Python – Introduction

Kurt Schmidt

Dept. of Computer Science, Drexel University

October 17, 2016

Intro

Python

- True, general-purpose language
- Applications for Python:
  - Web and Internet development
  - Scientific and Numeric computing
  - Teaching programming, introductory, and more advanced
- Tk GUI library is included with most distributions
- Python is often used as a support language for developers, for build conrol and management, for testing, in many other places

Interpreter

- These notes refer to Python 2.7
- I might mention some differences in Python 3, when interesting
- Python has a very convenient interactive interpreter
  - Can be used as a calculator
  - Documentation is handy

```
$ python
Python 2.7.6 ...
>>> 4 + 3
7
>>>
```

- Newline has special meaning
- Use `^D` to exit

Useful Tidbits

- Newlines separate statements in Python
  - Use ; to separate statements on the same line
- Escape the newline with \ to continue a statement

```
x = (x-1) * foo( 'a really long argument list' ) \
    + 12
```

- # introduces a line comment
  - We will see *docstrings* a bit later

```
i = 5   # Newtons
j = 3*i  # Very clever calculation
```

- `None` is the sentinel reference (the NULL pointer)

dir and help

- dir lists all the members of a class

```
>>>   # Name the type:
>>> dir( str )
...
>>>   # or, use an instance, or variable holding that type:
>>> dir( 'blah' )
...
```

  - Note, members surrounded by underscores are generally helper methods, used for defining operators

- help gives help on a class or member

```
>>> n = 'Turing'
>>>   # To see a nice descpription of the entire class:
>>> help( n )
...
>>>   # To get help on a method:
>>> help( str.rfind )
...
```

Types

Floats – `float`

- We have 2 division operators
  - Same behavior in Python3, on floats
  - `/` is float division
  - `//` is integer division (but yields a float type)

```
>>> 12.0 / 5
2.4
>>> 12 // 5
2.0
```

- We have an exponentiation operator, `**`

```
>>> 3.0 ** 4
81.0
>>> 2**0.5
1.4142135623730951
```

Integers – `int`

- In Python2, the two division operators have identical behavior on `int` type

```
Python 2.7.6 ...
>>> 12 / 5
2
>>> 12 // 5
2
```

- In Python3, `/` performs float division
- In Python3, `//` performs integer division

```
Python 3.4.3 ...
>>> 12 / 5
2.4
>>> 12 // 5
2
```

Large Integers – `long`

If operations on integers exceed the capacity of `int`, Python has the `long` type, which handles arbitrarily large integers.

- Since Python 2.5, this coercion happened quietly, as needed

```
Python 2.7.6 ....
>>> type( 10 )
<type 'int'>
>>> type( 10**20 )
<type 'long'>
```

- In Python 3, the distinction is removed entirely
- `int` is the only integer type, and it handles arbitrarily large integers

```
Python 3.4.3 ....
>>> type( 10 )
<class 'int'>
>>> type( 10**20 )
<class 'int'>
```

Complex Numbers

Python actually has built-in complex types.

- Components can be `int` or `float`

```
Python 2.7.6 ....
>>> type( 3+4j )
<type 'complex'>
>>> type( 3.0+4j )
<type 'complex'>
>>> ( 3+4j ) * ( 27-12j )
(129+72j)
>>> ( 12+17j ) / 5
(2.4+3.4j)
>>> ( 12+17j ) // 5
(2+0j)
>>> (3+4j)**2
(-7+24j)
```

Variables and Assignment

- Python variables are not declared
  - They are dynamically typed
- *Everything* is a reference in Python
- *Everything* is an object in Python
  - Some types are immutable

```
$ python
Python 2.7.6 ...
>>> x = 12
>>> type( x )
<type 'int'>
>>> x = 1.732050807
>>> type( x )
<type 'float'>
>>> x = "Zaphod Beeblebrox"
>>> type( x )
<type 'str'>
```

String – `str`

- Delimited by single quotes, or double quotes
  - No difference
- Concatenation, with +
- Repetition, with *

```
>>> f = 'Cookie'
>>> l = "Monster"
>>> n = f + l
>>> n
'CookieMonster'
>>> n = f + ' ' + l
>>> print n
Cookie Monster
>>> print 'Spam ' * 3
Spam Spam Spam
```

printf-Style Formatting

- % joins a printf-style format string to arguments:

```
>>> print '%s is %d, has a %.1f average\n'%('Kurt', 13, 93.6666)
Kurt is 13, has a 93.7 average
```

