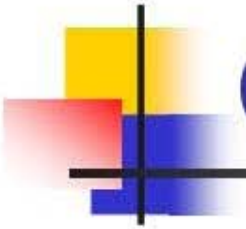


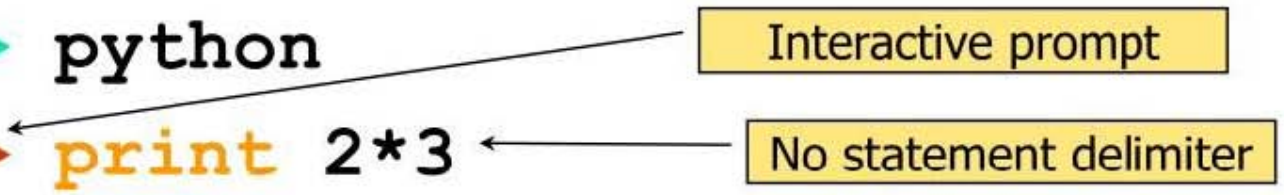
Getting Python

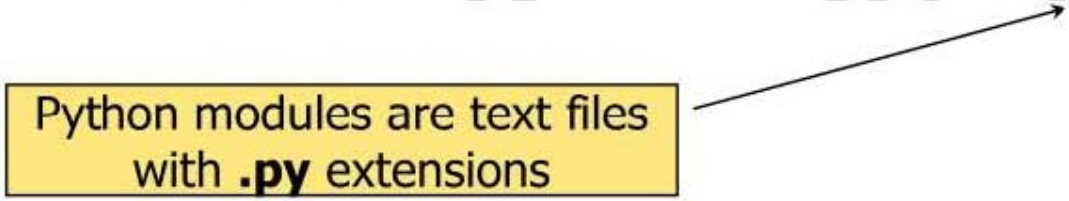
- On the Web:

www.python.org

Running Python (1)

- Interactively from console:
  - `C:> python`
  - `>>> print 2*3`
  - *6*
- As Python module files:
  - `C:> python mypgm.py`

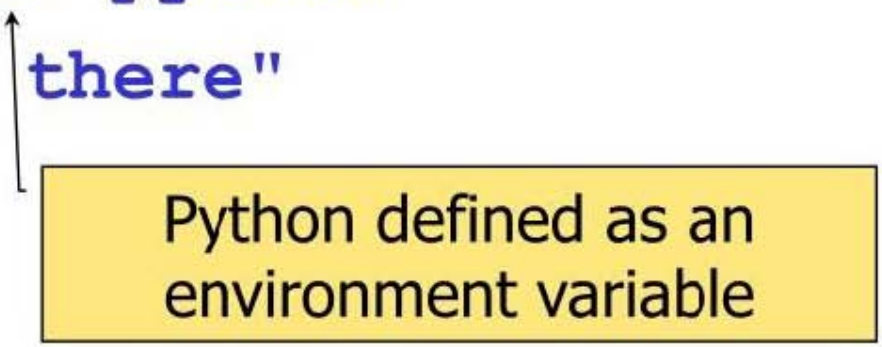


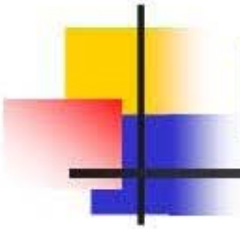


Running Python (2)

- From platform specific shells
  - `#!/usr/local/bin/python`
  - `print "Hello there"`

  Or
  - `#!/usr/bin/env python`
  - `print "Hello there"`





Running Python (3)

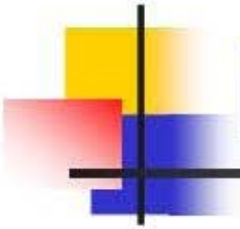
- Embedded in another system
  - `#include <Python.h>`
  - `// . . .`
  - `Py_Initialize();`
  - `PyRun_SimpleString("x=pfx+root+sfx");`
  - `// . . .`
- Platform-specific invocation
  - E.g., Double clicking `.py` files

Simple examples

- Built-in and explicit print
  - `>>> "Hello all"`
  - `'Hello all'`
  - `>>> print "A b"`
  - `A b`
  - `>>> ALongName = 177 / 3`
  - `>>> ALongName`
  - `59`

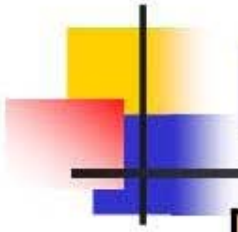
Builtin print gives double quotes as single quotes. " and ' quotes are same.

print statement removes quotes



Python Principles

- Python treats everything as an object
- Python is an interpreter
  - It gives immediate results
  - It generates byte code (similar to Java)



Built-in Object Types

Can be indexed/sliced?

Can be changed in place?

