DM561

Linear Algebra with Applications

Intoduction to Python - Part 2

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

[*Based on booklet Python Essentials*]

Outline

Outline

Exceptions

An exception formally indicates an error and terminates the program early.

```
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined

>>> [1, 2, 3].fly()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'list' object has no attribute 'fly'
```

Built-in Exceptions

Exception	Indication
`AttributeError`	An attribute reference or assignment failed.
`ImportError`	An `import` statement failed.
`IndexError`	A sequence subscript was out of range.
`NameError`	A local or global name was not found.
`TypeError`	An operation or function was applied to an object of inappropriate type.
`ValueError`	An operation or function received an argument that had the right type but an inappropriate value.
`ZeroDivisionError`	The second argument of a division or modulo operation was zero.

See `https://docs.python.org/3/library/exceptions.html` for the complete list of built-in exception

Raising Exceptions

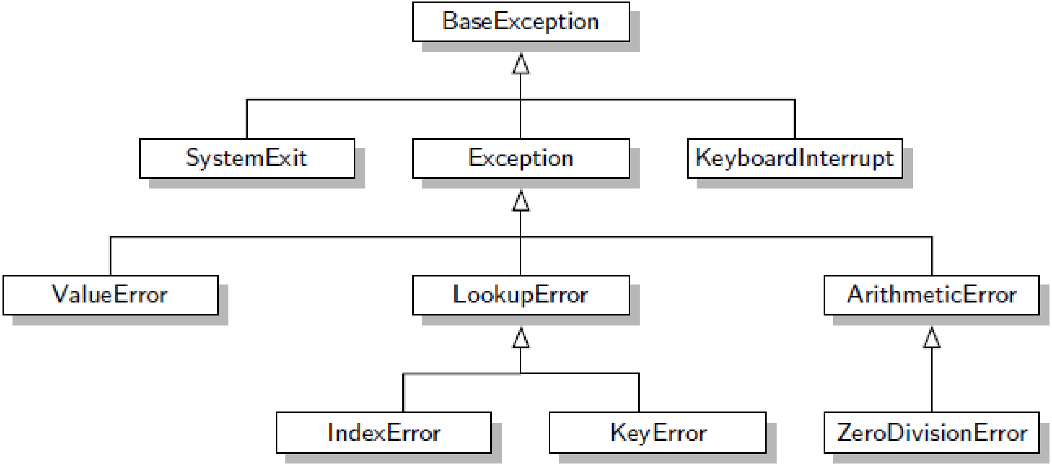
```
>>> if 7 is not 7.0:                    # Raise an exception with an error message.
...     raise Exception("ints and floats are different!")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: ints and floats are different!

>>> for x in range(10):
...     if x > 5:                       # Raise a specific kind of exception.
...         raise ValueError("'x' should not exceed 5.")
...     print(x, end=' ')
...
0 1 2 3 4 5
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
ValueError: 'x' should not exceed 5.
```

Handling Exceptions

To prevent an exception from halting the program, it must be handled by placing the problematic lines of code in a `try` block.

```
>>> try:
...     print("Entering try block...", end='')
...     house_on_fire = False
... except ValueError as e: # Skipped because there was no exception.
...     print("caught a ValueError.")
...     house_on_fire = True
... except TypeError as e:  # Also skipped (if just ``except:'' then always caught)
...     print("caught a TypeError.")
...     house_on_fire = True
... else:
...     print("no exceptions raised.")
... finally:  # always executed, even if a return statement or an uncaught ↪
    ↪exception occurs
...     print("The house is on fire:", house_on_fire)
...
Entering try block...no exceptions raised.
The house is on fire: False
```

Exeception Hierarchy



Working with Exception Objects

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Custom Exception Classes

```
>>> class TooHardError(Exception):
...     pass
...
>>> raise TooHardError("This lab is impossible!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.TooHardError: This lab is impossible!
```

Outline

File Reading

```
>>> myfile = open("hello_world.txt", 'r')    # Open a file for reading.
>>> print(myfile.read())                     # Print the contents of the file.
Hello,                                       # (it's a really small file.)
World!

>>> myfile.close()                           # Close the file connection.
```

the 'mode' can be `'r'`, `'w'`, `'x'`, `'a'`

A More Secure Way

```
>>> myfile = open("hello_world.txt", 'r')   # Open a file for reading.
>>> try:
...     contents = myfile.readlines()       # Read in the content by line.
... finally:
...     myfile.close()                      # Explicitly close the file.

# Equivalently, use a 'with' statement to take care of errors.
>>> with open("hello_world.txt", 'r') as myfile:
...    contents = myfile.readlines()
...                                         # The file is closed automatically.
```

Reading and Writing

Attribute	Description
`closed`	`True` if the object is closed.
`mode`	The access mode used to open the file object.
`name`	The name of the file.

