1 Python and Numpy Introduction II

Lab Objective: *Python is a powerful, general-purpose programming language. Using Python's built-in and NumPy's additional functions, it becomes a powerful tool in dealing with large data sets. In this second lab, we will cover the speed advantages of NumPy and the basic tools you'll need for numerical computing.*

Functions II

As you may recall from the previous lab, Python functions can be used in a variety of ways depending on the parameters that it accepts. (Remember that the Python function accepts an argument for each of its parameters.)

There are times that we may not have an argument for all of the parameters; in this case, it is possible to specify *default values* for a function's parameters. In the following example, the function `pad()` has three parameters, and the value of `c` defaults to 0. If it is not specified in the function call, the variable `c` will contain the value 0 when the function is executed.

```
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
...
>>> pad(1, 2, 3)               # Specify each parameter.
1 2 3
>>> pad(1, 2)                  # Specify only non-default parameters.
1 2 0
```

Arguments are passed to functions based on position or name, and positional arguments must be defined before named arguments. For example, `a` and `b` must come before `c` in the function definition of `pad()`. Examine the following code blocks demonstrating how positional and named arguments are used to call a function.

```
# Try defining printer with a named argument before a positional argument.
>>> def pad(c=0, a, b):
...     print(a, b, c)
...
```

```
SyntaxError: non-default argument follows default argument
```

```
# Correctly define pad() with the named argument after positional arguments.
>>> def pad(a, b, c=0):
...     """Print the arguments, plus a zero if c is not specified."""
...     print(a, b, c)
...

# Call pad() with 3 positional arguments.
>>> pad(2, 4, 6)
2 4 6

# Call pad() with 3 named arguments. Note the change in order.
>>> pad(b=3, c=5, a=7)
7 3 5

# Call pad() with 2 named arguments, excluding c.
>>> pad(b=1, a=2)
2 1 0

# Call pad() with 1 positional argument and 2 named arguments.
>>> pad(1, c=2, b=3)
1 3 2
```

The keyword `lambda` is a shortcut for creating one-line functions. Just like normal functions, `lambda` functions can take in arguments and parameters. For example, the polynomials $f(x) = 6x^3 + 4x^2 - x + 3$ and $g(x, y, z) = x + y^2 - z^3$ can be defined as functions in one line each.

```
# Define the polynomials the usual way using 'def'.
>>> def f(x):
...     return 6*x**3 + 4*x**2 - x + 3
>>> def g(x, y, z):
...     return x + y**2 - z**3

# Equivalently, define the polynomials quickly using 'lambda'.
>>> f = lambda x: 6*x**3 + 4*x**2 - x + 3
>>> g = lambda x, y, z: x + y**2 - z**3
```

Note

Documentation is important in every programming language. Every function should have a *docstring*—a string literal in triple quotes just under the function declaration—that describes the purpose of the function, the expected inputs and return values, and any other notes that are important to the user. Short docstrings are acceptable for very simple functions, but more complicated functions require careful and detailed explanations.

```
>>> def add(x, y):
...     """Return the sum of the two inputs."""
...     return x + y

>>> def area(width, height):
...     """Return the area of the rectangle with the specified width
...     and height.
...     """
...     return width * height
...
>>> def arithmetic(a, b):
...     """Return the difference and the product of the two inputs."""
...     return a - b, a * b
```

Lambda functions cannot have custom docstrings, so the `lambda` keyword should only be used as a shortcut for very simple or intuitive functions that do not need additional labeling.

Problem 1. The built-in `print()` function has the useful keyword arguments `sep` and `end`. It accepts any number of positional arguments and prints them out with `sep` inserted between values (defaulting to a space), then prints `end` (defaulting to the *newline character* `'\n'`).

Write a function called `isolate()` that accepts five arguments. Print the first three separated by 5 spaces, then print the rest with a single space between each output. For example,

```
>>> isolate(1, 2, 3, 4, 5)
1     2     3 4 5
```

Data Types and Structures

Strings II

Parts of a string can be accessed using *slicing*, which is indicated by square brackets `[ ]`. Slicing syntax is `[start:stop:step]`. The parameters `start` and `stop` default to the beginning and end of the string, respectively. The parameter `step` defaults to 1.

```
>>> my_string = "Hello world!"
>>> my_string[4]                # Indexing begins at 0.
'o'
>>> my_string[-1]               # Negative indices count backward from the end.
'!'

# Slice from the 0th to the 5th character (not including the 5th character).
>>> my_string[:5]
```

```
'Hello'

# Slice from the 6th character to the end.
>>> my_string[6:]
'world!'

# Slice from the 3rd to the 8th character (not including the 8th character).
>>> my_string[3:8]
'lo wo'

# Get every other character in the string.
>>> my_string[::2]
'Hlowrd'
```

Problem 2. Write two new functions called `first_half()` and `backward()`.

