A Basic Guide to Programming and Plotting in Python

Compiled by Sujan Koirala

Kanae Laboratory

Department of Mechanical and Environmental Informatics

Tokyo Institute of Technology

Python Documentation Series

Version 1-1
May, 2011

Contents

List of Figures

List of Tables

Chapter 1

Python Data Types and Operations

– Sujan Koirala

1-1 Introduction

In python there are various types of data. Every data has a type and a value. Every value has a fixed data type but it should not specified beforehand. The most basic data types in python are:

1. Boolean: These are data which have only two values: True or False.
2. Numbers: These are numeric data.
3. Strings: These data are sequences of unicode characters.
4. Bytes: An immutable sequence of numbers.

Furthermore, these data types can be combined and following types of datasets can be produced:

1. Lists: Ordered sequences of values.
2. Tuples: Ordered but immutable, i.e. cannot be modified, sequences of values.
3. Sets: Unordered bags of values.
4. Dictionaries: Unordered bag of key-value pairs.
5. Arrays: Ordered sequences of data of same type mentioned above.

1-2 Basic Data Types

In this section, a brief description of basic data types, their possible values, and various operations that can be applied to them are described.

1-2-1 Boolean Data

These data are either True or False. If an expression can produce either yes or no answers, booleans can be used to interpret the result. This kind of yes/no situations are known as boolean context. Here is a simple example.

- Assign some variable (size) as 1.

  >>> size = 1

- Check if size is less than 0.

  >>> size < 0

  >>> False

  ✓It is false as 1 > 0.

- Check if size is greater than 0.

  >>> size > 0

  >>> True

True or False can also be treated as numbers: True=1 and False=0.

1-2-2 Numbers

Python supports both integers and floating point numbers. There's no type declaration to distinguish them and Python automatically distinguishes them apart by the presence or absence of a decimal point.

- You can use type() function to check the type of any value or variable.

  >>> type(1)

  >>> <type 'int'>

  ✓As you expected, 1 is an int.

  >>> type(1.)

  >>> <type 'float'>

✓The decimal at the end make 1. a float.

```
>>> 1+1
>>> 2
```

✓Adding an int to an int yields an int.

```
>>> 1+1.
>>> 2.0
```

✓Adding an int to a float yields a float. Python coerces the int into a float to perform the addition, then returns a float as the result.

- Integer can be converted to float using float() and float can be converted to integer using int()

```
>>> float(2)
>>> 2.0
>>> int(2.6)
>>> 2
```

✓Python truncates the float to integer, 2.6 becomes 2 instead of 3. To round the float number use

```
>>> round(2.6)
>>> 3.0
```

1-2-2-1 Numerical Operations

- The '/' operator performs division.

```
>>> 1/2
>>> 0
>>> 1/2.
>>> 0.5
```

✓Be careful on float or integer data type as the result can be different as shown above.

- The '//' operator performs a division combined with truncating and rounding. When the result is positive, it truncates the result but when the result is negative, it rounds off the result to nearest integer but the result is always a float.

```
>>> 1.//2
>>> 0.0
>>> -1.//2
>>> -1.0
```

- The '**' operator means "raised to the power of". 11^2 is 121.

```
>>> 11**2
>>> 121
>>> 11**2.
>>> 121.0
```

  ✓Be careful on float or integer data type as the result can be different as shown above.

- The '%' operator gives the remainder after performing integer division.

```
>>> 11%2
>>> 1
```

  ✓11 divided by 2 is 5 with a remainder of 1, so the result here is 1.

1-2-2-2 Fractions

To start using fractions, import the fractions module. To define a fraction, create a Fraction object as

```
>>> import fractions
>>> fractions.Fraction(1,2)
>>> Fraction(1, 2)
```

You can perform all the usual mathematical operations with fractions as

```
>>> fractions.Fraction(1, 2)*2
>>> Fraction(1, 1)
```

1-2-2-3 Trigonometry

You can also do basic trigonometry in Python.

```
>>> import math
>>> math.pi
>>> 3.1415926535897931
>>> math.sin(math.pi / 2)
>>> 1.0
```

1-2-3 Strings

In Python, all strings are sequences of Unicode characters. It is an immutable sequence and cannot be modified.

- To create a string, enclose it in quotes. Python strings can be defined with either single quotes (' ') or double quotes (" ").

  ```
  >>> s='sujan'  or s="sujan"
  ```

- The built-in len() function returns the length of the string, i.e. the number of characters.

  ```
  >>> len(s)
  >>> 5
  ```

- You can get individual characters out of a string using index notation.

  ```
  >>> s[1]
  >>> u
  ```

- You can concatenate strings using the + operator.

  ```
  >>> s+' '+'koirala'
  >>> sujan koirala
  ```

  ✓Even space has to be specified as an empty string.

1-2-3-1 Common String Methods

Lets assume we have a string 'sujan koirala' stored as string s.

```
>>> = 'Sujan Koirala'
```

1. **split()**: It has one required argument, a delimiter. The method splits a string into a list of strings based on the delimiter. See Table 1-1 for examples.
2. **lower() and upper()**: Changes the string to lower case and upper case respectively.

   ```
   >>> s
   >>> 'Sujan Koirala'
   >>> s.lower()
   >>> 'sujan koirala'
   >>> s.upper()
   >>> 'SUJAN KOIRALA'
   ```

Table 1-1: Splitting the strings

>>> s.split()
 	
>>> ['Sujan', 'Koirala']
	✓blank space as delimiter	✓creates a list with elements separated at locations of blank space

>>> s.split('a') < 0
 	
>>> ['Suj', 'n Koir', 'l', '']
	✓'a'as delimiter	✓creates a list with elements separated at locations of 'a'

3. **count()**: Counts the number of occurrences of a substring.

   >>> s.count('a')

   >>> 3

   ✓There are 3 a’s in string s.

1-2-4 Bytes

An immutable sequence of numbers between 0 and 255 is called a bytes object. Each byte within the bytes object can be an ascii character or an encoded hexadecimal number from \x00 to \xff (0–255).

- To define a bytes object, use the b' 'syntax. This is commonly known as “byte literal” syntax.

  >>> by = b'abcd\x65'

  >>> by

  >>> 'abcde'

  ✓\x65 is 'e'.

- Just like strings, you can use len() function and use the + operator to concatenate bytes objects. But you cannot join strings and bytes.

  >>> len(by)

  >>> 5

  >>> by += b'\x66'

```
>>> by
>>> 'abcdef'
```

1-3 Combined Data Types

The basic data types explained in the previous section can be arranged in sequences to create combined data types. These combined data types can be modified, for e.g., lists or are immutable which cannot be modified, for e.g., tuples. This section provides brief description of these data and the common operations that can be used.

1-3-1 Lists

Lists are the sequence of data stored in an arranged form. It can hold different types of data (strings, numbers etc.) and it can be modified to add new data or remove old data.

1-3-1-1 Creating a List

To create a list: use square brackets “[ ]” to wrap a comma-separated list of values of any data types.

```
>>> a_list = [ 'a', 'b', 'mpilgrim', 'z', 'example', 2]
>>> a_list
>>> ['a', 'b', 'mpilgrim', 'z', 'example', 2]
```

✓All data except last data are strings. Last one is integer.

```
>>> a_list[ 0]
>>> 'a'
```

✓List data can be accessed using index.

```
>>> type(a_list[0])
>>> <type 'str'>
>>> type(a_list[-1])
>>> <type 'int'>
```

✓Type of data can be checked using type().

1-3-1-2 Slicing a List

Once a list has been created, a part of it can be taken as a new list. This is called slicing the list. A slice can be extracted using indices. Let’s consider same list as above:

```
>>> a_list = [ 'a', 'b', 'mpilgrim', 'z', 'example', 2]
```

- The length of the list can be obtained as:

  ```
  >>> len(a_list)
  >>> 6
  ```

  ✓the index can be from 0 to 5 if we count from left to right or -1 to -6 if we count from right to left.

- We can obtain any other list as:

  ```
  >>> b_list=a_list[0:3]
  >>> b_list
  >>> [ 'a', 'b', 'mpilgrim']
  ```

1-3-1-3 Adding Item to a List

There are 4 different ways to add item/items to a list. Let's consider same list as above:

```
>>> a_list = [ 'a', 'b', 'mpilgrim', 'z', 'example', 2]
```

1. **'+' operator**: The + operator concatenates lists to create a new list. A list can contain any number of items; there is no size limit.

   ```
   >>> b_list=a_list+['Hydro','Aqua']
   >>> b_list
   >>> [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua']
   ```

2. **append()**: The append() method adds a single item to the end of the list. Even if the added item is a list, the whole list is added as a single item in the old list.

   ```
   >>> b_list.append(True)
   >>> b_list
   >>> [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True]
   ```

   ✓This list has strings, integer, and boolean data.

   ```
   >>> len(b_list)
   >>> 9
   >>> b_list.append(['d','e'])
   >>> b_list
   >>> [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True, ['d', 'e']]
   ```

>>> len(b_list)

>>> 10

✓The length of b_list has increased by only one even though two items, ['d', 'e'], were added.

3. **extend()**: Similar to append but each item is added separately. For e.g., let's consider the list

   >>> b_list= ['a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True]

   >>> len(b_list)

   >>> 9

   >>> b_list.extend(['d','e'])

   >>> b_list

   >>> ['a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True, 'd', 'e']

   >>> len(b_list)

   >>> 11

   ✓The length of b_list has increased by two as two items in the list, ['d', 'e'], were added.

4. **insert()**: The insert() method inserts a single item into a list. The first argument is the index of the first item in the list that will get bumped out of position.

   >>> b_list= ['a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True]

   >>> b_list.insert(0,'d'])

   ✓Insert 'd' in the first position,i.e., index 0.

   >>> b_list

   >>> ['d', 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True]

   >>> b_list.insert(0,['x', 'y'])

   >>> b_list

   >>> [['x', 'y'], 'd', 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', True]

   ✓The list ['x', 'y'] is added as one item as in the case of append().

1-3-1-4 Search for Item in a List

Consider the following list:

```
>>> b_list= [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
```

- **count()** can be used as in the case of string

  ```
  >>> b_list.count('b')
  >>> 2
  ```

- **in** can be used to check if certain value exists in a list.

  ```
  >>> 'b' in b_list
  >>> True
  >>> 'c' in b_list
  >>> False
  ```

  ✓The output is boolean data, i.e., True or False.

- **index** can be used to find the index of search data.

  ```
  >>> b_list.index('a')
  >>> 0
  >>> b_list.index('b')
  >>> 1
  ```

  ✓Even though there are 2 'b', the index of first 'b' is returned.

1-3-1-5 Removing Item from a List

There are many ways to remove an item from a list. The list automatically adjusts its size after some element has been removed.

Removing Item by Index

The **del** command removes an item from a list if the index of an element that needs to be removed is provided.

- Consider the following list:

  ```
  >>> b_list= [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
  ```

- Suppose we want to remove the element 'mpilgrim' from the list. Its index is 2.

  ```
  >>> b_list[2]
  >>> 'mpilgrim'
  >>> del b_list[2]
  >>> b_list
  >>> [ 'a', 'b', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
  ```

  ✓'mpilgrim' is now removed.

The **pop()** command can also remove an item by specifying an index. But, it is even more versatile as it can be used without any argument to remove the last item of a list.