Type	Ordered	Mutable	Examples
Numbers	N/A	No	3.14, 123, 99L, 1+-2j, 071, 0x0a
Strings	Yes	No	'A string', "A double 'ed string"
Lists	Yes	Yes	[1, [2, 'three'], [5E-1, 10e3], -8L]
Dictionaries	No	Yes	{'hang':'man', 'fur':'ball'}
Tuples	Yes	No	(1, 'two', -3j, 04, 0x55, 6L)
Files	N/A	N/A	text = open('ham','r').read()

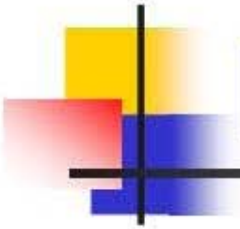
Operator Precedence

	Operators	Description
Low	x **or** y, **lambda** arguments**:** expression	Logical OR (y evaluated only if x false), anonymous function
	x **and** y	Logical AND (y evaluated only if x is true)
	not x	Logical negation
	<, <=, >, >=, ==, <>, !=, is, is not, in, not in	Comparison operators, identity tests, sequence membership
	x **|** y	Bitwise OR
	x **^** y	Bitwise EXCLUSIVE OR
	x **&** y	Bitwise AND
	x **<<** n, x **>>** n	Shift x left or right by n bits
	x **+** y, x **−** y	Numeric addition or sequence concatenation, subtraction
	x ***** y, x **/** y, x **%** y	Multiplication or sequence repetition, division, modulus
	-x, **+**x, **~**x	Unary negation, identity, bitwise negation
	x**[**i**]**, x**[**i**:**j**]**, x**.**y, x**(...)**	Indexing and slicing sequences, qualification, function call
High	**(...), [...], {...}**, **`...`**	Tuple, List, Dictionary, conversion to string

Basic Operations (1)

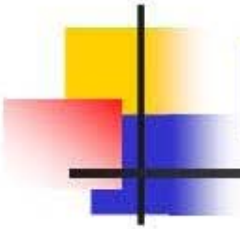
- Assignment creates names
  - `s = 'A string' # s is created`
- Names can be any length
- Names are case sensitive
  - `>>> A = 1; a = 2; A+a`
  - *3*

Semicolons separates statements on the same line



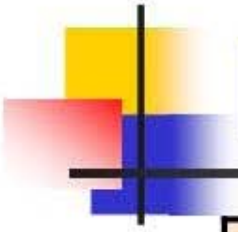
Basic Operations (2)

- Mixing numeric types promotes operands to most inclusive type
  - `>>> 1/2.0 # same as 1.0/2.0`
  - `0.5`



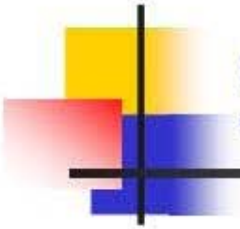
Basic Operations (3)

- Boolean True is non-zero, non-NULL, non-empty
  - `>>> "a"=='a', (1,2)==(1,2), [3]`
  - *`(1, 1, [3])`*
- Boolean False = not True
  - `>>> "a"!='a', (2)!=(2), not [3]`
  - *`(0, 0, 0)`*



Basic Numeric Operations

Expression	Result	Description
1 / 2.0	1.0 / 2.0 = 0.5	Mixing types promotes operands to most inclusive type.
x = 1 x << 2, x | 2	1 (4, 3)	Assigns built-in long variable x value 1 Bit shifts left 2 bits, Bitwise OR
99999999+1 99999999L+1	Overflow error 100000000	Integer value too large for long Long values can be any size
2 + -5j, 1j * 1J 2 + 3j * 2 (2+3j) * 3	((2-5j), (-1+0j)) (2+6j) (6+9j)	Complex numbers



Strings

- Sequence of immutable characters (characters can't be changed in-place).

`'a', "b"`	`('a', 'b')`
`"""Spans two lines"""`	`'Spans two\012lines'`
`'a' * 3 + 'b'`	`'aaab'`
`('a' + 'b') * 3`	`'ababab'`

String Operations

Range includes lower bound and excludes upper bound

`'abc'[2]`	*'c'*	*Index (zero based)*
`'abc'[1:]`	*'bc'*	*Slice to end*
`'abc'[:-1]`	*'ab'*	*Slice from start*
`'abc'[1:2]`	*'b'*	*Slice in middle*
`len('abc')`	*3*	*Length*
`for i in 'abc':` `  print i,`	*a b c*	*Iteration*
`'b' in 'abc'`	*1*	*Membership*

Suppress new line on output

String Formatting

Adjacent strings are concatenated, like in C

- Like C's printf with similar specifiers
  - `"It's " '%d great life!' % 1`
  - *`"It's 1 great life!"`*
  - `'%s %s much' % ("Python's", 2)`
  - *`"Python's 2 much"`*
- C's backslash conventions used
- Raw strings take backslashes literally
  - `print "a\tc"  # outputs a  c`
  - `print R"a\tc" # outputs a\tc`

