CS 5489 Machine Learning

Lecture 1a: Python Tutorial

Prof. Antoni B. Chan

Dept. of Computer Science, City University of Hong Kong

Why Python?

- General-purpose high-level programming language
- Design philosophy emphasizes programmer productivity and code readability
  - "executable pseudo-code"
- Supports multiple programming paradigms
  - object-oriented, imperative, functional
- Dynamic typing and automatic memory management

What is special about Python?

- Object-oriented: everything is an object
- Clean: usually one way to do something, not a dozen
- Easy-to-learn: learn in 1-2 days
- Easy-to-read
- Powerful: full-fledged programming language

Applications for Python

- Scientific Computing
  - numpy, scipy, ipython
- Data Science, Deep Learning
  - scikit-learn, matplotlib, pandas, keras, tensorflow
- Web & Internet Development
  - Django – complete web application framework
  - model-view-controller design pattern
  - templates, web server, object-relational mapper

Disadvantages of Python

- Not as fast as Java or C
- However, you can call C-compiled libraries from Python (e.g. Boost C++)
- Alternatively, Python code can be compiled to improve speed
  - Cython and PyPy
  - requires type of variables to be declared

Installing Python

- We will use Python 3
  - Python 3 is not backwards compatible with Python 2.7

- Anaconda (https://www.anaconda.com/download)
    - single bundle includes most scientific computing packages.
        - package manager for installing other libraries
    - make sure to pick version for **Python 3**.
    - easy install packages for Windows, Mac, Linux.
        - (single directory install)

Running Python

- Interactive shell (ipython)
    - good for learning the language, experimenting with code, testing modules

---

```
Nori:CS5489 abc$ ipython
Python 3.5.4 |Anaconda, Inc.| (default, Oct  5 2017, 02:58:14)
Type "copyright", "credits" or "license" for more information.

IPython 4.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hello, World")
Hello, World

In [2]:
Do you really want to exit ([y]/n)? y
Nori:CS5489 abc$
```

- Script file (hello.py)

```python
#!/usr/bin/python
print("Hello, World")
```

- Standalone script
    - explicitly using python interpreter

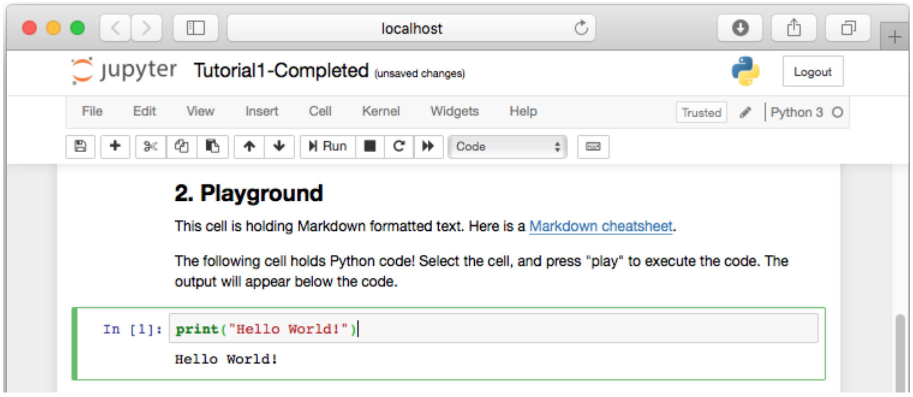
```
Nori:~ abc$ python hello.py
Hello, World
```

- using magic shebang (Linux, Mac OS X)

```
Nori:~ abc$ ./hello.py
Hello, World
```

Jupyter (ipython notebooks)

- Launch from *Anaconda Navigator*
- browser-based interactive computing environment
    - development, documenting, executing code, viewing results (inline images)
    - whole session stored in notebook document (.ipynb)
    - (also made and presented these slides!)



2. Playground

This cell is holding Markdown formatted text. Here is a Markdown cheatsheet.