- Consider the following list:

  ```
  >>> b_list= [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
  ```

- Suppose we want to remove the element 'mpilgrim' from the list. Its index is 2.

  ```
  >>> b_list[2]
  >>> 'mpilgrim'
  >>> b_list.pop(2)
  >>> 'mpilgrim'
  ```

  ✓The item to be removed will be displayed.

- Now the b_list is as follows

  ```
  >>> [ 'a', 'b', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
  ```

- If pop() is used without an argument.

  ```
  >>> b_list.pop()
  ```

- Now the b_list is as follows

  ```
  >>> [ 'a', 'b', 'z', 'example', 2, 'Hydro', 'Aqua']
  ```

  ✓The last 'b' is removed from the list.

- If pop() is used once again. The list will be as follows

  ```
  >>> [ 'a', 'b', 'z', 'example', 2, 'Hydro']
  ```

Removing Item by Value

The **remove** command removes item/items from a list if the value of the item is specified.

- Consider the following list:

  ```
  >>> b_list= [ 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b']
  ```

- Suppose we want to remove the elements 'b' from the list.

  ```
  >>> b_list.remove('b')
  >>> b_list
  >>> [ 'a', 'z', 'example', 2, 'Hydro', 'Aqua']
  ```

  ✓All the 'b' in the list are now removed.

1-3-2 Tuples

A tuple is an immutable list. A tuple can not be changed/modified in any way once it is created.

- A tuple is defined in the same way as a list, except that the whole set of elements is enclosed in parentheses instead of square brackets.
- The elements of a tuple have a defined order, just like a list. Tuples indices are zerobased, just like a list, so the first element of a nonempty tuple is always t[0].
- Negative indices count from the end of the tuple, just as with a list.
- Slicing works too, just like a list. Note that when you slice a list, you get a new list; when you slice a tuple, you get a new tuple.
- A tuple is used because reading/writing a tuple is faster than the same for lists. If you do not need to modify a set of item, a tuple can be used instead of list.

1-3-2-1 Creating Tuples

A tuple can be created just like the list but parentheses "( )" has to be used instead of square brackets"[ ]". For e.g.,

```
>>> a_tuple= ( 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b')
```

1-3-2-2 Tuple Operations

All the list operations excpet the ones that modify the list itself can be used for tuples too. For e.g., you cannot use append(), extend(), insert(), del, remove(), and pop() for tuples. For other operations, please follow the same steps as explained in the previous section. Here are some examples of tuple operations.

- Consider the following list:

  >>> a_tuple= ('a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b')

  >>> a_tuple.index('z')

  >>> 3

    ✓item 'z' is at the index 3, i.e., it is the fourth element of the tuple.

  >>> b_tuple=a_tuple[0:4]

  >>> b_tuple

  >>> ('a', 'b', 'mpilgrim', 'z')

    ✓New tuple can be created by slicing a tuple as original tuple does not change.

  >>> a_tuple

  >>> ('a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b')

1-3-3 Sets

A set is an unordered collection of unique values. A single set can contain values of any datatype.

1-3-3-1 Creating Set

There are basically two ways of creating set.

1. From scratch: Sets can be created like lists but curly brackets “{}” have to be used instead of square brackets “[ ]”. For e.g.,

   >>> a_set= { 'a', 'b', 'mpilgrim', 'z', 'example', 2, 'Hydro', 'Aqua', 'b'}

   >>> type(a_set)

   >>> <type 'set'>

   >>> a_set

   >>> set([2, 'Aqua', 'Hydro', 'a', 'b', 'example', 'mpilgrim', 'z'])

✓The set has different orders than the values given inside {} because it is unordered and original orders are ignored. Also, there is only one 'b' in the set even though two 'b' were given because a set is a collection of unique values. Duplicate values are taken as one.

2. From list or tuple: A set can be created from a list or tuple as,

   ```
   >>> set(a_list)
   >>> set(a_tuple)
   ```

1-3-3-2 Modifying Set

A set can be modified by adding an item or another set to it. Also, items of set can be removed.

Adding Elements

- Consider a set as follows,

  ```
  >>> a_set={2, 'Aqua', 'Hydro', 'a', 'b', 'example', 'mpilgrim', 'z'}
  ```

- Using **add**: To add single item to a set.

  ```
  >>> a_set.add('c')
  >>> a_set
  >>> set([2, 'Aqua', 'Hydro', 'a', 'b', 'c', 'example', 'mpilgrim', 'z'])
  ```

  ✓'c' is added after 'b'.

- Using **update**: To add multiple items as a set or list or tuple.

  ```
  >>> a_set.update('a','Sujan','Koirala')
  >>> a_set
  >>> set([2, 'Aqua', 'Hydro', 'Koirala', 'Sujan', 'a', 'b', 'c', 'example', 'mpilgrim', 'z'])
  ```

  ✓'Koirala' and 'Sujan' are added but 'a' is not added.

Removing Elements

- Consider a set as follows,

  ```
  >>> a_set={2, 'Aqua', 'Hydro', 'a', 'b', 'example', 'mpilgrim', 'z'}
  ```

- Using **remove()** and **discard()**: These are used to remove an item from a set.

  ```
  >>> a_set.remove('b')
  >>> a_set
  >>> set([2, 'Aqua', 'Hydro', 'Koirala', 'Sujan', 'a', 'c', 'example', 'mpilgrim', 'z'])
  ```

  ✓'b' has been removed.

  ```
  >>> a_set.discard('Hydro')
  >>> a_set
  >>> set([2, 'Aqua', 'Koirala', 'Sujan', 'a', 'c', 'example', 'mpilgrim', 'z'])
  ```

- Using **pop()** and **clear()**: pop() is same as list but it does not remove the last item as list. pop() removes one item ramdomly. clear() is used to clear the whole set and create an empty set.

  ```
  >>> a_set.pop()
  >>> a_set
  >>> set([2, 'Koirala', 'Sujan', 'a', 'c', 'example', 'mpilgrim', 'z'])
  ```

1-3-3-3 Set Operations

Two sets can be combined or common elements in two sets can be combined to form a new set. These functions are useful to combine two or more lists.

- Consider following two sets,

  ```
  >>> a_set={2,4,5,9,12,21,30,51,76,127,195}
  >>> b_set={1,2,3,5,6,8,9,12,15,17,18,21}
  ```

- **Union**: Can be used to combine two sets.

  ```
  >>> c_set=a_set.union(b_set)
  >>> c_set
  >>> set([1,2,195,4,5,6,8,12,76,15,17,18,3,21,30,51,9,127])
  ```

- **Intersection**: Can be used to create a set with elements common to two sets.

  ```
  >>> d_set=a_set.intersection(b_set)
  >>> d_set
  >>> set([9,2,12,5,21])
  ```

1-3-4 Dictionaries

A dictionary is an unordered set of key-value pairs. A value can be retrieved for a known key but the other-way is not possible.

1-3-4-1 Creating Dictionary

Creating a dictionary is similar to set in using curled brackets "{}" but key:value pairs are used instead of values. The following is an example,

```
>>> a_dict={'Hydro':'131.112.42.40','Aqua':'131.112.42.41'}
>>> a_dict
>>> {'Aqua':'192.168.1.154','Hydro':'131.112.42.40'}
```

✓The order is changed automatically like set.

```
>>> a_dict['Hydro']
>>> '131.112.42.40'
```

✓Key 'Hydro' can be used to access the value '131.112.42.40'.

1-3-4-2 Modifying Dictionary

Since the size of the dictionary is not fixed, new key:value pair can be freely added to the dictionary. Also values for a key can be modified.

- Consider the following dictionary.

  ```
  >>> a_dict={'Aqua':'192.168.1.154','Hydro':'131.112.42.40'}
  ```

- If you want to change the value of 'Aqua',

  ```
  >>> a_dict['Aqua']='192.168.1.154'
  >>> a_dict
  >>> {'Aqua':'192.168.1.154','Hydro':'131.112.42.40'}
  ```

- If you want to add new item to the dictionary,

  ```
  >>> a_dict['Lab']='Kanae'
  >>> a_dict
  >>> {'Aqua':'192.168.1.154','Hydro':'131.112.42.40','Lab':'Kanae'}
  ```

- Dictionary values can also be lists instead of single values. For e.g.,

  ```
  >>> k_lab={'Female':['Yoshikawa','Imada','Yamada','Sasaki','Watanabe','Sato'],
  ```

```
'Male':['Sujan','Iseri','Hagiwara','Shiraha','Ishida','Kusuhara','Hirochi','Endo']}
>>> k_lab['Female']
>>> ['Yoshikawa','Imada','Yamada','Sasaki','Watanabe','Sato']
```

1-3-5 Arrays

Arrays are similar to lists but it contains homogeneous data, i.e., data of same type only. Arrays are commonly used to store numbers and hence used in mathematical calculations.

1-3-5-1 Creating Arrays

Python arrays can be created in many ways. It can also be read from some data file in text or binary format, which are explained in latter chapters of this guide. Here, some commonly used methods are explained. For a detailed tutorial on python arrays, refer here.

1. From list: Arrays can be created from a list or a tuple using:

   ✓somearray=array(somelist). Consider the following examples.

   ```
   >>> b_list=['a','b',1,2]
   ```

   ✓The list has mixed datatypes. First two items are strings and last two are numbers.

   ```
   >>> b_array=array(b_list)
   >>> array(['a', 'b', '1', '2'], dtype='|S8')
   ```

   ✓Since first two elements are string, numbers are also converted to strings when array is created.

   ```
   >>> b_list2=[1,2,3,4]
   ```

   ✓All items are numbers.

   ```
   >>> b_array2=array(b_list2)
   >>> b_list2
   >>> array([1, 2, 3, 4])
   ```

   ✓Numeric array is created. Mathematical operations like addition, subtraction, division, etc. can be carried in this array.

2. Using built-in functions:

(a) From direct values:

```
>>> xx = array([2, 4, -11]
```

✓xx is array of length 3 or shape (1,3) ⇒ means 1 row and 3 columns.

(b) From **arange(number)**: Creates an array from the range of values. Examples are provided below. For details of arange follow chapter 4.

```
>>> yy=arange(2,5,1)
>>> yy
>>> array([2,3,4])
```

✓Creates an array from lower value (2) to upper value (5) in specified interval (1) excluding the last value (5).

```
>>> yy=arange(5)
>>> yy
>>> array([0,1,2,3,4])
```

✓If the lower value and interval are not specified, they are taken as 0 and 1, respectively.

```
>>> yy=arange(5,2,-1)
>>> yy
>>> array([5,4,3])
```

✓The interval can be negative.

(c) Arrays of fixed shape: Sometimes it is necessary to create some array to store the result of calculation. Fuctions **zeros(someshape)** and **ones(someshape)** can be used to create arrays with all values as zero or one, respectively.

```
>>> zz=zeros(20)
```

✓will create an array with 20 zeros.

```
>>> zz=zeros(20,20)
```

✓will create an array with 20 rows and 20 columns (total 20*20=400 elements) with all elements as zero.

```
>>> zz=zeros(20,20,20)
```

✓will create an array with 20 blocks with each block having 20 rows and 20 columns (total 20*20*20=8000 elements) with all elements as zero.

