# Python & its Modules

CONTENTS



## I. BASICS

### A. *Environments*

- Python usage: in & out
  - type `python` from a terminal to launch python in interactive mode
  - type `python fn.py` from Linux prompt to execute python file `fn.py`
  - type `execfile('fn.py')` from Python prompt to execute python file `fn.py`
  - quit Python by typing `quit()` or `exit()`
  - `exit()` also exits a code or function
- Python versions:
  - there are two major versions of Python, 2.x & 3.x
  - type `python --version`
- git:
  - *git* is a version control system
    - to download materials from git type
      `git clone web_address`
- GitHub:
  - *GitHub* is a code hosting platform for version control and collaboration
    - it lets you and others work together on projects from anywhere
  - a *repository* is usually used to organize a single project
    - repositories can contain folders and files, images, videos, spreadsheets, and data sets  anything your project needs
    - it is recommended to include a README, or a file with information about your project
    - to create a repository click on green icon *New repository*
    - saved changes are called *commits*
    - to delete a repository go to repository, then click on *settings* and then go to the bottom of settings page
  - the repository can have multiple *branches* of the repository: the *master branch* as well as alternative branches
  - Pull requests:
    - a *pull request*, requests others to review and pull in your contribution and merge them into their branch

    - pull requests show diffs, or differences, of the content from both branches
    - the changes, additions, and subtractions are shown in green and red
- PyCharm
  - *PyCharm* is an Integrated Development Environment (IDE) used for programming in Python
  - `https://www.jetbrains.com/pycharm/`
  - to add a directory to the project
    `File|Settings|Project|Project Structure`

*1) Anaconda:*

- General:
  - `https://www.continuum.io/why-anaconda`
  - *Anaconda* is an open source distribution of the Python and R programming languages for large-scale data processing, predictive analytics, and scientific computing
  - Anaconda aims to simplify management of packages and environments
    - creation of virtual environments make it easy to work on multiple projects
  - Anaconda is a distribution of software that comes with *conda* (see below), Python, and over 150 scientific packages and their dependencies
    - both Python versions are included in Anaconda
  - I have used Anaconda as a Python compiler
  - *Miniconda*, is a smaller distribution that includes only conda and Python
    - you can still install any of the available packages with conda
- `conda` as package manager:
  - conda is used exclusively from the command line
  - `conda` is Anaconda's package manager application that installs, runs, and updates packages and their dependencies
    - conda installs pre-compiled packages
  - basic commands:
    - `conda command_name --help`
      explains `command_name`
    - `conda --version`
    - `conda list` lists linked packages
  - key commands:
    - `conda install package_name`
    - `conda install -c conda-forge matplotlib=2.0.2`
    - `conda remove -n package_name --all`
    - `conda update package_name`
    - `conda search search_term`
  - installing through `https`
    `conda install --channel https://... x`
  - to remove `conda` installation
    `rm -rf ~/anaconda`
- `conda` versus `pip`
  - `conda` is similar to `pip` except that the available packages are focused around data science while `pip` is for general use
  - `conda` is not Python specific like `pip` is, it can also install non-Python packages
  - can install packages with `pip` if they not available from `conda` or `Anaconda.org`
- Conda as virtual environment manager:
  - environments allow the separation and isolation of the used packages for different projects
  - `conda create -n env_name packages`
    - `-n` (or `--name`), stands for *name*
    - `packages` are list of packages that need to be installed in the environment
    - it is recommended to install all programs in an environment at the same time
  - `conda` and python:
    - `conda` treats Python same as any other package
    - if Python version is not specified, `conda` installs the same version that was used during installation of `conda`
    - can specify python version as an argument
      `python=3`

- to install in a specific environment type
  `conda install --name env_name pck_name`
- to remove from a specific environment type
  `conda remove --name  env_name pck_name`
- can check installed packages in an environment by typing either one of
  `conda env list`
  `conda info --envs`
- save or share an environment by typing
  `conda env export > environment.yaml`
- to create an environment from an environment file use
  `conda env create -f environment.yaml`
- can remove environments use
  `conda env remove -n env_name`
- to cleanup downloaded libraries type
  `conda clean -tp`

• Conda activation:
  - activate an environment by typing
    `activate env_name` in Windows
    `source activate env_name` in Linux
  - environments are installed by default into the envs directory in your conda directory
  - to deactivate type
    `deactivate` in Windows
    `source deactivate` in Linux
    * deactivation takes it back to root directory

• Jupyter Notebook:
  - formerly *iPython Notebook*
  - interactively write documents that include code, text, output and *LateX*
  - to install from conda type
    `conda install jupyter notebook`
  - To start a notebook server, enter
    `jupyter notebook`

## B. *Fundementals*

• Some symbols
  - `x / y` returns true division
  - `x // y` returns floor division
  - symbol `x % y` returns the remainder
  - symbol `**` is power
  - to comment out a line, start with symbol `#`
  - to comment out a paragraph, enclose it between `"""`

• Data types & Variables:
  - variables are not declared
  - variables can dynamically change type
  - basic built-in types are
    * `bool`
    * `int`
    * `float`
    * `long`
    * `complex`
    * `None`, used to represent the absence of a value
      `x = None`

• Data type commands:
  - `type()` returns type of object
    * `type(None)` is 'NoneType'
  - `int()` converts string to int
  - `float()` converts string to float

