Introduction to Python

Ontario Summer School on High Performance Computing

Alexey Fedoseev

June 25, 2019

Computer programs

A computer program consists of *instructions* that tell the computer what to do and *data* that the program uses when it is running.

The data consists of constants or fixed values that never change and variable values. Usually, both constants and variables are defined as certain data types.

Each data type prescribes and limits the form of the data.

Examples of data types include: an integer expressed as a decimal number, or a string of text characters.

Programming languages

You tell computer what to do using a programming language. There are many programming languages and each of them has a unique set of keywords (words that it understands) and a special syntax for organizing program instructions.

Regardless of what language you use, eventually your program has to be converted into the machine language so that the computer can understand it. There are two ways to do this:

- Compile the program.
- Interpret the program.

Interpreter versus Compiler

An *interpreter* translates high-level instructions into an intermediate form, which it then executes.

In contrast, a *compiler* translates high-level instructions directly into machine language.

Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long.

The interpreter, on the other hand, can immediately execute high-level programs.

We will begin by using Python as the programming language in this class.

Python history

Python has been around for a while, and is well-developed:

- First released in February 1991.
- Python combines functional and syntactic aspects of the languages ABC, C, Modula-3, SETL, Lisp, Haskell, Icon, Perl.
- Python is a high-level, dynamic, interpreted language.
- Python supports many programming paradigms (procedural, object-oriented, functional, imperative).
- Python has automatic memory management, and garbage collection.

About Python

Some important things to know about Python:

- Python is a scripting language, meaning an interpreter executes commands one line at a time (not a compiled language).
- Python can be used interactively, with or without IDE.
- Python can also be run using scripts.
- Python has a large repository of community packages.
- Python is a general purpose language; it was not designed with data analysis in mind.
  - Several important features, such as numerics, are add-ons.
- Because Python is a general-purpose language it tends to be more versatile and flexible than R.

Using Anaconda on the Teach cluster

Anaconda is a distribution of the Python and R programming languages for scientific computing. It comes with 1,400 packages and different package versions are managed by the package management system `conda`.

Anaconda allows us to create a named, isolated, working copy of Python that maintains its own files, directories, and paths so that you can work with specific versions of libraries or Python itself without affecting other Python projects.

- Connect to the Teach login node

```
$ ssh username@teach.scinet.utoronto.ca
```

Now you are on the login node `teach01`. This node is shared between students. Use this node to develop and compile code, to run short tests, and to submit computations to the scheduler.

- Load Anaconda

```
$ module load anaconda3
```

Virtual environments

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

Virtual environments

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.

`python` versus `ipython`

There are two main ways to run Python interactively: regular Python and iPython ("Interactive Python"). They both have advantages and disadvantages:

- Regular Python is what you get when you type `python` at the command prompt.
- There aren't as many special features built into regular Python.
- But regular Python is what you get when you run Python scripts, so you're sure to get consistent behavior between your scripts and the Python command line.
- Type `ipython` to launch iPython. It has tab-line completion built in, interactive plotting and other features.
- But iPython is not what you have when you run scripts.

Version of Python

After starting Python the first thing you see is a version of Python you are using.

```
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
```

This is an important information because the developers of Python constantly instroducing new features and if you write a program using the latest version of Python and give it to someone who is using a very old version of Python, it might not work due to the changes made in the language.

Python prompt

After reading the greeting message you will find at the very bottom of it a sign '>>>' followed by the cursor. Here you can type in your instructions. Python executes them immediately after you hit Enter. Let us try to give our first instruction.

```
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> "hello"
'hello'
>>>
```

As soon as the execution of instruction is finished, Python displays the result of it and waits for new commands with a new prompt '>>>'.

Importing modules

Most of the default functions in Python are contained in packages. There are several ways to import packages:

```
>>> import platform
>>> platform.system()
'Darwin'
>>> import time as t
>>> t.time()
1550679097.931794
>>> from calendar import isleap
>>> isleap(2020)
True
```

Python data types

Python has several standard data types:

- Numbers
- Strings
- Booleans
- Container types
  - Lists
  - Sets
  - Tuples
  - Dictionaries

Integers in Python 3

In Python 3, there is effectively no limit to how long an integer value can be. It is only constrained by the amount of memory your system has.

```
>>> import math
>>> math.factorial(300)
306057512216440636035370461297268629388588804173576999416776741259476533
176716867465515291422477573349939147888701726368864263907759003154226842
927906974559841225476930271954604008012215776252176854255965356903506788
725264321896264299365204576448830388909753943489625436053225980776521270
822437639449120128678675368305712293681943649956460498166450227716500185
176546469340112226034729724066333258583506870150169794168850353752137554
910289126407157154830282284937952636580145235233156936482233436799254594
095276820608062232812387383880817049600000000000000000000000000000000000
000000000000000000000000000000000000000
```

Floating-Point Numbers

Floats have a decimal point and integers do not have a decimal point. So even though 4 and 4.0 are the same number, 4 is an integer while 4.0 is a float.

