An Introduction to Python

Presented to OCPUG, 2010-11-30

Why Learn Python: Part 1 of 3

- Easy to learn yet powerful
- Concise syntax: few words per idea expressed
- Usable for web, scripts, full blown standalone applications
- Runs on all major operating systems

Why Learn Python: Part 2 of 3

- No type declarations to speak of
- Friendly, helpful community
- The language is very "googleable"

Why Learn Python: Part 3 of 3

- Small core language
- One needn't learn a lot to become productive
- Large set of reusable modules available

Paradigms

- Procedural
- Object Oriented
- Functional (to an extent)

Major Implementations

CPython
Pypy
Jython
IronPython

CPython

- Latest Versions 2.7 and 3.1
- Written in C
- Has its own byte code
- Can sort of JIT with Psyco on x86
- The Reference Implementation

Pypy

- Version 1.4 implements Python 2.5
- Written in Python
- JIT available
- Can use C, JVM or .Net as backends
- Just recently considered “suitable for everyday development” and “production use”

Jython

- Written in Java
- Version 2.5 is Python 2.5
- JIT's
- Runs on the JVM

IronPython

- Written in C#
- Version 2.6 is Python 2.6
- JIT's
- Runs on .Net
- Doesn't have much of a Python Standard Library

Nouns and verbs

- Nouns: “data”
- Verbs: “operators”, “control flow”, “functions”, “methods”

The simplest control flow: sequential execution

```
print 'hello world'
print 'how are you?'
print 'goodbye'
# Like a recipe or chemistry experiment
```

(Scalar) types: Part 1 of 2

- int: whole number
- long: potentially large whole number
- float: whole number or fraction
- bool: logic 101 truth values
- None: special value representing “nothing”
- str: a sequence of characters

(Collection) types: Part 2 of 2

- list: a read/write sequence
- tuple: a readonly sequence
- dict: like a dictionary or “hash”
- set: from set theory
- file: a sequence of bytes or characters, usually on disk

Example Python 2.x int literals

0
1
999999

Example int, long and float operators

Addition: +
Subtraction: -
Multiplication: *
Integer (2.x) or float (3.x) division: /
Integer division: //
Modulus: %
Exponentiation: **

Example use of int

```
print(1+2)
# prints 3

print(5**2)
# prints 25
```

Example long literals in Python 2.x

1L
65535L
680564733841876926926749214863536 42L

Int vs. long in Python 2.x vs 3.x

- In Python 2.x, small integers are int's, and big integers are long's
- In Python 3.x, all integers are called int's, but actually look like 2.x's long's behind the scenes
- In 3.x, the “L” suffix is never used

Example float literals

1.0
3.14159
1.5e20

Example use of float

```
Print(3.14159)
# prints 3.14159

Print(2/9)
# prints 0.222222222222

Print(1.5e20)
# prints 1.5e+20
```

bool literals

True
False

Example bool operators

and
or
not

Common operators returning bool

Less than:	<
Less than or equal to:	<=
Greater than:	>
Greater than or equal to:	>=
Equal to:	==
Not equal to:	!=

Example use of bool

```
print(not True)
# prints False

Print(1 < 3)
# prints True

print(True and False)
# prints False

Print(True or False)
# prints True
```

None literal

None

Quick aside: variables

When you want a variable to have a value, you just assign it with =

```
x = 5
y = True
z = 2.71828
```

Some analogies for understanding variables

- Could be thought of as a sticky label you can place on a value
- Could also be thought of as a bucket you can put a value in
- They just assign a name to a value

Variables' degree of permanence

- Unlike in mathematics a variable, once assigned, does not necessarily retain that value forever.
- A subsequent assignment to the same variable changes its value.

Example of changing a variable

```
x = 5
print(x)
# prints 5

x = 10
print(x)
# prints 10; the previous  5 is lost
```

Example None operators

```
x == None

y is None
```

Example Python 2.x str literals

'abc'
“def”
'ghi”jkl'
“mno'pqr”
u' 公案 '

Python 3.x str literals

'abc'
“def”
'ghi”jkl'
“mno'pqr”
' 公案 '

...all str's are unicode in 3.x.

Example str operators

Catenation: +
Repetition: *
Slicing

Example use of str's

```
print 'abc' + “def”
# prints abcdef

print 'ghi' * 3
# prints ghighighi

print 'abcdefghi'[3:6]
# prints def
```

More on str slicing

print string[x,y] says:
print characters x through y-1

The leftmost character is character number zero

```
print '0123456789'[2:5]
# prints 234
```

Slicing with negative values

A negative number in a slice says “from the end”

```
string='abcdefghi'
print string[3:-2]
# prints defg
```

Example list literals

[]
[1]
[1, 2, 3, 4]
[20, 15, 5, 0]

Some other ways of getting a list: Python 2.x

```
print(range(3))
# prints [ 0, 1, 2 ]

print(range(5, 10))
# prints [ 5, 6, 7, 8, 9 ]
```