  - `str()` converts arguments to strings
  - `isinstance(x, int)` checks if x is integer, can check for float, bool etc.
- Boolean:
  - boolean variables are assigned to `True` or `False`
  - boolean operators for or & and are `or` & `and`
  - negation is `not`
  - not equal to is `!=`
- `print()`
  - `print('a string')=print("a string")`
  - `print` inserts new-line, `\n` at the end
    * if do not want new-line use option `end = ''`
  - `print('a = ', a)`
  - `print("A has {} dogs".format(var))`
  - `print("A = {:2.3f}".format(2.2))`
  - inside the `{}`:
    * < will left-align the text in a field, e.g. `{:<3}`
    * ^ will center the text in the field
    * > will right-align it
- `(text)`
  - if a text is too long to fit in a line use parenthesis
- `dir()`
  - without arguments, returns the list of names in the current local scope,
  - with an argument, or object, attempts to return a list of valid attributes for that object, `dir(str)`
- `vars(),locals(),globals()`
  - `globals()` returns the dictionary of the module namespace
  - `locals()` returns a dictionary of the current namespace
  - `vars()` returns either a dictionary of the current namespace (if called with no argument) or the dictionary of the argument
- `id()`
  - `id(object)` returns the identity of an object
  - this is an integer (or long integer)
  - the identity is guaranteed to be unique and constant for this object during its lifetime
- `input("prompt")`
  - take input from keyboard as string
  - returns the keyboard string
- ```
  if expression1:
      statement(s)
  elif expression2:
      statement(s)
  else:
      statement(s)
  ```
- Exception:
  - even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it
  - errors detected during execution are called *exceptions* and are not unconditionally fatal
  - Python has built-in exceptions, in the *Exception* class, but a code may need user-defined exceptions
    * exceptions should typically be derived from the Exception class, either directly or indirectly
- `try`, `except`, `raise` & `finally`
  - can defend your program from an exception, by placing the suspicious code in a `try, except` block
    ```
    try:
        suspicious statement(s)
    except:
        problem handling statement(s)
    ```
  - if no exception occurs, the `except` clause is skipped and execution of the try statement is finished
    ```

- if `try` block gives exception, the `except` clause handles the problem as elegantly as possible
- the `raise` statement forces an exception to occur
- the argument to `raise` indicates the exception to be raised
  * this must be either an exception instance or an exception class
- for example
  `raise ValueError`
  implies the value of a parameter error
- can also be more specific
  `raise ValueError, value not in dict`
- the `try` statement has another optional clause, `finally`, which is intended to define clean-up actions that must be executed under all circumstances
  * a `finally` clause is always executed before leaving the `try` statement, whether an exception has occurred or not

```
try:
    raise KeyboardInterrupt
finally:
    print('Goodbye, world!')
```

• Iterator:
- an *iterator* is an object representing a stream of data
  * an iterator is an object we can iterate over
  * see Section II-G for more on classes
- repeated calls to the iterators `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream
- iterators are required to have an `__iter__()` method that returns the iterator object itself
  * so every iterator is also iterable and may be used in most places where other iterables are accepted

• Iterable:
- an *iterable* is an object capable of returning its members one at a time
- e.g. lists, tuples, dictionaries, file objects
- iterables can be used in a for loop

•
```
for iterating_var in sequence:
    statements(s)
```
- the `for` statement assigns to `iterating_var` all the elements of the `sequence`,
- examples:

```
for _letter in _string:
for _element in _list:
for (i,j) in _list_of_2_tuples:
for _key in _dict:
```

- if the index is needed as well, `enumerate()` returns both index and member
  `for i,x in enumerate(list)`

•
```
while expression:
    statement(s)
```

• `open`
- `open('output.txt', mode='w')`
- `mode`
  * `mode='b'` means binary
  * `mode='r'` means read
  * `mode='w'` means write

• `with`
- used when two related operations which you'd like to execute as a pair, with a block of code in between
-
```
with open('output.txt', mode='w') as f:
    f.write('Hi there!')
```
- `with` statement will automatically close the file after the nested block of code

• `in`
- `in` is an operator that checks if $a \in A$
- `in` returns boolean outcome

- examples:
  `_letter in _string:`
  `_element in _list:`
  `_key in _dict:`

- `is`
  - `is` is similar to == but stronger
  - unlike ==, `is` does not make any conversions
    `if x is None`

- `__main__` & `global`
  - scripts are in `__main__`,
  - variables in `__main__` are `global`,
  - to reassign a global variable in a function, need to declare the variable as `global var`.
- `__name__`
  - before executing the code, Python defines a few special variables
  - `__name__` is such a variable, that stores the name of source file
  - when Python interpreter is running a source file, as the main program, it sets the special `__name__` variable to have a value `__main__`
  - if the source file is being imported from another module, `__name__` will be set to the module's name
- User defined functions:
  - `def fn(x):`
  - `    y= x**2`
  - `    return(y)`
  - the return value (if assigned one) of a void function is `None`,
  - tuples (see subsection II-F), allow multiple returns from a function
    `return(x,y)`
- `lambda`
  - lambda function is of the form
  - `g = lambda x: x**2`
- Generator
  - a *generator* is a function which returns a generator iterator
    * outside of Python, generators are referred to as *coroutines*
    * generator functions create generator iterators
    * a generator is a special type of iterator
    * since a generator is a type of iterator, it can be used in a for loop
  - instead of `return()` type `yield`
    * `return` implies returning control of execution
    * `yield` implies the transfer of control is temporary and voluntary, where the generator expects to regain it in the future
  - the yielded value is the "generated" value
  - the next time `next()` is called on the generator iterator, the generator resumes execution from where it called `yield`, not from the beginning of the function
    * to get the next value from a generator, use the same built-in function as for iterators: `next()`
- Other Python keywords:
  - `assert`
    * `assert flag, 'text'`
    * if `flag=False` then stops codes and prints `text`
  - `del`
    * `del a` deletes variable `a`
  - `pass`
    * is a null operation
    * e.g function placeholder
  - `break`
    * breaks out of the innermost enclosing `for` or `while` loop

- `continue`
  - continues with the next iteration of the loop

## II. STRUCTURES

- There are four native collections in Python:
  - *lists*, `list()`
  - *dictionaries* `dict()`, and
  - *tuples* `tuple()`
  - *sets* `set()`
- Strings and tuples are *immutable*, lists, sets and dictionaries are *mutable*.
- `len()`
  - returns the number of items in a string, list, tuple, set or dictionary

### A. Strings

- Strings:
  - a string is a sequence of characters
    - both single quotes and double quotes can be used
  - the string type is denoted as `str`
  - strings are *immutable*
    - cannot change an existing string
  - compare two strings using ==, > and <
    - ordering is through alphabetic order
- Overloading:
  - + operator concatenates strings
  - * operator repeats a string a given number of times
- Special characters:
  - `\n` is new line
- Slicing: syntax to access sublists, using symbol ':'
  - index starts at zero

```
a[a:b] #from a to (b-1)
a[a:]  #from a to the end
a[:b]  #from beginning to (b-1)
a[:]   #a copy of the whole array
a[-1]  #last item
a[-2]  #second last item
a[-2:] #last two items
a[:-1] #all except the last item
a[:-2] #all except the last two items
```