1-3-5-2 Array Operations

Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

```
>>> a = array( [20,30,40,50] )
>>> b = arange( 4 )
>>> b
>>> array( [0,1,2,3] )
>>> c = a-b
>>> c
>>> array([20, 29, 38, 47])
```

✓Each element in b is subtracted from respective element in a.

```
>>> b**2
>>> array([0, 1, 4, 9])
```

✓Square of each element in b.

```
>>> 10*sin(a)
>>> array([ 9.12945251, -9.88031624, 7.4511316 , -2.62374854])
```

✓Two operations can be carried out at once.

```
>>> a<35
>>> array([True, True, False, False], dtype=bool)
```

✓'True' if a<35 otherwise 'False'.

For comprehensive examples of all mathematical functions like mean, median, refer here.

Chapter 2

Read and Write Text Files

– Hannah Yamada and Go Hirochi

This chapter explains the method to read and write a free format text file and extract the text data.

2-1 Read Text File

This section explains the procedure to read a free format text file (ASCII data). To read excel data, save the file as 'filename.csv' or 'filename.txt'. When reading the file, it is often necessary to state what is separating each element in the data (called "delimiter"), e.g., tab, break, or comma.

- Read text file as "array":

  ```
  >>> a=loadtxt('filename', delimiter=',')
  ```

- Read text file as "list":

  ```
  >>> a=file('filename').readlines()
  ```

  ✓file('filename') means the file itself is stored as an object. The contents is not read yet.

  ✓readlines()- Reads contents (each line) of the file as a list. It also reads the "delimiters" together.

- Read text file as "string":

  ```
  >>> a=file('filename').read()
  ```

  ✓read()- All the elements in the file are read as one string data.

- Other options:

  >>> a=open('filename')

  ✓Same as “file”.

  >>> type(a)

  ✓Check the data type of variable.

2-2 Modify Text Data

Here are some examples of functions to modify the data. We will use a list a=[1,2,3,4,5,6,7,8,9,10] as an example.

- Check the dimension of the data:

  >>> a.shape

  ✓for array.

  >>> shape(a)

  ✓for list.

  ✓Note that the order is number of rows (longitudinal direction↓), number of columns (lateral direction→) for 2-dimensional arrays in python.

- Change the dimension of the data:

  >>> b=a.reshape(2,5)

  ✓can be used in arrays only.

  >>> b=array([[1,2,3,4,5],[6,7,8,9,10]])

  >>> b=reshape(a,(2,5))

  ✓can be used for both array and list. List is converted to array by using this function.

  >>> b=a.reshape(-1,5)

  ✓By using the ‘-1’ flag, the first dimension is automatically set to match the total size of the array. For e.g., if there are 10 elements in an array/list and 5 columns is specified during reshape, number of rows is automatically calculated as 2. The shape will be (2,5).

- Convert the data type to array:

  >>> b=array(a)

- Convert the data type to list:

  `>>> b=a.tolist()`

- Convert the data into float and integer:

  `>>> float(a[0])`

  `>>> int(a[0])`

  ✓these functions can only be used for one element at a time.

- Drop the '\n' sign at the end of each line:
  - When we read text file with a=file('filename').readlines(), there is the sign '\n' at the end of each row. This sign is used to indicate the end of line in ASCII encoding.
  - strip() function is used in python to remove these characters from the data read from a ASCII text file.
  - For e.g., if the data from a text file is like,

    ```
    1
    4
    6
    8
    12
    ```

  - After reading it with a=file('filename').readlines(), the data will have '\n' as,

    ```
    ['1\n',
    '4\n',
    '6\n',
    '8\n',
    '12\n']
    ```

  - To drop the '\n' for each element:

    `>>> b=[s.strip() for s in a]`
  - Furthermore, to convert each element into float:

    `>>> b=[float(s.strip()) for s in a]`

– After this adjustment, list b will be as shown

[1.0,
4.0,
6.0,
8.0,
12.0]

2-3 Extract Data

This section explains how to extract data from an array or a list. The following process can be used to take data for a region from global data, or for a limited period from long time series data. The process is called 'slicing' as explained in Chapter 1 also, and is accomplished by using index.

2-3-1 Extract from 1-D Data

Same method can be used for arrays and lists. Let's consider the following list,

```
>>> a=[1,2,3,4,5]
```

✓There are five items in the list.

Index Basics

Indexing can be carried out in two ways:

1. Positive Index: The counting order is from left to right. The index for the first element is 0 (not 1).

   ```
   >>> a[0]
   >>> 1
   >>> a[1]
   >>> 2
   >>> a[4]
   >>> 5
   ```

   ✓The fifth item (index=4) is 5.

2. Negative Index: The counting order is from right to left. The index for the last item is -1. In some cases, the list is very long and it is much easier to count from the end rather than the beginning.

   ```
   >>> a[-1]
   >>> 5
   ```

   ✓It is same as a[4] as shown above.

   ```
   >>> a[-2]
   >>> 4
   ```

Data Extraction

Data extraction is carried out by using indices. In this section, some examples of using indices are provided. Details of array indexing and slicing can be found here.

1. Using two indices:

   ```
   >>> somelist[first index:last index]
   >>> a[0:2]
   >>> [1,2]
   ```

   ✓a[0] and a[1] are included but a[2] is not included.

   ```
   >>> a[3:4]
   >>> 4
   ```

2. Using single index:

   ```
   >>> a[:2]
   ```

   ✓same as a[0:2].

   ```
   >>> [1,2]
   >>> a[2:]
   ```

   ✓same as a[2:5].

   ```
   >>> [3,4,5]
   ```

2-3-2 Extract from 2-D Data

Different method for array and list as indexing is different in two cases as explained below.

- Consider a 2-D list and 2-D array,

  ```
  >>> a_list=[[1,2,3],[4,5,6]]
  >>> a_array=array([[1,2,3],[4,5,6]])
  ```

- The shape of data can be found as,

  ```
  >>> shape(a_list)
  >>> (2,3)
  >>> a_array.shape
  >>> (2,3)
  ```

- The items can be accessed by using index as,

  ```
  >>> a_list[0]
  >>> [1,2,3]
  ```

  ✓which is a list.

  ```
  >>> a_array[0]
  >>> array([1,2,3])
  ```

  ✓which is an array.

- To extract data from list,

  ```
  >>> a_list[0][1]
  >>> 2
  >>> a_list[1][:2]
  >>> [4,5]
  ```

  ✓The index has to be provided in two different sets of square brackets "[ ]".

- To extract data from array,

  ```
  >>> a_array[0,1]
  >>> 2
  >>> a_array[1,:2]
  ```

```
>>> [4,5]
```

✓The index can be provided is one set of square brackets "[ ]".

2-3-3 Extract from more than 3-D Data

Similar to 2-D data, the indexing is different for array and list.

- Consider a 3-D list and 3-D array,

```
>>> a_list=[[[2,3],[4,5],[6,7],[8,9]],[[12, 13],[14,15],[16,17],[18,19]]]
>>> a_array=array([[[2,3],[4,5],[6,7],[8,9]],[[12, 13],[14,15],[16,17],[18,19]]])
```

- The shape of both data is (2,4,2).
- To extract from list,

```
>>> a_list[0][2]
>>> [6,7]
>>> a_list[0][2][1]
>>> 6
```

- To extract from array,

```
>>> a_array[0,2]
>>> array([6,7])
>>> a_array[0,2,1]
>>> 6
```

2-4 Save Text File

- If we want to be able to view the file easily, we can save the data into a text file.
- To save variable "a",

```
>>> savetxt('filename',a)
```

Chapter 3

Manipulating Binary Data

– Sayaka Yoshikawa and Tipaporn Homdee

This chapter explains the procedure of reading binary data (arrays), extracting data, using built-in functions, as well as and saving data to a binary file.

3-1 Read Binary Data

Python can directly read the binary data as,

```
>>> fromfile('filename','type code'):
```

- fromfile() reads binary file and stores data as an array.
- type code: can be defined as type code (e.g., 'f') or python type (e.g., 'float') as shown in Table 3-1. It determines the size and byte-order of items in the binary file.

```
>>> Ir=fromfile('/home/sayajo/irrigation/Newirr0.5_2000_km2.hlf','f')
>>> Ir
>>> array([ 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
>>> Ir.shape
>>> (252900,)
```

✓'Newirr0.5_2000_km2.hlf' is global data of 0.5° resolution (360 rows and 720 columns) in the binary format. But when the binary file is read in python using fromfile, the data is stored as 1-D array of size 360*720 (=252900) as shown above. To convert the 1-D array to 2-D array, use

```
>>> Ir=Ir.reshape(360,720)
```

Table 3-1: Data type of the returned array

Type code	C Type	Python Type	Minimum size in bytes
'c'	char	character	1
'b'	signed char	int	1
'B'	unsigned char	int	1
'u'	Py_UNICODE	Unicode character	2
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	2
'I'	unsigned int	long	2
'l'	signed long	int	4
'L'	unsigned long	long	4
'f'	float	float	4
'd'	double	float	8

3-2 Extracting Data

The basic extracting, adding and searching the data are same as that explained in chapter 2. Later, a more realistic example of extracting data is described in section 3-5.

3-3 Use of Built-in Functions

The Python interpreter has a number of functions built into it. This section documents the Python's built-in functions in easy-to-use order. Firstly, consider the following 2-D arrays,

```
>>> A=array([[-2, 2], [-5, 5]])
>>> B=array([[2, 2], [5, 5]])
>>> C=array([[2.53, 2.5556], [5.3678, 5.4568]])
```

3-3-1 Mathematical Functions

1. max(iterable): Returns the maximum from the passed elements or if a single iterable is passed, the max element in the iterable. With two or more arguments, return the largest value.

   ```
   >>> max([0,10,15,30,100,-5])
   >>> 100
   ```

```
>>> A.max()
>>> 5
```

2. min(iterable): Returns the minimum from the passed elements or if a single iterable is passed, the minimum element in the iterable. With two or more arguments, return the smallest value.

```
>>> min([0,10,15,30,100,-5])
>>> -5
>>> A.min()
>>> -5
```

3. mean(iterable): Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. For details, click here.

```
>>> mean ([0,10,15,30,100,-5])
>>> 75
>>> A.mean()
>>> 0.0
```

4. sum(iterable): Returns the sum of the array elements. It returns sum of array elements over an axis if axis is specified else sum of all elements. For details, click here.

```
>>> sum([1,2,3,4])
>>> 10
>>> A.sum()
>>> 0
```

5. abs(A): Returns the absolute value of a number, which can be an integer or a float, or an entire array.

```
>>> abs(A)
>>> array([[2,2],[5,5]])
>>> abs (B)
>>> array([2, 2],[5, 5])
```

6. divmod(x,y): Returns the quotient and remainder resulting from dividing the first argument (some number x or an array) by the second (some number y or an array).

   >>> divmod(2, 3)

   >>> (0, 2)

   ✓as 2 / 3 = 0 and remainder is 2.

   >>> divmod(4, 2)

   >>> (2, 0)

   ✓as 4 / 2 = 2 and remainder is 0.

   In case of two dimensional array data

   >>> divmod(A,B)

   >>> (array([[-1, 1], [-1, 1]]), array([[0, 0], [0, 0]]))

7. modulo (x%y): Returns the remainder of a division of x by y.

   >>> 5%2

   >>> 1

8. pow(x,y[, z]): Returns x to the power y. But, if z is present, returns x to the power y modulo z (more efficient than pow(x, y) % z). The pow(x, y) is equivalent to x**y.