Example list operators

Slicing
sort()
Catenation: +
append
extend

Lists defined

- A list is a collection of (potentially) many values, kept in order, indexed by whole numbers from 0 to num_values-1
- They are similar to arrays in many other languages, but are very flexible compared to arrays in C (another programming language)
- Modifying the end of a list is fast; modifying the beginning of a list is slow

(More) example list operators

Indexing: list_[5]
Slicing: list_[5, 10]
list_.append(5)
del list_[5]
list_.pop()
Comparison operators: <, ==, >=, etc.
len(list_)
list_.sort()

A note on strings

```
# This is sometimes quadratic (slow):
string = ''
for i in range(10000):
    string += str(i)

# This is linear (fast):
list_ = []
for i in range(10000):
    list_.append(i)
string = ''.join(list_)
```

Some brief notes about tuples

- Tuples are like lists, except they're readonly, and their literals use (), not []
- The main exception is that a tuple with a single element is written (1,) - for example
- It's unfortunately easy to end up with a tuple by writing x = 1,

Dictionaries Defined

- Are similar to a real-world dictionary on one's bookshelf
- Are like a “hash” or “map” or “associative array” in some other languages
- Are a collection of (potentially) many variables, that facilitate easily finding where you put something previously
- Are indexed by immutable values and can store mutable or immutable values

Examples of dictionary literals

```
{}
{ 'a': 'abc', 'b': 'bcd' }
{ 1: 'xyz', 2112: 'pdq', 'string': 5.0 }
```

Example use of a dictionary

```
d = {}
d[0] = 1
d[1] = 2
d[2] = 4
d[3] = 8
d[4] = 16
d[5] = 32
print(d[0])
# prints 1
print(d[4])
# prints 16
```

(Further) example operations on dictionaries

len(dict_)
d1.update(d2)
dict_.keys()
dict_.values()
dict_.items()
==
!=

Operations on dictionaries: Python 2.x vs 3.x

- In 2.x, .keys(), .values(), and .items() return lists
- In 3.x, they return iterators, achieving lazy evaluation
- In 2.x, for an iterator, you must use .iterkeys(), .itervalues() and .iteritems()
- If you don't know the difference, you're probably better off with an iterator than a list

Suitability of Dictionary Keys

- Dictionary keys must be immutable (readonly) values
- So you cannot index a dictionary by a list, but you can index a dictionary by a tuple
- You can still put pretty much anything you want into a dictionary as a value; it's keys that are restricted

Sets defined

(From wikipedia): A set is a collection of distinct objects, considered as an object in its own right. Sets are one of the most fundamental concepts in mathematics.

Sets compared to dictionaries

- Sets are a lot like dictionaries minus the values – all they have are the keys
  - No key-value pairs

Creating sets

2.7 and up: { 'a', 'b', 'c' }
2.5, 2.6, perhaps earlier: set(['a', 'b', 'c'])

Example set operations

Cardinality: len(s1)
Intersection: s3 = s1 & s2
Union: s4 = s1 | s2
Difference: s5 = s1 – s2

Comparing sets

Equality: s1 == s2
Inequality: s1 != s2
Subset: s1 <= s2
Proper subset: s1 < s2
Superset: s1 >= s2
Proper superset: s1 > s2

Definition of files

A sequence of characters or bytes, typically in a filesystem on disk

Examples of files

A spreadsheet .ods or .xls
A text file .txt
A python file .py
sys.stdout
sys.stderr
sys.stdin

Common file operations: reading

```
file_ = open('file.txt')
file_.read(10)
file_.readline()
file_.close()
```

Common File Operations: Writing

```
file_ = open('file2.txt', 'w')
file_.write('line of text\n')
file_.close()
```

Python's type system

- pretty strong typing: few implicit conversions
  - bool might be implicitly promoted to int
  - int (or long) might be implicitly promoted to float
  - Almost anything is usable in a boolean context

Explicit type conversions

Usually if you want to convert a variable x to type t and save it in variable y: y = t(x)

Examples:

- y = int('1')
- y = float(5)
- y = str(1/9.0)

Modules

- Modules are the main way Python encourages code reuse
- Modules are also an important way of keeping the core language small

Example of reusing a module

```
import decimal
variable1 = decimal.Decimal(6)
variable2 = decimal.Decimal('0.33')
variable3 = variable1 * variable2
print(variable3)
# prints 1.98
```

What are decimals?

- An arithmetic type similar to float's
- Stored base 10 rather than float's base 2
- Slower than float
- More precise than float if used with human-readable, base 10 inputs

More modules in the standard library

sys, os, os.path, collections, re, struct, StringIO, time, heapq, bisect, array, copy, pprint, math, itertools, functools, operator, anydbm, gdbm, dbhash, bsddb, gzip, bz2, zlib, zipfile, tarfile, csv, hashlib, ctypes, select, multiprocessing, mmap, subprocess, socket, ssl, xml: sax, dom, elementtree, signal, email, json, cgi, urllib, httplib, profile, parser...

