George F. Duck Memorial Lecture Series
Dept of Mathematics and Computer Science
March 27, 2012

Python Tutorial

(adapted for CSC461 Programming Languages Fall 2015)

Dr. John M. Weiss, Professor
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Rapid City, South Dakota, USA
John.Weiss@sdsmt.edu

Why Python?

- Python is a good, general purpose programming language, widely used, offering high programmer productivity
- Good candidate teaching language in CS0, maybe even CS1

Python Strengths

- High level of abstraction
- Expressive and powerful
- Simple syntax, easy to learn and use
- Small, portable, cross platform, free
- Support for many domains: database, text processing, scientific and numeric apps, graphics, GUIs, web apps, etc.

Python Weaknesses

- Less efficient than C++:
  - Python is a higher level language, with greater abstraction from the underlying hardware
  - Python is not fully compiled
  - However: high performance modules can be written in other languages (like C) as needed
- Dynamic typing:
  - Interpreter cannot automatically detect all type errors
  - However: dynamic typing makes it much easier to write general purpose routines

Python Philosophy

- "Small and simple is beautiful" philosophy
- Free, open source, cross platform
- Highly extensible and modular
- Multiparadigm PL, with support for procedural, object-oriented, and functional programming
- Encourages good programming practices
- Fun to learn and use (named after a well-known British comedy troop)

Python History

- Python 1.0 (Guido van Rossum, ~1990)
- Python 2.0 (2000) – still in wide use
- Python 3.0 (2009) – more consistent syntax
- Ongoing language development
- TIOBE language of the year in 2007, 2010, 2011 (greatest growth in popularity)
- One of top ten PL's by any metric

Applications

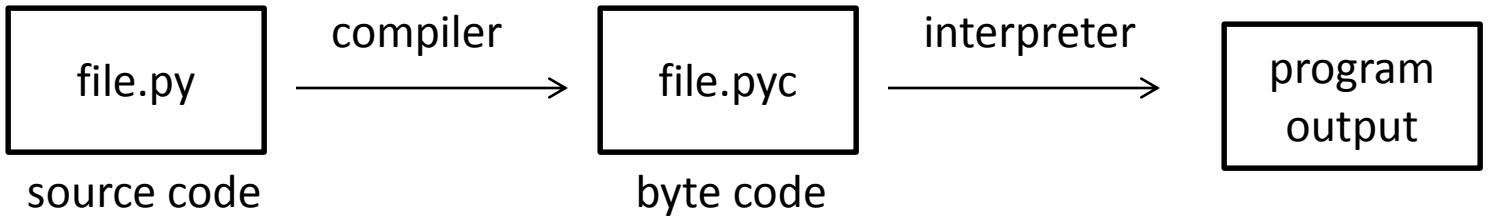
- Scripting language: install scripts, web applications, text processing, prototyping
- Embedded language (GIMP, Maya, ArcGIS, etc.)
- Library bindings (SQL, Qt, ROS, OpenCV, etc.)
- Used by Google, Yahoo, CERN, NASA, ILM, YouTube, BitTorrent, many more

Python Resources

- Website: www.python.org
- Extensive libraries
- Online documentation and tutorials
- Batteries included:
  - IDLE (IDE)
  - Tkinter (GUI)
  - Regular expressions
  - Unit testing
  - Threading and multiprocessing
  - Web programming
- Many third party packages (PyPi repository)

Usage

- Python interpreter: read-eval-print loop (REPL)
- Scripts and modules are partially compiled and partially interpreted:



- Scripts: **`python file.py`**
- Modules: **`import module`**

Python Implementations

- CPython –produces standard Python bytecode
- Jython – produces Java bytecode
- IronPython – targets the .NET platform, produces DLR bytecode (note also *Python Tools for Visual Studio*)

Python Program Execution

- Python is a (partially) interpreted language.
- Statements are translated and executed in sequence.
- No main() function, but easy to fake it. This pattern allows a program to be interpreted as either a script or a module:

```
if __name__ == '__main__':
    # "main" program stmts go here
```

Code Formatting

- *Indentation is required, not optional.* Statement blocks (function bodies, if-else, loops) *must* be indented to the same level.
- One statement per line recommended.
- Use backslash (\) to continue statement over multiple lines.
- Use semicolon (;) to separate multiple statements on one line.