- Iterable:
  - the elements of a string are iterable
    ```
    for char in string:
        statement(s)
    ```
- Methods:
  - `str.find(a,_start,_before)`
    - returns position of the first occurrence of `a`
    - search from position `_start` to `_before`
    - `_start` & `_before` are optional
    - useful when parsing text
  - `str.strip()`
    - returns string without `\r\n` control sequences, and without extra spacing at both ends
    - an optional argument string `chars` specifies the set of characters to be removed
  - `str.startswith()`
    - returns `True` if string starts with the `prefix`
  - `str.split()` breaks a string into words,
    - more specifically, `string.split(delimiter)` breaks the string into a list of words using a defined delimiter,
  - `str.join()`

    * is the reverse of split,
    * given a delimiter `s`,
    * given a sequence (list) of strings `seq`
    * `s.join(seq)` concatenates elements of `seq` into a single string.
- Capitalization:
  - `str.upper()`
    * capitalizes all the letters
  - `str.lower()`
    * lower-case all letters
  - `str.isuppper()`
    * returns a boolean
  - `word.capitalize()`
    * capitalizes the first letter
- Formatting:
  - `s = '%s %s %d' % ('hello','wo',2016)`

### *B. Files*

- Files are stored on secondary memory.
- `open`

```
fin = open(fn.txt) #fin=file handle
fin = open(fn.txt,'r') #same as above
fout= open(fn.txt,w) # write
```

- File handle as a sequence:
  - `for line in file_handle`
  - each line in the file is a string in the sequence,
  - if using `for` loop then no need to use explicit read functions.
- Read from file:

```
line = file_handle.readline()
all  = file_handle.read()
```

- Write to file:

```
fout.write('') #argument is string
fout.write(str(42))
fout.write('%d %s',42,'aa')
```

### *C. Lists*

- lists:
  - a *list* is is a sequence of values of any data type,
    * a linear collection of values that stay in order,
    * similar to cell in Matlab,
  - e.g. `A = [10, 'aa']`,
    * where `A[0]=10, A[1]='aa'`,
  - lists are mutable,
  - each item could be a list itself,
  - an empty list is [ ].
- `range(n)`
  - generates a list of `n` elements,
    `[0, 1, ... (n-1)]`
  - same as $[0 : n-1]$ in Matlab,
- `xrange()`
  - similar to `range()`, but returns an `xrange` object instead of a list,

  - the advantage of the `xrange` type is that an `xrange` object will always take the same amount of memory, no matter the size of the range it represents.
- Overloading & slicing is similar to strings.
- Iterator:
  - can traverse the elements in the list by doing `for i in range(len(list_name)):`,
  - the `in` operator works on lists.
- Methods:
  - `l.append(f)` adds the new element f to the end of the list,
  - `l.extend(m)` concatenates list `m` to `l`,
    - similar to `l+m`,
  - `l.sort()` alphabetically sorts elements,
  - `b = l.pop(i)`, takes out the $i^{\text{th}}$ element from `l`, and assigns it to `b`,
    - in the absence of argument `i`, `pop()` removes last element,
  - `a.remove(element)` to remove a specific element, without referring to index,
  - `del l[i]`, deletes the $i^{\text{th}}$ element,
  - `del l[1:5]` removes a subset.
- `list()`
  - `list()` creates an empty list,
  - `list(_str)` converts a string to a list,
    - it splits a string to individual characters,
  - `list(_dict)` converts a dictionary to a list,
    - it only converts the keys, and ignores the values.
- Vector:
  - vectors are lists of numbers,
  - functions for lists with number elements include `sum(l)`, `min(l)`, and `max(l)`.
- Lists are comparable:
  - start with the first elements,
  - continue until find the elements that are different,
  - ignore the remaining elements
    `[1, 7] < [2, 5]`
    `[2, 1, 2] < [2, 2, 0]`
- Two lists with same address:
  - consider two lists `t1=t2=[a, b]`,
  - changing value of one `t1[0]=c`, will also change `t2`,
  - this is different from C and Matlab,
  - to check whether two lists refer to the same object use operator `is`.
- Shortcut to create a list:
  - `[f(a) for a in A] #f(a) is a fn`
  - *list comprehension* creates a dynamic list

### D. *Dictionaries*

- Dictionary:
  - a dictionary is a mapping between *keys* and *values*
  - a key, or label, is used to access a value
  - a dictionary is a ”bag” of values, each with its own label
    - the order of items in a dictionary does not matter
  - keys can be of any type, not just integers
  - think of keys as a generalization to indices
  - dictionaries are mutable
  - dictionaries can be effectively used to count objects
- Creating & initalizing a dictionary:
  - two ways of creating an empty dictionary are
    `X=dict()`
    `X={}`

  - adds an item to the dictionary by typing
    `X['one']='meg'`
    * `X['one']` returns `'meg'`
  - can combine creation and initialization
    `X = {'one': 'meg', 'two':'yergou'}`
- From a loaded file `reader`:
  `X={row[0]:row[1] for row in reader}`
- Search:
  - for dictionary value retrieval, hashing is used
    `for _key in _dict`
    * the iterator traverses the keys, rather than the values, of the dictionary,
  - as a result, search time is about the same no matter how long the dictionary is,
  - when compared to list, a search algorithm is used, which lead to proportionally increasing searching time with the list length.
- methods:
  - `X.get(_key,_default_val)`
    * returns the *value* corresponding to `_key` if it exists, otherwise returns `_default_val`
    * motivation: note that if `_key` does not exist, Python will complain about the command `X[_key]`,
  - `X.keys()`
    * returns a list of keys
  - `X.values()`
    * returns a list of values
    * e.g., `for _value in _dict.values()`
  - `X.items()`
    * returns a list of tuples, where a tuple is a key/value pair
    * since `X.items()` is a list, it is possible to sort by keys

*E. Sets*

- Set:
  - a set is an unordered collection of distinct elements,
  - `animals = {'cat', 'dog'}`
  - `animals.add('fish')`
  - `animals.remove('fish')`
  - `'fish' in animals` is True or False.