1. `first_half()` should accept a parameter and return the first half of it, excluding the middle character if there is an odd number of characters.
   (Hint: the built-in function `len()` returns the length of the input.)
2. The `backward()` function should accept a parameter and reverse the order of its characters using slicing, then return the reversed string.
   (Hint: The `step` parameter used in slicing can be negative.)

Use IPython to quickly test your syntax for each function.

Lists II

Common list methods (functions) include `append()`, `insert()`, `remove()`, and `pop()`. Consult IPython for details on each of these methods using object introspection.

```
>>> my_list = [1, 2]                # Create a simple list of two integers.
>>> my_list.append(4)               # Append the integer 4 to the end.
>>> my_list.insert(2, 3)            # Insert 3 at location 2.
>>> my_list
[1, 2, 3, 4]
>>> my_list.remove(3)               # Remove 3 from the list.
>>> my_list.pop()                   # Remove (and return) the last entry.
4
>>> my_list
[1, 2]
```

Slicing is also very useful for replacing values in a list.

```
>>> my_list = [10, 20, 30, 40, 50]
>>> my_list[0] = -1
>>> my_list[3:] = [8, 9]
```

```
>>> print(my_list)
[-1, 20, 30, 8, 9]
```

The `in` operator quickly checks if a given value is in a list (or another iterable, including strings).

```
>>> my_list = [1, 2, 3, 4, 5]
>>> 2 in my_list
True
>>> 6 in my_list
False
>>> 'a' in "xylophone"              # 'in' also works on strings.
False
```

Multiple lists can be combined into one list through list concatenation.

```
>>> my_first_list = [1, 2, 3]
>>> my second_list = [4, 5, 6]
>>> my_full_list = my_first_list + my_second_list
>>> my_full_list
[1,2,3,4,5,6]
```

Tuples

A Python `tuple` is an ordered collection of elements, created by enclosing comma-separated values with parentheses `(` and `)`. Tuples are similar to lists, but they are much more rigid, have less built-in operations, and cannot be altered after creation. Therefore, lists are preferable for managing dynamic, ordered collections of objects.

When multiple objects are returned by a function, they are returned as a tuple. For example, recall that the `arithmetic()` function returns two values.

```
>>> def arithmetic(a, b):
...     return a - b, a * b         # Separate return values with commas.
...
>>> x, y = arithmetic(5,2)                      # Get each value individually,
>>> print(x, y)
3 10
>>> both = arithmetic(5,2)                      # or get them both as a tuple.
>>> print(both)
(3, 10)
```

Problem 3. Write a function called `list_ops()`. Define a list with the entries `"bear"`, `"ant"`, `"cat"`, and `"dog"`, in that order. Then perform the following operations on the list:

1. Append `"eagle"`.
2. Replace the entry at index 2 with `"fox"`.

3. Remove (or pop) the entry at index 1.
4. Sort the list in reverse alphabetical order.
5. Replace `"eagle"` with `"hawk"`.
   (Hint: the list's `index()` method may be helpful.)
6. Add the string `"hunter"` to the last entry in the list.

Return the resulting list.

Work out (on paper) what the result should be, then check that your function returns the correct list. Consider printing the list at each step to see the intermediate results.

Sets

A Python `set` is an unordered collection of distinct objects. Objects can be added to or removed from a set after its creation. Initialize a set with curly braces `{ }` and separate the values by commas, or use `set()` to create an empty set. Like mathematical sets, Python sets have operations such as union, intersection, difference, and symmetric difference.

```
# Initialize some sets. Note that repeats are not added.
>>> gym_members = {"Doe, John", "Doe, John", "Smith, Jane", "Brown, Bob"}
>>> print(gym_members)
{'Doe, John', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.add("Lytle, Josh")      # Add an object to the set.
>>> gym_members.discard("Doe, John")    # Delete an object from the set.
>>> print(gym_members)
{'Lytle, Josh', 'Brown, Bob', 'Smith, Jane'}

>>> gym_members.intersection({"Lytle, Josh", "Henriksen, Ian", "Webb, Jared"})
{'Lytle, Josh'}
>>> gym_members.difference({"Brown, Bob", "Sharp, Sarah"})
{'Lytle, Josh', 'Smith, Jane'}
```