   >>> pow(A,B)

   >>> array([[4, 4], [-3125, 3125]])

9. round(x,n): Returns the floating point value of x rounded to n digits after the decimal point.

   >>> round(2.675,2)

   >>> 2.67

10. around(A,n): Returns the floating point array A rounded to n digits after the decimal point.

    >>> around(C,2)

    >>> array([[2.53, 2.56], [5.37, 5.46]])

11. range([x],y[,z]) : This function creates lists of integers in an arithmetic progression. It is primarily used in for loops. The arguments must be plain integers.

    - If the step argument is omitted, it defaults to 1.
    - If the start argument (x) is omitted, it defaults to 0.

- The full form returns a list of plain integers [x, x + z, x + 2*z, ···,y-z].
- If step (z) is positive, the last element is the 'start (x) + i * step (z)' just less than 'y'.
- If step (z) is negative, the last element is the 'start (x) + i * step (z)' just greater than 'y'.
- If step (z) is zero, ValueError is raised.

```
>>> range(10)
>>> [0,1,2,3,4,5,6,7,8,9]
>>> range(1,11)
>>> [1,2,3,4,5,6,7,8,9,10]
>>> range(0,20,5)
>>> [0,5,10,15]
>>> range(0,-5,-1)
>>> [0,-1,-2,-3,-4,]
>>> range(0)
>>> [ ]
```

12. arange(x,y[,z]) : This function creates arrays of integers in an arithmetic progression. Same as in range().

```
>>> arange(10)
>>> array([0,1,2,3,4,5,6,7,8,9])
>>> arange(1,11)
>>> array([1,2,3,4,5,6,7,8,9,10])
>>> arange(0,20,5)
>>> array([0,5,10,15])
>>> arange(0,-5,-1)
>>> array([0,-1,-2,-3,-4])
>>> arange(0)
>>> array([], dtype=int64)
```

13. zip(A,B): Returns a list of tuples, where each tuple contains a pair of ith element of each argument sequences. The returned list is truncated to length of shortest sequence. For a single sequence argument, it returns a list with 1 tuple. With no arguments, it returns an empty list.

```
>>> zip(A,B)
>>> [(array([-2, 2]), array([2, 2])), (array([-5, 5]), array([5, 5]))]
```

14. sort(): Sorts the array elements in smallest to largest order.

    ```
    >>> D=array([10,2,3,10,100,54])
    >>> D.sort()
    >>> D
    >>> array([2, 3, 10, 10, 54, 100])
    ```

15. ravel(): Returns a flattened array. 2-D array is converted to 1-D array.

    ```
    >>> A.ravel()
    >>> array([-2, 2, -5, 5])
    ```

16. transpose(): Returns the transpose of an array (matrix) by permuting the dimensions.

    ```
    >>> A.transpose()
    >>> array([[-2, -5], [ 2, 5]])
    ```

17. diagonal(): Returns diagonal matrixs for pecified diagonals.

    ```
    >>> A.digonal()
    >>> array([-2, 5])
    ```

3-3-2 Other Useful Functions

1. astype('type code'): Returns an array with the same elements coerced to the type indicated by type code in Table 3-1. It is useful to save data as some type.

   ```
   >>> A.astype('f')
   >>> array([[-2., 2.],[-5., 5.]])
   ```

2. tolist(): Converts the array to an ordinary list with the same items.

   ```
   >>> A.tolist()
   >>> [[-2, 2], [-5, 5]]
   ```

3. byteswap(): Swaps the bytes in an array and returns the byteswapped array. If the first argument is 'True', it byteswaps and returns all items of the array in-place. Supported byte sizes are 1, 2, 4, or 8. It is useful when reading data from a file written on a machine with a different byte order. For details on machine dependency, refer this. To convert data from big endian to little endian or vice-versa, add byteswap() in same line where 'fromfile' is used. If your data is made by big endian (ex. H08 model's output), you don't need to put it.

3-4 Write Binary Data

- tofile('filename'): Write/save all items (as machine values) to the file object 'filename'. It is better to know the endianness of saved data so that it can be used while reading data using fromfile().
- If you want to save binary data as big endian type,

  ```
  >>> A.tofile('A.bin')
  ```
- If you want to save binary data as little endian type and floating data,

  ```
  >>> A.astype('f').byteswap().tofile('A.bin')
  ```

3-5 Exercise

Here, an example is presented showing the procedure to extract data around India (latitude: 3 35°N, Longitude: 60 ~ 95°E) from global data and plot it using python.

- Data name: Historical Irrigation area data
  - Spatial resolution: 0.5° (row: 360 × column: 720)
  - Periods: 1900-2003 (yearly)
  - Filename: /home/sayajo/irrigation/Newirr1900-2003_pct.hlf
  - Data type: binary
  - Data type code: float32
  - Endian type: little endian
- Sample Answer:

  ```
  >>> IR=('/home/sayajo/irrigation/Newirr1900-2003_km2.hlf','f').byteswap().reshape(-1,360,720)
  ```

  ```
  >>> IR
  ```

  ```
  array ([[[0.  0.  0.  ...  0.  0.  0.]
           [0.  0.  0.  ...  0.  0.  0.]
           [0.  0.  0.  ...  0.  0.  0.]
           ...
           [0.  0.  0.  ...  0.  0.  0.]
           [0.  0.  0.  ...  0.  0.  0.]
           [0.  0.  0.  ...  0.  0.  0.]]]  ,dtype = float32)
  ```

```
>>> IR.shape
>>> (104,360,720)
```

✓has 104 time steps (years) and 360 rows and 720 columns in each time step.

- To find the difference between two data (Differences between 1900 and 2000)

```
>>> IR[100]-IR[0]

        array  ([[[0.  0.  0.  ...  0.  0.   0.]
                  [0.  0.  0.  ...  0.  0.   0.]
                  [0.  0.  0.  ...  0.  0.   0.]
                  ...
                  [0.  0.  0.  ...  0.  0.   0.]
                  [0.  0.  0.  ...  0.  0.   0.]
                  [0.  0.  0.  ...  0.  0.   0.]]]  , dtype = float32)
```

- Extraction of data in Indian area
  - Get the index of co-ordinates for India

    ```
    >>> sLon=-180; sLat=90
    ```
    ✓Define the origin of global data.
    ```
    >>> LON=arrange(sLon,sLon+0.5*720,0.5)
    ```
    ✓make an array of longitude of each grid.
    ```
    >>> LAT=arrange(sLat,sLat-0.5*360,-0.5)
    ```
    ✓make an array of latitude of each grid.
    ```
    >>> LAT.tolist().index(3)
    ```
    ✓Get the index for latitude = 3° N.
    ```
    >>> 174
    >>> LAT.tolist().index(35)
    ```
    ✓Get the index for latitude = 35° N.
    ```
    >>> 110
    >>> LON.tolist().index(60)
    ```
    ✓Get the index for longitude = 60° E.
    ```
    >>> 480
    >>> LON.tolist().index(95)
    ```
    ✓Get the index for longitude = 95° E.
    ```
    >>> 550
    ```

– Extract the data using the index obtained above.

```
>>> India=IR[100,110:174,480:550]
```

✓100 means 100th time step.

```
array  ([[1.21320999e + 02   6.97241028e + 02   1.28660995e + 02   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]
         [0.00000000e + 00   2.58078003e + 01   1.48645996e + 02   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]
         [2.97949009e + 01   7.49999983e − 03   0.00000000e + 00   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]
                ...
         [0.00000000e + 00   0.00000000e + 00   0.00000000e + 00   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]
         [0.00000000e + 00   0.00000000e + 00   0.00000000e + 00   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]
         [0.00000000e + 00   0.00000000e + 00   0.00000000e + 00   ...
          0.00000000e + 00   0.00000000e + 00   0.00000000e + 00]]  , dtype = float32)
```

```
>>> India.max()
>>> 2559.8201
```

✓Maximum area: 2559.82 km^2

```
>>> India.sum()
>>> 788900.38
```

✓Total area: 788900.38 km^2

```
>>> figure();imshow(India)
```

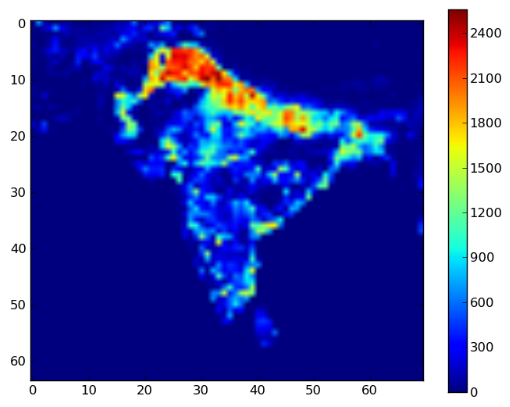


Figure 3-1: Irrigation area around India in 2000

If you can obtain Fig. 3-1, you are successful! ⋆Congratulations⋆

Chapter 4

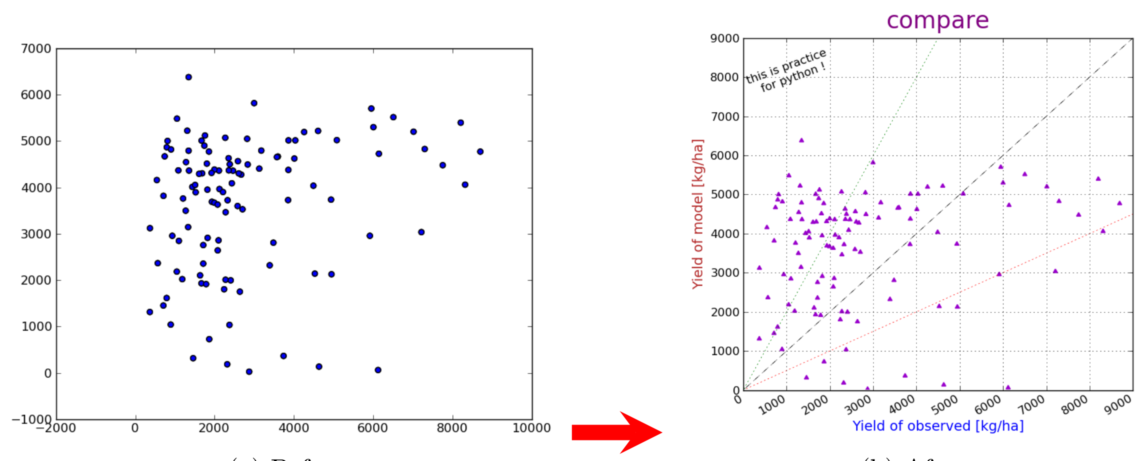
Plotting Figures

– Kensuke Hagiwara and Hiroyuki Ishida

This chapter provides some examples of plotting using Python. The matplotlib library of python is a powerful plotting tool based on plotting library of MATLAB. Various types of figures can be drawn using matplotlib. For comprehensive set of examples with source code used to plot the figure, click here.

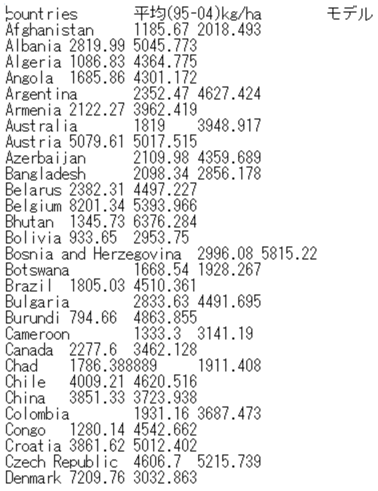
4-1 Basic Plot

Figure 4-1 compares observed yield and model simulated yield. The objective of this section is to turn Fig. 4-1a into Fig. 4-1b.



