Basics of Python

Wednesday, September 18

Contents

1. Introduction to Python

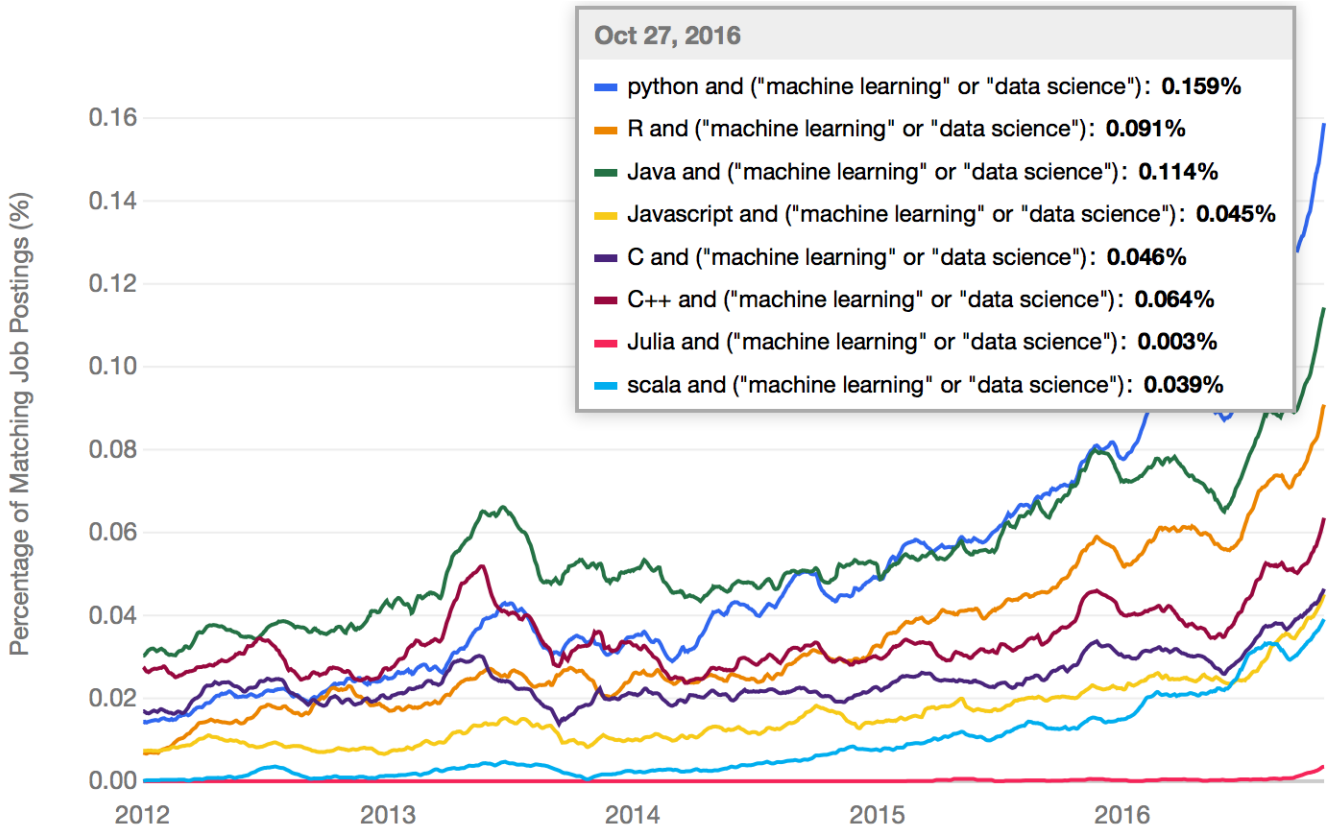
1.1. What is Python

Python is a high-level and dynamic programming language.

Unlike C/C++, Java, R and many other programming languages, Python is designed to be easy to read, write and understand. It is free, and open source, which has contributed to its wide adoption in tech companies and in academia. Python's versatility has made it an industry standard in machine learning, data management & processing, web development, scripting, and finance.

