# ELEG 5491: Introduction to Deep Learning

## Python Programming Basics

Prof. LI Hongsheng

Office: SHB 428
e-mail: hsli@ee.cuhk.edu.hk
web: https://blackboard.cuhk.edu.hk

Department of Electronic Engineering
The Chinese University of Hong Kong

Feb. 2023

# Python

- Open source general-purpose language
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment
- Does not need compile;
- Many good packages for various purposes: web, file, multi-thread, AI, etc.

- Documentation: http://www.python.org/doc/
- Free book: http://www.diveintopython.org

- **IMPORTANT:** Learn to use Google. At most times, it will bring you to some answers on Stack Overflow

# Python version and distributions

- Latest version is 3.7
- Version 2.7.x will not be supported after 2020

If you start from scratch, you should use Python 3.x

- Python has different distributions that pre-install a large number of useful packages: `numpy`, `scipy`, `scikit-learn`, `matplotlib`, etc.
- For scientific computing, Anaconda is recommended
- Download: https://www.anaconda.com/download/
- Search package in Anaconda package pools
  ```
  conda search scipy
  ```
- Install package from Anaconda package pools
  ```
  conda install scipy
  ```
- One can also use the Python Package Index (PyPI) for package management
  ```
  pip install scipy
  ```

# Python interactive interface

- After installation, in your commnd window, type in “python”
- Python has an interactive interface

  ```
  % python Python 3.6.0b2+ (3.6:84a3c5003510+, Oct 26 2016, 02:33:55)
  [GCC 6.2.0 20161005]
  Type “help”, “copyright”, “credits” or “license” for more information.
  >>>
  ```

- Python interpreter evaluates inputs

  ```
  >>> 3*(7+2)
  27
  ```

- Python prompts with “>>>”
- To exit Python:

  Ctrl-D

# Run Python program via commands

- A sample code

## Sample code

```
 def main():
    x = 34 - 23                 # A comment.
    y = "Hello"                 # Another one.
    z = 3.45
    if z == 3.45 or y == "Hello":
        x = x + 1
        y = y + " World"         # String concat.
    print (x)
    print (y)

if __name__ = "__main__":
    main()
```

- Execution results

## Running results

```
12
Hello World
```

# Run Python program via commands

- Write a Python program using text editor and save it as “myscript.py”
- Run Python program via command window

### Run Python program

```
python myscript.py arg1 arg2
```

# Syntax basics

- Assignment uses = and comparison uses ==
- For numbers, + - * / % are as expected
- Logical operators are words (`and`, `or`, `not`), not symbols
- The basic printing command is `print`
- The first assignment to a variable creates it
  - Variable types don't need to be declared
  - Python figures out the variable types on its own
  - One cannot access non-existent names
  - Avoid using system pre-defined names

- Whitespace
  - Whitespace is meaningful in Python: especially indentation and placement of newlines.
  - By default, four whitespace (" ") = one indentation
  - Use a newline to end a line of code. Use \ when must to the next line prematurally
  - No braces { } to mark blocks of code in Python. Use consistent indentation instead
  - The first line with less indentation is outside of the block
  - The first line with more indentation starts a nested block

# Syntax basics

- Comments
  - Start comments with #: the rest of line is ignored
  - Can include a "documentation string" as the first line of any new function or class that you define
  - It is of good practice to include one to explain the function

```
def my_function(x,y):
    """This is the docstring.
    This function does blah blah blah."""
    # The code goes here
```

- Basic datatypes
  - Integers (default for numbers)

    ```
    z = 5 / 2     # Answer is 2, integer division
    ```

  - Floats

    ```
    z = 3.456
    ```

  - String
    - Can use `""` or `' '` to specify.

      ```
      "abc" 'abs'
      ```

    - Unmatched can occur within the string

      ```
      "matt's"
      ```

    - Use triple double-quotes for multi-line strings or strings than contain both ' and " inside of them

      ```
      """a'b"c"""
      ```

# Syntax basics

- Naming rules
  - Names are case sensitive and cannot start with a number
  - They can contain letters, numbers, and underscores