(a) Before (b) After

Figure 4-1: Original and modified figures



```
countries	平均(95-04)kg/ha	モデル
Afghanistan	1185.67 2018.493
Albania 2819.99 5045.773
Algeria 1086.83 4364.775
Angola  1685.86 4301.172
Argentina	2352.47 4627.424
Armenia 2122.27 3962.419
Australia	1819	3948.917
Austria 5079.61 5017.515
Azerbaijan	2109.98 4359.689
Bangladesh	2098.34 2856.178
Belarus 2382.31 4497.227
Belgium 8201.34 5393.966
Bhutan  1345.73 6376.284
Bolivia 933.65  2953.75
Bosnia and Herzegovina  2996.08 5815.22
Botswana	1668.54 1928.267
Brazil  1805.03 4510.361
Bulgaria	2833.63 4491.695
Burundi 794.66  4863.855
Cameroon	1333.3  3141.19
Canada  2277.6  3462.128
Chad    1786.388889	1911.408
Chile   4009.21 4620.516
China   3851.33 3723.938
Colombia	1931.16 3687.473
Congo   1280.14 4542.662
Croatia 3861.62 5012.402
Czech Republic  4606.7  5215.739
Denmark 7209.76 3032.863
```

Figure 4-2: Basic data

Reading Data

1. The original data is in text format as shown in Fig. 4-2. Procedure for reading the text data is explained in chapter 2.

   - There are two wheat yields data. One is observed data, the other is model output.
   - The observed data is saved to variable 'FAOstat' and model output is saved to 'Model'.
   - Both are composed of 111 countries data.

     >>> shape(FAOstat) = (111,)

     >>> shape(Model) = (111,)

Default Basic Plot

Here is the process to make a simple scatter plot with default settings.

>>> fig=figure()

>>> scatter(Model, FAOstat)

✓We can obtain Fig 4-3 but it's too simple as default settings are used.

>>> draw()

✓The figure appears.

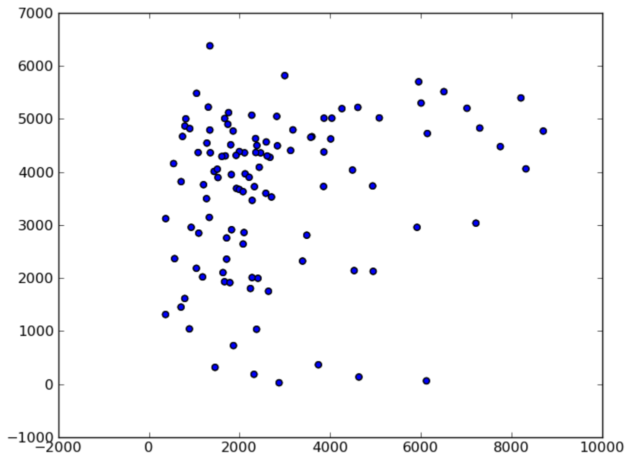


Figure 4-3: Basic plot

Setting Axis Limits

The axis limits can be set by using xlim() and ylim() as:

```
>>> xlim(0,9000)
>>> ylim(0,9000)
```

✓Change values of x-axis and y-axis. In this case, 0 to 9000 as shown in Fig. 4-4.

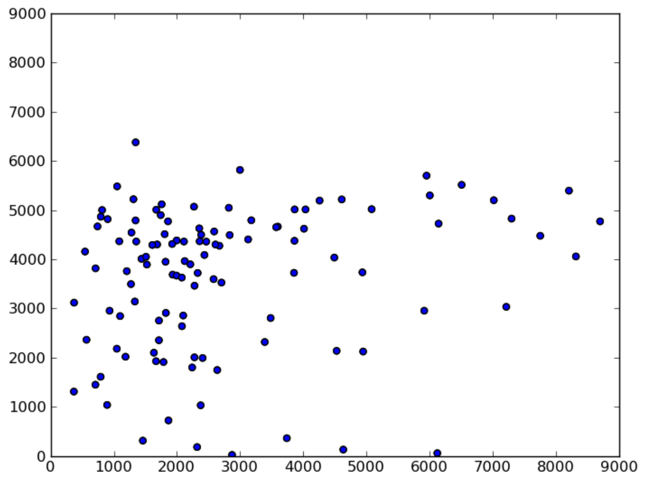


Figure 4-4: Axis limits

Setting Title and Axis Labels

The title of the figure as well as the x and y axes can be set as:

>>> title('compare',color='purple',fontsize=24)

✓puts title of figure.

✓Also, specify font color and size. For color, use color= some color name such 'red' or color= some hexadecimal color code such as '#0000FF'. For font size, use fontsize=number (number is > 0).

>>> grid()

✓turns the grids on or off.

>>> xlabel('Yield of observed [kg/ha]', color='blue',fontsize=14)

✓puts title of x-axis.

>>> ylabel('Yield of model [kg/ha]', color='firebrick',fontsize=14)

✓puts title of y-axis.

The modified figure is presented in Fig. 4-5.

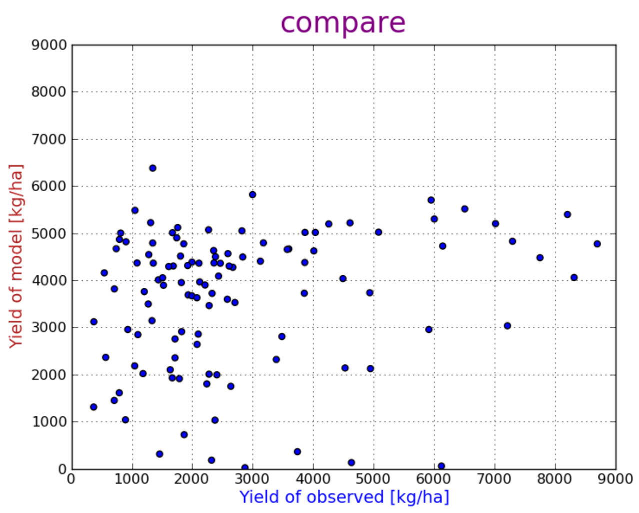


Figure 4-5: Title and labels

Draw Lines over the Plot

Lines can be drawn over the plot in many ways. Here, an example of using arrow() is provided.

>>> arrow(0,0,9000,9000,linestyle='dashed',linewidth=0.3)

>>> arrow(0,0,4500,9000,linestyle='dotted',linewidth=0.3,color='green')

>>> arrow(0,0,9000,4500,linestyle='dotted',linewidth=0.3,color='red')

✓"Arrow(a1,a2,a3,a4)" is used for plot a line in the figure.

✓(x,y)=(a1 ,a2) co-ordinate of start point.

✓(x,y)=(a1+a3 ,a2+a4) co-ordinate of end point.

✓ “Linestyle” specifies line type. Default is 'solid'.

✓ “Linewidth” specifies width of the line. Use as, linewidth = number (number is > 0).

Check Fig. 4-6 for the newly added lines.

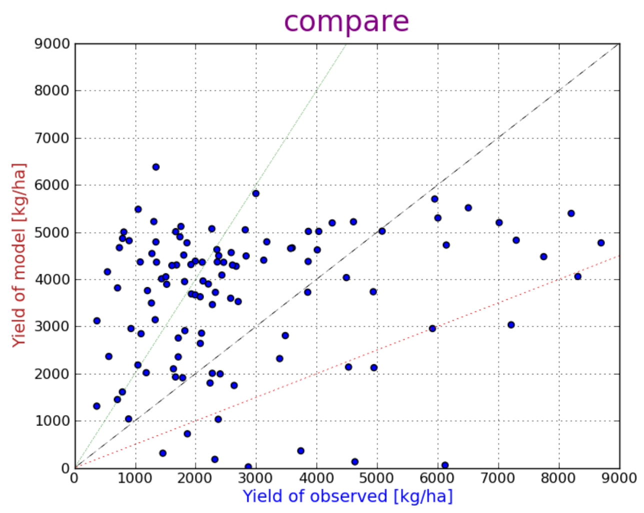


Figure 4-6: Draw lines

Add Text to Figure

Some text can be added to figure by specifying the co-ordinates of the point.

>>> figtext(0.13,0.78, 'this is practice \n for python !', rotation=20)

✓ “\n” indicates the end of line. So, anything after it will be written in new line.

✓ “figtext(a1,a2,'sentence')”.

⋆ a1 and a2 means location of the sentence. $0 \leq a1, a2 \leq 1$.

⋆ (a1,a2)=(0,0) is bottom left.

⋆ (a1,a2)=(1,1) is upper right.

⋆ (a1,a2)=(0.5,0.5) is center.

⋆ Rotation: rotation = number. Number is angle (in this case, 20°).

See Fig. 4-7 for newly added text.

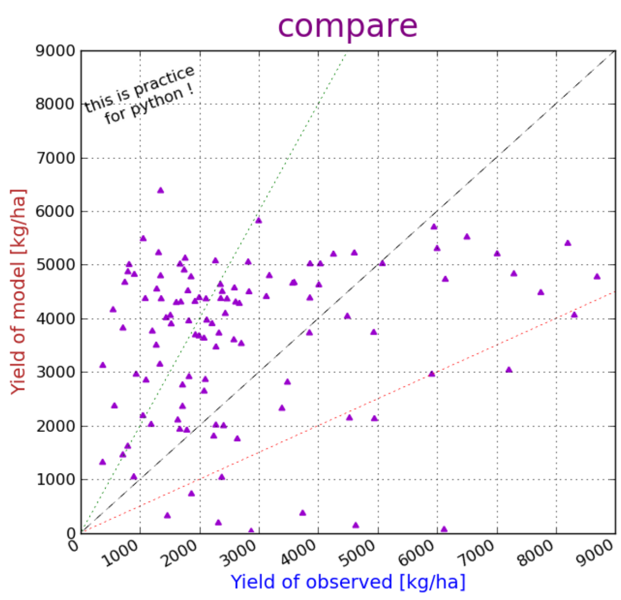


Figure 4-7: Adding text to figure

Additional Options

Here, some important options related to commands used in making a figure are briefly introduced.

- A new figure can be opened with

  >>> fig=fig(1,figsize=(number in inches,number in inches))

  ✓The size of the figure can be specified.

- scatter() also has various options as

  >>> scatter(Model, FAOstat, marker='∧',color='#9900CC'),

  ✓change the marker style and its color (check Table 4-1 for most of the available options).