Dictionaries

Like a set, a Python `dict` (dictionary) is an unordered data type. A dictionary stores key-value pairs, called *items*. The values of a dictionary are indexed by its keys. Dictionaries are initialized with curly braces, colons, and commas. Use `dict()` or `{}` to create an empty dictionary.

```
>>> my_dictionary = {"business": 4121, "math": 2061, "visual arts": 7321}
>>> print(my_dictionary["math"])
2061

# Add a value indexed by 'science' and delete the 'business' keypair.
>>> my_dictionary["science"] = 6284
>>> my_dictionary.pop("business")       # Use 'pop' or 'popitem' to remove.
4121
```

```
>>> print(my_dictionary)
{'math': 2061, 'visual arts': 7321, 'science': 6284}

# Display the keys and values.
>>> my_dictionary.keys()
dict_keys(['math', 'visual arts', 'science'])
>>> my_dictionary.values()
dict_values([2061, 7321, 6284])
```

As far as data access goes, lists are like dictionaries whose keys are the integers $0, 1, \ldots, n-1$, where n is the number of items in the list. The keys of a dictionary need not be integers, but they must be *immutable*, which means that they must be objects that cannot be modified after creation. We will discuss mutability more thoroughly in the Standard Library lab.

Type Casting

The names of each of Python's data types can be used as functions to cast a value as that type. This is particularly useful for converting between integers and floats.

```
# Cast numerical values as different kinds of numerical values.
>>> x = int(3.0)
>>> y = float(3)
>>> z = complex(3)
>>> print(x, y, z)
3 3.0 (3+0j)

# Cast a list as a set and vice versa.
>>> set([1, 2, 3, 4, 4])
{1, 2, 3, 4}
>>> list({'a', 'a', 'b', 'b', 'c'})
['a', 'c', 'b']

# Cast other objects as strings.
>>> str(['a', str(1), 'b', float(2)])
"['a', '1', 'b', 2.0]"
>>> str(list(set([complex(float(3))])))
'[(3+0j)]'
```

Comprehensions

A *list comprehension* uses for loop syntax between square brackets to create a list. This is a powerful, efficient way to build lists. The code is concise and runs quickly.

```
>>> [float(n) for n in range(5)]
[0.0, 1.0, 2.0, 3.0, 4.0]
```

List comprehensions can be thought of as "inverted loops", meaning that the body of the loop comes before the looping condition. The following loop and list comprehension produce the same list, but the list comprehension takes only about two-thirds the time to execute.

```
>>> loop_output = []
>>> for i in range(5):
...     loop_output.append(i**2)
...
>>> list_output = [i**2 for i in range(5)]
```

The syntax for a list comprehension can be used for more than just lists however. Now that we have learned about sets, dictionaries, and tuples, we can show you how to use similar methods for each data type.

```
#Recall the format for list comprehension
>>>[i**2 for i in range(-2,3)]
[4, 1, 0, 1, 4, 9]
#Following a similar format for sets
>>>{i**2 for i in range(-2,3)}
{0, 1, 4, 9}
#Dictionaries
>>>{i:i**2 for i in range(-2,3)}
{-2: 4, -1: 1, 0: 0, 1: 1, 2: 4}
```

Problem 4. The alternating harmonic series is defined as follows.

$$\sum_{n=1}^{\infty} \frac{(-1)^{(n+1)}}{n} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \ldots = \ln(2)$$

Write a function called `alt_harmonic()` that accepts an integer n. Use a list comprehension to quickly compute the sum of the first n terms of this series (be careful not to compute only $n-1$ terms). The sum of the first 500,000 terms of this series approximates $\ln(2)$ to five decimal places.
(Hint: consider using Python's built-in `sum()` function.)

Achtung!

This syntax does not work for tuples. If you use the same syntax it will create a python generator. This generator can then be used to to create a list, set, tuple, or dictionary.

```
#Trying to use tuple comprhension
>>>(i**2 for i in range(-2,3))
 <generator object <genexpr> at 0x00000286FC8F56D0>
#To use the generator, assign it to some variable
#The generator can only be cast once to another data type
```

```
>>> generator = (i**2 for i in range(-2,3))
>>>a = tuple(generator)
>>>a
(4, 1, 0, 1, 4, 9)
>>>b = list(a)
[]
```

Numpy II

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.