*F. Tuples*

- Tuple:
  - a tuple is a comma separated list of values,
    * `t = 2, 7, 'a', 14`
    * optionally enclosed in `()`,
    * if a single element is created, then end it in a comma to denote it as a tuple,
      `t=5,`
    * using tuples, Python allows two assignments in one
      `a,b = 1,2`
  - can create an empty tuple as `t = tuple()`.
- Tuples and lists:
  - tuples are similar to lists,
    * can iterate over tuples, just like lists,
    * can use functions like `max()` & `min()`,
    * can compare using operators such as <, >, etc.
  - unlike lists, tuples are immutable,
    * can not use methods such as `sort()`, `append()` etc., that modify the tuple,
    * compared to lists, tuples have a reduction in methods,
    * still can replace one tuple with another
      `t=('A') + t[1:]`,

- immutability makes tuples more efficient
  - quicker, less memory etc.,
  - an example is temporary variables.

### G. Classes

- Class definition and instantiation:
  - definition
    ```
    class class_name(object):
       'optional doc string'
       self.a = 0
       def func(self):
          ...
    ```
  - the class has an optional documentation string, which can be accessed as `class_name.__doc__`
  - instantiation
    `obj=class_name()`.
- Attribute:
  - `a` is an *attribute* of `class_name` and can be accessed as `class_name.a`,
    - objects are mutable,
    - `hasattr(object, name)` checks if an attribute exists or not,
    - `getattr(object, name)` returns the attributes value
    - `isinstance(object,class)` returns true if `object` is an instance of class Class or is an instance of a subclass of Class.
    - `del class_name.a` deletes attribute `a`,
    - `__dict__` is built-in class dictionary attribute that maps attribute names and values containing the class's namespace.
- Method:
  - a *method* is a function associated with a class,
    `def function_name(self,optional_var):`
  - the object itself is regarded as the first parameter of the method
  - `__init__`
    - `__init__` is a special method that gets invoked when an object is instantiated
      - it is the constructor
    - `def __init__(self)`
  - `__str__`
    - `__str__` is a special method that returns a string representation of an object
    - gets invoked when printing an object
  - when writing a new class, start with `__init__` followed by `__str__` methods
- Subclass:
  - if `A` is a base class, then can create a *subclass* `B` that inherits class `A`
  - `class B(A):`
- Overloading operators:
  - the method `__add__`, for example, allows overloading the '+' operator in a way described in this method,
  - in other words, the behavior of an operator can be changed so that it works with user-defined types.

## III. MODULES

- Module
  - a module is a Python file that contains a collection of functions.
- Importing modules:
  - one approach to import & use a function `function_name()` from a module `module_name` is
    ```
    import module_name
    module_name.function_name()
    ```
  - equivalently
    ```
    import module_name as mn
    mn.function_name()
    ```
  - can also directly import certain variables/functions
    ```
    from module_name import filename
    ```
- `__future__`
  - To write a Python 2/3 compatible code-base, add these lines to the top of each module,
  - `from __future__ import absolute_import`
    * with absolute import, command `import foo` means `foo` is a module or package reachable from sys.path
  - `from __future__ import division`
    * `x/y` returns "true division", while `x//y` returns the "floor division"
  - `from __future__ print_function`
    * `print("H", "w")` will print as `H w` rather than `('H', 'w')`
- `six`
  - provides simple utilities for wrapping over differences between Python 2/3,
  - $6 = 2 * 3$,
  - Python 3 reorganized the standard library and moved several functions to different modules,
    * `six` provides a consistent interface to them through the fake `six.moves` module.
  - this influences the following:
    * `urllib`, a high-level interface for fetching data across the World Wide Web,
    * `xrange`
- `urllib`
  - `urllib` is a high-level interface for fetching data across the World Wide Web
  - for example to retrieve a file from WWW
    ```
    from urllib.request import urlretrieve
    urlretrieve(url, file)
    ```
- `gzip`
  - this module provides a simple interface to compress and decompress files just like the GNU programs `gzip` and `gunzip` would.
- `zipfile`
  - handles zip files
- `tempfile`
  - this module creates temporary files and directories,
  - high level interfaces provide automatic cleanup and include
    * `TemporaryFile`
    * `NamedTemporaryFile`
    * `TemporaryDirectory`
    * `SpooledTemporaryFile`
- `ctypes`
  - `ctypes` module is used if working with C dlls,
  - `c_int` is int/long in python,
  - class ctypes.WinDLL(name, mode= DEFAULT_MODE, handle=None, use_errno=False, use_last_error=False)
    * Windows only: Instances of this class represent loaded shared libraries,
    * functions in these libraries use the stdcall calling convention, and are assumed to return int by default.