Method	Description
`close()`	Close the connection to the file.
`read()`	Read a given number of bytes; with no input, read the entire file.
`readline()`	Read a line of the file, including the newline character at the end.
`readlines()`	Call `readline()` repeatedly and return a list of the resulting lines.
`seek()`	Move the cursor to a new position.
`tell()`	Report the current position of the cursor.
`write()`	Write a single string to the file (spaces are **not** added).
`writelines()`	Write a list of strings to the file (newline characters are **not** added).

Writing

```
>>> with open("out.txt", 'w') as outfile:    # Open 'out.txt' for writing.
...     for i in range(10):
...         outfile.write(str(i**2)+' ')     # Write some strings (and spaces).
...
>>> outfile.closed                           # The file is closed automatically.
True
```

String Methods

Method	Returns
`count()`	The number of times a given substring occurs within the string.
`find()`	The lowest index where a given substring is found.
`isalpha()`	True if all characters in the string are alphabetic (a, b, c, ...).
`isdigit()`	True if all characters in the string are digits (0, 1, 2, ...).
`isspace()`	True if all characters in the string are whitespace (`" "`, `'\t'`, `'\n'`).
`join()`	The concatenation of the strings in a given iterable with a specified separator between entries.
`lower()`	A copy of the string converted to lowercase.
`upper()`	A copy of the string converted to uppercase.
`replace()`	A copy of the string with occurrences of a given substring replaced by a different specified substring.
`split()`	A list of segments of the string, using a given character or string as a delimiter.
`strip()`	A copy of the string with leading and trailing whitespace removed.

String Methods

```python
# str.join() puts the string between the entries of a list.
>>> words = ["state", "of", "the", "art"]
>>> "-".join(words)
'state-of-the-art'

# str.split() creates a list out of a string, given a delimiter.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split('\n')
['One fish', 'Two fish', 'Red fish', 'Blue fish', '']

# If no delimiter is provided, the string is split by whitespace characters.
>>> "One fish\nTwo fish\nRed fish\nBlue fish\n".split()
['One', 'fish', 'Two', 'fish', 'Red', 'fish', 'Blue', 'fish']
```

Format

```
# Join the data using string concatenation.
>>> day, month, year = 10, "June", 2017
>>> print("Is today", day, str(month) + ',', str(year) + "?")
Is today 10 June, 2017?

# Join the data using str.format().
>>> print("Is today {} {}, {}?".format(day, month, year))
Is today 10 June, 2017?
```

```
>>> iters = int(1e7)
>>> chunk = iters // 20
>>> for i in range(iters):
...     print("\r[{:<20}] i = {}".format('='*((i//chunk)+1), i), end='', flush=True)
```

```
# From version 3.6
>>> print(f'Is today {day} {month}, {year}? The vlaue of pi is: {math.pi:.3f}')
Is today 10 June, 2017? The value of pi is: 3.142.
```

JSON To dict

Reading a JSON (JavaScript Object Notation) file, content example:

```
{
  "first_name": "Søren",
  "last_name": "Sørensen",
  "age": 25,
  "address": {
    "street_address": "Rodvej 45",
    "city": "Odense",
    "postal_code": "5000"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "+45 7575757"
    },
    {
      "type": "mobile",
      "number": "+45 99999999"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

example.json

```python
import json

with open('example.json') as json_file:
    data = json.load(json_file)

print(json.dumps(data,indent=4))

