Northeastern

EECE5645
Parallel Processing for Data Analytics

Lecture 1: Introduction to Python

Based on:
http://tdc-www.harvard.edu/Python.pdf

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Functions
- ❑ Control Flow
- ❑ Modules
- ❑ Classes and Objects

Python

- Open source **general-purpose** language.
- Object Oriented, Procedural, Functional
- Powerful library of Modules
- Great interactive environment
- Interfaces with Spark

- Downloads: http://www.python.org
- Documentation:https://docs.python.org/3.3/tutorial/
- Discovery Cluster:
  - `module load python/3.3.6`

Batteries Included

https://docs.python.org/3/library/

- Advanced data structures (lists, dictionaries) part of language
- Numerical, math, statistics
  - `numpy, scipy`: matlab-like functionality
  - `pandas`: R-like data frames
  - `matplotlib`: beautiful plots
- File and directory access
- Data compression and archiving
- Cryptography
- Multithreading, OS
- Networking/Internet/WWW protocols
- Multimedia
- Optimization
- …

Running Python: The Python Interpreter

```
$python
Python 3.6.6 (default, Dec 3 2018, 16:42:15)
[GCC 4.8.5 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> 3*(7+2)
27
>>> CTRL-D                              # to exit
```

Executing Python programs

- Write a program in your favorite editor
- Save it with a .py file extention
- Type

```
python yourfilename.py
```

to execute it

A Simple Example

```
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)
```

Output:

12

Hello World

Enough to Understand the Code

- Assignment uses = and comparison uses ==.
- For numbers + - * / % are as expected.
  - Special use of + for string concatenation.
  - Special use of % for string formatting (as with printf in C)
- 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 **do not** need to be declared.
  - Python figures out the variable types on its own.

Basic Datatypes

NOT TRUE FOR python 2.x

Use `z = 5. / 2` **to be safe**

- **Integers (default for numbers)**

```
z = 5 / 2      # Answer is 2.5, floating point division.
z = 5 // 2     # Answer is 2, integer division.
```

- **Floats**

```
x = 3.456
```

- **Strings**
  - Can use " " or ' ' to specify.

```
"abc" 'abc'                    #Same thing
```

  - 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"""
```

Whitespace

Whitespace is meaningful in Python!!!!!: especially indentation and placement of newlines

- Use a newline to end a line of code.
  - Use \ when must go to next line prematurely.
- **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
- Often a colon : appears at the start of a new block. (E.g. for if statements, function and class definitions.)

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.
- The development environment, debugger, and other tools like help() use it: it's good style to include one.

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

Assignment Statements

```
>>> a = 1              #int
>>> b = 1.5            #float
>>> c = 'banana'       #str
>>> d = "apple"        #"..." same as '...'
>>> print(a,b,c,d)
1 1.5 banana apple
>>> e = (a >= b)       #Boolean expression
>>> print(e)
False
```

Accessing Non-Existent Names

If you try to access a name before it's been properly created (by placing it on the left side of an assignment), you'll get an error.

```
>>> y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Multiple Simultaneous Assignments

```
>>> x, y = 2, 3
>>> x
2
>>> y
3
```

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 words:

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**

Basic Operators

- Binary ops on numbers

```
>>> a = 1
>>> b = 1.5
>>> print(a+b, a-b, a/b, a*b, 2**b)
2.5 -0.5 0.666666666667 1.5 2.828427124746 1903
>>> print(5/2, 1.*5/2,  5%2 , 5.1//2)
2.5 2.5 1 2.0
```

- Some **overloaded** to work on strings

```
>>> 'banana'+'_'+'apple'    #concatenation
'banana_apple'
>>> 4*('apricot'+' ')     #replication
'apricot apricot apricot apricot '
```

Casting

```
>>> a = '1.5'
>>> b = float(a)+2                #cast as number
>>> print('Bank Balance: $' + str(b))   #cast as str
Bank Balance: $3.5

>>> print(eval('1+2*(1/3.0)'))
                #evaluate string as python expression
1.66666666667
```

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Functions
- ❑ Control Flow
- ❑ Modules
- ❑ Classes and Objects

Sequence Types (a.k.a. Collections, Containters)

1. **Tuple**
   - A simple *immutable* ordered sequence of items
   - items can be of mixed types, including collections
2. **String**
   - *Immutable*
   - Conceptually very much like a tuple
3. **List**
   - *Mutable* ordered sequence of items of mixed types

Similar Syntax

- **All three sequence types (tuples, strings, and lists) share much of the same syntax and functionality.**

- Key difference:
  - **Tuples** and **strings** are *immutable*
  - **Lists** are *mutable*

- Most operations shown in this section can be applied to all sequence types

Definitions

- **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."""
```

Accessing Elements

- We can access individual members of a tuple, list, or string using square bracket “array” notation.
- Note that all 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'
```

Positive and Negative Indices

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

Positive index: count from first element, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from last element, starting with –1.