The following cell holds Python code! Select the cell, and press "play" to execute the code. The output will appear below the code.

```
In [1]: print("Hello World!")

        Hello World!
```

Jupyter tips

- Keyboard shortcuts
  - there are a lot of keyboard shortcuts for moving between cells, running cells, deleting and inserting cells.
- Starting directory
  - use the `--notebook-dir=mydir` option to start the notebook in a particular directory.
  - Windows: create a shortcut to run `jupyter-notebook.exe --notebook-dir=%userprofile%`.

- Problems viewing SVG images in ipynb
  - SVG images may not display due to the serurity model of Jupyter.
  - select "Trust Notebook" from the "File" menu to show the SVG images.
- View ipynb in slideshow mode in a web browser (like this presentation!)

```
jupyter-nbconvert --to slides file.ipynb --post serve
```

- can also use the RISE plugin to present directly from the juptyer notebook.
- info, info
- Convert to HTML to view statically in web browser

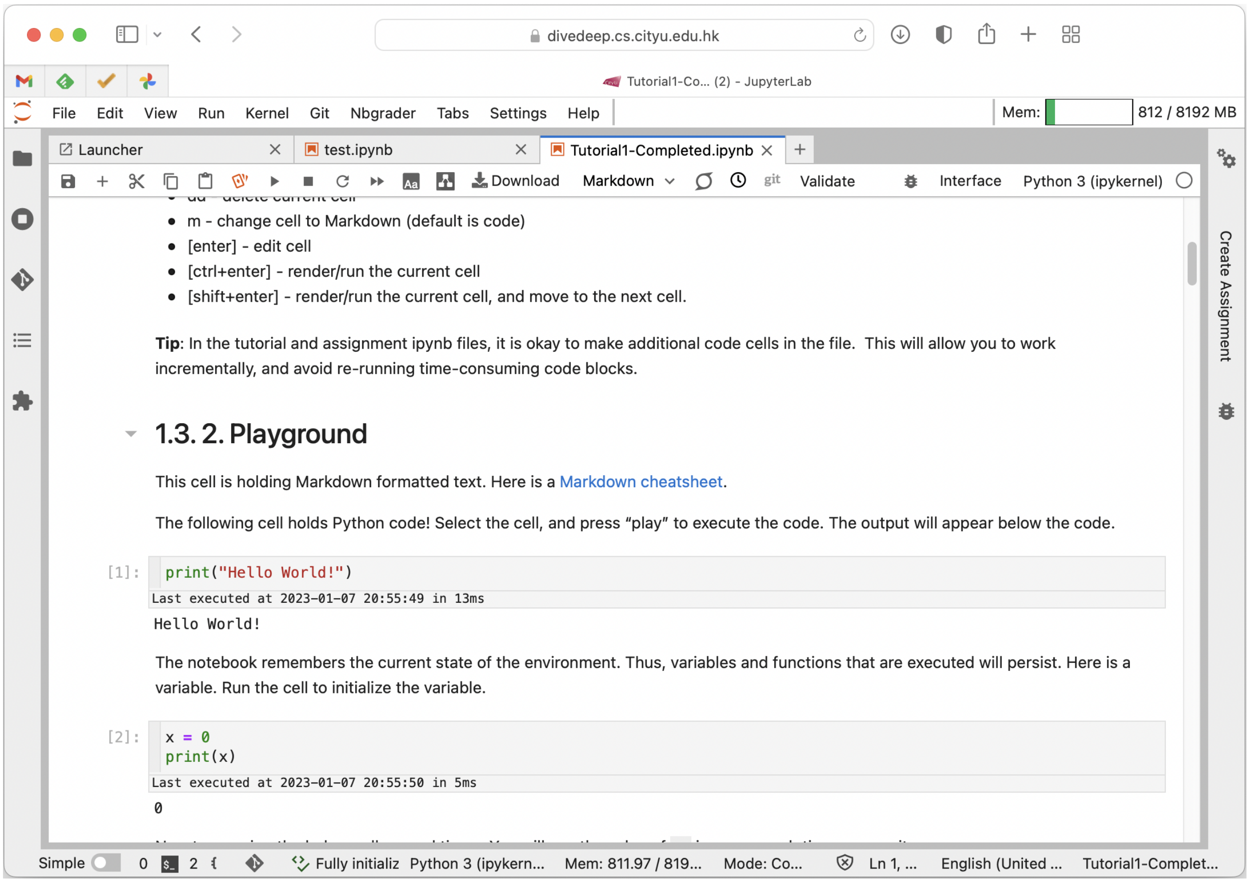
```
jupyter-nbconvert file.ipynb
```

- ValueError when using matplotlib in Jupyter
  - This mainly affects Mac where the OS locale is set to a non-English language. Open "Terminal" app and go to Preferences -> Profiles -> Terminal -> Enviornment. Deselect the option "Set locale variables automatically".
  - more info: http://stackoverflow.com/questions/15526996/ipython-notebook-locale-error
- MacOS and Anaconda
  - MacOS has a builtin python distribution. If you are using anaconda, make sure that you use the correct command-line commands. You can add "/anaconda3/bin/" in front of the command to make sure you are using the anaconda version (or the appropriate base direcotry for anaconda3). Otherwise, it may default to the builtin python.

CS Lab Resources

- JupyterHub
  - Jupyter notebooks run on a central server - shared CPU and GPU
  - JupyterLab (IDE): https://divedeep.cs.cityu.edu.hk/cs5489_22b
- Linux machines
  - there are several computing clusters in CS.
  - High Throughput GPU Cluster 1 (HTGC1)
  - High Throughput GPU Cluster 2 (HTGC2)
  - High Throughput GPU Cluster 3 (HTGC3)
- Windows machines