Before you start calculating with floats you should understand that the precision of floats has limits, due to Python and the architecture of a computer. Some examples of errors due to finite precision are displayed below.

```
>>> 1.13 - 1.1
0.029999999999999805
>>> 1 + .0000000000000001
1.0
```

Strings

Strings are among the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.

```
>>> word = "Hello World"
>>> print(word)
Hello World
>>> print(word + " again!")
Hello World again!
```

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
>>> print(word[0], word[6])
H W
>>> print(word[6:11])
World
```

Booleans

Python supports standard boolean variables and operations. Booleans behave as you would intuitively expect.

- "and" and "&" are the same.
- "or" and "|" are the same.
- "not" is written out (don't use the "!" symbol).

```
>>> truth = True
>>> truth and False
False
>>> not truth
False
>>> truth or False
True
>>> truth | False
True
```

Dynamic typing

Python uses dynamic typing which means you can re-use a variable over and over again.

```
>>> truth = 2
>>> print(truth)
2
>>> type(truth)
<class 'int'>
>>>
>>> truth = True
>>> print(truth)
True
>>> type(truth)
<class 'bool'>
```

Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type.

```
>>> mix_list = ["apples", 2, True, [4,10], False]
>>> print(mix_list)
['apples', 2, True, [4, 10], False]
```

List indices start at 0, and lists can be sliced. Slicing is done with `[start:finish]`, but does not include the "finish" element.

```
>>> print(mix_list[0])
apples
>>> print(mix_list[0:1])
['apples']
>>> print(mix_list[0:2])
['apples', 2]
```

Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the `append()` method.

```
>>> print(mix_list)
['apples', 2, True, [4, 10], False]
>>>
>>> mix_list[0] = "oranges"
>>> print(mix_list)
['oranges', 2, True, [4, 10], False]
>>>
>>> mix_list.append("books")
>>> print(mix_list)
['oranges', 2, True, [4, 10], False, 'books']
```

Dictionaries

A list is an ordered sequence of objects, whereas dictionaries are unordered sets. The main difference between lists and dictionaries is that items in dictionaries are accessed via keys and not via their position. Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type.

To access dictionary elements, you can use the square brackets along with the key to obtain its value.

```
>>> campus = {"name": "St. George", "location": "Toronto"}
>>> print(campus)
{'name': 'St. George', 'location': 'Toronto'}
>>> print(campus["name"])
St. George
```

Updating the dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown in a simple example given below.

```
>>> campus["name"] = "University of Toronto"
>>> campus["established"] = 1827
>>> del campus["location"]
>>> print(campus)
{'name': 'University of Toronto', 'established': 1827}
>>>
>>> campus.keys()
dict_keys(['name', 'established'])
>>> campus.values()
dict_values(['University of Toronto', 1827])
```

Writing scripts

Create the file `todo.py` with the following contents.

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("day",
                    choices=["today", "tomorrow"],
                    help="Select the day to see tasks")
args = parser.parse_args()

toDo = {"today": "sleep"}
toDo["tomorrow"] = "do nothing"

print(args.day.capitalize(), "I am going to", toDo[args.day])
```

```
$ python todo.py today
Today I am going to sleep
```

Running the script

```
$ python todo.py tomorrow
Tomorrow I am going to do nothing

$ python todo.py
usage: todo.py [-h] {today,tomorrow}
todo.py: error: the following arguments are required: day

$ python todo.py -h
usage: todo.py [-h] {today,tomorrow}

positional arguments:
  {today,tomorrow}  Select the day to see tasks

optional arguments:
  -h, --help        show this help message and exit
```

Conditional statements

`if` statement evaluates whether a statement is true or false, and runs code only in the case that the statement is true.

```
>>> temperature = -15
>>> if temperature < -10:
...     print("It is cold!")
...
It is cold!
```

If you want the program to do something even when an if statement evaluates to false use `else` statement.

```
>>> if temperature < -10:
...     print("It is cold!")
... else:
...     print("It is warm")
...
It is cold!
```

Else if statement

In many cases, we will want a program that evaluates more than two possible outcomes. For this, we will use an else if statement `elif`

```
>>> if temperature < -10:
...     print("It is cold!")
... elif (temperature > -10) and (temperature < 10):
...     print("It is mild")
... else:
...     print("It is warm")
...
It is cold!
```

Code blocks in Python start with “`:`” and are preceded by white space. The amount of indentation indicates which code block it’s part of. Different code blocks can have different amounts of indentation, but the indentation must be consistent within the same code block.

