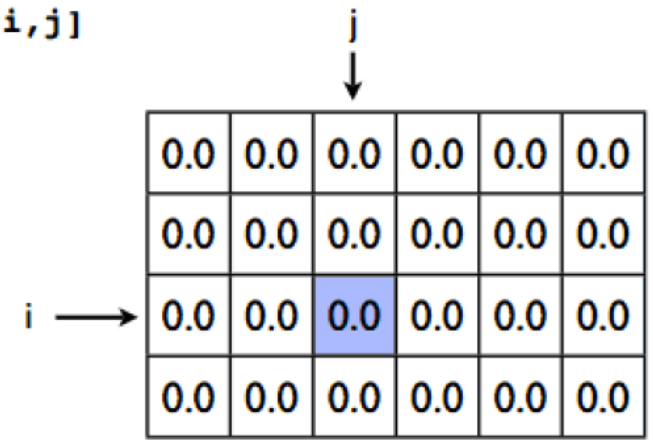
```
In [6]: np.eye(3)
Out[6]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

- Multiple diagonal values

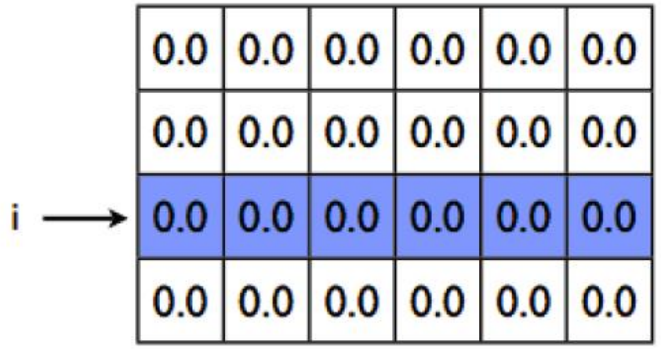
```
In [7]: np.diag([1,2,3,4])
Out[7]:
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
```

Array Items

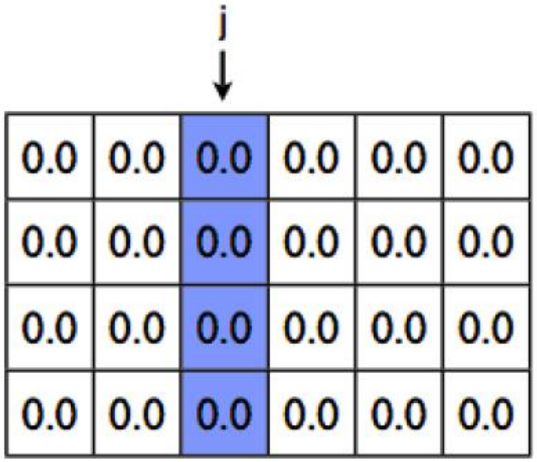
a[i,j]



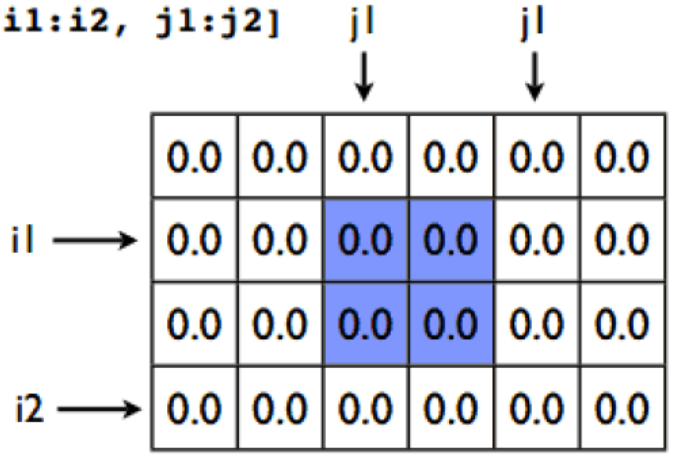
a[i,:]



a[:,j]



a[i1:i2, j1:j2]



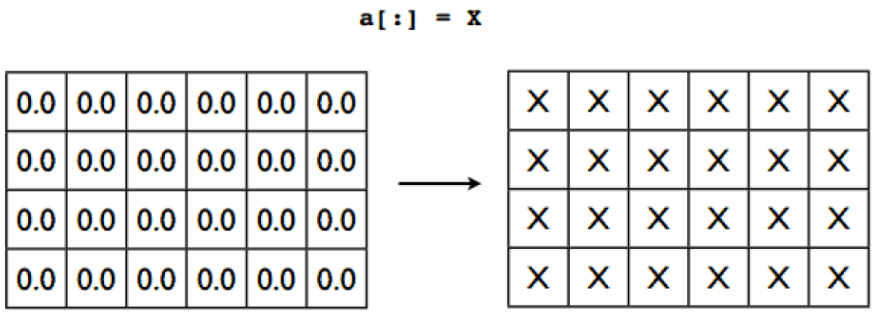
Array Slicing

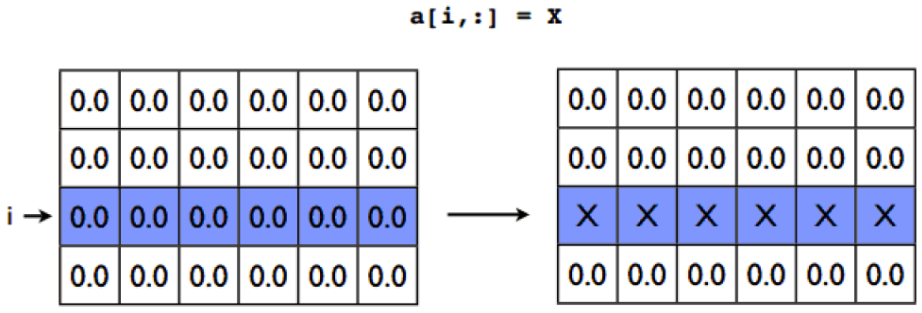
- **SLICES ARE NEVER COPIES**

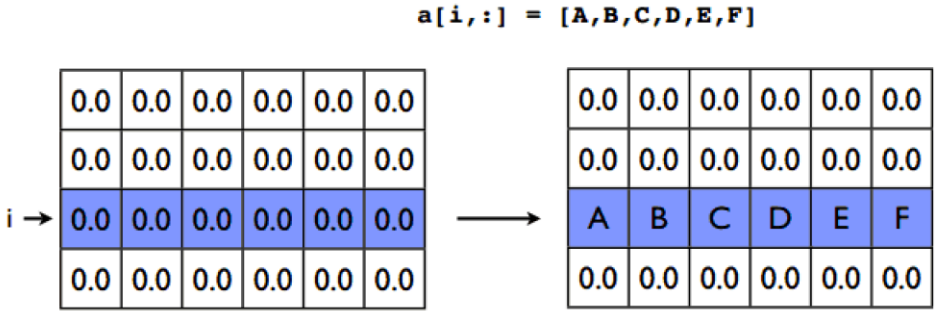
```
In [8]: a = np.arange(0, 5, 0.5)