- Superceded by `format` method since 2.6
  - Much fighting in the land, which one is better

```
>>> print '{0} is {1}, has a {2:.2f} average\n'.format(
>>>     'Kurt', 13, 93.6666 )
Kurt is 13, has a 93.67 average
```

- If you're familiar with neither, you should probably use `format`

String Concatenation & Types

- Python is pretty strongly typed
  - Can't add a number to a string, e.g.
- Every object has a `__str__` method
  - Called by the `str` operator
  - (Really, it's the constructor for the `str` class)

```
>>> print 'Jupiter' + str( 13 )
Jupiter13
>>> print str( 10 ) + 'Q'
10Q
>>> print "My list: " + str( [1,2,3] )
My list: [1, 2, 3]
```

Strings Are *Iterable*

- Indexed, starting at 0
  - Use index operator, `[]`
- `len` operator yields the length of an iterable
- These strings are *immutable*
  - Can not modify contents

```
>>> len( n )
13
>>> print n[0]
C
>>> print n[12]
r
>>> n[9] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

`split` and `join` Methods

- `split` takes a delimeter, returns a `list` of strings
  - (Coming soon)

```
>>> l = 'line,from,a,CSV'
>>> l.split( ',' )
['line', 'from', 'a', 'CSV']
```

- `join` takes a `list` of strings, returns a string

```
>>> l = [ 'Kurt', 'CS265', '013', '93', '81', '97' ]
>>> ':'.join( l )
'Kurt:CS265:013:93:81:97'
```

Some Other Handy Methods

Note, strings are immutable, so, methods that seem to make a change actually return a new `str`[1]

`strip lstrip rstrip` Strips off leading/trailing characters (whitespace by default)

`split join` Parse (create) a string to (from) a list of strings

`find rfind` Find a substring in a string (whitespace by default)

`replace` Replace a substring w/a new string.

[1] see `StringIO` or `MutableString` for efficient string processing

Use Indices to Get Substrings

- Index operators can take a range
  - Called a *slice*
- End position is a “one past the end” notion
  - That character not included
  - Leave empty to indicate rest of the string

```
>>> s = 'Isaac Asimov'
>>> s[4:9]
'c Asi'
>>> len( s )
12
>>> s[11]
'v'
>>> s[6:11]
'Asimo'
>>> s[6:]
'Asimov'
```

More on Slices

- Can leave start position blank to start at the beginning:

```
>>> s[0:7]
'Isaac A'
>>> s[:7]
'Isaac A'
```

- Use a negative position to count relative to the end

```
>>> s[:-1]  # All but the last letter
'Isaac Asimo'
>>> s[:-2]  # All but the last two
'Isaac Asim'
```

Raw Strings

- Any string, prefixed with an `r`
- Ignores escape sequences

```
>>> s = 'A\tline\040with\nwhitespace'
>>> s
'A\tline with\nwhitespace'
>>> print s
A  line with
whitespace
>>> r = r'A\tline\040with\ano\nwhitespace'
>>> r
'A\\tline\\040with\\nwhitespace'
>>> print r
A\tline\040with\nwhitespace
```

Multiline Strings

- Delimited with 3 single (’’’) or 3 double (""") quotes
- Just syntax to allow strings to contain newlines

```
>>> s = '''This is a line
... Another line, same string.
...    Leading white space (after the prefix) be preserved.'''
>>> print s
This is a line
Another line, same string.
    Leading white space (after the prefix) be preserved.
```

- Makes a handy doc string for functions and classes

```
def foo( a=5 ) :
   '''foo - a silly function
   Input: an integer, optionally
   Output: The answer to live, the universe, and everything
   Side-effects: Eats all your chocolate cookies'''
   ...
```

Operators

Implementing Operators

- Behavior for operators supplied by method in left-most operand

`+`	`__add__`	`str()`	`__str__`
`-`	`__sub__`	`>` `==` etc.	`__cmp__`
`*`	`__mult__`	`~`	`__invert__`
`/`	`__div__`	`&`	`__and__`
`//`	`__floordiv__`	`|`	`__or__`
`%`	`__mod__`	`^`	`__xor__`

Lists (Arrays)

Python *list* (Array)

- Indexed container
- Can hold any data
  - *Everything* is a reference in Python