- MMW2462 in CS lab contains GPU workstations.
- Google colab: https://colab.research.google.com/
  - provided by Google. Some limitations on running time (12 hours) and memory usage.
- More details are on Canvas.



- m - change cell to Markdown (default is code)
- [enter] - edit cell
- [ctrl+enter] - render/run the current cell
- [shift+enter] - render/run the current cell, and move to the next cell.

Tip: In the tutorial and assignment ipynb files, it is okay to make additional code cells in the file. This will allow you to work incrementally, and avoid re-running time-consuming code blocks.

1.3. 2. Playground

This cell is holding Markdown formatted text. Here is a Markdown cheatsheet.

The following cell holds Python code! Select the cell, and press “play” to execute the code. The output will appear below the code.

```
[1]: print("Hello World!")
Last executed at 2023-01-07 20:55:49 in 13ms
Hello World!
```

The notebook remembers the current state of the environment. Thus, variables and functions that are executed will persist. Here is a variable. Run the cell to initialize the variable.

```
[2]: x = 0
     print(x)
Last executed at 2023-01-07 20:55:50 in 5ms
0
```

Outline

1. Python Intro
2. **Python Basics (identifiers, types, operators)**
3. Control structures (conditional and loops)
4. Functions, Classes
5. File IO, Pickle, pandas
6. NumPy
7. matplotlib
8. probability review

Python Basics

- Formatting
  - case-sensitive
  - statements end in **newline** (not semicolon)
    - use semicolon for multiple statements in one line.
  - **indentation** for code blocks (after a colon).

```
In [1]: print("Hello")
print("Hello"); print("World")
name = "Bob"
if name == "George":
    print("Hi George")
else:
    print("Who are you?")
```

```
Hello
Hello
World
Who are you?
```

- single-line comments with `#`
- multi-line statements continued with backslash (`\`)
    - not required inside `{}`, `()`, or `[]` for data types

In [2]:

```
# this is a comment
a=1        # comments also can go after statements
b=2; c=3   # here too

# multiple line statement
x = a + \
    b + c

# backslash not needed when listing multi-line data
y = [1, 2,
     3, 4]
```