`for` loops

A `for` loop implements the repeated execution of code based on a loop counter or loop variable. This means that `for` loops are used most often when the number of iterations is known before entering the loop, unlike `while` loops which are conditionally based.

```
>>> shopping_list = ["apples", "oranges", "grapes", "tomatoes"]
>>> for item in shopping_list:
...     print("I need to buy", item)
...
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

Notice that code blocks in Python are specified using tabulation. It means that the code inside loops **must** be indented.

Nested code blocks

```
>>> temperature = -5
>>> shopping_list = ["apples", "oranges", "grapes", "tomatoes"]
>>> if temperature > -10:
...     print("I am going out!")
...     for item in shopping_list:
...             print("I need to buy", item)
... else:
...     print("I am staying home!")
...
I am going out!
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

range

Python's built-in range() function is handy when you need to perform an action a specific number of times. It generates the integer numbers between the given start integer to the stop integer, which is generally used to iterate over with a for loop.

```
>>> len(shopping_list)
4
>>> range(len(shopping_list))
range(0, 4)
>>> for index in range(len(shopping_list)):
...     print("I need to buy", shopping_list[index])
...
I need to buy apples
I need to buy oranges
I need to buy grapes
I need to buy tomatoes
```

while loops

A while loop implements the repeated execution of code based on a given boolean condition. The code that is in a while block will execute as long as the while statement evaluates to True.

You can think of the while loop as a repeating conditional statement. After an if statement, the program continues to execute code, but in a while loop, the program jumps back to the start of the while statement until the condition is False.

```
>>> temperature = 2
>>> while temperature > -2:
...     print("Temperature is", temperature)
...     temperature = temperature - 1
...
Temperature is 2
Temperature is 1
Temperature is 0
Temperature is -1
```

Defining functions

A function is a block of instructions that performs an action and, once defined, can be reused. Functions make code more modular, allowing you to use the same code over and over again.

Let us create a file named `temperature.py` and define a function `check_temp`.

```
def check_temp(temperature):
    if temperature < -10:
        print("It is cold")
    elif temperature < 10:
        print("It is mild")
    elif temperature < 25:
        print("It is warm")
    else:
        print("It is hot!")

check_temp(-5)
check_temp(15)
```

- As with all code blocks the body of a function starts with a colon and must be indented.
- A function can take arguments. When called, that which is passed to the function is assigned to the variable declared in the function definition.

Run the script `temperature.py` to see the output.

```
$ python temperature.py
It is mild
It is warm
```

Default values

Functions can take multiple arguments.

```
def check_temp(temperature = 0):
        if temperature < -10:
                print("It is cold")
        elif temperature < 10:
                print("It is mild")
        elif temperature < 25:
                print("It is warm")
        else:
                print("It is hot!")

check_temp()
check_temp(15)
```

- All non-optional arguments must be specified when a function is called.
- Notice that the argument values are assigned in the order they appear in the function declaration.
- The values of optional arguments are specified in the function definition.

```
$ python temperature.py
It is mild
It is warm
```

Global and local variables

- Variables which are declared within a function are called "local". They may only be accessed within the function.
- Variables which are declared outside of functions are called "global".
  - Global variables can be accessed from within functions, but this is a bad idea.
  - It's better to pass all information you need in your function as an argument.
- If you start using global variables within functions you will break the modularity of your code. Your code will become less portable and much harder to debug.

return statement

You can pass a parameter value into a function, and a function can also produce a value.

In Python you must include a return statement with your functions if you want to return something.

Create listUtils.py:

```
def list2str(lst):
    result = []
    for elem in lst:
        result.append(str(elem))
    return(result)

print(list2str([1,2,3,4]))
```

Run listUtils.py

```
$ python listUtils.py
['1', '2', '3', '4']
```

Importing modules

Writing a module is just like writing any other Python file. Modules can contain definitions of functions, classes, and variables that can then be utilized in other Python programs.