```
>>> l = list() # explicitly call constructor
>>> l = []  # Use language syntax
>>> m = [ 5.7, 'Dead Collector', 13 ]
>>> m2 = [ 'A', m, 42 ]
```

- Use `len`

```
>>> len( m )
3
>>> m2
['A', [5.7, 'Dead Collector' 13], 42]
>>> len( m2 )
3
```

Indices

- Lists are indexed

```
>>> l = [ 5.7, "Dead Collector", 13 ]
>>> m = [ 'A', 1, 42 ]
>>> l[2]
13
>>> m[0]
'A'
>>> m[1]
[5.7, "Dead Collector", 13]
>>> m[1][2]
13
```

- Lists are *not* immutable
  - We can assign particular elements

```
>>> l[1] = 'Black Knight'
>>> l
[5.7, 'Black Knight', 13]
>>> m  # Note, m has changed (apparently)
['A', [5.7, 'Black Knight', 13], 42]
```

Slices

- Just as with strings, we can take slices of a list
  - Returns new list
  - Does not modify original list

```
>>> l = [ 1, 2, 3, 4 ]
>>> l[1:3]
[2, 3]
>>> l[:-1]
[1, 2, 3]
```

- Take a slice of the entire array to make a copy:

```
>>> m = [ 'a', l[:], 'b' ]
>>> m
['a', [1, 2, 3, 4], 'b']
>>> l[2] = 42
>>> l
[1, 2, 42, 4]
>>> m
['a', [1, 2, 3, 4], 'b']
```

append, insert

- Use `append` to add an item to the end of the list:

```
>>> l = []
>>> l.append( 'a' )
>>> l.append( 'b' )
>>> l.append( 'c' )
>>> l
['a', 'b', 'c']
```

- Use `insert(idx, item)` to insert `item` at (prior to) `idx`

```
>>> l.insert( 1, 13 )
>>> l
['a', 13, 'b', 'c']
```

pop

- Use `pop(idx)` to remove element at `idx` (last item by default)

```
>>> l
['a', 'b', 'c', 1, 2, 3]
>>> l.pop()
3
>>> l
['a', 'b', 'c', 1, 2]
>>> l.pop( 1 )
'b'
>>> l
['a', 'c', 1, 2]
```

extend

- `extend` takes an iterable object, appends its elements, in order, onto list

```
>>> l = [ 'a', 'b', 'c' ]
>>> m = [ 1, 2, 3 ]
>>> l.append( m )
>>> l
['a', 'b', 'c', [1, 2, 3]]
>>> l.pop() # remove last element
[1, 2, 3]
>>> l.extend( m )
>>> l
['a', 'b', 'c', 1, 2, 3]
>>> l.extend( 'Bilbo' )
>>> l
['a', 'b', 'c', 1, 2, 3, 'B', 'i', 'l', 'b', 'o']
```

`tuple` – Immutable list

- Handy as a key in a dictionary (soon)
- Can be made from any iterable:

```
>>> l = [ 1, 2, 3, 4, 5 ]
>>> t = tuple( l )
>>> t
(1, 2, 3, 4, 5)
>>> s = tuple( 'Kurt' )
>>> s
('K', 'u', 'r', 't')
```

- Can be indexed:

```
>>> t[1:4]
(2, 3, 4)
```

- Can *not* be modified:

```
>>> t[3] = "Gollum"
...
TypeError: 'tuple' object does not support item assignment
```

Tuple Literals

- Tuples can be described directly:

```
>>> t = ( 3, 7.2, 'Radagast', 5 )
>>> t
(3, 7.2, 'Radagast', 5)
```

- Singletons have a slightly different syntax
  - To avoid confusion with parenthesised numeric expressions

```
>>> t = (4)
>>> t
4
>>> t = (4,)
>>> t
(4,)
```

Sorting Lists

Lists have a `sort` method:

`L.sort(cmp=None, key=None, reverse=False)`

- In place
- Stable

```
>>> l = [ 283.27, 24, 'Pippin', 12, 2.7, "Merry" ]
>>> l.sort()
>>> l
[2.7, 12, 24, 283.27, 'Merry', 'Pippin']
```

- Python has a `sorted` operator
  - Doesn't modify list