Identifiers and Variables

- Identifiers
    - same as in C
- Naming convention:
    - `ClassName` -- a class name
    - `varName` -- other identifier
    - `_privateVar` -- private identifier
    - `__veryPrivate` -- strongly private identifier
    - `__special__` -- language-defined special name

- Variables
    - no declaration needed
    - no need for declaring data type (automatic type)
    - need to assign to initialize
        - use of uninitialized variable raises exception
    - automatic garbage collection (reference counts)

Basic Types

- Integer number

In [3]:

```
4
int(4)
```

Out[3]:

```
4
```

- Real number (float)

In [4]:

```
4.0
float(4)
```

Out[4]:

```
4.0
```

- Boolean

In [5]:

```
True
False
```

Out[5]:

```
False
```

- String literal

In [6]:

```
"a string"
'a string'
"concatenate " "two string literals"
"""this is a multi-line string.
it keeps the newline."""
r'raw string\no escape chars'
```

Out[6]:

```
'raw string\\no escape chars'
```

Lists

- Lists can hold anything (even other lists)

In [7]:

```
myList = ['abcd', 786, 2.23]
print(myList)      # print the list
```

```
['abcd', 786, 2.23]
```

In [8]:

```
print(myList[0])  # print the first element (0-indexed)
```

```
abcd
```

- Creating lists of numbers

In [9]:

```
a = range(5)   # list of numbers from 0 to 4
print(a)
print(list(a))
```

```
range(0, 5)
[0, 1, 2, 3, 4]
```

In [10]:

```
b = range(2,12,3)  # numbers from 2 to 11, count by 3
print(b)
print(list(b))
```

```
range(2, 12, 3)
[2, 5, 8, 11]
```

- append and pop

In [11]:

```
a = list(range(0,5))
a.append('blah')  # add item to end
print(a)
```

```
[0, 1, 2, 3, 4, 'blah']
```

In [12]:

```
a.pop()  # remove last item and return it
```

Out[12]:

```
'blah'
```

- insert and delete

In [13]:

```
a.insert(0,42)  # insert 42 at index 0
print(a)
```

```
[42, 0, 1, 2, 3, 4]
```

In [14]:

```
del a[2]     # delete item 2
print(a)
```

```
[42, 0, 2, 3, 4]
```

- more list operations

In [15]:

```
a.reverse()    # reverse the entries
print(a)
```

```
[4, 3, 2, 0, 42]
```

In [16]:

```
a.sort()      # sort the entries
print(a)
```

```
[0, 2, 3, 4, 42]
```

Tuples

- Similar to a list
  - but immutable (read-only)
  - cannot change the contents (like a string constant)

In [17]:

```
# make some tuples
x = (1,2,'three')
print(x)
```

```
(1, 2, 'three')
```

In [18]:

```
y = 4,5,6             # parentheses not needed!
print(y)
```

```
(4, 5, 6)
```

In [19]:

```
z = (1,)    # tuple with 1 element (the trailing comma is required)
print(z)
```

```
(1,)
```

Operators on sequences

- *Same operators* for strings, lists, and tuples
- Slice a sublist with colon (`:`)
  - **Note**: the 2nd argument is not inclusive!

