

Introduction to Python

CS345 - Programming Languages

Dr. Greg Lavender

Department of Computer Sciences

The University of Texas at Austin

What is Python?

- An object-oriented “scripting” language
 - interpreted, interactive
 - facilitates a rapid edit-test-debug development cycle
 - object-oriented
 - modules, classes, exceptions
 - dynamically typed, automatic garbage collection
 - extensible
 - builtin interfaces to many C libraries
 - X11, MFC, networking with TCP/IP, etc.
 - can add new functionality by writing modules in C/C++
 - portable
 - runs on Unix/Linux, Mac, Windows, etc.
 - see www.python.org

Scripting Languages

- “Scripts” vs “Programs”
 - interpreted vs compiled
 - one script == a program
 - many {*.c,*.h} files == a program
- shell scripts vs C programs
 - scripts that execute OS programs (in C)
 - e.g., sh, csh, tcsh, bash
- Higher-level “glue” language
 - glue together larger program/library components
 - orchestrate larger-grained computations
 - versus programming fine-grained computations

The Python Interpreter

Running the interpreter

```
shell$ python
Python 2.3.5 (#1, Aug 19 2006, 00:49:11)
[GCC 4.0.1 (Apple Computer, Inc. build 5363)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> license()
A. HISTORY OF THE SOFTWARE
=====
```

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see <http://www.zope.com>). In 2001, the Python Software Foundation (PSF, see <http://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

. . .

The Python Interpreter

Python Help

```
>>> help()
```

```
Welcome to Python 2.3! This is the online help utility.
```

```
If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://www.python.org/doc/tut/.
```

```
Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".
```

```
To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".
```

```
help>
```

The Python Interpreter

Python Keywords

```
help> keywords
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.
```

```
and          else          import        raise
assert       except        in            return
break        exec          is            try
class        finally      lambda        while
continue     for           not           yield
def          from          or
del          global       pass
elif        if           print
```

The Python Interpreter

Python Modules

```
help> modules
```

Too many to list, but some important ones are:

```
compiler  
exceptions  
gc  
string  
sys
```

The Python Interpreter

Hello world

```
>>> print "Hello, world"
Hello, world
>>> print 'Hello, world'
Hello, world
>>> "Hello, world"
'Hello, world'
>>> 'Hello, world'
'Hello, world'
```

The Python Interpreter

Recursion and integer precision

```
>>> def fact(n):
...     if n==0:
...         return 1
...     else:
...         return (n*fact(n-1))
...
>>> fact(5)
120
>>> fact(100)
933262154439441526816992388562667004907159682643816214685929638952175
999932299156089414639761565182862536979208272237582511852109168640000
00000000000000000000L
>>> fact(1000)
...
RuntimeError: maximum recursion depth exceeded
>>> import sys
>>> sys.getrecursionlimit
1000
```

The Python Interpreter

```
>>> sys.setrecursionlimit(1002)
>>> fact(1000)
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208
486
969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627
727
188732519779505950995276120874975462497043601418278094646496291056393887437886487337119181
045
825783647849977012476632889835955735432513185323958463075557409114262417474349347553428646
576611667797396668820291207379143853719588249808126867838374559731746136085379534524221586
593201928090878297308431392844403281231558611036976801357304216168747609675871348312025478
589320767169132448426236131412508780208000261683151027341827977704784635868170164365024153
691398281264810213092761244896359928705114964975419909342221566832572080821333186116811553
615836546984046708975602900950537616475847728421889679646244945160765353408198901385442487
98495995331910172335556602139450399736280750137837615307127761926849034352625200015888535
147331611702103968175921510907788019393178114194545257223865541461062892187960223838971476
088506276862967146674697562911234082439208160153780889893964518263243671616762179168909779
911903754031274622289988005195444414282012187361745992642956581746628302955570299024324153
181617210465832036786906117260158783520751516284225540265170483304226143974286933061690897
968482590125458327168226458066526769958652682272807075781391858178889652208164348344825993
266043367660176999612831860788386150279465955131156552036093988180612138558600301435694527
224206344631797460594682573103790084024432438465657245014402821885252470935190620929023136
493273497565513958720559654228749774011413346962715422845862377387538230483865688976461927
383814900140767310446640259899490222221765904339901886018566526485061799702356193897017860
040811889729918311021171229845901641921068884387121855646124960798722908519296819372388642
614839657382291123125024186649353143970137428531926649875337218940694281434118520158014123
344828015051399694290153483077644569099073152433278288269864602789864321139083506217095002
59738986355427719674282248757586765752344220207573630569498825087968928162753848863396909
959826280956121450994871701244516461260379029309120889086942028510640182154399457156805941
872748998094254742173582401063677404595741785160829230135358081840096996372524230560855903
```