In [9]: a
Out[9]: array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,
4.5])

In [10]: a[::2]
Out[10]: array([ 0.,  1.,  2.,  3.,  4.])

In [11]: a[::2][0] = 999

In [12]: a
Out[12]: array([ 999. ,  0.5,  1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4.
,  4.5])
```

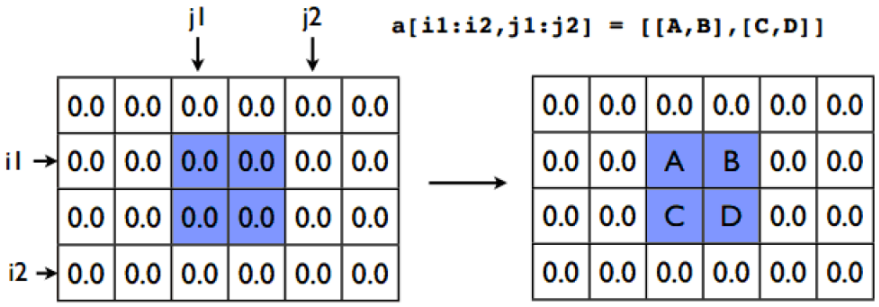
- Having shared data is different from list slicing, IT saves memory and makes operations more efficient

Array Assignment









Array Slicing Patterns

- Shifting values of an array: a[1:] = a[:-1]