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

# 'A' is a 2-D array with 2 rows, 3 columns, and 6 entries.
>>> print(A.ndim, A.shape, A.size)
2 (2, 3) 6
```

Note that `ndim` is the number of entries in `shape`, and that the `size` of the array is the product of the entries of `shape`.

Array Creation Routines

In addition to casting other structures as arrays via `np.array()`, NumPy provides efficient ways to create certain commonly-used arrays.

Function	Returns
`arange()`	Array of sequential integers (like `list(range())`).
`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.
`fromiter()`	Array formed from a python generator, must specify the desired data type

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`.

```
# 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.]])
```

Unlike native Python data structures, **all elements of a NumPy array must be of the same data type**. To change an existing array's data 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)        # Equivalent to x = np.float64(x).
>>> print(x)
[ 0.  1.  2.  3.  4.]               # Floats are displayed with periods.
>>> x.dtype
dtype('float64')
```

The following functions are for dealing with the diagonal, upper, or lower portion of an array.

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 be used to create a diagonal matrix from a 1-D array.
>>> np.diag([1, 11, 111])
array([[  1,   0,   0],
       [  0,  11,   0],
       [  0,   0, 111]])
```

See `http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html` for the official documentation on NumPy's array creation routines.

Data Access

Array Slicing

Indexing for a 1-D NumPy array uses the slicing syntax `x[start:stop:step]`. If there is no colon, a single entry of that dimension is accessed. With a colon, a range of values is accessed. For multi-dimensional arrays, use a comma to separate slicing syntax for each axis.

```
# Make an array of the integers from 0 to 10 (exclusive).
>>> x = np.arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Access elements of the array with slicing syntax.
>>> x[3]                          # The element at index 3.
3
>>> x[:3]                         # Everything before index 3.
array([0, 1, 2])
>>> x[3:]                         # Everything from index 3 on.
array([3, 4, 5, 6, 7, 8, 9])
>>> x[3:8]                        # Everything between index 3 and index 8 (↩
    exclusive).
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]])
```

```
# Use a comma to separate the dimensions for multi-dimensional arrays.
>>> 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]])
```

> **Note**
>
> Indexing and slicing operations return a *view* of the array. Changing a view of an array also changes the original array. In other words, **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, but copying an array uses more time and memory than getting a view.

Fancy Indexing

So-called *fancy indexing* is a second way to access or change the elements of an array. Instead of using slicing syntax, provide either an array of indices or an array of boolean values (called a *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 is especially useful for extracting or changing the values of an array that meet some sort of criterion. Use comparison operators like `<` and `==` to create masks.

```
>>> 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]
```

While indexing and slicing always return a view, fancy indexing always returns a copy.

Problem 5. Write a function that accepts a single array as input. Make a copy of the array, then use fancy indexing to set all negative entries of the copy to 0. Return the copy.

Array Manipulation

Shaping

An array's `shape` attribute describes its dimensions. Use `np.reshape()` or the array's `reshape()` method to give an array a new shape. The total number of entries in the old array and the new array must be the same in order for the shaping to work correctly. Using a `-1` in the new shape tuple makes the specified dimension as long as necessary.

```
>>> 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]])
```

Use `np.ravel()` to flatten a multi-dimensional array into a 1-D array and `np.transpose()` or the `T` attribute to transpose a 2-D array in the matrix sense.

```
>>> 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]])
```

Note

By default, all NumPy arrays that can be represented by a single dimension, including column slices, are automatically reshaped into "flat" 1-D arrays. For example, by default an array will have 10 elements instead of 10 arrays with one element each. Though we usually represent vectors vertically in mathematical notation, NumPy methods such as `dot()` are implemented to purposefully work well with 1-D "row arrays".

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

# Slicing out a column of A still produces a "flat" 1-D array.
>>> x = A[:,1]                      # All of the rows, column 1.
>>> x
array([1, 6])                       # Not array([[1],
>>> x.shape                         #             [6]])
(2,)
>>> x.ndim
1
```

However, it is occasionally necessary to change a 1-D array into a "column array". Use `np.reshape()`, `np.vstack()`, or slice the array and put `np.newaxis` on the second axis. Note that `np.transpose()` does not alter 1-D arrays.

```
>>> x = np.arange(3)
>>> x
array([0, 1, 2])

>>> x.reshape((-1,1))               # Or x[:,np.newaxis] or np.vstack(x).
array([[0],
       [1],
       [2]])
```

Do not force a 1-D vector to be a column vector unless necessary.