- If you want to write text like $\frac{kg}{ha}$ instead of kg/ha,

  >>> ylabel('$\\frac{kg}{ha}$')

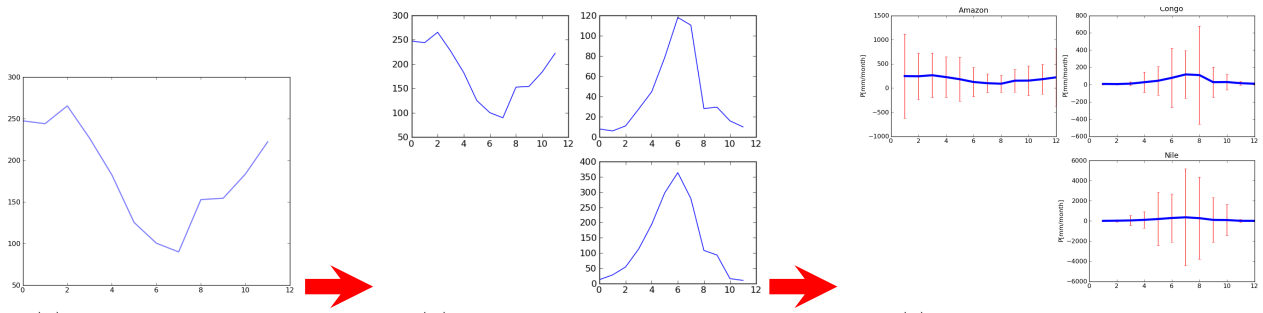
  ✓For details on writing mathematical expressions in python, refer here.

Table 4-1: Line and marker styles

Line style		Marker style	
Linestyle	Lines	Marker	Signs
'Solid'	—	'o'	Circle
'Dashed'	– –	'v'	Triangle_down
'Dotted'	· · ·	'∧'	Triangle_up
		'<'	Triangle_left
		'>'	Triangle_right
		's'	Square
		'h'	Hexagon
		'+'	Plus
		'x'	X
		'd'	Diamond
		'p'	pentagon

4-2 Making Subplots

One figure can have many small plots in it. The small plots are known as subplots. The obejective of this section is to arrange plots in one figure as shown in Fig. 4-8. Also, additional customization of the figure are also provided.



(a) 3 figures like this (b) Put together in 1 figure (c) Modified with Errorbars

Figure 4-8: Making many plots in one figure

Reading Data

The original data used in making the figure is explained here.

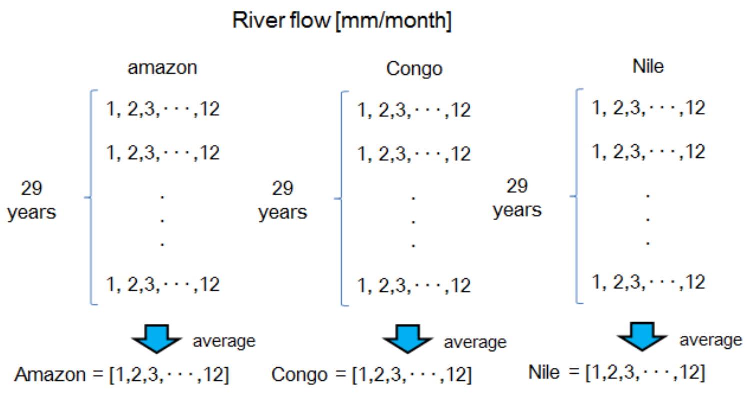


Figure 4-9: Sample data used in making subplots

- The data are monthly river flow data of 3 rivers for 29 years as shown in Fig. 4-9.
- The data for each river is saved to variable "rivername".
- There are data for 3 rivers. Shape of each river data is (29, 12); 29 rows is for 29 years and 12 columns is for 12 months of each year.
- The data is reshaped to get average values for each month (for e.g., average of data in January of each of the 29 years) as,

```
>>> Amazon=Amazon.mean(0)
>>> shape(Amazon)
>>> (12,)
>>> Congo=Congo.mean(0)
>>> shape(Congo)
>>> (12,)
>>> Nile=Nile.mean(0)
>>> shape(Nile)
>>> (12,)
```

Making 3 Subplots

Here basic subplots for three rivers are plotted. The procedure to prepare Fig. 4-10 is as follows.

```
>>> fig=figure()
>>> AX1=fig.add_subplot(221)
```

✓to arrange 3 revers data in one figure, we use subplot(a1a2a3) or subplot (a1,a2,a3).

- ⋆ a1 means a number of columns
- ⋆ a2 means a number of rows
- ⋆ a3 means location of figure.
- ⋆ a3 must be less than a1 × a2

```
>>> AX2=fig.add_subplot(222)
>>> AX3=fig.add_subplot(224)
>>> L1=AX1.plot(Amazon, linewidth=2,color='B',alpha=0.5)
>>> L2=AX2.plot(Congo, linewidth=2, color='B',alpha=0.5)
>>> L3=AX3.plot(Nile, linewidth=2,color='B',alpha=0.5)
>>> draw()
```

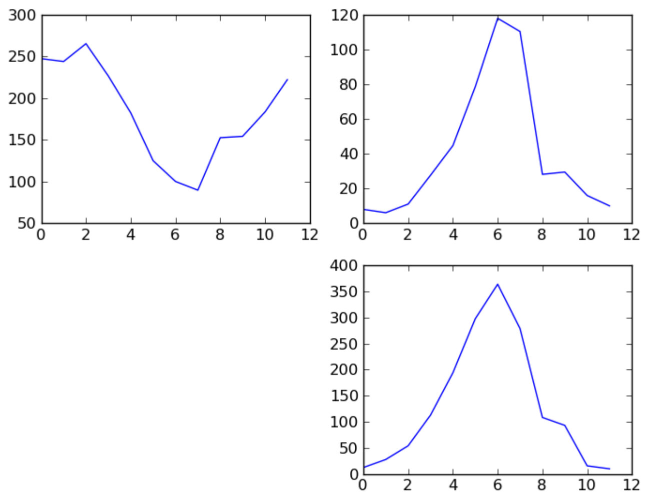


Figure 4-10: Three basic subplots

Modify Subplots

Each subplot can be modified as a separate figure.

```
>>> AX1.set_title('Amazon')
>>> AX2.set_title('Congo')
>>> AX3.set_title('Nile')
>>> AX1.set_ylim(0,400)
>>> AX2.set_ylim(0,400)
>>> AX3.set_ylim(0,400)
>>> AX1.set_ylabel('P[mm/month]')
>>> AX2.set_ylabel('P[mm/month]')
>>> AX3.set_ylabel('P[mm/month]')
```

✓To add information into the figure, use “X.set_Y”.

⋆ X is the subplot which you want to modify. In this case, AX1, AX2, and AX3 are X. Y is the property to be added or modified.

The modified subplots are shown in Fig. 4-11.

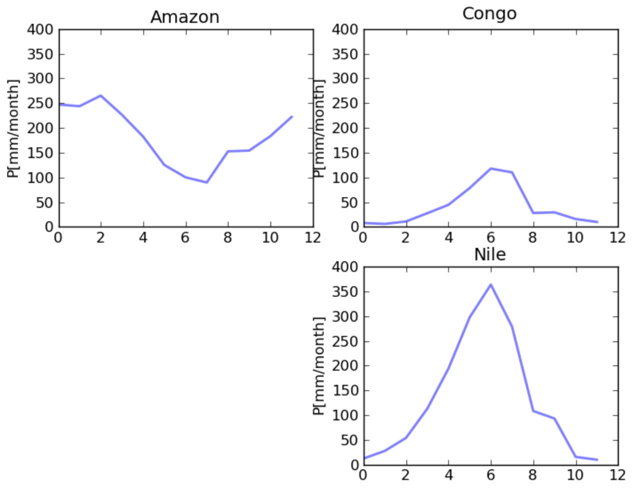


Figure 4-11: Setting limits and adding titles to each subplot

Add Errorbars

Errorbars can be added to line plots.

```
>>> v1=AmazonRiv.var(0)
>>> v2=AmazonRiv.var(0)
>>> v3=AmazonRiv.var(0)
>>> x = arange(1,13)
>>> L1=AX1.errorbar(x,Amazon,v1,color='B',linewidth=4,ecolor='R',elinewidth=1)
>>> L2=AX3.errorbar(x,Congo,v1,color='B',linewidth=4,ecolor='R',elinewidth=1)
>>> L3=AX3.errorbar(x,Nile,v1,color='B',linewidth=4,ecolor='R',elinewidth=1)
```

✓errorbar(a1,b1,c1). a1 is x axis. b1 is y values. c1 is attribute (in this case, c1 is variance).

✓ecolor specifies color of errorbars.

✓elinewidth specifies linewidth of errorbars.

The final figure is shown in Fig. 4-12.

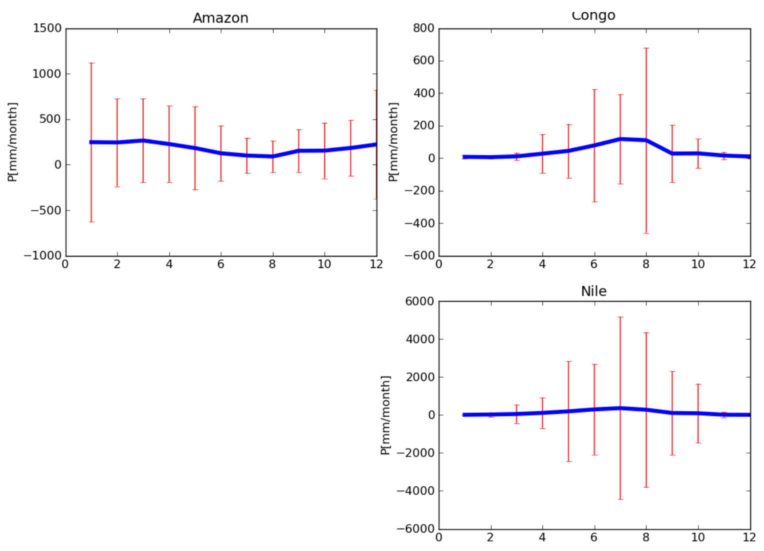


Figure 4-12: Adding errorbars to subplots

Chapter 5

Plotting 2-dimensional Maps

– Yoshihiko Iseri and Orie Sasaki

This chapter is divided into following sections.

1. Section 5-1 explains how to draw 2-dimensional contour plot.
   (a) Preparation of initial data for contour plot (section 5-1-1).
   (b) Drawing 2-dimensional contour plot (section 5-1-2).
   (c) Editing lines and labels in the contour plot (section 5-1-3 and section 5-1-3 respectively).
   (d) Editing colorbar and colormap (section 5-1-5 and section 5-1-6 respectively).
2. Section 5-2 describes how to make a global map.
3. Section 5-3 explains how to draw contours over a global map.

5-1 Drawing 2-D Contour Plot

This section explains the procedure to draw a 2-dimensional contour plot and change its appearance properties such as labels, lines, and colorbar. The data used to plot the figure is presented in section 5-1-1. After defining the data matrixes (X, Y and D) and plotting contours, section 5-1-3-section 5-1-6 are optional. These steps help to modify the plot and make it more clear or visible. It is not necessary to perform all these steps, for instance, section 5-1-4 can be performed even if section 5-1-3 is skipped.

5-1-1 Initial Data

Suppose the contour plot for the following 5 × 5 array data (call it D) is desired. The locations of each component of D on x-y plane are defined by matrix X and Y.
You can create array (matrix X, Y and D) shown in Fig. 5-1 by using the python code shown in Table 5-1.