  `bob Bob _bob _2_bob_ bob_2 BoB`
- Reserved names

  `and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while`

# Control of flow

- The following code are self-explanatory

```
if x == 3:
    print “X equals 3.”
elif x == 2:
    print “X equals 2.”
else:
    print “X equals something else.”
print “This is outside the ‘if’.”
```

```
assert(number_of_players < 5)
```

```
x = 3
while x < 10:
   if x > 7:
       x += 2
       continue
   x = x + 1
   print “Still in the loop.”
   if x == 8:
       break
print “Outside of the loop.”
```

```
for x in range(10):
   if x > 7:
       x += 2
       continue
   x = x + 1
   print “Still in the loop.”
   if x == 8:
       break
print “Outside of the loop.”
```

# Sequence types

- Tuple
  - A simple immutable ordered sequence of items
  - Items can be of mixed types, including collection types
- Strings
  - Immutable
  - Conceptually very much like a tuple
- List
  - Mutable ordered sequence of items of mixed types
- All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality
- Key difference: Lists are mutable
- The operations shown can be applied to all sequence types

# Sequence types

- Tuples are defined using parentheses (and commas)

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Lists are defined using square brackets (and commas)

```
>>> li = [“abc”, 34, 4.34, 23]
```

- Strings are defined using quotes (”, ’, or ”””).

```
>>> st = “Hello World”
>>> st = ‘Hello World’
>>> st = “““This is a multi-line
string that uses triple quotes.”””
```

- Can access individual members using square bracket “array” notation
- Indices are 0-based

```
>>> tu = (23, ‘abc’, 4.56, (2,3), ‘def’)
>>> tu[1]     # Second item in the tuple.
 ‘abc’

>>> li = [“abc”, 34, 4.34, 23]
>>> li[1]      # Second item in the list.
 34

>>> st = “Hello World”
>>> st[1]   # Second character in string.
 ‘e’
```

# Indices & slicing

```
>>> t = (23, ‘abc’, 4.56, (2,3), ‘def’)
```

- Positive index: count from the left, starting with 0

```
>>> t[1]
‘abc’
```

- Negative lookup: count from right, starting with −1

```
>>> t[-3]
4.56
```

- Slicing: return a **copy** of the container with a subset of the original members. Start copying at the first index, and stop copying **before** the second index

```
>>> t[1:4]
(‘abc’, 4.56, (2,3))
```

- Can also use negative indices when slicing

```
>>> t[1:-1]
(‘abc’, 4.56, (2,3))
```

- Omit the first index to make a copy starting from the beginning of the container

```
>>> t[:2]
(23, ‘abc’)
```

- Omit the second index to make a copy starting at the first index and going to the end of the container

```
>>> t[2:]
(4.56, (2,3), ‘def’)
```

# Operators

- **in**: test whether a value in a container or a substring in a string

```
>>> t = [1, 2, 4, 5]
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- **+**: produces a **new** tuple, list, or string whose value is the concatenation of its arguments

```
>>> (1, 2, 3) + (4, 5, 6)
 (1, 2, 3, 4, 5, 6)

>>> [1, 2, 3] + [4, 5, 6]
 [1, 2, 3, 4, 5, 6]

>>> “Hello” + “ ” + “World”
 ‘Hello World’
```

# Operators & assignment

- *: produces a new tuple, list, or string that "repeats" the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)

>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]

>>> “Hello” * 3
‘HelloHelloHello’
```

- For simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect
- For other mutable datatypes (lists, dictionaries, user-defined types), assignment works differently
  - When we change these data, we do it in place
  - We don't copy them into a new memory address each time
  - If we type `y=x` and then modify `y`, both `x` and `y` are changed

*immutable*

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

*mutable*

```
x = some mutable object
y = x
make a change to y
look at x
x will be changed as well
```

# Operations for lists

- Lists are mutable

```
>>> li = [‘abc’, 23, 4.34, 23]
>>> li[1] = 45
>>> li
  [‘abc’, 45, 4.34, 23]
```

- We can change lists in place
- Name `li` still points to the same memory reference when we are done
- The mutability of lists means that they are not as fast as tuples

- `append` & `insert`

```
>>> li = [1, 11, 3, 4, 5]

>>> li.append(‘a’)   # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, ‘a’]

>>> li.insert(2, ‘i’)
>>>li
[1, 11, ‘i’, 3, 4, 5, ‘a’]
```

# Operations for lists: extend and +

- + creates a fresh list (with a new memory reference)
- extend operates on list li in place

```
>>> li.extend([9, 8, 7])
>>>li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7]
```

**Differences:**

- extend takes a list as an argument
- append takes a singleton as an argument

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, ‘i’, 3, 4, 5, ‘a’, 9, 8, 7, [10, 11, 12]]
```

# Other operations for lists