1.2. Why do we choose Python

- **Simplicity**: Python code is designed to be as simple and intuitive as the pseudo code.
- **Popularity**: Python is the most popular programming language in the data analysis and machine learning community. See below a graph of trends in requirements among job adverts for data scientists over time.
- **Community support**: As a result of its popularity, Python is now extremely well supported and extensively documented.



Graph from [IBM blogs (https://www.ibm.com/developerworks/community/blogs/roller-ui/homepage?lang=en)](https://www.ibm.com/developerworks/community/blogs/roller-ui/homepage?lang=en)

1.3. Zen of Python

A collection of 19 aphorisms embodies the guiding principles that made Python a popular porogramming language: the "Zen of Python". The Zen of Python was originally written by Tim Peters and lives in PEP20. It is good to keep these in mind when writing code.

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.

- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
- Although practicality beats purity.
- Errors should never pass silently.
- Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
- Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
- Although never is often better than *right* now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- Namespaces are one honking great idea -- let's do more of those!

2. Data types in Python

2.1. Basic data types

Just like R or MATLAB, Python has several basic data types to represent numbers, booleans and strings.

Numbers: Integers and floats work as the same in MATLAB/R

In [1]:

```
# Integers
x = 3
print(type(x)) # Prints "<class 'int'>"
print(x)       # Prints "3"
print(x + 1)   # Addition; prints "4"
print(x - 1)   # Subtraction; prints "2"
print(x * 2)   # Multiplication; prints "6"
print(x ** 2)  # Exponentiation; prints "9"

# Operations refereing to the already named variable
x += 1
print(x)       # Prints "4"
x *= 2

# Floats
print(x)       # Prints "8"
y = 2.5
print(type(y))                    # Prints "<class 'float'>"
print(y, y + 1, y * 2, y ** 2)    # Prints "2.5 3.5 5.0 6.25"
```

```
<class 'int'>
3
4
2
6
9
4
8
<class 'float'>
2.5 3.5 5.0 6.25
```

Booleans: Python has full support for boolean operators. Importantly, and unlike STATA, boolean operators are English words rather than symbols like `&` or `|`

In [2]:

```
t = True
f = False
print(type(t))  # Prints "<class 'bool'>"
print(t and f)  # Logical AND; prints "False"
print(t or f)   # Logical OR; prints "True"
print(not t)    # Logical NOT; prints "False"
print(t != f)   # Logical XOR; prints "True"
```

```
<class 'bool'>
False
True
False
True
```

Strings: strings are enclosed by single or double quotes

In [3]:

```
hello = 'hello'                         # String literals can use single quotes
world = "world"                         # or double quotes; it does not matter.
print(hello)                            # Prints "hello"
print(len(hello))                       # String length; prints "5"
hw = hello + ' ' + world                # String concatenation
print(hw)                               # prints "hello world"
hw12 = '%s %s %d' % (hello, world, 12)  # sprintf style string formatting
print(hw12)                             # prints "hello world 12"
```

```
hello
5
hello world
hello world 12
```

Python also support some convenient functions for string operations. This will come handy when cleaning data.

In [4]:

```
s = "hello"
print(s.capitalize())           # Capitalize a string; prints "Hello"
print(s.upper())                # Convert a string to uppercase; prints "HELLO"
print(s.rjust(7))               # Right-justify a string, padding with spaces; prints "
hello"
print(s.center(7))              # Center a string, padding with spaces; prints " hello
 "
print(s.replace('l', '(ell)'))  # Replace all instances of one substring with another;
                                # prints "he(ell)(ell)o"
print('  world '.strip())       # Strip leading and trailing whitespace; prints "world"
```

```
Hello
HELLO
  hello
 hello
he(ell)(ell)o
world
```

2.2. Containers

A container is a data structure that stores objects in an organised way. Accessing the various elements of a container must be done by following specific rules.