Discoverability

- python
- import decimal
- help(decimal)
- dir(decimal)

Creating your own modules

- Place the following in a file named foo.py and put it somewhere on your Python path (described by sys.path) or in “.”:

```
#!/usr/bin/python
print 'hello'
```

- And then in some other python file, you print the word “hello” with:

```
import foo
```

Getting an intuition for control flow

Winpdb! (or something like it)
http://winpdb.org/

Using winpdb

- Install winpdb
  - Ubuntu/Debian: Synaptic
  - Windows: wxWindows .exe + winpdb .zip + setup.py
- Create your script as (EG) foo.py
- At a shell prompt type: winpdb foo.py

Example if statement

```
if 1 == 1:
    print 'expression'
    print 'was'
    print 'True'
```

if statement described

A way of doing something 0 or 1 times

If using an oven, preheat.
If using a toaster oven, don't worry about it.

if/else

```
if canned_beans:
    print 'open can'
else:
    print 'soak beans overnight'
```

If/elif/else

```
if x < 10:
    print 'less than 10'
elif 10 <= x < 20:
    print 'between 10 and 20'
elif 20 <= x < 30:
    print 'between 20 and 30'
else:
    print 'something else'
```

Case/switch

- Python has no case statement or switch statement
- Instead use if/elif/else

Example while statement

```
x = 5
while x < 10:
    print(x)
    x += 1
# prints the values 5, 6, 7, 8 and 9, each on
# a separate line
```

while statement described

- Execute something 0 or more times
- Maybe 100 times
- Maybe forever

while analogy

```
put_food_in_oven()
while not is_cooked_clear_through():
    time.sleep(5*60)
remove_from_oven()
```

Example for statement

```
for i in range(5):
    print i
# prints 0, 1, 2, 3, 4 – each on a different line
```

for statement described

Do something once for each thing in a specific sequence

EG, if you were making apple pie, you might core an apple once for each apple

Exceptions

```
import sys
n = int(sys.argv[1])
try:
    print 1.0 / n
except ZeroDivisionError:
    print 'no reciprocal'
```

Example of a user-defined function

```
def square(x):
    result = x * x
    return result
print square(1)
# prints 1
print square(5)
# prints 25
```

User-defined functions described

- A way of doing something from more than one place in a program
- A way of introducing a “scope” to avoid variable name collisions
- A way of hiding detail

Generator example

```
def my_range(n):
    i = 0
    while i < n:
        yield i

for j in my_range(3):
    print j
# prints 0, 1, 2 each on a separate line
```

List comprehensions

```
list1 = range(5)
list2 = [ x / 10.0 for x in list1 if x % 3 == 0 ]
```

Parallelism

- CPython's threading is poor for CPU-bound processes, decent for I/O-bound processes
- CPython is good at “multiprocessing”: multiple processes and shared memory
- Jython and IronPython can thread well
- Stackless
- Pypy (Stackless)
- CPython: greenlets

Another way of getting a sequence in Python 2.x

```
for i in xrange(3):
   print(i)
# prints:
# 0
# 1
# 2
```

...and it's evaluated *lazily*

On range and xrange in Python 3.x

- xrange is gone in 3.x
- range in 3.x is like xrange in 2.x
- If you really do need an eagerly expanded list in 3.x, use list(range(x))

Example of reading a file line by line

```
file_ = open('foo.txt', 'r')
for line in file_:
    print line
file_.close()
```

Object Orientation

- Big topic
- class statement
- Like a “jack in the box”

Quick class Example

```
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __add__(self, other):
    result = Point(0, 0)
    result.x = self.x + other.x
    result.y = self.y + other.y
    return result

  def magnitude(self):
    return (self.x ** 2 + self.y ** 2) ** 0.5

  def __str__(self):
    return 'Point(%f, %f)' % (self.x, self.y)
```

Using the example class

```
point1 = Point(5, 10)
point2 = Point(6, 15)
print point1
print point2
print point1 + point2
print point1.magnitude()
# Outputs:
# Point(5.000000, 10.000000)
# Point(6.000000, 15.000000)
# Point(11.000000, 25.000000)
# 11.1803398875
```

Static Analyzers

- Pylint
- PyChecker
- Pyflakes

Further Resources – Part 1 of 3

- The Python Tutorial:
  http://docs.python.org/tutorial/
- Dive into Python: http://diveintopython.org/
- Python koans:
  http://bitbucket.org/mcrute/python_koans/downloads
- Cheat sheets: http://rgruet.free.fr/#QuickRef
- Google http://www.google.com/

Further Resources – Part 2 of 3

- Choice of 2.x vs 3.x:
  http://wiki.python.org/moin/Python2orPython3
- python-list (comp.lang.python):
  http://mail.python.org/mailman/listinfo/python-list
- Your local Python User Group

Further Resources – Part 3 of 3

- Why Python?
  http://www.linuxjournal.com/article/3882
- Why learn Python?
  http://www.keithbraithwaite.demon.co.uk/professional/presentations/2003/ot/why_learn_python.pdf

Questions?