In [20]:

```
"hello"[0]     # the first element
```

Out[20]:

```
'h'
```

In [21]:

```
"hello"[-1]    # the last element (index from end)
```

Out[21]:

```
'o'
```

In [22]:

```
"hello"[1:4]  # the 2nd through 4th elements
```

Out[22]:

```
'ell'
```

In [23]:

```
"hello"[2:]    # the 3rd through last elements
```

Out[23]:

```
'llo'
```

In [24]:

```
"hello"[0:5:2] # indices 0,2,4 (by 2)
```

Out[24]:
```
'hlo'
```

- Other operators on string, list, tuple

In [25]:
```
len("hello")   # length
```

Out[25]:
```
5
```

In [26]:
```
"he" + "llo"   # concatenation
```

Out[26]:
```
'hello'
```

In [27]:
```
"hello"*3      # repetition
```

Out[27]:
```
'hellohellohello'
```

String methods

- Useful methods

In [28]:
```
"112211".count("11")         # 2
"this.com".endswith(".com")  # True
"wxyz".startswith("wx")      # True
"abc".find("c")              # finds first: 2
",".join(['a', 'b', 'c'])    # join list: 'a,b,c'
"aba".replace("a", "d")      # replace all: "dbd"
"a,b,c".split(',')           # make list: ['a', 'b', 'c']
"  abc    ".strip()          # "abc",  also rstrip(), lstrip()
```

Out[28]:
```
'abc'
```

- String formatting: automatically fill in type

In [29]:
```
"{} and {} and {}".format('string', 123, 1.6789)
```

Out[29]:
```
'string and 123 and 1.6789'
```

- String formatting: specify type (similar to C)

In [30]:
```
"{:d} and {:f} and {:0.2f}".format(False, 3, 1.234)
```

Out[30]:
```
'0 and 3.000000 and 1.23'
```

Dictionaries

- Stores key-value pairs (associative array or hash table)
  - key can be a string, number, or tuple

In [31]:
```
mydict = {'name': 'john', 42: 'sales', ('hello', 'world'): 6734}
print(mydict)
```

```
{'name': 'john', 42: 'sales', ('hello', 'world'): 6734}
```

- Access

In [32]:
```
print(mydict['name'])         # get value for key 'name'
```

```
john
```

In [33]:
```
mydict['name'] = 'jon' # change value for key 'name'
mydict[2] = 5          # insert a new key-value pair
```

```
print(mydict)
```

```
{'name': 'jon', 42: 'sales', ('hello', 'world'): 6734, 2: 5}
```

In [34]:

```
del mydict[2]           # delete entry for key 2
print(mydict)
```

```
{'name': 'jon', 42: 'sales', ('hello', 'world'): 6734}
```

- Other operations:

In [35]:

```
mydict.keys()            # iterator of all keys (no random access)
```

Out[35]:

```
dict_keys(['name', 42, ('hello', 'world')])
```

In [36]:

```
list(mydict.keys())      # convert to a list for random access
```

Out[36]:

```
['name', 42, ('hello', 'world')]
```

In [37]:

```
mydict.values()          # iterator of all values
```

Out[37]:

```
dict_values(['jon', 'sales', 6734])
```

In [38]:

```
mydict.items()           # iterator of tuples (key, value)
```

Out[38]:

```
dict_items([('name', 'jon'), (42, 'sales'), (('hello', 'world'), 6734)])
```

In [39]:

```
'name' in mydict  # check the presence of a key
```

Out[39]:

```
True
```

Operators

- Arithmetic: `+`, `-`, `*`, `/`, `%`, `**` (exponent), `//` (floor division)

In [40]:

```
print(6/4)      # float division
```

```
1.5
```

In [41]:

```
print(6//4)    # integer division
```

```
1
```

In [42]:

```
print(6//4.0)  # floor division
```

```
1.0
```

- Assignment: `=`, `+=`, `-=`, `/=`, `%=`, `**=`, `//=`
- Equality: `==`, `!=`
- Compare: `>`, `>=`, `<`, `<=`
- Logical: `and`, `or`, `not`

- Membership: `in`, `not in`

In [43]:

```
2 in [2, 3, 4]
```

Out[43]:

```
True
```

- Identity: `is`, `is not`
  - checks reference to the same object

In [44]:

```
x = [1,2,3]
y = x
```

```
x is y    # same variable?
```

Out[44]:

```
True
```

In [45]:

```
z = x[:]  # create a copy
```

In [46]:

```
z is x    # same variable?
```

Out[46]:

```
False
```

- Tuple packing and unpacking

In [47]:

```
point = (1,2,3)
(x,y,z) = point
print(x)
print(y)
print(z)
```

```
1
2
3
```