Comments

- Inline comments use #

```
x = 10          # inline comment
```

- Multiline comments use triple quoted strings
- These also serve as docstrings for online help

```
'''This is a multiline
   (or docstring) comment'''
```

Data Types

- Simple types: int, float, complex, boolean
- Aggregate types: string, list, tuple, dict, set, range, file
- No arrays! (well, there is an array module…)
- Dynamic typing: data type is associated with value, not variable name
- Dynamic memory (de)allocation

Data Types

- Integers (**int**) are infinite precision
- **Floats** are IEEE double precision
- **Boolean** are True, False
- Strings (**str**) are immutable (create a new string instead), support both ASCII and Unicode
- Many string methods, also overloaded ops + (concatenation) and * (repetition)

Data Types: Lists

- **Lists** are a very general data type: `[ e1, e2, … ]`
- Subscript like arrays: `mylist[i]`
- Slices: `mylist[i:j]`
- May store heterogeneous values
- Superset of stacks, queues, deques
- Implement nonlinear structures (trees, graphs, multidimensional arrays) via nested sublists
- Many list methods, also overloaded op + (append)

Data Types

- **Tuples** are immutable lists: **(e1, e2, …)**
- **Sets** provide membership (*in* op), union, intersection, difference: **{ e1, e2, … }**
- Dictionaries (**dicts**), aka associative arrays or maps, are hash tables that store (key, value) pairs: **{ k1:v1, k2:v2, … }**

Data Types

- **Range** type provides a range of integer values
  - range(start, end, increment)
  - Often used for iteration
- **File** type
  - May open file for read/write/append access
  - Text vs. binary files, sequential vs. random access
  - *Pickle* module allows you to write (“dump”) data structure to a file, and read it back in (“load”)

Expressions

- Operators are similar to C++
- Similar precedence and associativity
- Float division: x / y
- Integer division: x // y
- Exponentiation: x ** y
- No increment (++) or decrement (--) ops

Modules

- Modules are libraries
- Usage: import modulename
  modulename.resourcename

```
import math
x = float(input("Enter a value: "))
y = math.pi * math.sqrt( x )
```

Online Help

- dir(object) – list of object attributes
- help(topic) – online help

```
help()              # go into online help utility

help(list)          # help on list type

import math
dir(math)     # list all math module attributes
help(math)    # help on all math functions
help(math.sqrt)   # only math sqrt function
```

Ten Statement Python

1. **Assignment**: var = expr
2. **Input**: var = input("prompt")
3. **Output**: print(expr1, expr2, …)
4. **File i/o**: open(), close(), read(), write()
5. **Function definition**: def sqr(x): return x*x

Ten Statement Python (cont.)

6. **Selection**: if stmt
7. **Repetition**: while loop
8. **Repetition**: for(each) loop
9. **Exception handling**: try, except
10. **Class definition**: class C(parent): methods

Of course there's more…

Assignment

- Variable name is a reference to an object
- Not declared or typed (although you may declare variables to be global)
- Cannot reference a variable until it has been bound to a value via assignment: var = expr
- When value changes, data type may change
- Tradeoff: flexibility vs. reliability (type checking) and efficiency

Assignment

- Simple:

```
var = expr
```

- Compound:

```
var += expr, var -= expr, etc.
```

- Extended:

```
var1, var2 = expr1, expr2
```

Console Input

- `var = input( "prompt" )`
- Prompt string is optional
- Returns input as string; use typecast to convert to desired data type:

```
x = int(input("Enter an integer: "))
```