The Python Interpreter

What about tail recursion?

```
>>> def fact(n,m=1):                # note the use of a default
...     if n==0:                    argument
...         return m
...     else:
...         return fact(n-1,m*n)
...
>>> fact(1001)
...
RuntimeError: maximum recursion depth exceeded
```

So the interpreter does not recognize tail recursion and does not perform automatic tail call optimization, as in Scheme and other languages!

Built-in Types

- **Numbers**
 - Integers: C "long" and arbitrary precision
 - decimal, decimal long, octal and hex literals
 - Boolean: True, False and bool(x) -> bool
 - Floating point: C "double"
 - Complex numbers
- **Sequences**
 - String, List, Tuple
- Dictionary (association list)
- File

Arithmetic Operators

```
>>> help("int")
```

```
int(x [, base]) -> integer
|
| x.__abs__() <==> abs(x)
| x.__add__(y) <==> x+y
| x.__and__(y) <==> x&y
| x.__cmp__(y) <==> cmp(x,y)
| x.__coerce__(y) <==> coerce(x, y)
| x.__div__(y) <==> x/y
| x.__divmod__(y) <==> xdivmod(x, y)y
| x.__float__() <==> float(x)
| x.__floordiv__(y) <==> x//y
| x.__getattr__('name') <==> x.name
| x.__hash__() <==> hash(x)
| x.__hex__() <==> hex(x)
| x.__int__() <==> int(x)
| x.__invert__() <==> ~x
| x.__long__() <==> long(x)
| x.__lshift__(y) <==> x<<y
| x.__mod__(y) <==> x%y
| x.__mul__(y) <==> x*y
| x.__neg__() <==> -x
| x.__nonzero__() <==> x != 0
| x.__oct__() <==> oct(x)
| x.__or__(y) <==> x|y
| x.__pos__() <==> +x
| x.__pow__(y[, z]) <==> pow(x, y[, z])
```

Arithmetic Operators (cont.)

```
| x.__radd__(y) <==> y+x  
| x.__rand__(y) <==> y&x  
| x.__rdiv__(y) <==> y/x  
| x.__rdivmod__(y) <==> ydivmod(y, x)x  
| x.__repr__() <==> repr(x)  
| x.__rfloordiv__(y) <==> y//x  
| x.__rlshift__(y) <==> y<<x  
| x.__rmod__(y) <==> y%x  
| x.__rmul__(y) <==> y*x  
| x.__ror__(y) <==> y|x  
| y.__rpow__(x[, z]) <==> pow(x, y[, z])  
| x.__rrshift__(y) <==> y>>x  
| x.__rshift__(y) <==> x>>y  
| x.__rsub__(y) <==> y-x  
| x.__rtruediv__(y) <==> y/x  
| x.__rxor__(y) <==> y^x  
| x.__str__() <==> str(x)  
| x.__sub__(y) <==> x-y  
| x.__truediv__(y) <==> x/y  
| x.__xor__(y) <==> x^y
```

Booleans

```
>>> bool
<type 'bool'>
>>> True, False
(True, False)
>>> bool(0), bool(1), bool(-1)
(False, True, True)
>>> bool(3.14), bool("hello, world")
(True, True)
>>> True & True, True & False, False & False
(True, False, False)
>>> True | True, True | False, False | False
(True, True, False)
>>> True ^ True, True ^ False, False ^ False
(False, True, False)
>>> str(True), repr(True)
('True', 'True')
>>> print False
False
>>> print True
True
```

Floating Point

```
>>> float
<type 'float'>
>>> 3.14
3.1400000000000001
>>> print 3.14
3.14
>>> repr(3.14)
'3.1400000000000001'
>>> str(3.14)
'3.14'
>>> 3.14//2, 3.14/2
(1.0, 1.5700000000000001)
>>> import math
>>>> math.pi, math.e
(3.1415926535897931, 2.7182818284590451)
>>> help ("math")    # check it out to see what functions are available
```

Strings

Ordered immutable sequence of characters

```
>>> help ("str") # check out help to learn more
>>> str
<type 'str'>
>>> str.upper("abcdefghijklmnopqrstuvwxyz")
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> "" # empty string
''
>>> print ""
'
>>> print '''
"
>>> "a %s parrot" % 'dead' # inline string formatting
'a dead parrot'
>>> "monty " + 'python'
'monty python'
>>> 'm' in 'monty python'
True
>>> 'm' in "Monty Python"
False
>>> s="monty python"
>>> len(s), s.find('py'), s.split()
(12, 6, ['monty', 'python'])
>>> s[0],s[1],s[2:3],s[4:]
('m', 'o', 'n', 'y python')
>>> s[6:99]
'python'
```

Lists

```
>>> help ("list") # check out help to learn more
>>> [] #empty list
[]
>>> x = [1, "this", 3.14]
>>> x
[1, 'this', 3.1400000000000001]
>>> len(x)
3
>>> for i in x: # iterate over the list x
...     print i
...
1
this
3.14
>>> x+[] == []+x
True
>>> x + ['a', 2, []]
[1, 'this', 3.1400000000000001, 'a', 2, []]
>>> x
[1, 'this', 3.1400000000000001]
>>> x.append(5)
>>> x.reverse()
>>> x
>>> [5, 3.1400000000000001, 'this', 1]
>>> x*2
[5, 3.1400000000000001, 'this', 1, 5, 3.1400000000000001, 'this', 1]
>>> x.sort()
>>> x
[1, 3.1400000000000001, 5, 'this']
```

Tuples

```
>>> help ("tuple") # check out help to learn more
>>> () #empty tuple
()
>>> x = (1, 'a', 'bcd')
>>> len(x)
3
>>> 'a' in x
True
>>> x[0],x[1],x[:-1]
(1, 'a', (1, 'a'))
>>> (1,2) + (3,4)
(1, 2, 3, 4)
>>> (1,2)*2
(1, 2, 1, 2)
>>> y = (20) # not a 1 item tuple!
>>> y
20
>>> y = (20,)
>>> y
(20,)
>>> y=(20,15,9,1,-5)
>>> tmp = list(y) # convert tuple to a list
>>> tmp.sort()
>>> y=tuple(tmp) # convert sorted list back to a tuple
>>> y
(-5, 1, 9, 15, 20)
```

Files

```
>>> help ("file") # check out help to learn more
>>> output = open('/tmp/test', 'w') # open tmp file for writing
>>> input = open('/etc/passwd', 'r') # open Unix passwd file for reading
>>> S = input.read() # read entire file into string s
>>> S = input.readline() # read next line
>>> L = input.readlines() # read entire file into list of line strings
>>> output.write(S) # write string S
>>> output.write(L) # write all lines in L
>>> output.close()
>>> input.close()
```

Statements & Expressions

- Assignment
- Blocks
- Control Flow
 - Conditional, loops
- Functions
 - Lambda expressions

Assignment Forms

- Basic form
 - `a = 1`
 - `b = 'spam'`
- Tuple positional assignment
 - `a, b = 2, 'ham'`
 - `a == 2, b == 'ham' => (True, True)`
- List positional assignment
 - `[a,b] = [3,'eggs']`
 - `a==3, b=='eggs' => (True, True)`
- Multi-assignment
 - `a = b = 10`

Compound Statements

- Python does not use 'begin-end' or '{...}' to denote a block
 - You must remember to indent statements at the same level positionally to place them into the same block
 - This is probably the most annoying aspect of python until you get used to it
 - Example:

```
if n == 0:  
.....return 1  
else:  
.....return n*fact(n-1)
```

Conditional Statement

- `if <test1>:`
 `<statement-1 block>`
`elif <test2>:`
 `<statement-2 block>`
...
`else:`
 `<statement-n block>`

Conditional Statement

```
>>> if 1:
...     print True
      File "<stdin>", line 2
        print True
          ^
IndentationError: expected an indented block
>>> if 1:
...     print True
...
True
>>> choice='ham'
>>> if choice=='spam':
...     print 1
... elif choice=='ham':
...     print 2
... elif choice=='eggs':
...     print 3
... else:
...     print "None"
...
2
```

Conditional Statement

There is no C-like 'switch' statement in Python, but you can use a dictionary as a way to encode a set of choices and then use a "key" to select one of the elements

```
>>>choice='ham'  
>>>dict = {'spam' : 1, 'ham' : 2, 'eggs' : 3 }  
>>> dict  
{'eggs': 3, 'ham': 2, 'spam': 1}  
>>> print dict [choice]  
2  
>>>
```

While Loops

```
>>> x='spam'  
>>> while x:  
...     print x,  
...     x=x[1:]  
...  
spam pam am m
```

```
>>> a=0; b=10  
>>> while a < b:  
...     print a,  
...     a += 1  
...  
0 1 2 3 4 5 6 7 8 9
```

For Loops

```
>>> for x in ['spam', 'ham', 'eggs']:
...     print x,
...
spam ham eggs
>>> for x in [1,2,3,4]:
...     print x
...
1
2
3
4
>>> sum=0
>>> for x in [1,2,3,4,5]:
...     sum += x
...
>>> sum
15
>>> T=[(1,'a'), (2,'b'), (3,'c')]
>>> for (a,b) in T:
...     print a, b
...
1 a
2 b
3 c
```

Functions

```
>>> def mul(x,y): return x * y
...
>>> mul(3,4)
12
>>> mul(math.pi,2.0)
6.2831853071795862
>>> mul([1,2],2)
[1, 2, 1, 2]
>>> mul(('a','b'),3)
('a', 'b', 'a', 'b', 'a', 'b')
>>> mul("boo", 5)
'booboobooboobo'
>>> def fact(n):
...     m=1
...     while n > 0: m*=n; n-=1
...     return m
...
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229
9156089414639761565182862536979208272237582511852109168640000000000000000000
0000L
```

Flexible Function Arguments

```
>>> def f(a,b,c): print a, b, c
...
>>> f(1,2,3)
1 2 3
>>> f(a=2,b=4,c=12)
2 4 12
>>> f(c=12,a=2,b=4)
2 4 12
>>> def f(*args): print args
...
>>> f("this is an argument")
('this is an argument',)
>>> f(1,2,3,4,5)
(1, 2, 3, 4, 5)
>>> def f(*args):
...     while args:
...         print args,
...         args=args[1:]
...
>>> f(1,2,3,4,5)
(1, 2, 3, 4, 5) (2, 3, 4, 5) (3, 4, 5) (4, 5) (5,)
>>> def f(**args): print args
...
>>> f(a='spam',b=1)
{'a': 'spam', 'b': 1}
>>> f(a='spam',b=1,c='ham',d=2,e='eggs',f=3)
{'a': 'spam', 'c': 'ham', 'b': 1, 'e': 'eggs', 'd': 2, 'f': 3}
```

Lambda Expressions (anonymous functions)

```
>>> mul = lambda x, y: x * y
>>> mul
<function <lambda> at 0x4788b0>
>>> apply(mul, (2,3))
6
>>> map (lambda x:x*x, [1,2,3,4,5])
[1, 4, 9, 16, 25]
>>> filter (lambda x:x>0, [-5,-4,-3,-2,-1,0,1,2,3,4,5])
[1, 2, 3, 4, 5]
>>> reduce (lambda x,y: x+y, [1,2,3,4,5])
15
>>> reduce (lambda x,y:x*y, [1,2,3,4,5])
120
>>> range(1,5)
[1,2,3,4]
>>> def fact(n): return reduce (lambda x,y:x*y, range(1,n+1))
...
>>> fact(5)
120
>>> fact(100)
93326215443944152681699238856266700490715968264381621468592963895217599993229
9156089414639761565182862536979208272237582511852109168640000000000000000000
0000L
```