Let us say that we put several useful functions into the file `listUtils.py`. In order to use the function this file we need to `import` them first.

```
$ python
>>> import listUtils
['1', '2', '3', '4']
>>> listUtils.list2str([5,4,3,2,1])
['5', '4', '3', '2', '1']
```

Finding your functions

Let us say I moved my utility script to the directory /Users/alexey/code. We need to point Python to the new directory.

```
$ python
>>> import listUtils
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'listUtils'
>>> import sys
>>> sys.path
['/Users/alexey/anaconda3/lib/python3.6', ...,
'/Users/alexey/anaconda3/lib/python3.6/site-packages']
>>> sys.path.append('/Users/alexey/code')
>>> import listUtils
['1', '2', '3', '4']
```

Getting help

Once a module or package has been imported, you can get information about how to run that package: the `help` command gives information about the package or function.

Type "`q`" to get out of help screen.

```
>>> import os
>>> help(os)
...
>>>
>>> import time
>>> help(time)
...
>>>
```

Making your own help

You can create your own help entry for your functions, called “docstrings”.

- Put `"""` around your docstring text
- It must be at the start of your function.
- You can also make docstrings for the whole module.

```
>>> import listUtils
>>> help(listUtils.list2str)
Help on function list2str in
module listUtils:

list2str(lst)
    Function list2str converts
    every element of a list to
    a string value
```

```
# listUtils.py
"""
This module contains helper functions
"""

def list2str(lst):
    """
    Function list2str converts
    every element of a list to
    a string value
    """
    result = []
    for elem in lst:
        result.append(str(elem))
    return(result)
```

Command line arguments

We already saw how we can pass the parameters to the script using package argparse. Python has another way to retrieve these parameters

```
# cmd_param.py
import sys
print("There are", len(sys.argv), "arguments.")
print("The command line arguments are:", sys.argv)
```

```
$ python cmd_param.py param_1 99.9
There are 3 arguments.
The command line arguments are: ['cmd_param.py', 'param_1', '99.9']
```

Remember that by default, the arguments are strings.

List comprehensions

Very often we need to transform every element of a list. For example:

```
>>> mynums = [1, 2, 3, 4]
>>> squared_nums = []
>>> for num in mynums:
...     squared_nums.append(num * num)
...
>>> print(squared_nums)
[1, 4, 9, 16]
```

List comprehensions provide a concise way to create such lists.

```
>>> [num * num for num in mynums]
[1, 4, 9, 16]
>>> [num * num for num in mynums if num % 2 == 0]
[4, 16]
```

The % (modulo) operator calculates the remainder from the division.

NumPy & SciPy

Multidimensional lists in Python

The element of a list in Python can be of any type, including a list, that is we can create a list of lists or multidimensional list. For example, this is how you can create a Vandermonde matrix:

```
>>> vander_matrix = [[1.0, 1.0, 1.0], [1.0, 2.0, 4.0], [1.0, 3.0, 9.0]]
```

Here we have a three-element list where each element consists of a three-element list.

```
>>> vander_matrix[0]
[1.0, 1.0, 1.0]
>>> vander_matrix[1]
[1.0, 2.0, 4.0]
>>> vander_matrix[0][0]
1.0
>>> vander_matrix[1][1]
2.0
```

Remember that list indices in Python start at 0.

NumPy arrays

The NumPy array is similar to a list but where all the elements of the list are of the same type.

NumPy has a number of functions for creating arrays. The first of these, the `array` function, converts a list to an array.

```
>>> import numpy
>>> vander_matrix
[[1.0, 1.0, 1.0], [1.0, 2.0, 4.0], [1.0, 3.0, 9.0]]
>>> vander_matrix_numpy = numpy.array(vander_matrix)
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
```

Remember to `import numpy` module in your script.

NumPy arrays

The second way arrays can be created is using the NumPy `linspace` function. It creates an array of `N` evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, N)`. If the third argument `N` is omitted, then `N = 50`.

```
>>> numpy.linspace(0, 3, 7)
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. ])
```

The third way arrays can be created is using the NumPy `arange` function. The form of the function is `arange(start, stop, step)`. If the third argument is omitted `step = 1`. If the first and third arguments are omitted, then `start = 0` and `step = 1`.

```
>>> numpy.arange(0, 1, 0.1)
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
>>> numpy.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy arrays

A fourth way to create an array is with the `zeros` and `ones` functions which create arrays where all the elements are either zeros or ones.