Python provides four built-in containers: lists, dictionaries, sets, and tuples. We quickly review them below.

2.2.1. Lists

A list is an array of elements in Python. It can be modified after its creation, it is resizable and can contain elements of different types. All elements in a list are indexed by integers. Lists are delimited by squared brackets `[...]` .

In [5]:

```
xs = [3, 1, 2]              # Create a list
print(xs, xs[2])            # Prints "[3, 1, 2] 2"
print(xs[-1])               # Negative indices count from the end of the list; prints "2"
xs[2] = 'foo'               # Lists can contain elements of different types
print(xs)                   # Prints "[3, 1, 'foo']"
xs.append('bar')            # Add a new element to the end of the list
print(xs)                   # Prints "[3, 1, 'foo', 'bar']"
x = xs.pop()                # Remove and return the last element of the list
print(x, xs)                # Prints "bar [3, 1, 'foo']"
```

```
[3, 1, 2] 2
2
[3, 1, 'foo']
[3, 1, 'foo', 'bar']
bar [3, 1, 'foo']
```

In Python, it is easy to slice, loop and transform list. See examples below.

In [6]:

```
#### Slicing: get sublist
nums = list(range(5))     # range is a built-in function that creates a list of integer
s
print(nums)               # Prints "[0, 1, 2, 3, 4]"
print(nums[2:4])          # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print(nums[2:])           # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print(nums[:2])           # Get a slice from the start to index 2 (exclusive); prints "
[0, 1]"
print(nums[:])            # Get a slice of the whole list; prints "[0, 1, 2, 3, 4]"
print(nums[:-1])          # Slice indices can be negative; prints "[0, 1, 2, 3]"
nums[2:4] = [8, 9]        # Assign a new sublist to a slice
print(nums)               # Prints "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

In [7]:

```
#### Looping: loop over elements
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)             # Prints "cat", "dog", "monkey", each on its own line.
```

```
cat
dog
monkey
```

In [8]:

```
#### list comprehensions
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)   # Prints [0, 1, 4, 9, 16]
```

```
[0, 1, 4, 9, 16]
```

2.2.2. Dictionaries

Sometimes, we need more freedom in defining indices in a container. Lists only allow us to use integers as indices, and they have to be ordered in increasing order.

A dictionary offers a solution to this problem. Just like a normal dictionary is a list of word-definition pairs, a Python dictionary is a container of `(key, value)` pairs. Dictionaries are defined with curly brackets, and colons separate keys from values: `{ : , : , ... , : }`. Values can be any data types, just like with lists, but here keys can also be any data type.

We access the values of a dictionary just like those of a list: by writing the name of the dictionary and the key of the value we are interested in between square brackets. For instance `my_dict['key1']`.

In [9]:

```
d = {'cat': 'cute', 'dog': 'furry'}   # Create a new dictionary with some data

print(d['cat'])                       # Get an entry from a dictionary; prints "cute"
print('cat' in d)                     # Check if a dictionary has a given key; prints
 "True"
d['fish'] = 'wet'                     # Set an entry in a dictionary
print(d['fish'])                      # Prints "wet"
# print(d['monkey'])                  # KeyError: 'monkey' not a key of d
print(d.get('monkey', 'N/A'))         # Get an element with a default; prints "N/A"
print(d.get('fish', 'N/A'))           # Get an element with a default; prints "wet"

del d['fish']                         # Remove an element from a dictionary
print(d.get('fish', 'N/A'))           # "fish" is no longer a key; prints "N/A"
```

```
cute
True
wet
N/A
wet
N/A
```

Just like with lists, we can loop over values of a dictionary. Unlike lists however, we can also loop over key-value pairs.

In [10]:

```
#### over keys
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"

#### over keys and values
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# Prints "A person has 2 legs", "A cat has 4 legs", "A spider has 8 legs"
```

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

In [11]:

```
#### dict comprehensions
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square)  # Prints "{0: 0, 2: 4, 4: 16}"
```