Sets

- a set is a collection of unique items

In [48]:

```
a=[1, 2, 2, 2, 4, 5, 5]
sA = set(a)
sA
```

Out[48]:

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

- set operations

In [49]:

```
sB = {4, 5, 6, 7}
print(sA - sB)    # set difference
```

```
{1, 2}
```

In [50]:

```
print (sA | sB)    # set union
```

```
{1, 2, 4, 5, 6, 7}
```

In [51]:

```
print (sA & sB)    # set intersect
```

```
{4, 5}
```

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. **Control structures (conditional and loops)**
4. Functions, Classes
5. File IO, Pickle, pandas
6. NumPy
7. matplotlib
8. probability review

Conditional Statements

- indentation used for code blocks after colon (:)
- if-elif-else statement

In [52]:

```
if x==2:
    print("foo")
elif x==3:
    print("bar")
else:
    print("baz")
```

```
baz
```

- nested if

In [53]:

```
if x>1:
    if x==2:
        print("foo")
    else:
        print("bar")
else:
    print("baz")
```

```
baz
```

- single-line

In [54]:

```
if x==1: print("blah")
```

```
blah
```

- check existence using "if in"

In [55]:

```
mydict = {'name': 'john', 42: 'sales'}
if 'name' in mydict:
    print("mydict has name field")
```

```
mydict has name field
```

In [56]:

```
if 'str' in 'this is a long string':
    print('str is inside')
```

```
str is inside
```

Loops

- "for-in" loop over values in a list

In [57]:

```
ns = range(1,6,2)     # list of numbers from 1 to 6, by 2
for n in ns:
    print(n)
```

```
1
3
5
```

- loop over index-value pairs

In [58]:

```
x = ['a', 'b', 'c']
for i,n in enumerate(x):
    print(i, n)
```

```
0 a
1 b
2 c
```

- looping over two lists at the same time

In [59]:

```
x = ['a', 'b', 'c']
y = ['A', 'B', 'C']
for i,j in zip(x,y):
    print(i,j)
```

```
a A
b B
c C
```

- `zip` creates pairs of items between the two lists
  - (actually creates an iterator over them)

In [60]:

```
list(zip(x,y))     # convert to a list (for random access)
```

Out[60]:

```
[('a', 'A'), ('b', 'B'), ('c', 'C')]
```

- looping over dictionary

In [61]:

```
x = {'a':1, 'b':2, 'c':3}
for (key,val) in x.items():
    print(key, val)
```

```
a 1
b 2
c 3
```

- while loop

In [62]:

```
x=0
while x<5:
    x += 1
print(x)
```

```
5
```

In [63]:

```
# single line
while x<10: x += 1
print(x)
```

```
10
```

- loop control (same as C)
  - `break`, `continue`
- else clause
  - runs after list is exhausted
  - does *not* run if loop break

In [64]:

```
for i in [0, 1, 6]:
    print(i)
else:
    print("end of list reached!")
```

```
0
1
6
end of list reached!
```

List Comprehension

- build a new list with a "for" loop

In [65]:

```
myList = [1, 2, 2, 2, 4, 5, 5]
myList4 = [4*item for item in myList]   # multiply each item by 4
myList4
```

Out[65]:

```
[4, 8, 8, 8, 16, 20, 20]
```

In [66]:

```
# equivalent code
myList4=[]
for item in myList:
    myList4.append(4*item)
myList4
```

Out[66]:

```
[4, 8, 8, 8, 16, 20, 20]
```

In [67]:

```
# can also use conditional to select items
[4*item*4 for item in myList if item>2]
```

Out[67]:

```
[64, 80, 80]
```

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. Control structures (conditional and loops)
4. **Functions, Classes**
5. File IO, Pickle, pandas
6. NumPy
7. matplotlib
8. probability review