```
>>> numpy.zeros(5)
array([0., 0., 0., 0., 0.])
>>> numpy.ones(5)
array([1., 1., 1., 1., 1.])
```

Very common you find that instead of typing the name of the module `numpy`, it is imported with a short alias `np`.

```
>>> import numpy as np
>>> np.ones(3)
array([1., 1., 1.])
```

Mathematical operations with arrays

It is very easy to perform mathematical operations on every element in the array.

```
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
>>> vander_matrix_numpy * 2
array([[ 2.,  2.,  2.],
       [ 2.,  4.,  8.],
       [ 2.,  6., 18.]])
```

This works not only for multiplication, but for any other mathematical operation.

```
>>> vander_matrix_numpy - 1
array([[0., 0., 0.],
       [0., 1., 3.],
       [0., 2., 8.]])
```

Mathematical operations with arrays

Multiplication of two arrays is performed element-wise.

```
>>> vander_matrix_numpy * vander_matrix_numpy
array([[ 1.,  1.,  1.],
       [ 1.,  4., 16.],
       [ 1.,  9., 81.]])
```

To calculate the dot product of two arrays use function `np.dot`.

```
>>> np.dot(vander_matrix_numpy, vander_matrix_numpy)
array([[ 3.,  6., 14.],
       [ 7., 17., 45.],
       [13., 34., 94.]])
```

These kinds of operations with arrays are called vectorized operations because the entire array, or "vector", is processed as a unit. Vectorized operations are much faster than processing each element of arrays one by one.

Multidimensional arrays

We can create a multidimensional array by applying `array` function to the multidimensional list

```
>>> numpy.array([[1,2,3,4,5],[6,7,8,9,10]])
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

To create a multidimensional array using the `zeros` and `ones` functions we need to specify number of rows and number of columns. In NumPy rows are always specified first.

```
>>> numpy.ones((3, 4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

Notice the way we specified the number of rows and columns: `(3, 4)`. This structure is called tuple. Tuples are very similar to lists, but the main difference between them is that the tuples cannot be changed unlike lists.

Array indexing

NumPy offers several ways to index arrays.

```
>>> all_data = numpy.arange(10, 0, -1)
>>> all_data
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>> all_data[0]
10
>>> all_data[-1] # supports negative indices
1
>>> all_data[2:]
array([8, 7, 6, 5, 4, 3, 2, 1])
>>> all_data[:2]
array([10,  9])
>>> all_data[0:2] # slice items between indexes
array([10,  9])
```

While slicing between indices, the `start` index is included and the `stop` index is not included.

Boolean indexing

Frequently we want to select or modify only the elements of an array satisfying some condition (fancy indexing).

```
>>> all_data
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>> (all_data <= 7) & (all_data >= 5)
array([False, False, False,  True,  True,  True, False, False, False, False]
>>> all_data[(all_data <= 7) & (all_data >= 5)]
array([7, 6, 5])
>>> even_nums = all_data[(all_data % 2) == 0]
>>> even_nums
array([10,  8,  6,  4,  2])
```

The "%" symbol is the modulo operator.

Multidimensional slices

You can slice multidimensional arrays in a similar way.

```
>>> vander_matrix_numpy
array([[1., 1., 1.],
       [1., 2., 4.],
       [1., 3., 9.]])
>>> vander_matrix_numpy[1,1]
2.0
>>> vander_matrix_numpy[2,:]
array([1., 3., 9.])
>>> vander_matrix_numpy[1:,1:]
array([[2., 4.],
       [3., 9.]])
```

Shape and reshape

The `shape` property return a tuple of array's dimensions and can be used to change the dimensions of an array.

```
>>> seq_array = numpy.arange(1,11)
>>> seq_array
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> seq_array.shape
(10,)
```

Here the shape (10,) means the array is indexed by a single index which runs from 0 to 9.

NumPy allows you to modify the shape of an array once it already exists. The `reshape` function gives a new shape to an array without changing the data.

```
>>> seq_array2d = seq_array.reshape((2,5))
>>> seq_array2d
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

Shape and reshape

Reshaping array doesn't change the data in the memory. Instead, it creates a new view that describes a different way to interpret the data.