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

Slicing: Return a Copy of a Subset

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

Get subsequence.

```
>>> t[1:3]
('abc',4.56)
```

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

```
>>> t[:3]
(23, 'abc',4.56)
```

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')
```

Copy the Entire Sequence

To make a copy of the entire sequence, use [:].

```
>>> t[:]
(23, 'abc', 4.56, (2,3), 'def')
```

Note the difference between these two assignments

```
>>> list2 = list1    # 2 names refer to same list
  # Changing one affects both
```

```
>>> list2 = list1[:] # Creates new independent copy
```

Membership test

Boolean test whether a value is inside a container:

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

For strings, also tests for substrings

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

Note: `in` also used in syntax of `for` *loops* and *list comprehensions*

Concatenation

The + operator 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"
```

Replication

The * operator 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"
```

Mutability: Tuples are Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
Traceback (most recent call last):
   File "", line 1, in -toplevel-
     tu[2] = 3.14
TypeError: object doesn't support item assignment
```

You can't change a tuple.

You *can* make a fresh tuple and assign its reference to a previously used variable name.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Mutability: Lists are Mutable

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

- **You can change a list *in place*. (i.e., no additional memory used)**
- **Variable li still points to the same memory location after assignment**
- **No free lunch: supporting mutability makes lists *slower* than tuples.**

Operations Only on Lists

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

>>> li.append('a')     # Our first exposure to an object method
>>> li
[1, 11, 3, 4, 5, 'a']

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

>>> li.sort()
[1, 3, 4, 5, 11, 'a', 'i']

>>> help(li)       # see all methods that can be applied to a list
```

Lists made out of other lists: List Comprehension

```
li = range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


squares = [x**2 for x in li]
print squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]


even_squares = [x for x in squares if x%2==0]
[0, 4, 16, 36, 64]


[ (x,1) for x in even_squares]
[(0, 1), (4, 1), (16, 1), (36, 1), (64, 1)]


[ range(x)  for x in range(4)]
[[], [0], [0, 1], [0, 1, 2]]


[  y   for x in range(4) for y in range(x)]          # "nested" for loop
[0, 0, 1, 0, 1, 2]
```

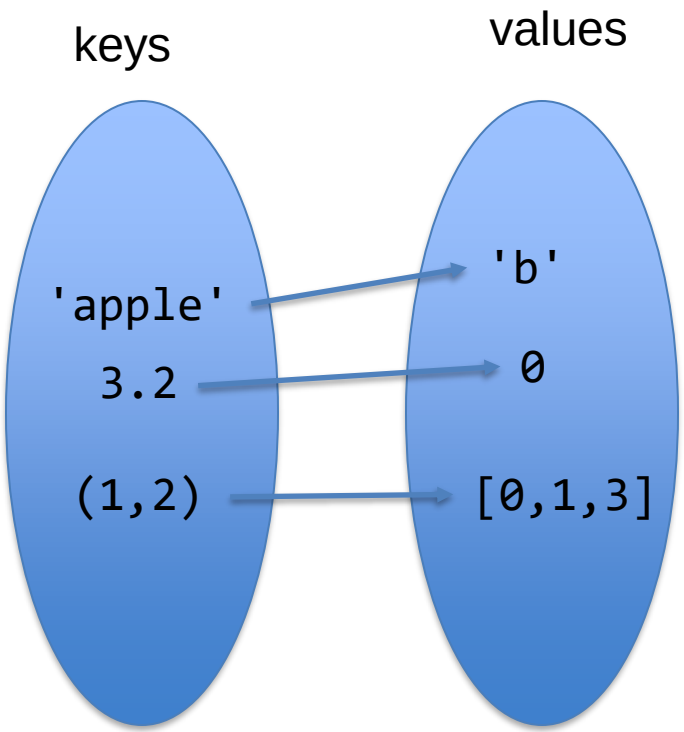
Tuples vs. Lists

- Lists are slower but more powerful than tuples
- Is your data going to be *accessed* but *not changed*?
  - Use tuple
- Do you need to support modifications?
  - Use list
- Convert between tuples and lists through casting:

```
>>> li = list(tu)
>>> tu = tuple(li)
```

Dictionaries: a Mapping type

- **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
- **You can define, modify, view, lookup, and delete key-value pairs in the dictionary**



Dictionary Examples

```
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user']
'bozo'
>>> d['pswd']
1234
>>> d['banana']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'banana'
```

```
# Change value for existing key
>>> d = {'user':'bozo', 'pswd':1234}
>>> d['user'] = 'clown'
>>> d
{'pswd': 1234, 'user': 'clown'}
```

```
# Add new key-value pair
>>> d['id'] = 25
>>> d
{'id': 25, 'pswd': 1234, 'user': 'clown'}
```

```
# remove key-value map
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> del d['user']
>>> d
{'i': 34, 'p': 1234}
>>> d.clear()
>>> d
{}
```

```
#change value for existing key
>>> d = {'user':'bozo', 'p':1234, 'i':34}
>>> d.keys()                    # List of keys
['i', 'p', 'user']
>>> d.values()          # List of values
[34, 1234, 'bozo']
>>> d.items()                   # List of pairs as tuples.
[('i', 34), ('p', 1234), ('user', 'bozo')]
```

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Control Flow
- ❑ Functions
- ❑ Modules
- ❑ Classes and Objects

`if` statement

```
if x == 3:
        print "X equals 3."            #indentation defines blocks!
elif x == 2:
        print "X equals 2."
else:
        print "X equals something else."
print "This is outside the 'if'."
```

`while` statement

Fibonacci Series:

```
>>> a, b = 0, 1          #multiple var assignment
>>> while a < 10:
        print(a)          #indentation defines block!
        a, b = b, a+b
0
1
1
2
3
5
8
```

for loop

Primality Test:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n//x)
            break
    else:                                   # executed only if loop does not break early
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Control Flow
- ❑ Functions
- ❑ Modules
- ❑ Classes and Objects

Function Definitions

- 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
```

Optional Arguments

- **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
```

Gotchas

- **All functions return a value:**
  - If no return statement, function returns None

- **Functions can be used like *any other data type*! They can be**
  - Arguments to other functions
  - Return values of other functions
  - Assigned to variables
  - Parts of typles lists, etc.

```
def square(x):
        return x * x
```

```
>>> z = square
>>> z(2)
4
>>> map(square,[1,2,3,4])          # map's 1st input is a function!
[1, 4, 9, 16]
```

Anonymous Functions: the lambda operator

```
def square(x):
        return x * x

>>> map(square, [1,2,3,4])
[1, 4, 9, 16]

>>> map( lambda x: x*x , [1,2,3,4])
[1, 4, 9, 16]


def power_generator(n):
        return lambda x : x ** n     #power_generator returns a function!

>>> square = power_generator(2)
>>> cube = power_generator(3)

>>> map(square, [1,2,3,4])
[1, 4, 9, 16]
>>> map(cube, [1,2,3,4])
[1, 8, 27, 64]
```

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Control Flow
- ❑ Functions
- ❑ Modules
- ❑ Classes and Objects

Python Modules

- Modules comprise functions and variables defined in separate files
- Functions or variables from a module are imported using from or import

```
from numpy import sqrt          #imports only sqrt
sqrt(2343523)

import numpy                                #imports entire module
numpy.sqrt(2343523)

import numpy as np                          #imports entire module as "np"
np.sqrt(2343523)

from numpy import *                         #imports everything in module
sqrt(1243)
exp(-1.23)
```

- Type help(module) to see module functionality

numpy

- Functions galore:

exp,log,log10,sin,cos,tan,sqrt,...

- Numerical Arrays and Matrices!

```
import numpy as np
a = np.array([1.0, 2.0, 2.1])
b = np.array([2.0, 1.0, -3.1])

print a + b                                   #vector addition
[ 3.  3. -1.]

print np.dot(a,b)                  #vector dot product
-2.51

print np.outer(a,b)                           #vector outer product
[[ 2.   1.  -3.1 ]
 [ 4.   2.  -6.2 ]
 [ 4.2  2.1 -6.51]]
```

numpy.linalg

help(numpy.linalg)

NAME
numpy.linalg

FILE
/Library/Python/2.7/site-packages/numpy-1.11.0-py2.7-macosx-10.10-intel.egg/numpy/linalg/__init__.py

DESCRIPTION
Core Linear Algebra Tools

Linear algebra basics:

- norm Vector or matrix norm
- inv Inverse of a square matrix
- solve Solve a linear system of equations
- det Determinant of a square matrix
- lstsq Solve linear least-squares problem
- pinv Pseudo-inverse (Moore-Penrose) calculated using a singular value decomposition
- matrix_power Integer power of a square matrix

Eigenvalues and decompositions:

- eig Eigenvalues and vectors of a square matrix
- eigh Eigenvalues and eigenvectors of a Hermitian matrix
- eigvals Eigenvalues of a square matrix
- eigvalsh Eigenvalues of a Hermitian matrix
- qr QR decomposition of a matrix
- svd Singular value decomposition of a matrix
- cholesky Cholesky decomposition of a matrix

Tensor operations:

- tensorsolve Solve a linear tensor equation
- tensorinv Calculate an inverse of a tensor

A few useful modules

- See documentation for following modules:

```
scipy                       # Numerical integration, linear algebra,
                            # signal processing, Fourier transforms
numpy.random          # Sample random numbers
argparse                    # parse command line arguments
sklearn                     # Machine learning algorithms
sklearn.linear_model  # Linear & logistic regression
sys                         # I/O
time                        # time utilities
matplotlib.pyplot     # plotting tools
```

Overview

- ❑ Python Basics
- ❑ Advanced Data Types
- ❑ Control Flow
- ❑ Functions
- ❑ Modules
- ❑ Classes & Objects

What is an Object?

- A data structure that contains *variables* and *methods*

- Object Oriented Design abides by the principles of:
  - **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[self.atno]

        def __str__(self): # overloads str() function
                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 Class

- Overloaded the default constructor
- Defined class variables (atno,position) and a class method (symbol)
  - accessed as self.atno within the class definition
  - accessed as at.atno outside class for atom at
- Good way to manage shared memory:
  - instead of passing long lists of arguments, encapsulate some of this data into an object, and pass the object.
  - much cleaner, easier to interpret programs as a result
- **Overloaded** the str() operator
- We now want to use the atom class to build molecules...

Molecules

```
class molecule:
    def __init__(self,name='Generic'):
        self.name = name
        self.atomlist = []
    def addatom(self,atom):
        self.atomlist.append(atom)
    def __str__(self):
        s = 'This is a molecule named %s\n' % self.name
        s = s +'It has %d atoms\n' % len(self.atomlist)
        for atom in self.atomlist:
            s  = s  + str(atom) + '\n'
        return s
```

Using the Molecule Class

```
>>>  mol = molecule('Water')
>>> at = atom(8, 0 ,0 ,0)
>>> mol.addatom(at)
>>> mol.addatom(atom(1,0.,0.,1.))
>>> mol.addatom(atom(1,0.,1.,0.))
>>> print(mol)
This is a molecule named Water
It has 3 atoms
8    0.0000    0.0000    0.0000
1    0.0000    0.0000    1.0000
1    0.0000    1.0000    0.0000
```

- Note that the str function calls the atoms str function
  - Code reuse: only have to type the code that prints an atom once; this means that if you change the atom specification, you only have one place to update.

Inheritance

```
class dna_molecule(molecule):
    def clone(self):
        new_molecule  = dna_molecule(molecule.name)
        for at in self.atomlist:
        new_molecule.addatom(atom(at.atno,
                                  at.position[0],
                                  at.position[1],
                                  at.position[2]))
         return new_molecule
```

- __init__, __str__, and addatom are *inherited* from the parent class (molecule)!
- **dna_molecule** is augmented with an additional method (clone)
- Another example of code reuse

Overriding Parent Methods

```
class dna_molecule(molecule):
    def __init__(self,name='DNA molecule',isOrganic=True):
        self.isOrganic=isOrganic
        super(dna_molecule, self).__init__(name)
    def clone(self):
        new_molecule  = dna_molecule(molecule.name)
        for at in self.atomlist:
        new_molecule.addatom(atom(at.atno,
                                  at.position[0],
                                  at.position[1],
                                  at.position[2]))
         return new_molecule
```

- A parent method (__init__) gets a new definition, using also the parent definion

Private and public variables and methods

- In Python anything with two leading underscores is private

__a, __my_variable

- Anything with one leading underscore is semiprivate, and you should feel guilty accessing this data directly.

_b

- Sometimes useful as an intermediate step to making variable private

Overloading Operators

__**contains**__(...)
x.__contains__(y) <==> y in x

__**eq**__(...)
x.__eq__(y) <==> x==y

__**ge**__(...)
x.__ge__(y) <==> x>=y

__**add**__(...)
x.__add__(y) <==> x+y

__**lt**__(...)
x.__lt__(y) <==> x<y

__**mul**__(...)
x.__mul__(n) <==> x*n

Use Docstrings!

```
class atom(object):
        """This will be shown whenever someone calls help(atom), along
        with list of public methods """

        def symbol(self):
                """This will be shown whenever someone
                calls help(atom.symbol) or help(x.symbol)
                where x is an atom object"""
                return Atno_to_Symbol[atno]
```

Running Python on Discovery Cluster

- Reserve an interactive node and connect to it:

```
srun --pty --export=ALL --reservation=eece5645 --tasks-per-node 1 --nodes 1 --mem=10Gb --time=00:30:00 /bin/bash
```

- Load python module

```
module load python/3.7.1
```

- **See more in "Programming Resources"**
- **Load module in your `.bashrc` file**