```
>>> l = [ 283.27, 24, 'Pippin', 12, 2.7, "Merry" ]
>>> m = sorted( l )
>>> m
[2.7, 12, 24, 283.27, 'Merry', 'Pippin']
>>> l
[283.27, 24, 'Pippin', 12, 2.7, 'Merry']
```

Tests

Relational Operators

Python has two boolean literals:[1]

`False True`

We have the normal relational operators:

`< <= == != >= >`

```
>>> 3 < 22
True
>>> 'bulb' < 'flower'
True
>>> 'bulb' < 'Flower'
False
>>> 42 < 'life'
True
>>> 42 < '17'
True
```

[1] Not really; just globals defined to be 0 and 1

Logical, Membership

We have friendly logical operators:

`not and or`

- Listed in order of decreasing precedence
- Use parentheses

Python has a membership operator, `in`:

```
>>> 3 in [1, 2, 3]
True
>>> 2 in (1, 2, 3)
True
>>> 'i' in 'team'
False
>>> 'am' in 'team'
True
>>> 'i' not in 'team'
True
```

Identity Operator

- Remember, everything in Python is a reference
- `is` operator tests references

```
>>> l = [ 'a', 'b', 'c' ]
>>> m = l
>>> m == l
True
>>> m is l
True
>>> c = l[:]
>>> c == l
True
>>> c is l
False
>>> c is not l
True
```

Other Types as `bool`

- `None` is false
- The number 0 (and 0.0) is false
- All other numbers are true
- The empty string is false
- All other strings are true
- More generally, empty containers are false
  - Non-empty containers are true

Control Statements

Branches – Basic (`if`)


```
if cond :
    body
```

If consequent is a single statement, you can do this:

```
if cond : stmt
```

- Note, the lack of parentheses
  - Not syntax for If statement or a Loop
- Body of consequent is uniformly indented

If-Else

More generally:

```
if cond :
   body
elif cond :
   body
...
else :
   body
```

```
if g >= 90 :
   print "A"
elif g >= 80 :
   print 'B'
elif g >= 70 :
   print 'C'
else :
   print 'F'
```

`for` Loops

- Python has `for` and `while` loops
- No `until` nor `do` loops
- We have the usual `break` and `continue` statements

For Loops

- Used with any iterable
- Body of loop is indented (consistently)[1]

```
>>> for c in 'ABC' :
...     print c
...
A
B
C
>>> l = [ 87, 27, 17, 62 ]
>>> for i in sorted( l ) :
...     print i
...
17
27
62
87
```

[1] Be careful, mixing tabs and spaces

range

Can be called a couple ways:

range(*end*)
range(*start*, *end*[, *step*])

- Returns a list
- `end` is one-past-the-end

```
>>> range( 8 )
[0, 1, 2, 3, 4, 5, 6, 7]
>>> range( 3, 13, 2 )
[3, 5, 7, 9, 11]
>>> range( 14, 2, -3 )
[14, 11, 8, 5]
```

xrange and For Loops

- `xrange` returns an *iterator*
- Can create C-style for loops:

```
>>> for i in xrange( 8 ) :
...   print i,
...
0 1 2 3 4 5 6 7
```

Python 3

In Python3 `range` behaves as `xrange`; `xrange` doesn't exist

List Comprehensions

- An efficient, easy way to map a function over a list

```
>>> [ i**2 for i in range( 1, 13 ) ]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144]
>>> [ (i, i**2) for i in range( 1, 7 ) ]
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36)]
>>> [ str(e) for e in [ 13, 'Jaga', 3.28 ]]
['13', 'Jaga', '3.28']
>>> [ (len(s), s) for s in [ 'Happy', 'Doc', 'Sneezy', 'Dopey' ]]
[(5, 'Happy'), (3, 'Doc'), (6, 'Sneezy'), (5, 'Dopey')]
```

- We can do outer joins[1]:

```
>>> [ (i, j) for i in range(1,5) for j in range(1,5) ]
[(1, 1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4),
(3, 1), (3, 2), (3, 3), (3, 4), (4, 1), (4, 2), (4, 3), (4, 4)]
```

[1] see `zip` for inner joins

while Loop

- Again, statements in body denoted by new (consistent) indent level
- Body of loop is indented (consistently)[1]

```
import sys

print 'Enter a # to square (neg. to quit) => '
i = int( sys.stdin.readline() )
while i >= 0 :
   print 'Enter a # to square (neg. to quit) => '
   i = int( sys.stdin.readline() )
```

[1] Be careful, mixing tabs and spaces

Dictionary

Associative Array – `dict`

- Key can be any immutable type
- Value can be anything