```
In [22]: a
Out[22]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [23]: a[1:] = a[:-1]

In [24]: a
Out[24]: array([0, 0, 1, 2, 3, 4, 5, 6, 7, 8])
```

- Reverse an array: a[::-1]

```
In [32]: a
Out[32]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [33]: a[::-1]
Out[33]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

Numpy array math

- Operations with scalars apply to all elements (unlike Matlab)
- Operations on other arrays are element-wise (think /.)
- These operations create new arrays for the result (whew!)

Let's get vectorized

- Conditional operations make boolean arrays

```
In [26]: a
Out[26]: array([0, 1, 2, 3, 4, 5])

In [27]: a > 2
Out[27]: array([False, False, False, True,  True,  True], dtype=bool)
```

- np.where selects from an array

```
In [45]: a
Out[45]: array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])

In [46]: np.where(a > 15)
Out[46]: (array([6, 7, 8, 9]),)

In [47]: np.where(a > 15, a, False)
Out[47]: array([ 0,  0,  0,  0,  0,  0, 16, 17, 18, 19])
```

Array methods

- Predicate
  - a.any(), a.all()
- Reduce
  - a.mean(), a.argmin(), a.argmax(), a.trace(), a.cumsum(), a.cumprod()
- Manipulate
  - a.argsort(), a.transpose(), a.reshape(...), a.ravel(), a.fill(...), a.clip(...)
- Complex numbers
  - a.real, a.imag, a.conj()

Array reshaping

- The size of an array is the product of the shape values: anarray with a shape of (3,4,3) has 3x4x3=36 elements
- reshape will reshape an array to any similar size
- Takes tuples as arguments: a.reshape((3,1,1,1))
  - Use -1 for a wildcard dimension
- gets filled in based on the shape of the array

```
In [140]: a = np.arange(30)
In [141]: b = a.reshape((3,-1,2))
In [142]: b.shape
Out[142]: (3, 5, 2) #because 30 / (3 * 2) = 5!!
```

- np.squeeze removes singular dimensions

Built-in NumPy universal functions

- comparison: <, <=, ==, !=, >=, >
- arithmetic: +, -, *, /, reciprocal, square
- exponential: exp, expm1, exp2, log, log10, log1p, log2,
- power, sqrt
- trig: sin, cos, tan, acsin, arccos, atctan, sinh, cosh,
- tanh, acsinh, arccosh, atctanh
- bitwise: &, |, ~, ^, leftshift, rightshift
- logical operations: and, logical_xor, not, or
- predicates: isfinite, isinf, isnan, signbit
- other: abs, ceil, floor, mod, modf, round, sinc, sign, trunc

NumPy other functions

- numpy.fft — Fast Fourier transforms
- numpy.polynomial — Orthogonal polynomials, spline fitting
- numpy.linalg — Linear algebra
  - cholesky, det, eig, eigvals, inv, lstsq, norm, qr, svd
- numpy.math — C standard library math functions
- numpy.random — Random number generation
  - beta, gamma, geometric, hypergeometric, lognormal,
- normal, poisson, uniform, weibull

SciPy

SciPy

- Complement to NumPy
- More domain oriented in its content
- Some code is developed in the scikit series as well
  - scikit-image
  - scikit-learn
  - scikit-statsmodels

Statistics

- Scipy provides a uniform interface for continuous and discrete probability distributions

- Useful for statistical tools and probabilistic descriptions of
  - random processes
  - Working with random variables
  - Constructing distributions
  - Creating samples from distributions
  - Fitting parameters of a distribution to data
  - Non-parametric density estimation