A Brief Introduction to Python for those who know Java

(Last extensive revision:
Daniel Moroz, fall 2015)
Plan 1

- Baby steps
  - History, Python environments, Docs

- Absolute Fundamentals
  - Objects, Types
  - Math and Strings basics
  - References and Mutability

- Data Types
  - Strings, Tuples, Lists, Dicts

- Looping
  - Comprehensions

- Iterators
  - Generators

- To Be Continued…
Python

- Developed by Guido van Rossum in the early 90s
  - Originally Dutch, in USA since 1995, now works for Dropbox
  - Benevolent Dictator for Life (BDFL)
- Available on Eniac and to download at python.org
- Named after the Monty Python comedy group
  - Homework :)

CIS 391 – Fall 2015
Some Positive Features of Python

- **Fast development:**
  - Concise, intuitive syntax
    - Whitespace delimited
  - Garbage collected

- **Portable:**
  - Programs run on major platforms without change
  - cpython: common Python implementation in C.

- **Various built-in types:**
  - lists, dictionaries, sets: useful for AI
  - (cf. Matlab, best for linear algebra)

- **Large collection of support libraries:**
  - eg. NumPy for Matlab like programming
Recommended Reading

- **Python Overview**
  - The Official Python Tutorial (http://docs.python.org/tutorial/)
  - Slides for CIS192, available online (used in these slides)

- **PEPs – Python Enhancement Proposals**
  - PEP 8 - Official Style Guide for Python Code (Guido et al)
    - Style is about consistency. 4 space indents, < 80 char lines
    - Naming convention for functions and variables: lower_w_under
    - Use the automatic pep8 checker!
  - PEP 20 – The Zen of Python (Tim Peters) (try: import this)
    - Beautiful is better than ugly
    - Simple is better than complex
    - There should be one obvious way to do it
    - That way may not be obvious at first unless you're Dutch
    - Readability counts
Which Python?

- **Python 2.7**
  - Current version on Eniac, so we’ll use it
  - Last stable release before version 3
- **NOT Python 3**
  - Many elegant but incompatible changes
  - More existing third party software is compatible with Python 2 than Python 3 right now
**Python Environments**

- **REPL**
  - Read Evaluate Print Loop
  - Type “python” at the terminal
  - Convenient for testing
  - GUI – IDLE

```bash
cis391@plus:~> python
Python 2.7.8 (default, Sep 30 2014, 15:34:38) [GCC] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World!')
Hello World!
'Hello World!'
>>> [2*i for i in range(10)]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> exit()
cis391@plus:~>
```
Python Environments Cont’d

- **Scripts**
  - Not REPL, need to explicitly print
  - Type “Python script_name.py” at the terminal to run
  - Homework submitted as scripts

```python
import random

def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0)""
    return ["%.2f" % random.random() for i in range(10)]

print '1/2 = ', 1/2
if __name__ == '__main__':
    rand_fn()
    print rand_fn()

CIS521@plus:~$ cat foo.py
import random

def rand_fn():
    """outputs list of 10 random floats between [0.0, 1.0)""
    return ["%.2f" % random.random() for i in range(10)]

print '1/2 = ', 1/2
if __name__ == '__main__':
    rand_fn()
    print rand_fn()

cis391@plus:~$ python foo.py
1/2 = 0
['0.14', '0.33', '0.15', '0.93', '0.44', '0.28', '0.44', '0.62', '0.33', '0.64']
```
Structure of Python File

- **Whitespace is meaningful in Python**
- **Use a newline to end a line of code.**
  - Use \ when must go to next line prematurely.
- **Block structure is indicated by indentation**
  - 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 function and class definitions.)
Objects and Types

- **All data treated as objects**
  - An object is deleted (by garbage collection) once unreachable.

- **Strong Typing**
  - Every object has a fixed type, interpreter doesn’t allow things incompatible with that type (e.g. “foo” + 2)
  - `type(object)`
  - `isinstance(object, type)`

- **Examples of Types:**
  - `int, float`
  - `str, tuple, dict, list`
  - `bool: True, False`
  - `None, generator, function`
Static vs Dynamic Typing

- **Java: statically typed**
  - Variables can only refer to objects of a declared type
  - Methods use type signatures to enforce contracts

- **Python: dynamic typed**
  - Variables come into existence when first assigned.
    - ```
      >>> x = "foo"
      >>> x = 2
    ```
  - `type(var)` automatically determined by what object assigned
  - If assigned again, can always refer to object of any type
  - Functions have no type signatures
  - Drawback: type errors are only caught at runtime
Math Basics

- **Literals**
  - Integers: 1, 2
  - Floats: 1.0, 2e10
  - Boolean: True, False

- **Operations**
  - Arithmetic: + - * /
  - Power: **
  - Modulus: %
  - Comparison: , <=, >, =, !=
  - Logic: (and, or, not) not symbols

- **Assignment Operators**
  - += *= /= &= ...
  - No ++ or --
Strings

- **Creation**
  - Can use either single or double quotes
  - Triple quote for multiline string and docstring

- **Concatenating strings**
  - By separating string literals with whitespace
  - Special use of ‘+’

- **Prefixing with r means raw.**
  - No need to escape special characters: r"\n"

- **String formatting**
  - Special use of ‘%’ (as in printf in C)

- **Immutable**
A Simple Code Sample (in IDLE)

```python
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
```

Intro to AI
References and Mutability

```python
>>> x = 'foo '  
>>> y = x  
>>> x = x.strip() #new obj  
>>> x
'foo'  
>>> y
'foo '  
```

- **strings are immutable**
- `==` checks whether variables point to objects of the same value
- `is` checks whether variables point to the same object

```python
>>> x = [1,2,3]  
>>> y = x  
>>> x.append(5) #same obj  
>>> y
[1, 2, 3, 5]  
>>> x
[1, 2, 3, 5]  
```

- **lists are mutable**
- use `y = x[:]` to get a (shallow) copy of any sequence, ie a new object of the same value
Sequence types:
Tuples, Lists, and Strings
Sequence Types

- **Tuple**
  - A simple *immutable* ordered sequence of items
    - *Immutable*: a tuple cannot be modified once created
  - Items can be of mixed types, including collection types

- **Strings**
  - *Immutable*
  - Very much like a tuple with different syntax
  - Regular strings use 8-bit characters. Unicode strings use 2-byte characters. (All this is changed in Python 3.)

- **List**
  - *Mutable* ordered sequence of items of mixed types
Sequence Types

- The three sequence types share much of the same syntax and functionality.

```python
>>> tu = (23, 'abc', 4.56, (2, 3), 'def') # tuple
>>> li = ['abc', 34, 4.34, 23] # list
>>> st = "Hello World"; st = 'Hello World' # strings

>>> tu[1]  # Accessing second item in the tuple.
'abc'

>>> t[-3]  #negative lookup from right, from -1
4.56
```
Slicing: Return Copy of a Subset

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

>>> t[1:4] # slicing ends before last index
('abc', 4.56, (2,3))

>>> t[1:-1] # using negative index
('abc', 4.56, (2,3))

>>> t[1:-1:2] # selection of every nth item.
('abc', (2,3))

>>> t[:2] # copy from beginning of sequence
(23, 'abc')

>>> t[2:] # copy to the very end of the sequence
(4.56, (2,3), 'def')
```
Operations on Lists

```python
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a')  # Note the method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
>>> 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']
```
Operations on Lists II

```python
>>> li = [5, 2, 6, 8]
```

```python
>>> li.reverse()    # reverse the list *in place* (modify)
>>> li
[8, 6, 2, 5]
```

```python
>>> li.sort()       # sort the list *in place*
>>> li
[2, 5, 6, 8]
```

```python
>>> li.sort(some_function)
# sort in place using user-defined comparison

>>> sorted(li)       #return a *copy* sorted
```
Operations on Strings

```python
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
>>> words
['Pretend', 'this', 'sentence', 'makes', 'sense. ']
>>> ".join(words) #join method of obj "_"
'Pretend_this_sentence_makes_sense.'

>>> s = 'dog'
>>> s.capitalize()
'Dog'
>>> s.upper()
'DOG'
>>> ' hi --'.strip(' --')
'hi --'
https://docs.python.org/2/library/string.html
```
Tuples

```python
>>> a = ["apple", "orange", "banana"]

>>> for (index, fruit) in enumerate(a):
...     print str(index) + ": " + fruit

0: apple
1: orange
2: banana

>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c', 'd']

>>> zip(a, b)
[(1, 'a'), (2, 'b'), (3, 'c')]

>>> zip("foo", "bar")
[('f', 'b'), ('o', 'a'), ('o', 'r')]

>>> x, y, z = 'a', 'b', 'c'
```
Dictionaries: a *mapping* collection type
Dict: Create, Access, Update

- Dictionaries are unordered & work by hashing, so keys must be immutable
- Constant average time add, lookup, update

```python
>>> d = {'user': 'bozo', 'pswd': 1234}

>>> d['user']
'bozo'

>>> d['bozo']
Traceback (most recent call last):
  File '<ipython-input-1-8f3e7b5c7793>' in <ipython-input-1-8f3e7b5c7793>, line 1
    d['bozo']
KeyError: bozo

>>> d['user'] = 'clown'  # Assigning to an existing key replaces its value.

>>> d
{'user': 'clown', 'pswd': 1234}
```
Dict: Useful Methods

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

```python
>>> from collections import defaultdict
>>> d = defaultdict(int)
>>> d['a']
0
```

- `defaultdict` automatically initializes nonexistent dictionary values
For Loops
For Loops

```python
for <item> in <collection>:
    <statements>
```

- `<item>` can be more complex than a single variable name.
  ```python
  for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:
      print x
  ```

- **Range:**
  - `range(5)` returns `[0,1,2,3,4]`
  - So we can say:
    ```python
    for x in range(5):
        print x
    ```
  - `xrange()` returns an *iterator* that provides the same functionality more efficiently (more later)
List Comprehensions replace loops!

```python
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]

>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```
List Comprehension extra for

[x for x in lst1 if x > 2\n for y in lst2\n for z in lst3 if x + y + z < 8]

res = [] # translation
for x in lst1:
    if x > 2:
        for y in lst2:
            for z in lst3:
                if x + y + z > 8:
                    res.append(x)
Dictionary, Set Comprehensions

```python
{k: v for k,v in lst}
d = dict() # translation
for k, v in lst:
    d[k] = v

{x for x in lst}
s = set() # translation
for x in lst:
    s.add(x)
```
Iterators
Iterator Objects

- Iterable objects can be used in a for loop because they have an `__iter__` magic method, which converts them to iterator objects:

```python
>>> k = [1,2,3]
>>> k.__iter__()
<listiterator object at 0x104f8ca50>
>>> iter(k)
<listiterator object at 0x104f8ca10>
```
Iterators

- Iterators are objects with a `next()` method:

```python
>>> i = iter(k)
>>> i.next()
1
>>> i.next()
2
>>> i.next()
3
>>> i.next()
StopIteration
```

- Python iterators do not have a `hasnext()` method!
- Just catch the `StopIteration` exception
Iterators: For.. In..

- for <item> in <iterable>:
  <statements>

- First line is just syntactic sugar for:
  - 1. Initialize: Call <iterable>.__iter__() to create an iterator
    Each iteration:
    - 2. Call iterator.next() and bind <item>
    - 2a. Catch StopIteration exceptions

- To be iterable: has __iter__ method
  - which returns an iterator obj

- To be iterator: has next method
  - which throws StopIteration when done
Iterator Class

class Reverse:

    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
    def __iter__(self):
        return self

>>> for char in Reverse('spam'):
    print char
m
a
p
s
Iterators use memory efficiently

Eg: File Objects

```python
>>> for line in open(“script.py”): #returns iterator
...     print(line.upper())
```

```python
import sys
print(sys.path)
X = 2
print(2 ** 3)
```

instead of

```python
>>> for line in open(“script.py”).readlines(): #returns list
...     print(line.upper())
```
Generators
Generators

- Generators are iterators (have next() method)
- Creating Generators: `yield`
  - Functions that contain the `yield` keyword automatically return a generator when called
  - Generator Comprehensions

```python
>>> def f(n):
    ...   yield n
    ...   yield n+1
    ...

>>> type(f)
<type 'function'>
>>> type(f(5))
<type 'generator'>
>>> [i for i in f(6)]
[6, 7]
```
Generators: What does `yield` do?

- Each time we call the `next` method of the generator, the method runs until it encounters a `yield` statement, and then it stops and returns the value that was yielded. Next time, it resumes where it left off.

```python
>>> gen = f(5)  # no need to say f(5).__iter__()
>>> gen<generator object f at 0x1008cc9b0>
>>> gen.next()
5
>>> gen.next()
6
>>> gen.next()
StopIteration
```
Generators

- \texttt{xrange(n)} vs \texttt{range(n)}
  - \texttt{xrange} acts like a generator
  - \texttt{range(n)} keeps all n values in memory before starting a loop even if \( n \) is huge: \texttt{for k in range(n)}
  - \texttt{sum(xrange(n))} much faster than \texttt{sum(range(n))} for large \( n \)

- Benefits
  - Less code than writing a standard iterator
  - Maintains local state automatically
  - Values are computed one at a time, as they’re needed
  - Avoids storing the entire sequence in memory
  - Good for aggregating (summing, counting) items. One pass.
  - Crucial for infinite sequences
  - Bad if you need to inspect the individual values
Using generators: merging sequences

- Problem: merge two sorted lists, using the output as a stream (i.e. not storing it).

```python
def merge(l, r):
    llen, rlen, i, j = len(l), len(r), 0, 0
    while i < llen or j < rlen:
        if j == rlen or (i < llen and l[i] < r[j]):
            yield l[i]
            i += 1
        else:
            yield r[j]
            j += 1
```
Using generators

```python
>>> g = merge([2,4], [1, 3, 5]) #g is an iterator
>>> while True:
    print g.next()
1
2
3
4
5

Traceback (most recent call last):
  File "<pyshell#73>", line 2, in <module>
    print g.next()
StopIteration

>>> [x for x in merge([1,3,5],[2,4])]
[1, 2, 3, 4, 5]
```
Generators and exceptions

```python
>>> g = merge([2,4], [1, 3, 5])
>>> while True:
    try:
        print g.next()
    except StopIteration:
        print 'Done'
        break

1
2
3
4
5
Done
```
Plan 2

- Import
- Functions
  - Args, kwargs
- Classes
  - “magic” methods (objects behave like built-in types)
- Profiling
  - timeit
  - cProfile