Console Output

- **`print( expr1, expr2, … )`**
- Expressions must be strings, but most types are automatically converted to printable representations
- Options:
  - sep=‘separator string’ (defaults to space)
  - end=‘end string’ (defaults to newline)
  - file=ofp (defaults to stdout)
- Two methods for producing formatted output

Formatted Output

- **format operator (%)**
  - `"format string" % (expr1, expr2, …)`
  - Format string contains format specifiers similar to C printf: %d, %f, etc.
- **string type format method**
  - `"format string".format(expr1, expr2, …)`
  - Format string uses different format specifiers, but functionality is equivalent

Formatted Output

- Format operator (%)

```
for i in range( 10, 50, 10 ):
  print('%3d %7.2f' % (i, sqrt(i)))
```

- String format method

```
for i in range( 10, 50, 10 ):
  print('{0:3d} {1:7.2f}'.format(i, sqrt(i)))
```

- Output (both methods)

```
 10    3.16
 20    4.47
 30    5.48
 40    6.32
```

Selection: if statement

- Similar to C++ else-if stmt:

```
if test:
    stmt block
[ elif test:
    stmt block ]*
[ else:
    stmt block ]
```

- Note: [] means optional, []* means 0 or more
- No switch/case statement

Selection: if statement

```
if x < 0:
    print( x, "is negative" )
elif x > 0:
    print( x, "is positive" )
else:
    print( x, "is zero" )
```

Iteration: while loop

- Similar to C++ while loop

```
while test:
    stmt block
[ else:
    stmt block ]
```

- *else* block executes on normal termination (not when *break* out of loop)

Iteration: while loop

```
x = 0
while x <= 0:
    x = int( input( "Enter a "
                    "positive integer:")
else:
    print( "You entered", x )
```

Iteration: for (each) loop

- Iterates through almost any type of collection

```
for loopvar in iterable:
    stmt block
[ else:
    stmt block ]
```

- *else* block executes on normal termination (not when *break* out of loop)

Iteration: for (each) loop

- Typical C++ code:

```
for ( int i = 0; i < n; i++ )
    sum += numlst[i];
```

- Python equivalent:

```
for i in numlst:
    sum += i
```

Iteration: for (each) loop

```
for i in range(0,100,10):
    print( i, end = ' ' )
else:
    print( "\nThat's all, folks!" )
```

Output

```
0 10 20 30 40 50 60 70 80 90
That's all, folks!
```

Iteration: for (each) loop

```
for i in "HELLO":
    print( i )
```

Output

```
H
E
L
L
O
```

Iteration: for (each) loop

```
for i in [ 100, 3.14159, "hi" ]:
    print( i )
```

Output

```
100
3.14159
hi
```

File I/O: Input

- Open file for reading:

```
ifp = open( "input file" )
```

- Read data from file:

  `ifp.read()` - entire file contents

  `ifp.readline()` - one line of file

- Or iterate line-by-line with for loop:

```
for line in ifp: print( line )
```

- Close file:

```
ifp.close()
```

File I/O: Output

- Open file for writing:

```
ofp = open( "output file", "w" )
```

- Write expression to file:

```
ofp.write( expr )
print( expr, file = ofp )
```

- Close file:

```
ofp.close()
```

- **help(open)** – lists various file r/w modes

File I/O

- Copy a text file

```
fin = open( "input.txt" )
fout = open( "output.txt", "w" )
fin.read( buffer )
fout.write( buffer )
fin.close()
fout.close()
```

- Line by line copy

```
for line in fin: fout.write( line )
```

Function Definitions

- Function parameters and return values are dynamically typed, which makes it trivial to write generic functions

```
def funcname( param1, param2, ... ):
    '''docstring''' # produces online help
    stmt1
    stmt2
    . . .
    return value
```

Function Definitions

```
def sumlist( numlst ):
  '''sum a list of any numeric type'''
  sum = 0
  for i in numlst:
    sum += i
  return sum
```