Functions

- Defining a function
  - *required* and *optional* inputs (similar to C++)
  - "docstring" for optional documentation

In [68]:

```
def sum3(a, b=1, c=2):
    "sum a few values"
    mysum = a+b+c
    print("{}+{}+{}={}".format(a,b,c,mysum))
    return mysum
```

- Calling a function

In [69]:

```
sum3(2,3,4)   # call function: 2+3+4
```

```
2+3+4=9
```

Out[69]:

```
9
```

In [70]:

```
sum3(0)    # use default inputs: 0+1+2
```

```
0+1+2=3
```

Out[70]:

```
3
```

In [71]:

```
sum3(b=1, a=5, c=2)  # use keyword arguments: 5+1+2
```

```
5+1+2=8
```

Out[71]:

```
8
```

- unpacking a list as function arguments

In [72]:

```
args = [1, 5, 2]
sum3(*args)
```

```
1+5+2=8
```

Out[72]:

```
8
```

- unpacking a dictionary as function keyword arguments

In [73]:

```
argsd = {'b':1, 'a':5, 'c':2}
sum3(**argsd)
```

```
5+1+2=8
```

Out[73]:

```
8
```

In [74]:

```
help(sum3)    # show documentation
```

```
Help on function sum3 in module __main__:

sum3(a, b=1, c=2)
    sum a few values
```

In [75]:

```
# ipython magic -- shows a help window about the function
? sum3
```

Classes

- Defining a class
  - `self` is a reference to the object instance (passed *implicitly*)

In [76]:

```
class MyList:
    "class documentation string"
    num = 0                       # a class variable
    def  __init__(self, b):       # constructor
        self.x = [b]              # an instance variable
        MyList.num += 1           # modify class variable
    def appendx(self, b):         # a class method
        self.x.append(b)          # modify an instance variable
        self.app = 1              # create new instance variable
```

- Using the class

In [77]:

```
c = MyList(0)          # create an instance of MyList
print(c.x)
```

```
[0]
```

In [78]:

```
c.appendx(1)           # c.x = [0, 1]
print(c.x)
```

```
[0, 1]
```

In [79]:

```
c.appendx(2)           # c.x = [0, 1, 2]
print(c.x)
```

```
[0, 1, 2]
```

In [80]:

```
print(MyList.num)      # access class variable (same as c.num)
```

```
1
```

More on Classes

- There are *no* "private" members
    - everything is accessible
    - convention to indicate *private*:
        - `_variable` means private method or variable (but still accessible)
    - convention for *very private*:
        - `__variable` is not directly visible
        - actually it is renamed to `_classname__variable`

- Instance variable rules
    - On *use* via instance (`self.x`), scope search order is:
        - (1) instance, (2) class, (3) base classes
        - also the same for method lookup
    - On *assignment* via instance (`self.x=...`):
        - always makes an instance variable
    - Class variables "default" for instance variables
        - *class* variable: one copy *shared* by all
        - *instance* variable: each instance has its own

Inheritence

- Child class inherits attributes from parents

In [81]:

```
class MyListAll(MyList):
    def __init__(self, a):   # overrides MyList
        self.allx = [a]
        MyList.__init__(self, a)   # call base class constructor
    def popx(self):
        return self.x.pop()
    def appendx(self, a):          # overrides MyList
        self.allx.append(a)
        MyList.appendx(self, a)    # "super" method call
```

- Multiple inheritence
    - `class ChildClass(Parent1, Parent2, ...)`
    - calling method in parent
        - `super(ChildClass, self).method(args)`

Class methods & Built-in Attributes

- Useful methods to override in class

In [82]:

```
class MyList2:
    ...
    def __str__(self):      # string representation
        ...
    def __cmp__(self, x):   # object comparison
        ...
    def __del__(self):      # destructor
        ...
```