Stacking

NumPy has functions for *stacking* two or more arrays with similar dimensions into a single block matrix. Each of these methods takes in a single tuple of arrays to be stacked in sequence.

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.

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

# vstack() stacks arrays vertically (row-wise).
>>> np.vstack((A,B,A))
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.]])

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

# hstack() stacks arrays horizontally (column-wise).
>>> 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.]])

# column_stack() stacks arrays horizontally, including 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.]])
```

See `http://docs.scipy.org/doc/numpy-1.10.1/reference/routines.array-manipulation.html` for more array manipulation routines and documentation.

Problem 6. Write a function that defines the following matrices as NumPy arrays.

$$A = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix} \qquad B = \begin{bmatrix} 3 & 0 & 0 \\ 3 & 3 & 0 \\ 3 & 3 & 3 \end{bmatrix} \qquad C = \begin{bmatrix} -2 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & -2 \end{bmatrix}$$

Use NumPy's stacking functions to create and return the block matrix:

$$\begin{bmatrix} \mathbf{0} & A^\mathsf{T} & I \\ A & \mathbf{0} & \mathbf{0} \\ B & \mathbf{0} & C \end{bmatrix},$$

where I is the 3×3 identity matrix and each $\mathbf{0}$ is a matrix of all zeros of appropriate size.

A block matrix of this form is used in the interior point method for linear optimization.

Array Broadcasting

Many matrix operations make sense only when the two operands have the same shape, such as element-wise addition. *Array broadcasting* extends such operations to accept some (but not all) operands with different shapes, and occurs automatically whenever possible.

Suppose, for example, that we would like to add different values to the columns of an $m \times n$ matrix A. Adding a 1-D array x with the n entries to A will automatically do this correctly. To add different values to the different rows of A, first reshape a 1-D array of m values into a column array. Broadcasting then correctly takes care of the operation.

Broadcasting can also occur between two 1-D arrays, once they are reshaped appropriately.

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

# Add the entries of 'x' to the corresponding columns of 'A'.
>>> A + x
array([[ 0,  2,  4],
       [ 3,  5,  7],
       [ 6,  8, 10],
       [ 9, 11, 13]])

>>> y = np.arange(0, 40, 10).reshape((4,1))
>>> y
array([[ 0],
       [10],
       [20],
```

```
       [30]])

# Add the entries of 'y' to the corresponding rows of 'A'.
>>> A + y
array([[ 0,  1,  2],
       [13, 14, 15],
       [26, 27, 28],
       [39, 40, 41]])

# Add 'x' and 'y' together with array broadcasting.
>>> x + y
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

Numerical Computing with NumPy

Universal Functions

A *universal function* is one that operates on an entire array element-wise. Universal functions are significantly more efficient than using a loop to operate individually on each element of an array.

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])
```

See `http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs` for a more comprehensive list of universal functions.

Achtung!

The `math` module has many useful functions for numerical computations. However, most of these functions can only act on single numbers, not on arrays. NumPy functions can act on either scalars or entire arrays, but `math` functions tend to be a little faster for acting on scalars.

```
>>> import math

# Math and NumPy functions can both operate on scalars.
>>> print(math.exp(3), np.exp(3))
20.085536923187668 20.0855369232

# However, math functions cannot operate on arrays.
>>> x = np.arange(-2, 3)
>>> np.tan(x)
array([ 2.18503986, -1.55740772,  0.        ,  1.55740772, -2.18503986])
>>> math.tan(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Always use universal NumPy functions, not the `math` module, when working with arrays.

Other Array Methods

The `np.ndarray` class itself has many useful methods for numerical computations.

Method	Returns
`all()`	`True` if all elements evaluate to `True`.
`any()`	`True` if any elements evaluate to `True`.
`argmax()`	Index of the maximum value.
`argmin()`	Index of the minimum value.
`argsort()`	Indices that would sort the array.
`clip()`	restrict values in an array to fit within a given range
`max()`	The maximum element of the array.
`mean()`	The average value of the array.
`min()`	The minimum element of the array.
`sort()`	Return nothing; sort the array in-place.
`std()`	The standard deviation of the array.
`sum()`	The sum of the elements of the array.
`var()`	The variance of the array.

Each of these `np.ndarray` methods has an equivalent NumPy function. For example, `A.max()` and `np.max(A)` operate the same way. The one exception is the `sort()` function: `np.sort()` returns a sorted copy of the array, while `A.sort()` sorts the array in-place and returns nothing.

Every method listed can operate *along an axis* via the keyword argument `axis`. If `axis` is specified for a method on an n-D array, the return value is an $(n-1)$-D array, the specified axis having been collapsed in the evaluation process. If `axis` is not specified, the return value is usually a scalar. Refer to the NumPy Visual Guide in the appendix for more visual examples.