- `sys`
  - `sys` module provides information about constants, functions and methods of the Python interpreter,
  - `sys.path` shows current paths stored.
  - `sys.path.append(YOUR_PATH)`.
- `os`
  - `os` module provides functions for working with files and directories,
  - `getcwd()`, gets current directory,
  - `path.abspath()`, gets the absolute path to a file,
  - `path.exists()`, checks whether a file exists,
  - `path.isdir()`, checks whether it is a directory,
  - `path.isfile()`, checks whether it is a file
  - `path.listdir()`, returns a list of files in the given directory
  - `path.dirname(path)`, returns the directory name of pathname `path`
  - `path.join`, takes a directory and a file name and joins them into a complete path.
- `argparse`
  - the `argparse`, or `ArgumentParser`, module makes it easy to write user-friendly command-line interfaces,
  - to create an `ArgumentParser` object,
    `parser=argparse.ArgumentParser()`
  - `add_argument()` method takes strings on command line and turn them into objects
  - `parse_args()` stores and uses the information.
- `copy`
  - the copy module includes two functions, `copy()` & `deepcopy()`, for duplicating existing objects,
  - `copy.copy(y)` is a *shallow copy* since it only copies everything except the embedded objects,
  - `copy.deepcopy(y)` is *deep copy* which copies everything.
- `collections`
  - implements specialized container data-types providing alternatives to Pythons general purpose built-in containers,
  - one of the data-types is `namedtuple(type_name, field_names)` factory function for creating tuple subclasses with named fields,
    * returns returns a new tuple subclass named `type_name`.
- `moviepy`
  - `moviepy.editor.VideoFileClip()`
    * `clip1 = VideoFileClip(mp4_fn)`
    * reads video-file & stores all frames in `clip1`
  - `object.fl_image()`
    * `x = clip1.fl_image(process_image)`
    * modifies every frames of `clip1`, transformed by function `process_image()`
  - `object.write_videofile()`
    * `%time clip1.write_videofile(fn)`
    * saves `clip1` in file `fn`
  - `object.save_frame()`
    * `clip1.save_frame("x.png", t=2)`
    * saves a single frame from time $t$
  - `object.subclip()`
    * `clip2 = clip1.subclip(t_s,t_e)`
    * returns video between time `t_s` and `t_e`
  - `object.iter_frames()`
    * `for frame in clip1.iter_frames():`
    * iteration through all frames
  - `concatenate_videoclips()`
    * `c=concatenate_videoclips([a,b])`
    * concatenate video clips `a` and `b` to `c`
  - `TextClip()`
    * `t = TextClip('text',options)`
    * adds text to a video
    * can control vareous attributes, such as duration,

    t.set_duration(3)

- `math`
  - includes `log10`, `log`, `sin`, `cos`, `pi`, `exp`
- `scipi`
  - *SciPy* builds on NumPy, see sub-section III-A, & providing functions that operate on NumPy arrays
  - image processing functions:
    - `img  = scipy.misc.imread('assets/cat.jpg')`
    - `img_t=scipy.misc.imresize(img, (300, 300))`
    - `scipy.misc.imsave('cat_t.jpg', img_t)`
    - `image, labels = scipy.ndimage.measurements.label(heatmap)`
      - `image` is an array the size of the `heatmap` input image where pixels are set to $0$ for background, $1$ for object number 1, 2 for car number 2 etc
      - `labels` is the number of labels found
  - Matlab mat read/write functions:
    - `scipy.io.loadmat`
    - `scipy.io.savemat`
- `PIL`
  - `PIL` stands for *Python Imaging Library*
  - `PIL` has image processing capabilities

```
from PIL import Image
im = Image.open("bride.jpg")
im.rotate(45).show()
```

- `pickle`
  - the `pickle` module implements protocols for serializing and de-serializing a Python object structure

```
pickle.dump(obj,file)
pickle.load(file)
```

- `csv`
  - `rdr = csv.reader(csv_fp)`
  - `header = next(rdr)` skips header
  - `for _line in rdr`

```
    ...
```

- `glob`
  - `glob.glob("name*.jpg")`
    - reads multiple filenames from directory
    - returns a list of filenames
- `random`
  - `x = random.random()`
  - `index = random.randint(p,q)`
- `skimage`
  - *scikit-image* is a collection of algorithms for image processing
  - `skimage.feature.hog(im_grey,...)`
    - finds *histogram of oriented gradients* (HOG) of a 1D image
    - call function as `features = hog(...)`
    - arguments:
      - `orientations=9`, or bins
      - `pixels_per_cell=(8, 8)`
      - `cells_per_block=(3, 3)` specifies the local area over which the histogram will be normalized. The HOG features for all cells in each block are computed at each block position. Furthermore, the block steps across and down through the image cell by cell
      - if there are $n$ cells horizontally/vertically, then

$$(n - \text{cells-per-block} + 1)^2 \tag{1}$$

blocks are considered

      - number of features extracted are

$$(n - \text{cells-per-block} + 1) \times (n - \text{cells-per-block} + 1) \times \text{cells-per-block} \times \text{cells-per-block} \times \text{orientations} \tag{2}$$

      - `feature_vec`: if `False` returns histograms (`features`), as 5 dimensional array as shown in (2)(for reuse), while `True` returns as a single dimensional feature vector
      - `visualize`: if `True` returns a second object which is a image visualization of hog
      - `block_norm='L1'` specifies the local area over which the histogram counts in a given cell will be normalized
- Other modules:
  - `time`
    - can use this module by measure time before and after a program is executed
    - `time.time()`
  - `SymPy` is a package for symbolic computation
  - `hashlib` to hash strings
  - `tqdm` make loops show a progress meter

    ```
    for i in tqdm(range(10000)):
    ```

### A. NumPy

- NumPy:
  - *NumPy* stands for *numerical Python*, and is pronounced "Numb Pie"
  - NumPy is the core library for scientific computing
  - NumPy adds support for large, multi-dimensional arrays and matrices along with a large library of high-level mathematical functions to operate on these arrays
  - NumPy provides familiar mathematical functions called universal functions (ufunc)
  - roughly speaking
$$\{\text{Python}, \text{NumPy}, \text{Matplotlib}\} \approx \text{Matlab}, \tag{3}$$
  - see, `http://www.numpy.org/`
  - *SciPy* (`https://www.scipy.org/`), or *scientific algorithm*, sits on top of NumPy
  - to import library, type
    `import numpy as np`
- `ndarray`
  - in NumPy the basic type is a multidimensional array
  - this class of multidimensional arrays is called `ndarray`
  - the dimensions are the *axes*
  - the number of axes is the *rank*
- `matrix`
  - `matrix` is a subclass of the array class, used for doing linear algebra,
  - `A=np.matrix([1,2,3])`
  - symbol $*$ is used for matrix multiplication,
  - `multiply()` function is used for element-wise multiplication.
- `dtype`
  - the `dtype` of an ndarray `x` is obtained by typing
    `x.dtype`
  - every element of numpy array is of the same type
  - if not explicitly stated, Numpy tries to guess a data-type when you create an array
  - in addition to python data types, NumPy provides additional types such as
    * `np.bool_` Boolean (True/False) stored as a byte
    * `np.int_` default integer type
    * `np.int8`, `int16, int32, int64`
    * `np.uint8`, `uint16, uint32`, `uint64`
    * `np.float16, float32`, `float64, float128`
    * `np.complex64`
  - can change dtype of `x` to `float32` by typing
    `y=x.astype(np.float32)`
- `NaN`
  - `NaN` (not-a-number) is a placeholder in NumPy for missing data,
  - `NaN` allows for vectorized operations,
  - `NaN` is a float value, while `None`, by definition, forces object type, which basically disables all efficiency in NumPy.
- `ndarray` properties:
  - `ndarray.ndim`
    returns the number of axes of the array
  - `ndarray.shape`
    returns the dimensions of the array, returned as a tuple
  - `ndarray.size`
    returns the total number of elements of the array
  - `ndarray.itemsize`
    returns the size in bytes of each element of the array
- Creating an `ndarray`:
  - `np.empty([2,2])`
  - `np.array(mylist)` list to np
  - `np.float32([[1,2],[3,4],[5,6]])`
  - `np.array([[1,2],[3,4]]`
  - `np.array([[1,2],[3,4]],dtype=int32)`