```
>>> d = dict()
>>> d[ 'Gandalf' ] = 'grey'
>>> d[ 17 ] = 'seventeen'
>>> d[ ('one', 'two') ] = 'buckled shoe'
>>> d
{17: 'seventeen', 'Gandalf': 'grey', ('one', 'two'): 'buckled shoe'}
>>> d[3.1416] = "Pi-ish"
>>> 7.0**2
49.0
>>> 49**(0.5)
7.0
>>> d
{17: 'seventeen', 'Gandalf': 'grey', ('one', 'two'): 'buckled shoe',
3.1416: 'Pi-ish'}
>>> d[ 'Gandalf' ] = 'White'
>>> d
{17: 'seventeen', 'Gandalf': 'White', ('one', 'two'): 'buckled shoe',
3.1416: 'Pi-ish'}
```

Testing for Membership

- Use the `has_key` method

```
>>> d.has_key( 'Kim' )
True
>>> d.has_key( 'kim' )
False
>>> d.has_key( '13' )
False
```

- Or, use the keyword `in`

```
>>> 'Kim' in d
True
>>> "Kim" in d
True
>>> 'kim' in d
False
```

Dictionary – Keys

- An iterator over a `dict` is an iterator over its keys
- Or, use the `keys` method

```
>>> d.keys()
[17, 'Gandalf', ('one', 'two'), 3.1416]
>>> 'Gandalf' in d
True
>>> d.has_key( 3.14 )
False
>>> for k in d :
...   print k, d[k]
...
17 seventeen
Gandalf White
('one', 'two') buckled shoe
3.1416 Pi-ish
```

Dictionary – Remove Key

- Use `pop()` to remove an entry

```
f = { 1:3, 2:3, 3:4, 5:13 }
>>> f.pop( 3 )
4
>>> f
{1: 3, 2: 3, 5: 13}
```

Dictionary – more traversals

- If you need to sort keys:

```
>>> for k in sorted( d.keys() ) :
...   print k, d[k]
...
3.1416 Pi-ish
17 seventeen
Gandalf White
('one', 'two') buckled shoe
```

- `items()` returns a list of (key, value) pairs, as tuples

```
for (k,v) in d.items() :
...   print k, v
...
17 seventeen
Gandalf White
('one', 'two') buckled shoe
3.1416 Pi-ish
```

Built-in `set`, `frozenset`

- `frozenset` is simply an immutable `set`
- Can provide a constructor with any iterable:

```
s = set()
s = set( ['hello', 13, 6.022094] )
```

- Since 2.7, we do have dedicated syntax (non-empty sets):

```
s = { 'hello', 13, 6.022094 }
```

- Use `add(`*item*`)`, `discard` v. `remove`
- All of the normal set operations are available
  - Most operators work on sets
- `set` also has mutator versions of these operations

Functions

Functions


```
def foo( n, name=None ) :
   '''foo - handy example function
   Takes an integer, n (in furlongs) and, optionally, a name
   Returns 13, invariably'''

   if name is None : name = 'Skippy'
   if type(n) is not int : n = 13
   print 'Hello, ' + str(Name)

   return 13
```

```
>>> r = foo( 8 )
Hello, Skippy
>>> r
13
>>> r = foo( 42, 'Zaphod' )
Hello, Zaphod
```

Functions (cont.)

- A string delimited by 3 quotes (or double quotes) is called a *multiline* string
- If one appears immediately following a function's preamble, it is the *doc string*
- Self-documenting code:

```
>>> help( foo )

Help on function foo in module __main__:

foo(n, name=None)
    foo - example function
    Takes an integer, n (in furlongs) and, optionally, a name
    Returns 13, invariably
```

Recursion

- Python is quite happy with recursion

```
def gcd( a, b ) :
   if b==0 :
      return a
   return gcd( b, a%b )
```

```
>>> gcd( 15, 12 )
3
>>> gcd( 1003, 18 )
1
```

Functions are First Class Objects

- Functions can be passed to, and returned from, other functions

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

def myMap( fn, l ) :
   rv = []
   for f in l :
      rv.append( fn( f ))
   return rv
```

```
>>> myMap( square, [ 3, 11, 14, 25 ])
[9, 121, 196, 625]
```