Lists (1)

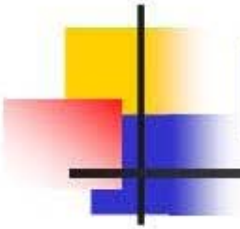
Concatenation of similar object types

Append is only way of growing list

Only way of deleting an element

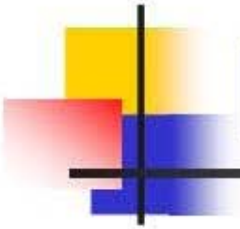
- Sequence of mutable heterogeneous objects (items can be changed in-place).

`[1, "a", [3, 4]]`	*`[1, 'a', [3, 4]]`*
`[1, 2, 3][1:2]`	*`[2]`*
`[1] + list('ab' + `76`)`	*`[1, 'a', 'b', '7', '6']`*
`L = [1, 2, 3]; L[1] = 5; L`	*`[1, 5, 3]`*
`L = [1, 2, 3]; del L[1]; L`	*`[1, 3]`*
`L.append(7); L`	*`[1, 3, 7]`*



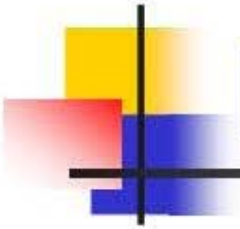
Lists (2)

- List methods work on lists, **not** copies
- Built-in operations work on copies
  - `>>> L = [1, 3]; L.append('a'); L`
  - `[1, 3, 'a']`
  - `>>> L + ['b'] # copies to new list`
  - `[1, 3, 'a', 'b']`
  - `>>> L`
  - `[1, 3, 'a']`



Lists (3)

- Shared references
  - `>>> X = [1 ,2, 3]`
  - `>>> L = ['a', X, 'c']; L`
  - `['a', [1, 2, 3], 'c']`
  - `>>> X[1] = -9; X, L`
  - `([1, -9, 3], ['a', [1, -9, 3], 'c'])`
  - `>>> M = X[:]  # make copy of X`
  - `>>> X[0] = 'c'; X, M`
  - `(['c', 2, 3], [1, 2, 3])`



Only way of deleting an element

Dictionaries

- Mapping of unordered immutable keys to mutable heterogeneous objects.

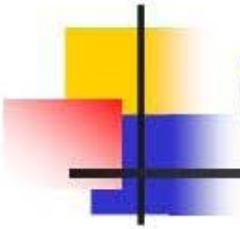
`D={'a':1,'b':[2,3]}`	`{'b': [2, 3], 'a': 1}`
`D['a'], D['b']`	`(1, [2, 3])`
`D.keys(), len(D)`	`(['b', 'a'], 2)`
`D.has_key('a')`	`1`
`D['c']=list('xy'); D`	`{'b':[2,3],c:['x','y'],'a':1}`
`D.values()`	`[[2, 3], ['x', 'y'], 1]`
`del D['b']; D`	`{'c': ['x', 'y'], 'a': 1}`

Tuples

Used to distinguish tuple from expression

- Sequence of ordered immutable heterogeneous objects.
- Can **not** change number of elements in tuple.

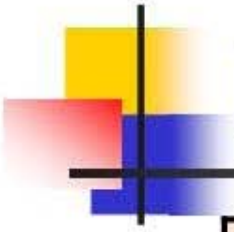
`t = ('a',{'b': 2});t`	`('a', {'b': 2})`
`t[1]['b'], len(t)+1`	`(2, 3)`
`tuple(t[0]) + t`	`('a', 'a', {'b': 2})`
`u = ('c',), u`	`('c')`
`for i in t: print i,`	`a {'b': 2}`



Comparisons, Equality

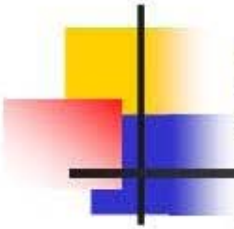
- In comparisons, Python automatically traverses data structures checking all objects
- Equivalence (==) tests value equality
- Identity (**is**) compares objects addresses

`Non-null sequences: 'ab', [3], {'a':3}, (2,)`	*True*
`Null sequences: "", [], {}, ()`	*False*
`Non-zero numeric: 1`	*True*
`Zero numeric: 0.0, 0x00`	*False*
`None`	*False*



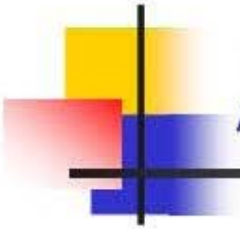
Reserved Words

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



Statements

Statements normally go to the end of line	`a = "xxx" #comment`
Statements can be continued across lines if:	
There is an open syntactic unit: (), [], {}	`a = [1, # comment1` `  2] # comment2`
The statement line ends in a backslash	`b = 'a' \` `  'b'`
The statement contains part of a triple quote (literal includes new line char (\n))	`c = """This is` `a triple quote"""`
Multiple statements separated by semicolons (;) on same line	`d = "abc"; print d`



Assignment Statement

- Defines variables names referring to objects
- Forms RHS tuples and assigns pair-wise to LHS
- Implicit assignments: `import`, `from`, `def`, `class`, `for`, `function`, `argument`, etc.