```
>>> li = [‘a’, ‘b’, ‘c’, ‘b’]
>>> li.index(‘b’)      # index of first occurrence
1
>>> li.count(‘b’)      # number of occurrences
2
>>> li.remove(‘b’)     # remove first occurrence
>>> li
  [‘a’, ‘c’, ‘b’]


>>> li = [5, 2, 6, 8]
>>> li.reverse()     # reverse the list *in place*
>>> li
  [8, 6, 2, 5]
>>> li.sort()        # sort the list *in place*
>>> li
  [2, 5, 6, 8]
>>> li.sort(some_function)
    # sort in place using user-defined comparison
```

- **IMPORTANT:** Learn to read documentation
  https://docs.python.org/3/tutorial/datastructures.html

# String formatting

- Since Python 3.x, `str.format()` is recommended for formatting strings
- With `str.format()`, the replacement fields are marked by curly braces

```
>>> name = ‘Eric’
>>> age = 74
>>> "Hello, {}.  You are {}.".format(name, age)
‘Hello, Eric.  You are 74.’
```

- Reference variables in any order by referencing their index

```
>>> "Hello, {1}.  You are {0}.".format(age, name)
‘Hello, Eric.  You are 74.’
```

- Reference variable names

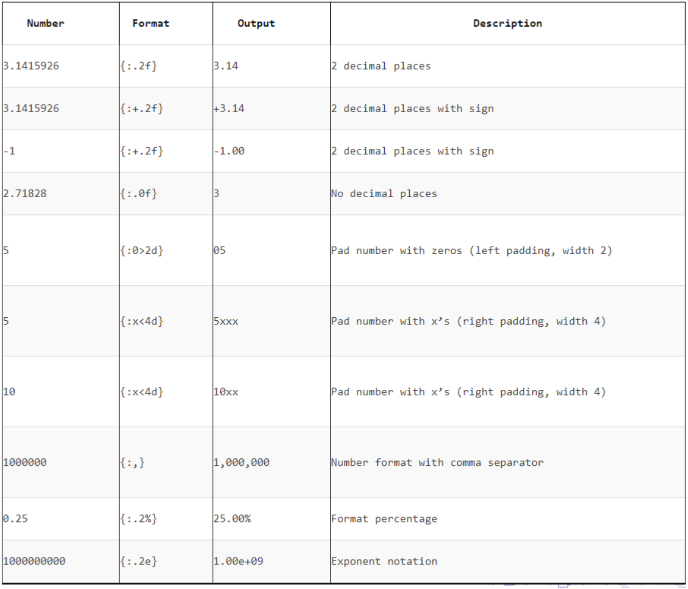
```
>>> "Hello, {name}.  You are {age}.".format(name=’Eric’,
age=74)
‘Hello, Eric.  You are 74.’
```

- Specify detailed number format

```
>>> "Hello, {name}.  You are {age:+.2f}.".format(
name=‘Eric’, age=74)
‘Hello, Eric.  You are +74.00.’
```

# String formatting

A reference table for string formatting



| Number | Format | Output | Description |
|---|---|---|---|
| 3.1415926 | {:.2f} | 3.14 | 2 decimal places |
| 3.1415926 | {:+.2f} | +3.14 | 2 decimal places with sign |
| -1 | {:+.2f} | -1.00 | 2 decimal places with sign |
| 2.71828 | {:.0f} | 3 | No decimal places |
| 5 | {:0>2d} | 05 | Pad number with zeros (left padding, width 2) |
| 5 | {:x<4d} | 5xxx | Pad number with x’s (right padding, width 4) |
| 10 | {:x<4d} | 10xx | Pad number with x’s (right padding, width 4) |
| 1000000 | {:,} | 1,000,000 | Number format with comma separator |
| 0.25 | {:.2%} | 25.00% | Format percentage |
| 1000000000 | {:.2e} | 1.00e+09 | Exponent notation |

# Dictionaries

- Dictionaries store a mapping between a set of keys and a set of values
- Keys can be any immutable type
- Values can be any type
- A single dictionary can store values of different types
- Can define, modify, view, lookup, and delete the key-value pairs in the dictionary

```
>>> d = {‘user’:‘bozo’, ‘pswd’:1234}
>>> d[‘user’]
‘bozo’
>>> d[‘pswd’]
1234
>>> d[‘bozo’]

Traceback (innermost last):
  File ‘<interactive input>’ line 1, in ?
KeyError: bozo