Table 5-1: Sample code to create a 2-D array

```
D=zeros([5,5])
for i in range(0,3):
        D[0:3-i,0:3-i]=3-i
        D[2+i::,2+i::]=3+i
x=arange(1,6)
y=arange(5,0,-1)
X, Y=meshgrid(x,y)
```

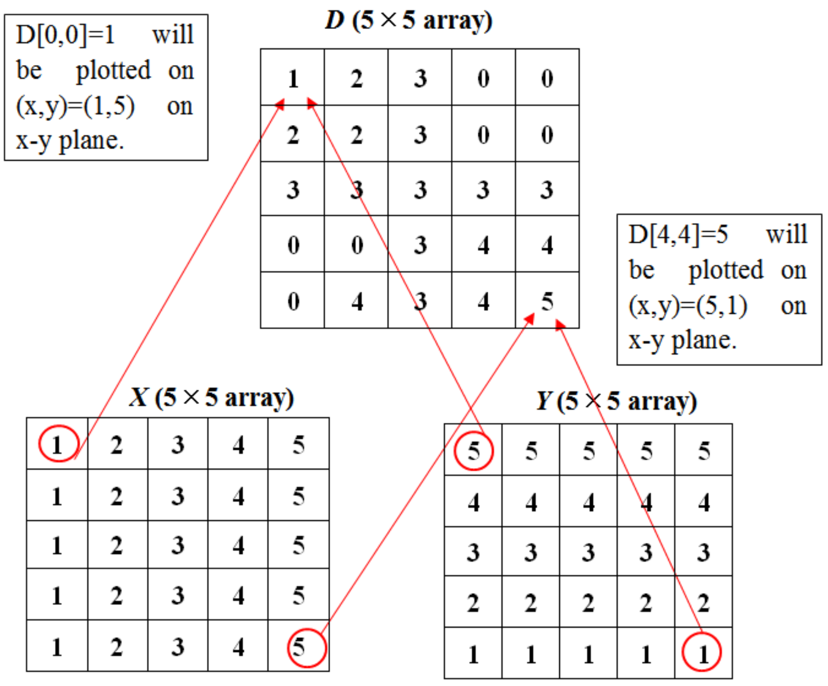


Figure 5-1: Basic 2-D data

5-1-2 Drawing Contours

Suppose there are matrixes X, Y and D, which are defined in section 5-1-1.

- Following draws contour plot of D against the x-y plane specified by X and Y:

  ```
  >>> contour(X,Y,D)
  ```

- To specify the contour levels:

  ```
  >>> V=arange(0,5,0.5)
  >>> contour(X,Y,D,V)
  ```

  ✓contour lines are drawn for every 0.5 step from 0 to 4.5

As of now, the plot will look like Fig. 5-2.

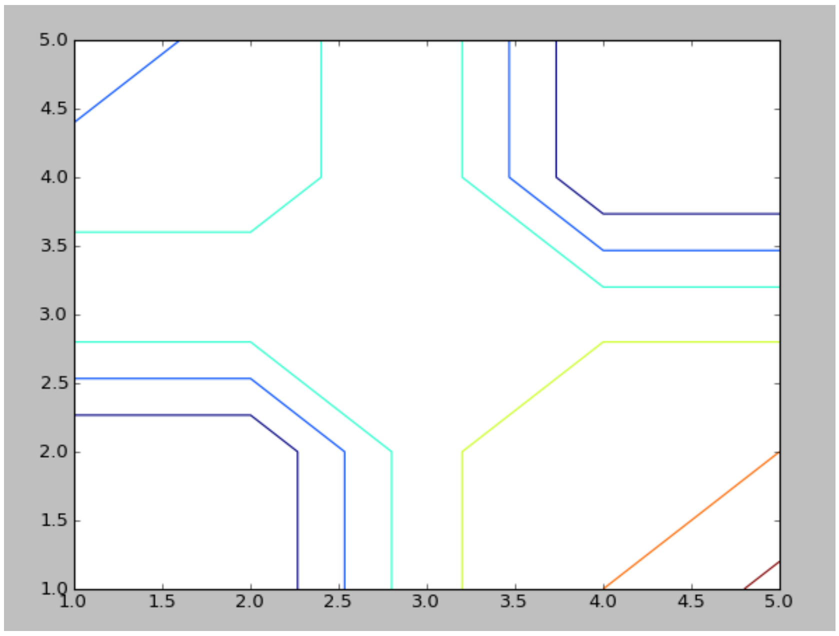


Figure 5-2: Basic contour plot

5-1-3 Editing Lines in Contour Plot

- To specify the width of contour lines:

  ```
  >>> contour(X,Y,D,linewidth=5)
  ```

  ✓width of each line is 5. It can be any value > 0.

- To draw all contour lines with the same color:

  >>> contour(X,Y,D,colors='r')

  ✓all line is drawn with red. Even hexadecimal color codes can be used. 'r' can be replaced by '#ff0000' (hexadecimal code for red). For a list of all the available hexadecimal colors that can be used, click here.

- To change the linestyle:

  >>> contour(X,Y,D,linestyles='dashed')

  ✓dashed (−−−) lines. You can select following linestyles: 'solid'(—), 'dashed', 'dashdot'(-.-), 'dotted'(···).

5-1-4 Editing Labels in Contour Plot

- To draw labels for contour lines:

  >>> cs = contour(X,Y,D)

  >>> clabel(cs, fontsize=12)

  ✓fontsize of labels is 12. It can be any value > 0.

- To draw labels for specified levels CL:

  >>> V=arange(0,5,0.5)

  >>> CL=arange(0,6,2)

  >>> cs=contour(X,Y,D,V)

  ✓contour lines are drawn for [0,1,2,3,4,5].

  >>> clabel(cs,fontsize=12,CL)

  ✓contour labels of fontsize 12 are written for [0,2,4].

More details of editing the labels can be found here.

5-1-5 Editing Colorbars in Contour Plot

- To specify orientation of colorbar,

  >>> colorbar()

  ✓default orientation is vertical colorbar on right side of the main plot.

  >>> colorbar(orientation='h')

  ✓will make a horizontal colorbar below the main plot.

- To specify the area fraction of total plot area occupied by colorbar:

  >>> colorbar()

  ✓default fraction is 0.15 (see Fig. 5-3a).

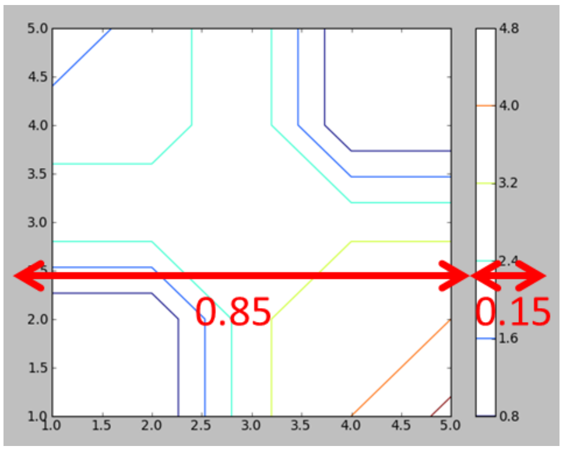
  >>> colorbar(fraction=0.5)

  ✓50% of the plot area is used by colorbar (see Fig. 5-3b).

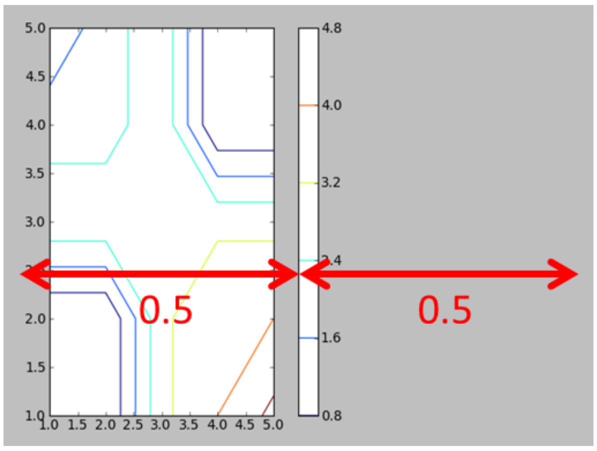
- To specify the ratio of length to width of colorbar:

  >>> colorbar(aspect=20)

  ✓length:width = 20:1.



(a) Large Fraction



(b) Small Fraction

Figure 5-3: Colorbar fraction of total plot area

More details of the options for colorbar can be found here.

5-1-6 Editing Colormap in Contour Plot

- To specify colormap for 2-D maps:

  >>> imshow(D,cmap=cm.colormap)

  ✓map are drawn by colors of selected 'colormap'.

  >>> contour(X,Y,D,cmap=cm.colormap)

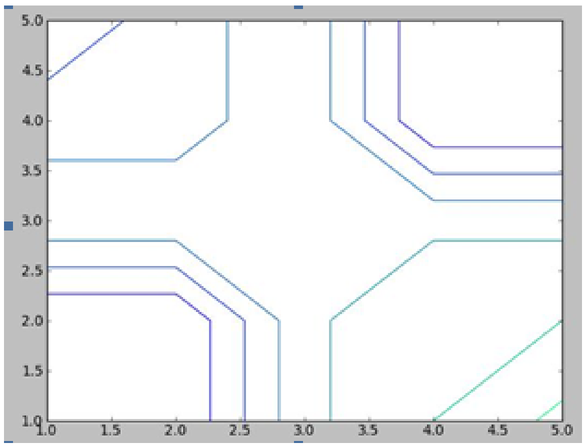
- To change colormap for 2-D maps:

  >>> contour(X,Y,D)

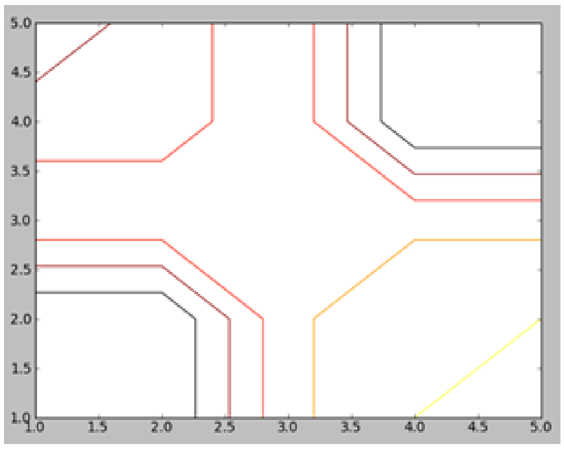
  ✓default colormap (= jet) is selected. Fig. 5-4a will be drawn.

  >>> hot()

  ✓If 'hot' colormap is selected, Fig. 5-4a will change to Fig. 5-4b.



(a) Using 'jet' colormap (default)



(b) Using 'hot' colormap

Figure 5-4: Contour plots using different colormaps

Various other colormaps are available in python. Fig. 5-5 shows some commonly used colorbars and the names for it.

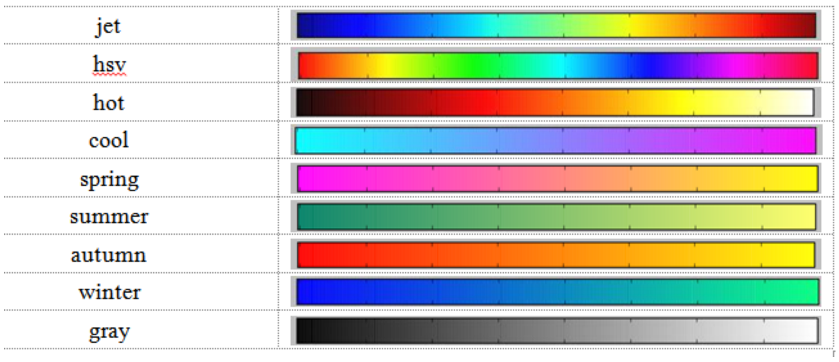


Figure 5-5: Some commonly used colormaps

For a list of all the colormaps available in python, click here.