  - `np.zeros((3,4))`
  - `np.zeros_like(a)`
  - `np.ones((3,4))`
  - `np.full((3,4), 7)` constant matrix
  - `np.eye(3)`
  - `np.diag([1,2,3])`
  - `np.arange(_start,_end,_int)`
  - `np.linspace(_start,_end,_num)`
  - `np.meshgrid()`
    * creates a set of coordinates inside a rectangle defined by vectors `x` and `y`
    * an implementation of MATLAB's meshgrid function
    * `XX, YY = np.meshgrid(x, y)`
    * `XX` and `YY` are matrices of coordinates
  - `np.mgrid[]`
    * similar to `meshgrid()` but reversed output
    * `YY, XX = mgrid[-2:2, -1:1]`
  - `np.random.rand(shape)`
  - `np.random.uniform(low=0,high=1,size=None)`
  - `np.random.randn(shape)`
  - `np.random.normal(mean,std,size)`
  - `np.fromfunction(f,shape)`
  - growing an array is expensive, try sizing it right at creation time
- Copies:
  - when operating on arrays, their content is sometimes copied into a new array and sometimes not:
    * with `b=a` no new object it created
    * Python passes mutable objects as references, so function calls also make no copy
    * arrays have pass-by-reference semantics
  - `b = copy(a)` makes a separate copy
- Slicing:
  - `row_r1 = a[1,:]` is a 1 dimensional view of the second row of `a`
  - `row_r2 = a[1:2,:]` is a 2 dimensional view of the second row of `a`
  - `row_r1.shape` prints "(4,)"
  - `row_r2.shape` print "(1, 4)"
  - `x[[a,b,c],[d,e,f]]` is the vector
    `x[a,d],x[b,e],x[c,f]`
- `ndarray` modifications:
  - `expand_dims(a, 0)` adds dim on an axis
  - `reshape(a,b)` modifies shape
  - `reshape(a,-1)` forces a
  - `resize(a,b)` modifies array
  - `flatten()` flattens copy
  - `ravel()` flattens original
  - `transpose()` or `np.T`
  - `fliplr(a) #left-right`
  - `flipud(a) #up-down`
  - `vstack((a,b))`
    * stack along 1st axis
    * i.e. row wise or vertically
  - `hstack((a,b))`
    * stack along 2nd axis
    * i.e. horizontally, or column wise
  - `dstack((a,b,...,c))`
    * stack along the 3rd axis
    * stacks 2D arrays into a single 3D array
  - `stack((a,b,c),axis=a)`
    * join a sequence of arrays along a new axis `a`

  - `np.concatenate((a,.. c), axis=0)`
    - join a sequence of arrays along an existing axis
    - slow compared to `append()`
- Some universal functions:
  - `https://docs.scipy.org/doc/numpy/reference/ufuncs.html`
  - `np.absolute(a)`
  - `np.exp(a)`
  - `np.log(a)`
  - `np.sqrt(a)`
  - `np.square(a)`
  - `np.sign(a)`
  - `np.sin(a)`
  - `np.cos(a)`
  - `np.tan(a)`
  - `np.arctan(a)`
  - `np.arctan2(a,b)`
- `ndarray` operations:
  - operations on `ndarray` are element-wise operations
  - many of the operators are overloaded:
    - addition as `+`
    - subtraction as `-`
    - element-wise multiplication as `*`
    - element-wise division as `/`
    - exponentiation as `**`
  - can use operations `*=`, `+=` & `-=`
  - use `dot()` for matrix multiplication
- `ndarray` computations:
  - `np.dot(b)` does matrix multiplication
  - `np.dot(a,b)` same as above
  - `np.outer(a,b)` does vector outer product
  - `np.convolve(a,b)`
  - `np.sum()`
  - `np.prod()`
  - `np.cumsum()`
  - `np.min()`
  - `np.max((a,b),axis=0)`
  - `np.argmin()`
  - `np.argsort()`
  - `np.unique(a)` unique elements
  - `np.nonzero(a)` nonzero element indices
  - `np.polyfit()`
    - generates the coefficient of a polynomial, of order `n`, that fits data points
    - `coef=np.polyfit((x1,y1),(x2,y2),n)`
    - `coef=np.polyfit(x,y,3)`
  - `np.polyval()`
    - evaluates a polynomial
      `y = np.polyval(coef,x)`
  - `np.histogram()`
    - `h, e = np.histogram(im,bins=10)`
    - `h` is the histogram
    - `e` is the edges
    - argument `range={a,b)` is optional
- Miscellaneous functions:
  - `load()`
    - `net_data = np.load("alexnet.npy")`
  - `set_printoptions()`

- `np.set_printoptions(precision=3)`
- when printing an array, it sets decimal precision