with open('example2.json', 'w') as outfile:
    json.dump(data, outfile)
```

example.py

Add the `ensure_ascii=False` argument to the json.dump in order to print corectly the 'ø' character.

Unicode

Python's string type uses the Unicode Standard for representing characters

code point | glyph | name

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

code point values integer in the range `0` to `0x10FFFF` (about 1.1 million values)

notation `U+265E` $\leadsto$ `0x265e` (9822 in decimal).

Encoding

Unicode code points are encoded in binary format, below shown in exadecimal system:

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

(little endian order)

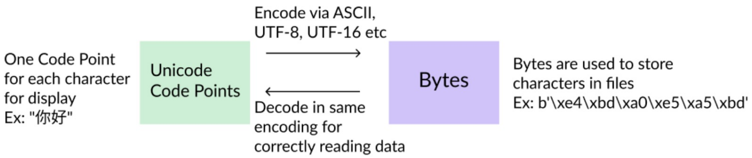
The standard encoding is UTF-8 that uses a variable number of bytes, 1, 2, 3 or 4.

Displaying the correct glyph from unicode is generally the job of a GUI toolkit or a terminal's font renderer.

Python and Unicode

Encode: from unicode code points to Bytes
Decode: from Bytes to unicode code points

```
>>> "\u0394" # or \U00000394
Δ # if GUI cannot visualize it then \u0394
>>> "\N{GREEK CAPITAL LETTER DELTA}"
Δ
>>> "\N{GREEK CAPITAL LETTER DELTA}".encode("utf-8") # str.encode()
b'\xce\x94'
>>> b'\xce\x94'.decode("utf-8")  # bytes.decode()
Δ
```



Since Python 3.0, the default encoding for Python source code is UTF-8, so you can simply include a Unicode character in a string literal and in identifiers:

```
>>> word = "øvelse" # u"øvelse" is obsolete
```

(`str` internal representation for strings is unicode code points)

```
>>> øv = 3
... øv
```

`ord` integer representing the Unicode code point of that character. `chr` the inverse.

```
>>> s = "a\xac\u1234\u20ac\U00008000"
... #     ^^^^ two-digit hex escape
... #         ^^^^^^ four-digit Unicode escape
... #                     ^^^^^^^^^^ eight-digit Unicode escape
>>> [ord(c) for c in s]
[97, 172, 4660, 8364, 32768]  # the code point
```

```
>>> u = chr(40960) + 'abcd' + chr(1972)
>>> u.encode('utf-8')
b'\xea\x80\x80abcd\xde\xb4'
```

Outline

Numpy

Module implementing multi-dimensional arrays useful for applied and computational mathematics.

```
>>> import numpy as np

# Create a 1-D array by passing a list into NumPy's array() function.
>>> np.array([8, 4, 6, 0, 2])
array([8, 4, 6, 0, 2])
```

`ndarray` object. Each dimension is called an axis: For a 2-D array, the 0-axis indexes the rows and the 1-axis indexes the columns.

```
# Create a 2-D array by passing a list of lists into array().
>>> A = np.array( [ [1, 2, 3],[4, 5, 6] ] )
>>> print(A)
[[1 2 3]
 [4 5 6]]
```

```
# Access to elements:
>>> print(A[0, 1], A[1, 2])
2 6

# Elements of a 2D array are 1D arrays.
>>> A[0]
array([1, 2, 3])
```

Basic Array Operations

Operators + and * for built-in lists (and strings too) :

```
# Addition concatenates lists together.
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]

# Multiplication concatenates a list with itself a given number of times.
>>> [1, 2, 3] * 4
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Basic Array Operations

```
>>> x, y = np.array([1, 2, 3]), np.array([4, 5, 6])

# Addition or multiplication by a scalar acts on each element of the array.
>>> x + 10                          # Add 10 to each entry of x.
array([11, 12, 13])
>>> x * 4                           # Multiply each entry of x by 4.
array([ 4,  8, 12])

# Add two arrays together (component-wise).
>>> x + y
array([5, 7, 9])

# Multiply two arrays together (component-wise).
>>> x * y
array([ 4, 10, 18])
```

Array Attributes

An `ndarray` object has several attributes, some of which are listed below.

Attribute	Description
`dtype`	The type of the elements in the array.
`ndim`	The number of axes (dimensions) of the array.
`shape`	A tuple of integers indicating the size in each dimension.
`size`	The total number of elements in the array.

Data Types

All elements of a NumPy array must have the same data type!!