Function Definitions

```
def fib( n ):
    """nth Fibonacci number (iterative)"""
    a, b = 0, 1      # initial values
    for i in range( n ):
        a, b = b, a + b
    return a
```

Function Definitions

```
def fib( n ):
    """nth Fibonacci number (recursive)"""
    if n < 2:
        return n;
    else:
        return fib( n - 1 ) + fib( n - 2 )
```

Functions

- parameter passing is by value (more accurately, by assignment)
- optional arguments (with default values)
- variable number of arguments
- keyword arguments
- multiple return values (put in list)

Functions

- Functions are first-class objects in Python:
  - pass as function parameters
  - return as function value
  - assign to symbols
- Scope: define function prior to call
- Prototyping: use stub

```
def fname( params ): pass
def fname( params ): ...
```

Handling Exceptions

- Exceptions are runtime errors, such as divide by zero
- Python exception handling is similar to C++
- “try” block encloses code in which exception might occur
- Exception handlers are placed in “except” blocks
- Optional “else” and “finally” blocks

Handling Exceptions

```
def divide( x, y ):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero")
    except:      # catches all exceptions
        print("some other error")
    else:        # executed if no exception
        print("result is", result)
    finally:     # always executed
        print("that's all folks!")
```

Python Classes

- Python supports multiple inheritance, with an object class at the root of the hierarchy
- Class definitions are similar to C++ and Java, but only methods are declared (dynamic typing!)
- First argument (current object) is explicitly listed, usually called “self”
- Constructor is called __init__
- Overloaded operators are called __eq__, __add__, etc.

Classes

```
class Cmplx( object ):
  def __init__( self, r = 0, i = 0 ):
    self.r = r; self.i = i
  def __str__( self ):
    return "%g+%gi" % ( self.r, self.i )
  def __eq__( self, z ):
    return self.r == z.r and self.i == z.i
  def conj( self ):
    return Cmplx( self.r, -self.i )

z1 = Cmplx( 3, 4 ); z2 = Cmplx( -2.1, 6.3 )
if ( z1 == z2 ): print( "z1  =", z1 )
else:            print( "z1* =", z1.conj() )
```

Advanced Stuff

- List comprehensions
  - compact form of iteration over lists and other sequences
- Regular expression module
  - powerful string matching capabilities
  - match set of symbols, repeated occurrences, etc.

List Comprehensions

- Interesting, concise form of iteration
- [expr for var in sequence]

```
a = [ x * x for x in range(1,9) ]
```

vs.

```
b = []
for x in range(1,9):
    b += x * x
```

- Both forms produce `[1,4,9,16,25,36,49,64]`

List Comprehensions

- [expr for var in sequence if test]

```
a=[ x * x for x in range(1,9) if x % 2 == 0 ]
```

vs.

```
b = []
for x in range(1,9):
    if x % 2 == 0:
        b += x * x
```

- Both forms produce `[4,16,36,64]`

Example: Word Concordance

- Parse text file into words, compute word frequencies, print words sorted first by frequency and then alphabetically
- Good application of hash tables
- 450 lines of C++ code (total)
- 30 lines of Python code (total), only 6 (six!) needed to construct concordance
- Performance penalty: Python runs 1.25-2.00X more slowly than C++ but…
- Processing the complete works of Shakespeare (5M text file) only takes 1 second in Python

Python Word Concordance

```
# 1) read the file into one long string
words = open( filename ).read()
# 2) convert to lowercase, split into words
wrdlst = re.findall('[a-z]+', words.lower())
# 3) count occurrence of each word
d = dict()
for w in wrdlst: d[w] = d.get(w,0) + 1
# 4) invert the dictionary
wc = [ ( d[k],k ) for k in d ]
# 5) sort by word frequency
wc.sort()
```

Questions?

- Slides can be downloaded as PythonTutorial.pdf *
- Comments and corrections are welcome: john.weiss@sdsmt.edu

**** http://www.mcs.sdsmt.edu/csc461/Resources/Python/PythonTutorial.pdf***