- Built-in attributes

In [83]:

```
print(c.__dict__)     # Dictionary with the namespace.
print(c.__doc__)      # Class documentation string
print(c.__module__)  # Module which defines the class
```

```
{'x': [0, 1, 2], 'app': 1}
class documentation string
__main__
```

In [84]:

```
print(MyList.__name__)    # Class name
print(MyList.__bases__)   # tuple of base classes
```

```
MyList
(<class 'object'>,)
```

Outline

1. Python Intro
2. Python Basics (identifiers, types, operators)
3. Control structures (conditional and loops)
4. Functions, Classes
5. **File IO, Pickle, pandas**
6. NumPy
7. matplotlib
8. probability review

File I/O

- Write a file

In [85]:

```
with open("myfile.txt", "w") as f:
    f.write("blah\n")
    f.writelines(['line1\n', 'line2\n', 'line3\n'])

# NOTE: using "with" will automatically close the file
```

- Read a whole file

In [86]:

```
with open("myfile.txt", "r") as f:
    contents = f.read()   # read the whole file as a string
    print(contents)
```

```
blah
line1
line2
line3
```

- Read line or remaining lines

In [87]:

```
f = open("myfile.txt", 'r')
print(f.readline())    # read a single line.
```

```
blah
```

In [88]:

```
print(f.readlines())   # read remaining lines in a list.
f.close()
```

```
['line1\n', 'line2\n', 'line3\n']
```

- Read line by line with a loop

In [89]:

```
with open("myfile.txt", 'r') as f:
    for line in f:
        print(line)     # still contains newline char
```

```
blah

line1

line2

line3
```

Saving Objects with Pickle

- Turns almost **any** Python **object** into a string representation for saving into a file.

In [90]:

```
import pickle                          # load the pickle library
mylist = MyList(0)                     # an object
# open file to save object (write bytes)
with open('alist.pickle', 'wb') as file:
    pickle.dump(mylist, file)          # save the object using pickle
```

- Load object from file

In [91]:

```
with open('alist.pickle', 'rb') as file:  # (read bytes)
    mylist2 = pickle.load(file)        # load pickled object from file
print(mylist2)
print(mylist2.x)
```

```
<__main__.MyList object at 0x104330310>
[0]
```

- cPickle is a faster version (1,000 times faster!)

Exception Handling

- Catching an exception
  - `except` block catches exceptions
  - `else` block executes if no exception occurs
  - `finally` block always executes at end

In [92]:

```
try:
    file = open('blah.pickle', 'r')
    blah = pickle.load(file)
    file.close()
except:                  # catch everything
    print("No file!")
else:                    # executes if no exception occurred
    print("No exception!")
finally:
    print("Bye!")         # always executes
```

```
No file!
Bye!
```

pandas

- pandas is a Python library for data wrangling and analysis.
- `Dataframe` is a table of entries (like an Excel spreadsheet).
  - each column does not need to be the same type
  - operations to modify and operate on the table

In [93]:

```
# setup pandas and display
import pandas as pd
```

In [94]:

```
# read CSV file
df = pd.read_csv('mycsv.csv')

# print the dataframe
df
```

Out[94]:

	Name	Location	Age
0	John	New York	24
1	Anna	Paris	13
2	Peter	Berlin	53
3	Linda	London	33

- select a column

In [95]:

```
df['Name']
```

Out[95]:

```
0     John
1     Anna
2    Peter
3    Linda
Name: Name, dtype: object
```

- query the table

In [96]:

```
# select Age greater than 30
df[df.Age > 30]
```

Out[96]:

	Name	Location	Age
2	Peter	Berlin	53
3	Linda	London	33

- compute statistics

In [97]:

```
df.mean()
```

```
/var/folders/d8/20tc63h54bgcpjl90_dt4bh80000gp/T/ipykernel_12428/3698961737.py:1:
FutureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numeric
_only=None') is deprecated; in a future version this will raise TypeError.  Select
only valid columns before calling the reduction.
  df.mean()
```

Out[97]:

```
Age    30.75
dtype: float64
```