Data type	Description
`bool_`	Boolean
`int8`	8-bit integer
`int16`	16-bit integer
`int32`	32-bit integer
`int64`	64-bit integer
`uint8`	Unsigned 8-bit integer
`uint16`	Unsigned 16-bit integer
`uint32`	Unsigned 32-bit integer
`uint64`	Unsigned 64-bit integer
`float16`	Half-precision float
`float32`	Single-precision float
`float64`	Double-precision float (default type for most computations)
`complex64`	Complex number represented by two single-precision floats
`complex128`	Complex number represented by two double-precision floats

Change Data Types

To change an existing array's type, use the array's `astype()` method.

```
# A list of integers becomes an array of integers.
>>> x = np.array([0, 1, 2, 3, 4])
>>> print(x)
[0 1 2 3 4]
>>> x.dtype
dtype('int64')

# Change the data type to one of NumPy's float types.
>>> x = x.astype(np.float64)
>>> print(x)
[ 0.  1.  2.  3.  4.]
>>> x.dtype
dtype('float64')
```

Array Creation Routines

Function	Returns
`arange()`	Array of sequential integers (like `list(range())`).
`linspace()`	Array of given shape and type of evenly spaced numbers over a specified interval.
`empty()`	Array of given shape and type, without initializing entries.
`identity()`	The square identity array.
`eye()`	2-D array with ones on the diagonal and zeros elsewhere.
`ones()`	Array of given shape and type, filled with ones.
`ones_like()`	Array of ones with the same shape and type as a given array.
`zeros()`	Array of given shape and type, filled with zeros.
`zeros_like()`	Array of zeros with the same shape and type as a given array.
`full()`	Array of given shape and type, filled with a specified value.
`full_like()`	Full array with the same shape and type as a given array.

Each of these functions accepts the keyword argument `dtype` to specify the data type. Common types include `np.bool_`, `np.int64`, `np.float64`, and `np.complex128`.

Array Creation Routines

```
>>> np.arange(3,7,2) # np.arange(start, stop, step)
array([3, 5])

>>> np.linspace(2.0, 3.0, num=5) # np.linspace(start, stop, num)
array([2.  , 2.25, 2.5 , 2.75, 3.  ])
```

Array Creation Routines

```
# A 1-D array of 5 zeros.
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])

# A 2x5 matrix (2-D array) of integer ones.
>>> np.ones((2,5), dtype=np.int)    # The shape is specified as a tuple.
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])

# The 2x2 identity matrix.
>>> I = np.eye(2)
>>> print(I)
[[ 1.  0.]
 [ 0.  1.]]

# Array of 3s the same size as 'I'.
>>> np.full_like(I, 3)              # Equivalent to np.full(I.shape, 3).
array([[ 3.,  3.],
       [ 3.,  3.]])
```

Array Creation Routines

Function	Description
`diag()`	Extract a diagonal or construct a diagonal array.
`tril()`	Get the lower-triangular portion of an array by replacing entries above the diagonal with zeros.
`triu()`	Get the upper-triangular portion of an array by replacing entries below the diagonal with zeros.

```
>>> A = np.array([[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 9]])

# Get only the upper triangular ↪
    ↪entries of 'A'.
>>> np.triu(A)
array([[1, 2, 3],
       [0, 5, 6],
       [0, 0, 9]])
```

```
# Get the diagonal entries of 'A' as a↪
    ↪ 1-D array.
>>> np.diag(A)
array([1, 5, 9])

# diag() can also create a diagonal ↪
    ↪matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[  1,   0,   0],
       [  0,  11,   0],
       [  0,   0, 111]])
```

Random Sampling

Similar to standard library's 'Random' module but more efficient

Function	Description
`choice()`	Take random samples from a 1-D array.
`random()`	Uniformly distributed floats over [0, 1).
`randint()`	Random integers over a half-open interval.
`random_integers()`	Random integers over a closed interval.
`randn()`	Sample from the standard normal distribution.
`permutation()`	Randomly permute a sequence / generate a random sequence.
Function	**Distribution**
`beta()`	Beta distribution over [0, 1].
`binomial()`	Binomial distribution.
`exponential()`	Exponential distribution.
`gamma()`	Gamma distribution.
`geometric()`	Geometric distribution.
`multinomial()`	Multivariate generalization of the binomial distribution.
`multivariate_normal()`	Multivariate generalization of the normal distribution.
`normal()`	Normal / Gaussian distribution.
`poisson()`	Poisson distribution.
`uniform()`	Uniform distribution.