```
>>> A = np.arange(9).reshape((3,3))
>>> A
array([[0, 1, 2],
       [3, 4, 5],
```

```
       [6, 7, 8]])

# Find the maximum value in the entire array.
>>> A.max()
8

# Find the minimum value of each column.
>>> A.min(axis=0)                  # np.array([min(A[:,i]) for i in range(3)])
array([0, 1, 2])

# Compute the sum of each row.
>>> A.sum(axis=1)                  # np.array([sum(A[i,:]) for i in range(3)])
array([3, 12, 21])
```

See `http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html` for a more comprehensive list of array methods.

Problem 7. A matrix is called *row-stochastic*[a] if its rows each sum to 1. Stochastic matrices are fundamentally important for finite discrete random processes and some machine learning algorithms.

Write a function than accepts a matrix (as a 2-D array). Divide each row of the matrix by the row sum and return the new row-stochastic matrix. Use array broadcasting and the `axis` argument instead of a loop.

[a] Similarly, a matrix is called *column-stochastic* if its columns each sum to 1.

Problem 8. This problem comes from `https://projecteuler.net`.

In the 20×20 grid below, four numbers along a diagonal line have been marked in red.

08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
32 98 81 28 64 23 67 10 26 38 40 67 59 54 70 66 18 38 64 70
67 26 20 68 02 62 12 20 95 63 94 39 63 08 40 91 66 49 94 21
24 55 58 05 66 73 99 26 97 17 78 78 96 83 14 88 34 89 63 72
21 36 23 09 75 00 76 44 20 45 35 14 00 61 33 97 34 31 33 95
78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48

The product of these numbers is $26 \times 63 \times 78 \times 14 = 1788696$. Write a function that returns the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the grid.

For convenience, this array has been saved in the file `grid.npy`. Use the following syntax to extract the array:

```
>>> grid = np.load("grid.npy")
```

One way to approach this problem is to iterate through the rows and columns of the array, checking small slices of the array at each iteration and updating the current largest product. Array slicing, however, provides a much more efficient solution.

The naïve method for computing the greatest product of four adjacent numbers in a horizontal row might be as follows:

```
>>> winner = 0
>>> for i in range(20):
...     for j in range(17):
...         winner = max(np.prod(grid[i,j:j+4]), winner)
...
>>> winner
48477312
```

Instead, use array slicing to construct a single array where the (i, j)th entry is the product of the four numbers to the right of the (i, j)th entry in the original grid. Then find the largest element in the new array.

```
>>> np.max(grid[:,:-3] * grid[:,1:-2] * grid[:,2:-1] * grid[:,3:])
48477312
```

Use slicing to similarly find the greatest products of four vertical, right diagonal, and left diagonal adjacent numbers.
(Hint: Consider drawing the portions of the grid that each slice in the above code covers, like the examples in the visual guide. Then draw the slices that produce vertical, right diagonal, or left diagonal sequences, and translate the pictures into slicing syntax.)

Achtung!

All of the examples in this lab use NumPy arrays, objects of type `np.ndarray`. NumPy also has a "matrix" data structure called `np.matrix` that was built specifically for MATLAB users who are transitioning to Python and NumPy. It behaves slightly differently than the regular array class, and can cause some unexpected and subtle problems.

For consistency (and your sanity), **never** use a NumPy matrix; **always** use NumPy arrays. If necessary, cast a matrix object as an array with `np.array()`.

Additional Material

Further Reading

Refer back to this and other introductory labs often as you continue getting used to Python syntax and data types. As you continue your study of Python, we strongly recommend the following readings.

- The official Python tutorial: `http://docs.python.org/3.6/tutorial/introduction.html` (especially chapters 3, 4, and 5).
- Section 1.2 of the SciPy lecture notes: `http://scipy-lectures.github.io/`.
- PEP8 - Python style guide: `http://www.python.org/dev/peps/pep-0008/`.

Python Main

If you have programmed in another language, you may have needed a `main` function in order for the program to run. Because Python is a high level language, it can run with or without a `main`. Below is the syntax for writing a `main`.