```
{0: 0, 2: 4, 4: 16}
```

2.2.3. Sets

When both the ordering of elements and their possible duplicity do not matter, we can use sets instead of lists. Just like sets in mathematics, they are simply defined by the elements they contain. Importantly, a set in Python is made up of immutable objects like integers, tuples or strings. That means they cannot contain lists, which can be amended.

Sets are written with curly brackets `{ , , ... , }`

The example below makes clear how Python treats sets.

In [12]:

```
example_set = {3, 2, 1, 2}
print(example_set)
```

```
{1, 2, 3}
```

Below are some functions over sets that you may find useful: addition, deletion and boolean operations.

In [13]:

```
animals = {'cat', 'dog'}
print('cat' in animals)   # Check if an element is in a set; prints "True"
print('fish' in animals)  # prints "False"

animals.add('fish')       # Add an element to a set
print('fish' in animals)  # Prints "True"
print(len(animals))       # Number of elements in a set; prints "3"

animals.add('cat')        # Adding an element that is already in the set does nothing
print(len(animals))       # Prints "3"

animals.remove('cat')     # Remove an element from a set
print(len(animals))       # Prints "2"
```

```
True
False
True
3
3
2
```

Even if the elements of a set are not ordered, we can build `for` -loops iterating over them.

In [14]:

```
animals = {'cat', 'dog', 'fish'}
for animal in animals:
    print(animal)
```

```
dog
fish
cat
```

In [15]:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
# Prints "#1: fish", "#2: dog", "#3: cat"
```

```
#1: dog
#2: fish
#3: cat
```

In [16]:

```
#### set comprehensions
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)  # Prints "{0, 1, 2, 3, 4, 5}"
```

```
{0, 1, 2, 3, 4, 5}
```

2.2.4. Tuples

Tuples are ordered and immutable lists of values. A tuple is in many ways similar to a list; one of the most important differences is that tuples can be used as keys in dictionaries and as elements of sets, while lists cannot.

Tuples are delimited by round brackets `( , , ... , )`.

In [17]:

```
d = {(x, x + 1): x for x in range(10)}   # Create a dictionary with tuple keys
t = (5, 6)                               # Create a tuple

print(type(t))                           # Prints "<class 'tuple'>"
print(d[t])                              # Prints "5"
print(d[(1, 2)])                         # Prints "1"
```

```
<class 'tuple'>
5
1
```

3. Functions

User-written functions are essential to make the logic of the code clearer, and to avoid copying the same routine several times when a task needs to be repeated.

Python functions are defined using the `def` and `return` keywords. We need to specify which arguments are being fed to a function when defining it.

See for instance a example of a function signing the integer or floeat fed to it.

In [18]:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# Prints "negative", "zero", "positive"
```

```
negative
zero
positive
```

We can also set default values to some of the arguments

In [19]:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)

hello('Bob') # Prints "Hello, Bob"
hello('Fred', loud=True)  # Prints "HELLO, FRED!"
```

```
Hello, Bob
HELLO, FRED!
```

Python functions are very versatile:

- They can even be defined within other functions
- They can take other functions as arguments
- They can output other functions

It is always useful to document the functions we write. To do so, we can use Python's `docstring` functionality. A doctsring is simply a piece of text written in the body of a function. It is meant to give us information about the function we are using, or more plausibly, about the function a diligent colleague has written.

The docstring is introduced and concluded by three quotes `"""` . We get access to a function's docstring by writting the name of the function followed by one or two interrogation marks.

In [20]:

```
def hello2(name, loud=False):
    """
    This function is a documented version of `hello`
    It returns `Hello` + a name given in arguments
    If `loud=True` is also entered as an argument, the output will be in capitalised le
tters
    """
    if loud:
        print('HELLO, %s!' % name.upper())
    else:
        print('Hello, %s' % name)
```

In [21]:

```
hello2?
```

In [22]:

```
hello??
```

4. Classes

Python supports the object-oriented programming and the definition of class is very straightforward.