Random Sampling

```
# 5 uniformly distributed values in the interval [0, 1).
>>> np.random.random(5)
array([ 0.21845499,  0.73352537,  0.28064456,  0.66878454,  0.44138609])

# A 2x5 matrix (2-D array) of integers in the interval [10, 20).
>>> np.random.randint(10, 20, (2,5))
array([[17, 12, 13, 13, 18],
       [16, 10, 12, 18, 12]])
```

Outline

Array Slicing

$$\texttt{A[0]} = \begin{bmatrix} \boxed{\times \quad \times \quad \times \quad \times \quad \times} \\ \times \quad \times \quad \times \quad \times \quad \times \\ \times \quad \times \quad \times \quad \times \quad \times \\ \times \quad \times \quad \times \quad \times \quad \times \end{bmatrix}$$

$$\texttt{A[2,1]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \boxed{\times} & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Array Slicing

$$\texttt{A[1]} = \texttt{A[1,:]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[:,2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[1:,:2]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

$$\texttt{A[1:-1,1:-1]} = \begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}$$

Array Slicing

```
>>> x = np.arange(10); x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> x[3]                                    # The element at index 3.
3
>>> x[:3]                                   # Everything up to index 3 (exclusive).
array([0, 1, 2])
>>> x[3:]                                   # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                                  # The elements from index 3 to 8.
array([3, 4, 5, 6, 7])
>>> A = np.array([[0,1,2,3,4],[5,6,7,8,9]])
>>> A
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> A[1, 2]                                 # The element at row 1, column 2.
7
>>> A[:, 2:]                                # All of the rows, from column 2 on.
array([[2, 3, 4],
       [7, 8, 9]])
```

Array Slicing

- Indexing and slicing operations return a view of the array.
- Changing a view of an array also changes the original array.
- That is, **arrays are mutable**.
- To create a copy of an array, use `np.copy()` or the array's `copy()` method.
- Changes to a copy of an array does not affect the original array
- Copying is less efficient than getting a view.

Fancy Indexing

Via either an array of indices or an array of boolean values (mask) to extract specific elements.

```
>>> x = np.arange(0, 50, 10)        # The integers from 0 to 50 by tens.
>>> x
array([ 0, 10, 20, 30, 40])

# An array of integers extracts the entries of 'x' at the given indices.
>>> index = np.array([3, 1, 4])     # Get the 3rd, 1st, and 4th elements.
>>> x[index]                        # Same as np.array([x[i] for i in index]).
array([30, 10, 40])

# A boolean array extracts the elements of 'x' at the same places as 'True'.
>>> mask = np.array([True, False, False, True, False])
>>> x[mask]                         # Get the 0th and 3rd entries.
array([ 0, 30])
```

Fancy Indexing

```python
>>> y = np.arange(10, 20, 2)        # Every other integers from 10 to 20.
>>> y
array([10, 12, 14, 16, 18])

# Extract the values of 'y' larger than 15.
>>> mask = y > 15                   # Same as np.array([i > 15 for i in y]).
>>> mask
array([False, False, False,  True,  True], dtype=bool)
>>> y[mask]                         # Same as y[y > 15]
array([16, 18])

# Change the values of 'y' that are larger than 15 to 100.
>>> y[mask] = 100
>>> print(y)
[10 12 14  100  100]
```

Shaping

```
>>> A = np.arange(12)                   # The integers from 0 to 12 (exclusive).
>>> print(A)
[ 0  1  2  3  4  5  6  7  8  9 10 11]

# 'A' has 12 entries, so it can be reshaped into a 3x4 matrix.
>>> A.reshape((3,4))                    # The new shape is specified as a tuple.
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Reshape 'A' into an array with 2 rows and the appropriate number of columns.
>>> A.reshape((2,-1))
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

Shaping

```
>>> A = np.arange(12).reshape((3,4))
>>> A
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

# Flatten 'A' into a one-dimensional array.
>>> np.ravel(A)                     # Equivalent to A.reshape(A.size)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# Transpose the matrix 'A'.
>>> A.T                             # Equivalent to np.transpose(A).
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

- Caution: reshape is just a view on the array, it does not copy the data. Beware: the following does shallow copy:

```
>>> N=A
>>> N[0,0] = 0 # now A[0,0] has become 0
>>> M=A.reshape(1,9)
>>> M[0,7]=10 # now A[2,1] has become 10
>>> A
array([[  0.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,  10.,   9.]])
```

- We can see the base object by:

```
>>> print(A.base)
None
>>> print(M.base) # the base object of M is the matrix A
[[  0.   2.   3.]
 [  4.   5.   6.]
 [  7.  10.   9.]]
```

- To implement a deep copy you must use `np.copy(A)`

Note on Vector Shape

- By default, all NumPy 1D arrays (including column slices) are automatically reshaped into "flat" (ie, row) 1D arrays.
- This contrasts with mathematical notation where vectors are represented vertically
- NumPy methods such as `dot()` are implemented to purposefully work well with 1D "row arrays".
- `np.transpose()` does not alter 1D arrays.
- Do not force a 1D vector to be a column vector unless necessary.
- To force a "column array" use `np.reshape()`, `np.vstack()`

Stacking

Function	Description
`concatenate()`	Join a sequence of arrays along an existing axis
`hstack()`	Stack arrays in sequence horizontally (column wise).
`vstack()`	Stack arrays in sequence vertically (row wise).
`column_stack()`	Stack 1-D arrays as columns into a 2-D array.
`row_stack()`	Stack 1-D arrays as rows into a 2-D array.

Stacking

```
>>> A = np.arange(6).reshape((2,3))
>>> B = np.zeros((4,3))

>>> np.vstack((A,B,A)) # same as np.concatenate([A,B,A],axis=0) and row_stack()
array([[ 0.,  1.,  2.],                # A
       [ 3.,  4.,  5.],
       [ 0.,  0.,  0.],                # B
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  1.,  2.],                # A
       [ 3.,  4.,  5.]])
```

Stacking

```
>>> A = A.T
>>> B = np.ones((3,4))

# hstack() # same as np.concatenate([A,B,A],axis=1) and column_stack()
>>> np.hstack((A,B,A))
array([[ 0.,  3.,  1.,  1.,  1.,  1.,  0.,  3.],
       [ 1.,  4.,  1.,  1.,  1.,  1.,  1.,  4.],
       [ 2.,  5.,  1.,  1.,  1.,  1.,  2.,  5.]])
```

You need to use `column_stack()` to stack arrays horizontally 1-D arrays.

```
>>> np.column_stack((A, np.zeros(3), np.ones(3), np.full(3, 2)))
array([[ 0.,  3.,  0.,  1.,  2.],
       [ 1.,  4.,  0.,  1.,  2.],
       [ 2.,  5.,  0.,  1.,  2.]])
```

http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html

Broadcasting

NumPy tries to automatically align arrays for component-wise operations whenever possible.

$$\texttt{A} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \qquad \texttt{x} = \begin{bmatrix} 10 & 20 & 30 \end{bmatrix}$$

$$\texttt{A + x} = \begin{matrix} \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} \\ + \\ [10 \quad 20 \quad 30] \end{matrix} = \begin{bmatrix} 11 & 22 & 33 \\ 11 & 22 & 33 \\ 11 & 22 & 33 \end{bmatrix}$$

$$\texttt{A + x.reshape((3,-1))} = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} + \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} = \begin{bmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{bmatrix}$$

http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Outline

Universal Functions

Universal function: operates on an entire array element-wise. More efficent than looping.

Function	Description
«abs()» or absolute()	Calculate the absolute value element-wise.
exp() / log()	Exponential (e^x) / natural log element-wise.
maximum() / minimum()	Element-wise maximum / minimum of two arrays.
sqrt()	The positive square-root, element-wise.
sin(), cos(), tan(), etc.	Element-wise trigonometric operations.

```
>>> x = np.arange(-2,3)
>>> print(x, np.abs(x))           # Like np.array([abs(i) for i in x]).
[-2 -1  0  1  2] [2 1 0 1 2]

>>> np.sin(x)                     # Like np.array([math.sin(i) for i in x]).
array([-0.90929743, -0.84147098,  0.        ,  0.84147098,  0.90929743])
```

Universal Functions

- Some functions from the module `math` do not work with array.
- It is possible to make them element-wise via `vectorize`.

```
>>> def f(x):
        return 0 if x<=5 else 1
# f(A) # error
>>> np.vectorize(f)(A)
array([[0, 0, 0],
       [0, 0, 1],
       [1, 1, 1]])
```

Operations along an Axis

Most array methods have an `axis` argument that allows an operation to be done along a given axis.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

$$\texttt{A.sum(axis=0)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 4 & 8 & 12 & 16 \end{bmatrix}$$

$$\texttt{A.sum(axis=1)} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 10 & 10 \end{bmatrix}$$

Summary

1. Exception Handling
2. File Input and Output
3. Numpy
   - Data Access
   - Numerical Computing with NumPy