## B. *Matplotlib*

- `matplotlib`
  - *Matplotlib*, is a 2D plotting library that emulates the Matlab interface
  - integrates seamlessly with `Numpy`
  - *Seaborn* sits on top of Matplotlib
    - it augments Matplotlib capabilities
  - `http://matplotlib.org/`
  - to import library, type
    `import matplotlib.pyplot as plt`
  - an IPython-specific directive that displays matplotlib plots in a notebook cell rather than in another window is
    `%matplotlib inline`
- Common methods:
  - `plt.figure(figsize=(w,h))`
    - the width (`w`) & the height (`h`) are in inches
  - `plt.subplot(2, 1, 1)`
  - `f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))`
    is an alternative approach
  - `ax1.set_title('Stacked thresholds')`
  - `ax1.imshow(color_binary)`
  - `plt.plot(x, z)`
  - `plt.hist(_x, _bins, rwidth=0.8)`
  - `plt.axis("off")`
    does not show x-y axes
  - `plt.xlabel('x axis label')`
  - `plt.ylabel('y axis label')`
  - `plt.title('z=%d' % var_name)`
  - `plt.grid()`
  - `plt.legend(['z', 'w'])`
  - `plt.show()`
- Plotting images:
  - similar to SciPy, `matplotlib` can handle images
  - `import matplotlib.image as mpimg`
  - `image = mpimg.imread('test.jpg')`
    - color space is in RGB
    - `.jpg` images are read on a scale of 0 to 255
    - `.png` images are read on a scale of 0.0 to 1.0
  - `plt.imshow(np.uint8(_img))`
  - `plt.imshow(_img, cmap='gray')`
    where `cmap` stands for *color map*

### C. *sklearn*

- `sklearn`
  - `sklearn` or *scikit-learn* is a machine learning, data mining and data analysis module
  - built on top of NumPy, SciPy, and matplotlib
- Basic machine learning tools:
  - `sklearn.utils.shuffle()`
    * `X, Y = shuffle(X, Y)`
    * shuffles the data
  - `sklearn.preprocessing.LabelBinarizer()`
    * `label_binarizer = LabelBinarizer()`
    * `y1hot=label_binarizer.fit_transform(y)`
  - `sklearn.model_selection.train_test_split()`
    * `Xt,Xv,yt,y_v=train_test_split(features,labels,test_size=0.33,random_state=0)`
    * performs both the shuffle and split
  - `sklearn.preprocessing.StandardScaler(with_mean=True, with_std=True)`
    * class that normalizes data by removing the mean and scaling to unit variance
    * `with_mean=True` removes mean value
    * `with_std=True` normalizes std
    * method `fit(X)`
      · `X` is the training data set
      · `fit()` computes the mean and std to be used for later scaling
    * method `transform(X)`
      · after determining mean and variance using `fit()`, can do normalization (standardization) using `transform()`
  - `sklearn.model_selection.GridSearchCV`
    * class that exhaustively searches parameter space
  - `sklearn.model_selection.RandomizedSearchCV`
    * class that randomly searches the parameter space
- Some supervised classifiers:
  - `clf = GaussianNB()`
  - `clf = SVC()`, or `clf = LinearSVC()`
  - `clf = DecisionTreeClassifier()`
- Common supervised classifier methods:
  - `clf.fit(X,y)`
    * `fit()` trains the classifier
    * `X` = feature vectors
    * `y` = labels
  - `y_hat = clf.predict([[-0.8, -1]])`
    * given some feature vector $x$, `predict(x)` predicts the label associated with $x$
  - `clf.score(X_test,y_true)`
    * `score()` gives the accuracy rate of the classifier on the test data `X_test`
- `sklearn.naive_bayes.GaussianNB()`
  - NB stands for naive Bayes
  - feature components are assumed independent
  - noise is assumed Gaussian
- `sklearn.svm.SVC()`
  - `svm` stands for *support vector machines*
  - `SVC` stands for *support vector classification*
  - arguments to `SVC()`:
    * `kernel` choices include
      · `kernel = 'rbf'` (default)
      · `kernel = 'linear'`
      · `kernel = 'poly'`
      · `kernel = 'sigmoid'`
    * `C` scales the soft margin cost function

        - `C` scales the total violation
        - it trades error penalty for stability
        - large `C` may minimize error rate but may also overfit
    - `gamma` is the free parameter of the Gaussian radial basis function (`rbf`)
        - term in exponent, inverse of variance
        - if small then far away points also influence decision boundary
- `sklearn.tree.DecisionTreeClassifier()`
  - some of the arguments:
    - `criterion` could be `'gini'` (default), or `'entropy'`
    - `min_samples_split=n` stops the process of splitting the tree if available samples are $< n$

*D. OpenCV*

- Generalities:
  - *OpenCV* stands for Open-Source Computer Vision
    - see opencv.org
  - 2 types of Python interfaces, `cv` & `cv2`
- `cv2`
  - `cv2` is the latest release
  - in `cv2`, everything is returned as NumPy objects, like `ndarray`, and native Python objects like lists, tuples, dictionary, etc.
  - `import cv2`
- `imread()`
  - `im = cv2.imread(fn)`
  - the default color space is BGR
- `imwrite()`
  - `cv2.imwrite('xyz.png',img)`
- `cvtColor()`
  - to convert (i.e. cvt) the color format of an image use function
    `im_o = cv2.cvtColor(im_i, flag)`
  - `flag` determines type of conversion, e.g.
    - `flag = cv2.COLOR_RGB2HSV`
    - `flag = cv2.COLOR_HSV2RGB`
    - `flag = cv2.COLOR_RGB2GRAY`
    - `flag = cv2.COLOR_BGR2RGB`
    - `flag = cv2.COLOR_BGR2HLS`
  - for an 8-bit image, the range of H is from $0 - 179$
- `resize()`
  - resizes an image
  - `_img_out=cv2.resize(_img_in,(32,32))`
  - can choose specific interpolator by adding optional argument `interpolation`
    - e.g., `interpolation=cv2.INTER_AREA`
- `findChessboardCorners()`
  - to calibrate a camera, this function determines the corners of a (possibly) distorted chessboard image
  - `ret,corn=cv2.findChessboardCorners`
    `(_im, (nx, ny), None)`
    - `_im` a distorted chessboard image in grayscale
    - `(nx, ny)` are the number of corners per row and column respectively
    - `None` means no flags
    - `ret` is a boolean variable to convey whether the corners were found
    - `corn` are the coordinates of the corners
- `drawChessboardCorners()`
  - superimposes corner coordinates on an image
  - `cv2.drawChessboardCorners(`
    `img, (nx, ny), corn, ret)`
  - parameters similar to above
- `calibrateCamera()`

```
ret, mtx, dist, rvecs, tvecs =
  cv2.calibrateCamera(objpoints,
  imgpts,gray.shape[::-1],None,None)
```

  - `objpoints` are the desired point coordinates
  - `imgpoints` are obtained, for example, from a chessboard corners
    - `objpoints` and `imgpoints` may come from multiple images
  - `,gray.shape[::-1]` is the shape of the image
  - `dist` are the distortion coefficients
  - `mtx` is the camera matrix
  - `rvecs` (rotation vector) and `trecs` (translation vector) )are the position of the camera

- `undistort()`
  - given the camera matrix `mtx` and the distortion coefficients `dist`, this function transforms distorted images to be undistorted
  - `dst = cv2.undistort(img, mtx,`
    `dist, None, mtx)`
    * `img` is a distorted image
    * `dst` is undistorted, destination image
- `getPerspectiveTransform()`
  - `M = cv2.getPerspectiveTransform(src,dst)`
  - `src` are a set of point to be be transformed to `dst`
  - `M` is the transformation
- `warpPerspective()`
  - `warped = cv2.warpPerspective(img, M,`
    `img_size, flags=cv2.INTER_LINEAR)`
  - `M` is the transform matrix
- `warpAffine()`
  - `img_o = cv2.warpAffine(img_i,M,dsize)`
  - computes an affine transformation $Ax + b$, where `M = [A b]` is $(2 \times 3)$ matrix
- `getRotationMatrix2D()`
  - to generates a rotation transformation $R$, type
    `R = cv2.getRotationMatrix2D(c,a,s)`
  - `c` is center of the rotation in image `(x,y)`
  - `a` is angle in degrees
  - `s` scales the image, `s=1` does not scale
  - `R` is $(2 \times 3)$ matrix that can be used for an affine transformation
- `line()`
  - `line` adds a line on an image `img`, given two points $(x1, y1)$ & $(x2, y2)$
  - `cv2.line(im,(x1,y1),(x2,y2),clr,thk)`
  - `clr` is color triplet, and `thk` is line thickness
- `rectangle()`
  - `cv2.rectangle(im,(x1,y1),(x2,y2),`
    `clr,thk)`
  - `clr` is color triplet, and `thk` is line thickness
- `putText()`
  `cv2.putText(im,'text',pos,font,1,c,4)` where
  - `pos` is starting position $(x, y)$
  - `font=cv2.FONT_HERSHEY_SIMPLEX`
  - `1` is the text size
  - `c` is color $(r, g, b)$
  - `4` is thickness
- `addWeighted()`
  - weighted sum of two arrays can be computed as
    `c=cv2.addWeighted(im1,alph,im2,bet,0)`
- `fillPoly()`
  - to fill the area bounded by a polygon type
    `cv2.fillPoly(img,vertices,`
    `ignore_mask_color)`
- `inRange()`
  - `inRange` checks if array elements of `src`, lie between two thresholds `lowerb` and `upperb`
    `cv2.inRange(src,lowerb,upperb,dst)`
  - `dst(I)` is set to
    * 255 if `src(I)` is within the specified region, and
    * 0 otherwise
- `bitwise_and()`
  - `bitwise_and` is used to mask images

`masked_im=cv2.bitwise_and(im,mask)`

- `GaussianBlur()`
  - Gaussian smoothing can be achieved by function `i=cv2.GaussianBlur(im,(k_s,k_s),0)`
  - for example, set the kernel size `k_s=3`
- `Sobel`
  - `sobelx=cv2.Sobel(im,cv2.CV_64F,1,0)`
  - `sobely=cv2.Sobel(im,cv2.CV_64F,0,1)`
  - the image `im`, should have depth 1, e.g. grey
  - option `ksize=5` sets kernel size
    - default is 3
- `Canny()`
  - Canny Edge Detection `edges=cv2.Canny(gray,low_thr,high_thr)`
  - applies a $5 \times 5$ Gaussian smoothing internally
  - first detects pixels with gradient above the `high_thr`, and reject pixels below the `low_thr`
    - pixels with values between `low_thr` and `high_thr` will be included as long as they are connected to strong edges
  - `edges` is a binary image over a black background, with white pixels tracing the detected edges
  - John Canny recommended a low to high ratio of $1:2$ or $1:3$
- `HoughLinesP()`
  - Hough transformation is performed by the function

    ```
    lines = cv2.HoughLinesP(edges, rho,
       theta, thr, np.array([]),
       min_line_length, max_line_gap)
    ```

  - `edges` is an image of edges, e.g. from `Canny`
  - `lines` is an array containing the endpoints $(x_1, y_1, x_2, y_2)$ of all detected line segments
  - `rho` & `theta` are the distance & angular resolutions
    - `rho` in units of pixels and `theta` in units of radians
  - `thr` specifies the minimum number of votes (intersections in a given grid cell) a candidate line needs to have to make it into the output
  - `np.array([])` is just an empty placeholder
  - `min_line_length` is the minimum length of a line (in pixels) that you will accept in the output
  - `max_line_gap` is the maximum distance (again, in pixels) between segments that you will allow to be connected into a single line
- `VideoCapture()`
  - `x = cv2.VideoCapture('x.mp4')`
  - `success,img = vidcap.read()`
  - `cv2.imwrite("frame%d.jpg"%count,img)`
- `matchTemplate()`
  - `r=cv2.matchTemplate(im,tmp,method)`
  - compares a template `tmp` against overlapped image regions of `im`
  - it returns `r`, a single channel image whose entries are the correlation matches for each point
  - multiple methods are possible such as `cv2.TM_CCORR_NORMED`
- `minMaxLoc()`
  - `minV,maxV,minL,maxL=cv2.minMaxLoc(r)`
  - `r` is like the output of `matchTemplate()`
  - returns minimum/maximum values and locations
  - a location is a pair `(x,y)`
  - for detection use `minV` or `maxV` depending on `method` used in `matchTemplate()`