What is Class

Class refers to the user-defined data-type. In class, you can define the stored internal data and also the associated functions (called **methods**).

How to create a class

For example, we could define a new data-type, called `Circle` in which we have a `radius` to store its radius and a `area()` mmethod to compute the size of the circle

In [23]:

```
class Circle(object):
    # Constructor
    def __init__(self, r):
        self.radius = r  # Create an instance variable

    # Instance method
    def area(self):
        return 3.14 * self.radius**2
```

Every class should contain a initialization function, called *Constructor*, to specify which internal variables you want to create to store the data.

Here, we use `radius` variable to store the data and the `self.` is necessary, meaning that the `radius` is an internal variable.

How to use class

Once class is defined, we can create *object* from the class, which is an instance of the class. When creating the object, we need to specify all the data to store by the internal variable.

And if we want to use the methods, we will type the name of the object followed by `.` and the name of the methods.

In [24]:

```
c1 = Circle(0.5)
c1.area()
```

Out[24]:

```
0.785
```

Here we have a more complicated class example.

In [25]:

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

```
Hello, Fred
HELLO, FRED!
```

More details about class and object-oriented programming can be found here: MIT6_0001F16_Lec8 (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-slides-code/MIT6_0001F16_Lec8.pdf)

5. Numpy

Numpy is a core numerical computing library for Python which supports high-performance operations over multidimensional arrays. If you are familiar with MATLAB, you can master the Numpy library very quickly.

5.1. Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

Numpy arrays are basically like standard Python lists, but they are optimised for scientific calculations and can store multidimensional objects as values.

In [26]:

```
import numpy as np

a = np.array([1, 2, 3])   # Create a rank 1 array
print(type(a))            # Prints "<class 'numpy.ndarray'>"
print(a.shape)            # Prints "(3,)"
print(a[0], a[1], a[2])   # Prints "1 2 3"
a[0] = 5                  # Change an element of the array
print(a)                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4"
```

```
<class 'numpy.ndarray'>
(3,)
1 2 3
[5 2 3]
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrrays. This is useful to initialise matrices or to create a variable of ones to get an intercept in OLS for instance.

In [27]:

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"

b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"

c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"

e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #              [ 0.68744134  0.87236687]]"
```

```
[[0. 0.]
 [0. 0.]]
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.51160833 0.79963087]
 [0.1422646  0.00224346]]
```

Array indexing

In [28]:

```
import numpy as np

#### slicing

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77"


# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #         [ 6]
                             #         [10]] (3, 1)"
```

```
2
77
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
[ 2  6 10] (3,)
[[ 2]
 [ 6]
 [10]] (3, 1)
```

Integer array indexing

In [29]:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"

# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"

# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"

# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

```
[1 4 5]
[1 4 5]
[2 2]
[2 2]
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
[ 1  6  7 11]
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

Boolean array indexing

In [30]:

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
[[False False]
 [ True  True]
 [ True  True]]
[3 4 5 6]
[3 4 5 6]
```

5.2. Mathematical operations with Numpy arrays

In [31]:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, * is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

In [32]:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))

# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```
219
219
[29 67]
[29 67]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

In [33]:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the `T` attribute of an array object:

In [34]:

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

```
[[1 2]
 [3 4]]
[[1 3]
 [2 4]]
[1 2 3]
[1 2 3]
```

5.3. Broadcasting

Broadcasting is a powerful mechanism that allows Numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix. We could do it like this:

In [35]:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

This works; however when the matrix `x` is very large, computing an explicit loop in Python could be slow. Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`. We could implement this approach like this:

In [36]:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each other
print(vv)                 # Prints "[[1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]
                          #         [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

```
[[1 0 1]
 [1 0 1]
 [1 0 1]
 [1 0 1]]
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`. Consider this version, using broadcasting:

In [37]:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #         [ 5  5  7]
          #         [ 8  8 10]
          #         [11 11 13]]"
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise. Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension