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 :)
Python Environments

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

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

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 (eg. “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 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

```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

- 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'
>>> tu[-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', (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
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
[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
['a', 'c', 'b']
```

Operations on Strings

```python
>>> s = "Pretend this sentence makes sense."
>>> words = s.split(" ")
['Pretend', 'this', 'sentence', 'makes', 'sense. ']
>>> "_".join(words) # join method of obj
_Pretend_this_sentence_makes_sense_. '
>>> s = 'dog'
>>> s.capitalize() # capitalize
'Dog'
>>> s.upper() # upper
'DOG'
>>> ' hi --'.strip('–')
'hi --'
```

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

```python
>>> a = ['apple', 'orange', 'banana']
>>> for (index, fruit) in enumerate(a):
...   print('str(index) = "' + str(index) + '": ' + fruit)
...   ...
str(index) = "0": apple
str(index) = "1": orange
str(index) = "2": banana
```
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']
```

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

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

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

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

List Comprehensions replace loops!
- `[x for x in lst1 if x > 2]`
  ```python
  >>> li = [3, 6, 2, 7]
  >>> [x for x in li if x > 2]
  [3, 6, 7]
  ```
  ```python
  >>> li = [3, 6, 2, 7, 1, 9]
  >>> [x for x in li if x > 4]
  [3, 6, 7, 9]
  ```
  ```python
  >>> li = [3, 6, 2, 7, 1, 9]
  >>> [x for x in li if x > 4]
  [3, 6, 7, 9]
  ```

List Comprehension extra for
- `[x for x in lst1 if x > 2]
  for y in lst2]`
  ```python
  >>> li1 = [1, 4, 3, 2, 7]
  >>> li2 = [1, 5, 6, 7, 9]
  >>> [x for x in li1 if x > 2] for y in li2
  ```
  ```python
  res = [] # translation
  for x in li1:
      if x > 2:
          for y in li2:
              for z in li3:
                  if x + y + z < 8:
                      res.append(x)
  ```
Dictionary, Set Comprehensions

(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:

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

 Iterators
• Iterators are objects with a next() method:

```
>>> 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

```python
>>> for char in Reverse('spam'):
   print char
m
a
p
```

Iterators use memory efficiently

```
<table>
<thead>
<tr>
<th>Eg: File Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;&gt; for line in open(&quot;script.py&quot;): #returns iterator</td>
</tr>
<tr>
<td>...     print(line.upper())</td>
</tr>
<tr>
<td>IMPORT SYS</td>
</tr>
<tr>
<td>PRINT(SYS.PATH)</td>
</tr>
<tr>
<td>X = 2</td>
</tr>
<tr>
<td>PRINT(2 ** 3)</td>
</tr>
<tr>
<td>instead of</td>
</tr>
<tr>
<td>&gt;&gt;&gt; for line in open(&quot;script.py&quot;).readlines(): #returns list</td>
</tr>
<tr>
<td>...     print(line.upper())</td>
</tr>
</tbody>
</table>
```

Instead of:

```
| X = 2 |
| PRINT(2 ** 3) |
```

---

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.

```
>>> 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
```

---

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).

```
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
```

---

**Plan 2**

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