5-2 Drawing the Global Map

This section explains how to draw a global map using python. Following is a step-by-step procedure.

1. Firstly, the Basemap toolkit needs to be imported as,

   >>> from mpl_toolkits.basemap import Basemap

   Then, the simplest way to specify the Basemap is:

   >>> M=Basemap()

   Alternatively, you can also select several options for Basemap. Some options are:

   >>> M=Basemap(resolution='i', llcrnrlon=60., llcrnrlat=-15., urcrnrlon=160., urcrnrlat=60.)

   ✓resolution: specifies the resolution of the map. 'c', 'l', 'i', 'h', 'f'or None can be used. 'c'(crude), 'l'(low), 'i'(intermediate), 'h'(high) and 'f'(full).

   ✓The lontitude and latitude for lower left corner and upper right corner can be specified by llcrnrlon, llcrnrlat, urcrnrlon and urcrnrlat:

   ✓llcrnrlon: LONgitude of Lower Left hand CoRNeR of the desired map.

   ✓llcrnrlat: LATitude of Lower Left hand CoRNeR of the desired map.

   ✓urcrnrlon: LONgitude of Upper Right hand CoRNeR of the desired map.

   ✓urcrnrlat: LATitude of Upper Right hand CoRNeR of the desired map.

2. To draw coastlines, country boundaries and rivers:

   >>> M.drawcoastlines(color='k',linewidth=0.8)

   ✓coastlines with black color and linewidth 0.8.

   >>> M.drawcountries(color='brown', linewidth=0.3)

   ✓draws country boundaries.

   >>> M.drawrivers(color='navy', linewidth=0.3)

   ✓draw rivers

3. To add longitude and latitude labels:

   >>> m.drawparallels(range(-80,80,20),color='gray',linewidth=0.5,labels=[1,0,0,1],xoffset=13)

   >>> m.drawmeridians(range(0,360,30),color='gray',linewidth=0.5,labels=[1,0,0,1],yoffset=13)

✓range: Defines the list of latitudes (parallels) and longitude (meridians) to be plotted over the map. In the above example, the parallels (meridians) are drawn from 80°S to 80 °N in every 20° (from 0° to 360° in every 30°).

✓color: Color of parallels (meridians).

✓linewidth: Width of parallels (meridians). If you want to draw only axis label and don't want to draw parallels (meridians) on the map, linewidths should be 0.

✓labels: List of 4 values (default [0,0,0,0]) that control whether parallels are labelled where they intersect the left, right, top or bottom of the plot. For e.g., labels=[1,0,0,1] will cause parallels to be labelled where they intersect the left and bottom of the plot, but not the right and top.

✓xoffset: Distance of latitude labels against vertical axis.

✓yoffset: Distance of longitude labels against horizontal axis.

5-3 Drawing Contours over Global Map

Following data is used to make contours.

- Projected temperature difference January, 2081-2100 vs. 1981-2000. This temperature projection is obtained from MPI-ECHAM5 climate model.
- The model resolution is 192 × 96 (192 grids for longitude and 96 grids for latitude).
- The data file name is "mpi_echam.txt". This data consist of [18432 rows and 3 columns]. Each row of the file correspond to each grid. The first column contains longitude, the second column contains latitude and the third column contains the temperature difference at the corresponding grid.

The procedure to draw contours over a global map are explained in the following. The objective of this section is to make Fig. 5-6.

1. As the first step, read the input file:

   ```
   >>> data=loadtxt('mpi_echam.txt')
   ```

   ✓Now, the data is [18432 × 3] matrix. The first column contains longitude, the second column contains latitude and the third column contains the temperature difference at the corresponding grid. Then, make the 2 D matrix to be used for contour plot (see section 5-1-1):

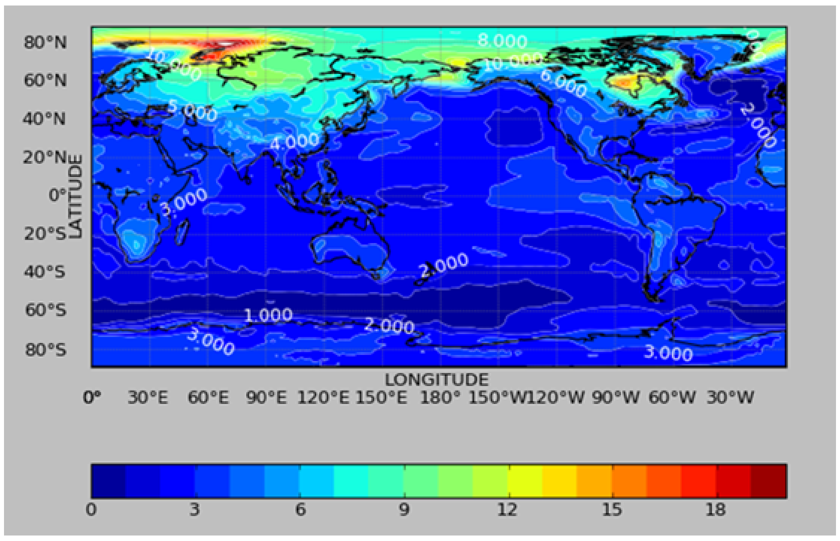


Figure 5-6: Global map with contours

```
>>> X=data[:,0].reshape(96,192)
>>> Y=data[:,1].reshape(96,192)
>>> Z=data[:,2],reshape(96,192)
```

2. Call the Basemap library and set up the Basemap:

```
>>> from mpl_toolkits.basemap import Basemap
>>> M = Basemap(resolution='i', llcrnrlon=0, llcrnrlat=-85,urcrnrlon=360,urcrnrlat=85)
```

3. Draw filled contour and contour lines:

```
>>> clevel=arange(0,20)
```

✓specify the level of contours

```
>>> contourf(X,Y,Z,clevel)
```

✓filled contour

```
>>> cs=contour(X,Y,Z,clevel,colors='w',linewidth=0.3)
```

✓draw contour lines

```
>>> clabel(cs,clevel,colors='r')
```

✓add contour labels

4. Add longitude and latitude labels:

```
>>> m.drawparallels(range(-80,80,20),color='gray', linewidth=0.5,labels=[1,0,0,1],xoffset=13)
>>> m.drawmeridians(range(0,360,30),color='gray', linewidth=0.5,labels=[1,0,0,1],yoffset=13)
```

A sample script to draw contour plot on the global map used to draw Fig. 5-6 is presented in Table 5-2.

Table 5-2: Sample code to draw contour plot on the global map

#! /usr/local/bin/python	
from mpl_toolkits.basemap import Basemap	
import numpy as np	
import matplotlib.pyplot as plt	
data=np.loadtxt('mpi_echam.txt')	# read input file
X = data[:,0].reshape(96,192)	# make 2-D matrix X
Y = data[:,1].reshape(96,192)	# make 2-D matrix Y
Z = data[:,2].reshape(96,192)	# make 2-D matrix Z
x_minmax=[np.min(X),np.max(X)]	
y_minmax=[np.min(Y),np.max(Y)]	
m = Basemap(resolution='c',llcrnrlon=x_minmax[0], llcrnrlat=y_minmax[0],urcrnrlon= x_minmax[1], urcrnrlat=y_minmax[1])	# Basemap properties
fig = plt.figure()	
m.drawcoastlines()	# draw coastlines
clevel=np.arange(0,21)	# specify the levels of contour
plt.contourf(X, Y, Z,clevel)	# contour plot with filling colors
plt.colorbar(orientation='h')	# add colorbar
cs=plt.contour(X, Y, Z, clevel, colors='w', linewidths=0.2)	# draw contour lines with white color
label=[1.,2.,3.,4.,5.,6.,8.,10.,15.,20.]	
plt.clabel(cs, label, colors='w')	# draw contour labels
m.drawparallels(range(-80,81,20),color='gray', linewidth=0.5,labels=[1,0,0,1],xoffset=13)	
m.drawmeridians(range(0,361,30),color='gray', linewidth=0.5,labels=[1,0,0,1],yoffset=13)	
plt.xlabel('LONGITUDE')	
plt.ylabel('LATITUDE')	
savename='fig3.png'	
plt.savefig(savename)	# save current figure
plt.show()	# show the figure on display

Chapter 6

Defining Functions and Calling UNIX Commands

– Kiyohara Shiraha and Keisuke Kusuhara

6-1 Defining Python Functions

This section provide a brief introduction to defining functions in python. For details on defining fuction, refer here.

- def ("definition".): When a function is defined, it should be written in a paragraph of def.
- Every defined fuction has parameter/s which is substituted in during calculation. For e.g., parameter is like x in f(x).
- An example of python function for calculating the least common multiple is presented in Table 6-1.
- After a file which includes a function is created and saved, the function can be used in interactive shell within the directory (with the file) or in other files in the same directory as a module.

  ✓If the saved filename is "xx.py", the function can be called in from another program file in the same directory. An example is shown in Table 6-2.

  ✓If the program is run, you can get the number 87, which is the least common multiple of 3 and 29.

```
>>> !./sample_module.py
>>> 87
```

Table 6-1: A sample python function

#!/usr/local/bin/python	#comment
from pylab import *	
def lcommon(m,n):	#def function's name with parameters as, funcname(parameter)
if m>0 and n>0 and type(m)==int and type(n)==int:	#int means integer.
a=[]	#make list"a"
for i in range(1,n+1):	#i is from 1 to n_th+1 ;that is [1, 2, . . . , (n-1)+1]
M=m*i	
for k in range(1,m+1):	
N=n*k	
if M==N:	#M, N is common multiple of m and n
a.append(M)	#input the common multiple of m and n into list a
return min(a)	#display the minimum value in list a
	#It is the least common multiple
else:	
return "error"	

Table 6-2: Calling and using a python function

#!/usr/local/bin/python	
from xx import lcommon	#from module's name import function's name
printlcommon(3,29)	#3 and 39 is substituted into m and n in"lcommon"

6-2 Use of UNIX Commands from within Python

UNIX commands are not directly available for use from within a python interactive shell or a program. But, if a module called 'OS module' is imported from python library, UNIX commands can be used within python program. This is useful to use python program as a shell script.

6-2-1 OS Module

This module provides a unified interface to a number of operating system functions. There are lots of useful functions for process management and file object creation in this module. Among them, it is especially useful to use functions for manipulating file and directory, which are briefly introduced below. For details on 'OS module', click here.

File and Directory Commands

Before using file and directory commands, it is necessary to import OS module as,

```
>>> import os
>>> os.getcwd()
```

✓same as pwd in UNIX. Stands for present working directory and displays the absolute path to the current directory.

```
>>> os.mkdir('dirname')
```

✓same as mkdir in UNIX. Makes a new directory. dirname can be absolute or relative path to the directory you want to create.

```
>>> os.remove('filename')
```

✓same as rm in UNIX. Removes a file.

```
>>> os.rmdir('dirname')
```

✓same as rm -r in UNIX. Removes a directory.

```
>>> os.chdir('dirpath')
```

✓same as cd in UNIX. Change directory to the location shown by dirpath. dirpath can be absolute or relative.

```
>>> os.listdir('dirpath')
```

✓same as ls in UNIX. Lists all files in a directory located at dirpath.

If you want to know more about these functions, follow this.