A Brief Introduction to Python
for those who know Java

(Last extensive revision: Daniel Moroz, fall 2015)

Plan Day 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; download at python.org
- Named after the Monty Python comedy group
  - Homework :)

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, designed for linear algebra)
- Large collection of support libraries:
  - eg. NumPy for Matlab like programming

Recommended Reading
- Python Overview
  - The Official Python Tutorial (https://docs.python.org/2/tutorial/)
  - Slides for CIS192, Fall 2015, (used in these slides) (https://www.cis.upenn.edu/~cis192/fall2015/)
- 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 yet Python 3
  - Many elegant but incompatible changes
  - More existing third party software is still 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

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

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: static typing
  - Variables can only refer to objects of a declared type
  - Methods use type signatures to enforce contracts
- Python: dynamic typing
  - Variables come into existence when first assigned.
    ```python
    >>> 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)
- Assignment Operators
  - +=, -=, *=, /=
  - No ++ or --
  - 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.
x = 3.45
if x == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World" # String concat.
print x
print y
```

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

**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. (Regular string in Unicode in Python 3.)
- **List**
  - Mutable ordered sequence of items of mixed types

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

```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 to the very end of the sequence (4.56, (2,3), 'def')
CIS 421/521 - Fall 2017
```

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, 'i', 11, 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']
CIS 421/521 - Fall 2017
```

Operations on Lists II

```python
>>> li = [5, 2, 6, 8]
>>> li.reverse() # reverse the list *in place* (modify)
>>> 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
>>> sorted(li) # return a *copy* sorted
CIS 421/521 - Fall 2017
```

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.capitalize()
'Dog'
>>> s.upper()
'DOG'
>>> ' hi --'.strip('–
')
'hi'
https://docs.python.org/2/library/string.html
CIS 421/521 - Fall 2017
```

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'
CIS 421/521 - Fall 2017
```

Dictionaries: a mapping collection type

```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'
CIS 421/521 - Fall 2017
```
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'
Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo

>>> d['user'] = 'clown'
# Assigning to an existing key replaces its value.
```
List Comprehension extra for

[x for x in lst1 if x > 2]
  for y in lst2:
    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

(k: v for k,v in lst)
d = dict() # translation
for x, y in lst:
  d[x] = y
(k for x in lst)
s = set() # translation
for x in lst:
  s.add(x)

Iterators

- Iterators are objects with a next() method:
  >>> i = iter(k)
  >>> i.next()
  1
  >>> i.next()
  2
  >>> i.next()
  3

- Python iterators do not have a hasNext() method!
- Just catch the StopIteration exception

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
An 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

Generators use memory efficiently

Eg: File Objects
>>> for line in open("script.py"): #returns iterator
    ...     print(line.upper())

IMPORT SYS
PRINT(SYS.PATH)
X = 2
PRINT(2 ** 3)

instead of
>>> for line in open("script.py").readlines(): #returns list
    ...     print(line.upper())

Generators

Generators: using yield

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

>>> def f(n):
    ...     yield n
    ...     yield n+1
    ...     yield n+2
    ...

>>> type(f)
<type 'function'>

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.

>>> 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: xrange(n) vs range(n)

▪ xrange acts like a generator
▪ range(n) keeps all n values in memory before starting a loop
  even if n is huge; for k in range(n)
▪ sum(xrange(n)) much faster than 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
```

```python
>>> [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
```

Generator comprehensions

- Review: `sum(xrange(n))` much faster than `sum(range(n)) for large n`

- Similarly,
  ```python
  >>> sum(x for x in xrange(10**8) if x%5==0)
  999999950000000L
  which uses a generator comprehension
  is much faster than
  >>> sum([x for x in xrange(10**8) if x%5==0])
  999999950000000L
  which creates the entire list before computing the sum
  ```

Plan for next time

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