>>> d = {‘user’:‘bozo’, ‘pswd’:1234}
>>> d[‘user’] = ‘clown’
>>> d
{‘user’:‘clown’, ‘pswd’:1234}

>>> d[‘id’] = 45
>>> d
{‘user’:‘clown’, ‘id’:45, ‘pswd’:1234}
```

```
>>> d = {‘user’:‘bozo’, ‘p’:1234, ‘i’:34}
>>> del d[‘user’]           # Remove one.
>>> d
{‘p’:1234, ‘i’:34}
>>> d.clear()               # Remove all.
>>> d
{}

>>> d = {‘user’:‘bozo’, ‘p’:1234, ‘i’:34}
>>> d.keys()             # List of keys.
[‘user’, ‘p’, ‘i’]
>>> d.values()           # List of values.
[‘bozo’, 1234, 34]
>>> d.items()       # List of item tuples.
[(‘user’,‘bozo’), (‘p’,1234), (‘i’,34)]
```

# For loop

- The for loop in python generally loop over a sequence or a dictionary

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
```

- range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):
    print(x)
```

- Increment the sequence from 2 to 30 (but not including 20) with 3 (default is 1)

```
for x in range(2, 30, 3):
    print(x)
```

- Use enumerate() to output the numeric index in the loop

```
my_list = [’apple’, ’banana’, ’grapes’, ’pear’]
for c, value in enumerate(my_list):
    print(c, value)

# Outputs:
# 1 apple
# 2 banana
```

# Functions

- def creates a function and assigns it a name. return sends a result back to the caller
- Arguments are passed by assignment
- Arguments and return types are not declared

```
def <name>(arg1, arg2, ..., argN):
  <statements>
  return <value>

def times(x,y):
  return x*y
```

- Arguments are passed by assignment
- Passed arguments are assigned to local names
- Assignment to argument names don't affect the caller
- Changing a mutable argument may affect the caller

```
def changer (x,y):
  x = 2                       # changes local value of x only
  y[0] = 'hi'                 # changes shared object
```

# Functions

- Can define default values for arguments that need not be passed

```
def func(a, b, c=10, d=100):
 print a, b, c, d

>>> func(1,2)
1 2 10 100

>>> func(1,2,3,4)
1,2,3,4
```

- All functions in Python have a return value (even no `return` in the code)
- Functions without a return return the special value `None`
- There is no function overloading (functions can't have the same name) in Python
- Functions can be used as any other datatypes. They can be
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc

# Modules

- Code reuse:
  - routines can be called multiple times within a program
  - Routines can be used from multiple programs
- Namespace partitioning
  - group data together with functions used for that data
- Implementing shared services or data
  - Can provide global data structure that is accessed by multiple subprograms

- Modules are functions and variables defined in separate files
- Items are imported using from or import

```
from module import function
function()

import module
module.function()

import module as md
md.function()
```

- Modules are namespaces. Can be used to organize variable names

```
atom.position = atom.position - molecule.position
```

# Classes & objects

- A software item that contains variables and methods
- Object Oriented Design focuses on
  - Encapsulation: dividing the code into a public interface, and a private implementation of that interface
  - Polymorphism: the ability to overload standard operators so that they have appropriate behavior based on their context
  - Inheritance: the ability to create subclasses that contain specializations of their parents
- Example

```
class atom(object):
  def __init__(self,atno,x,y,z):
      self.atno = atno
      self.position = (x,y,z)
  def symbol(self):   # a class method
      return Atno_to_Symbol[atno]
  def __repr__(self): # overloads printing
      return '%d %10.4f %10.4f %10.4f' %
             (self.atno, self.position[0],
              self.position[1],self.position[2])

>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```

# Atom & molecule classes

- Overloaded the default constructor
- Object Oriented Design focuses on
- Defined class variables (atno,position) that are persistent and local to the atom object
- Good way to manage shared memory
- Overloaded the print operator

- Use the atom class to build molecules

```
class atom(object):
  def __init__(self,atno,x,y,z):
      self.atno = atno
      self.position = (x,y,z)
  def symbol(self):   # a class method
      return Atno_to_Symbol[atno]
  def __repr__(self): # overloads printing
      return '%d %10.4f %10.4f %10.4f' %
             (self.atno, self.position[0],
              self.position[1],self.position[2])

>>> at = atom(6,0.0,1.0,2.0)
>>> print at
6  0.0000  1.0000 2.0000
>>> at.symbol()
'C'
```