```
if __name__ == "__main__":
    print("Hello, world!")          # Indent with four spaces (NOT a tab).
```

A common place to use the `if __name__ == "__main__"` syntax is if the file you are writing can be used as a stand alone file or be called by another Python Script. If it is run by itself, it will enter the `if __name__ == "__main__"` branch. If it is being called by another script, it won't enter the `if __name__ == "__main__"` branch, but it still gives access to the function and variables stored in the file.

Ipython Object Introspection

One of the biggest advantages of IPython is that it supports *object introspection*, whereas the regular Python interpreter does not. Object introspection quickly reveals all methods and attributes associated with an object. IPython also has a built-in `help()` function that provides interactive help.

```
    # A list is a basic Python data structure. To see the methods associated ↩
        with
    # a list, type the object name (list), followed by a period, and press tab.
    In [1]: list.   # Press 'tab'.
    append()  count()   insert()  remove()
    clear()   extend()  mro()     reverse()
    copy()    index()   pop()     sort()

    # To learn more about a specific method, use a '?' and hit 'Enter'.
    In [1]: list.append?
    Docstring: L.append(object) -> None -- append object to end
    Type:      method_descriptor

    In [2]: help()                      # Start IPython's interactive help ↩
        utility.
```

```
help> list                          # Get documentation on the list class.
Help on class list in module __builtin__:

class list(object)
|  list() -> new empty list
|  # ...                           # Press 'q' to exit the info screen.

help> quit                          # End the interactive help session.
```

Generalized Function Input

On rare occasion, it is necessary to define a function without knowing exactly what the parameters will be like or how many there will be. This is usually done by defining the function with the parameters `*args` and `**kwargs`. Here `*args` is a list of the positional arguments and `**kwargs` is a dictionary mapping the keywords to their argument. This is the most general form of a function definition.

```
>>> def report(*args, **kwargs):
...     for i, arg in enumerate(args):
...         print("Argument " + str(i) + ":", arg)
...     for key in kwargs:
...         print("Keyword", key, "-->", kwargs[key])
...
>>> report("TK", 421, exceptional=False, missing=True)
Argument 0: TK
Argument 1: 421
Keyword exceptional --> False
Keyword missing --> True
```

See `https://docs.python.org/3.6/tutorial/controlflow.html` for more on this topic.

Function Decorators

A *function decorator* is a special function that "wraps" other functions. It takes in a function as input and returns a new function that pre-processes the inputs or post-processes the outputs of the original function.

```
>>> def typewriter(func):
...     """Decorator for printing the type of output a function returns"""
...     def wrapper(*args, **kwargs):
...         output = func(*args, **kwargs)      # Call the decorated function.
...         print("output type:", type(output)) # Process before finishing.
...         return output                       # Return the function output.
...     return wrapper
```

The outer function, `typewriter()`, returns the new function `wrapper()`. Since `wrapper()` accepts `*args` and `**kwargs` as arguments, the input function `func()` accepts any number of positional or keyword arguments.

Apply a decorator to a function by tagging the function's definition with an @ symbol and the decorator name.

```
>>> @typewriter
... def combine(a, b, c):
...     return a*b // c
```

Placing the tag above the definition is equivalent to adding the following line of code after the function definition:

```
>>> combine = typewriter(combine)
```

Now, calling `combine()` actually calls `wrapper()`, which then calls the original `combine()`.

```
>>> combine(3, 4, 6)
output type: <class 'int'>
2
>>> combine(3.0, 4, 6)
output type: <class 'float'>
2.0
```

Function decorators can also be customized with arguments. This requires another level of nesting: the outermost function must define and return a decorator that defines and returns a wrapper.

```
>>> def repeat(times):
...     """Decorator for calling a function several times."""
...     def decorator(func):
...         def wrapper(*args, **kwargs):
...             for _ in range(times):
...                 output = func(*args, **kwargs)
...             return output
...         return wrapper
...     return decorator
...
>>> @repeat(3)
... def hello_world():
...     print("Hello, world!")
...
>>> hello_world()
Hello, world!
Hello, world!
Hello, world!
```

See `https://www.python.org/dev/peps/pep-0318/` for more details.