The `shape` of an multidimensional array is a tuple of its dimensions where first element of the tuple represents the number of rows and the second is the number of columns.

```
>>> seq_array2d
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
>>> seq_array2d.shape
(2, 5)
```

Shape and reshape

Specifying -1 as one of the dimensions while reshaping, forces NumPy to calculate this dimension based on the total amount of elements in the array and already specified dimensions.

```
>>> seq_array2d.reshape((3,-1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 10 into shape (3,newaxis)
>>> numpy.arange(9).reshape((-1,3))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

The `linalg` submodule

The `linalg` submodule of SciPy contains useful functions for matrix algebra.

- Typical matrix functions: `inv`, `det`, `norm`, etc.
- More advanced functions: `eig`, `SVD`, `cholesky`, etc.
- Both NumPy and SciPy have a `linalg` module. Use SciPy, because it is compiled with optimized BLAS/LAPACK support.

```
>>> import numpy
>>> import scipy
>>> from scipy import linalg
>>> A = numpy.array([[1,2,3], [3,4,5], [1,1,2]])
>>> linalg.det(A)
-2.0
>>> scipy.dot(A, linalg.inv(A))
array([[ 1.00000000e+00,  2.22044605e-16, -2.22044605e-16],
       [ 1.66533454e-16,  1.00000000e+00, -6.66133815e-16],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
```

Solving systems of equations

The `solve` function in the `linalg` module is used to solve the system of equations $Ax = b$.

```
>>> A
array([[1, 2, 3],
       [3, 4, 5],
       [1, 1, 2]])
>>> b = numpy.array([3, 4, 2])
>>> b
array([3, 4, 2])
>>> x = linalg.solve(A, b)
>>> x
array([-0.5, -0.5,  1.5])
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} -0.5 \\ -0.5 \\ 1.5 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 2 \end{bmatrix}$$

Visualization

Visualization

There are a number of high-quality visualization packages available in Python

- `matplotlib` focuses on generating publication-quality plots
- `seaborn` targets statistical data analysis
- `ggplot` is based on the famous `R` package
- `Plotly` and `Bokeh` focus on interactivity
- and others

Installing Matplotlib

To install `matplotlib` package run the following command in your terminal

```
$ pip install matplotlib
```

Anaconda base environment comes with pre-installed `matplotlib` package. If you need to install it in a new environment run

```
$ conda install matplotlib
```

`matplotlib` is imported using the following command

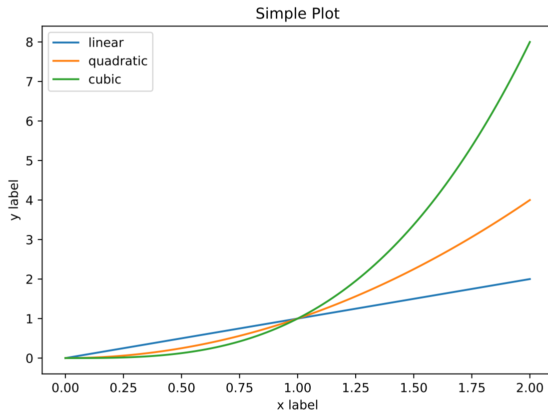
```
>>> import matplotlib.pyplot as plt
```

Also import `numpy` as it is frequently used together with `matplotlib`

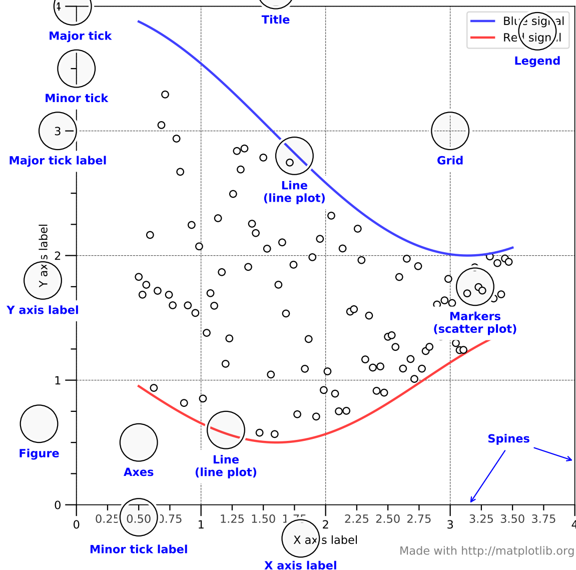
```
>>> import numpy as np
```

Simple plot in matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

plt.show()
```



To save your plot use the command: `plt.savefig(filename)`


Anatomy of a figure
Title
Major tick
Minor tick
Major tick label
Y axis label
Figure
Axes
Line
(line plot)
Grid
Legend
Blue signal
Red signal
Markers
(scatter plot)
Spines
Minor tick label
X axis label
Made with http://matplotlib.org