`a = "Normal assign"; a`	`'Normal assign'`
`[a, b] = [1, 2]; a, b`	`(1, 2)`
`[a, b] = [b, a]; a, b`	`(2, 1)`
`a = b = "men"; b = "mice"; a, b`	`('men', 'mice')`
`for c in "abc": print c,`	`a b c`

IF Statement

Required after conditional and **else**

- General form example:

```
if 'a' <= c <= 'z':
   print 'Lower case letter'
elif 'A' <= c <= 'Z': # optional
   print 'Upper case letter'
else:                 # optional
   print 'Not a letter'
```

Blocks (a.k.a. Suites)

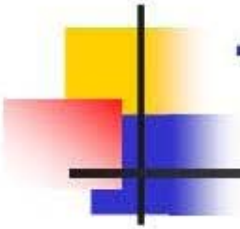
Required after conditional and **else**

All statements indented the same amount are members of the same block (or suite), until another less indented statement ends the block (or suite).

```
if 'a' <= c <= 'z':
    print 'Lower case letter'
    if d[c] == '':
        print "Not in dictionary"
    else: print "Found it" # OK for one stmt
else:
    print "Could not check"
```

Suite 1

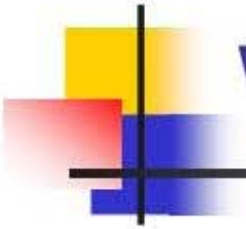
Suite 2



Truth Tests

- Comparisons and equality return 1 or 0.
- Boolean **and** and **or** use "short circuit" logic to return true or false objects
- In boolean **and** expressions, first false is returned or last true value in expression.
- In boolean **or** expressions, first true is returned or last false value in expression.

`2 > 32, 4 < 6, 31 == 31`	*`(0, 1, 1)`*
`3 and 4, [3, 4] and []`	*`(4, [])`*
`[] and {}`	*`[]`*
`(3 < 2) or (0,), [] or {}`	*`((0,), {})`*



WHILE Statement

- General format:

```
while <test> :    # loop conditional
  <stmt-block1>   # loop body
else :            # optional - run
  <stmt-block2>   #  if no break used
```

```
a = 0; b = 5
while a < b :
  print a,        # outputs 0 1 2 3 4
  a = a + 1
```

BREAK, CONTINUE, PASS (1)

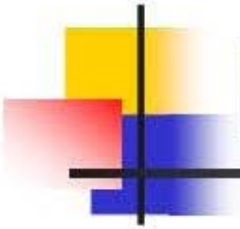
- **break** terminates the innermost executing loop and transfer control after the loop.
- **continue** immediately transfers control to the top of the innermost executing loop.
- **pass** is the no-op statement in Python.

```
while <test0> : # loop header
   <stmts1>       # run if test0 true
   if <test1> : break    # exit, skip else
   if <test2> : continue # go to loop header
   <stmts2>       # not run if test2 true
else :
   <stmts3>       # run if didn't hit break
```

BREAK, CONTINUE, PASS (2)

- Examples of **break** and **continue** in **while**

```
y = 2
while y < 10 :
  y = y + 1
  if y % 2 == 0 : continue # only do odd #s
  x = y / 2
  while x > 1 :
    if y % x == 0 :
      print "%d has factor %d" % (y, x)
      break   # causes else to be skipped
    x = x - 1
  else :
    print "%d is prime" % y
```



FOR Statement

- General format:

```
for <target> in <object> : # loop header
  <stmt-block1> # loop body
else :                  # optional, run else clause
  <stmt-block2> #  if no break used
```

```
sum = 0
for x in [1, 2, 3, 5] : sum = sum + x
sum                   # outputs 11
```

BREAK, CONTINUE, PASS (3)

- Examples of **break** and **continue** in **for**.

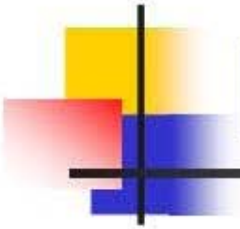
```
S = [(1, 2), (3, 4), (7, 8)]
for (x, y) in S :
  print [x, y], # outputs [1, 2] [3, 4] [7, 8]
L = ['Tom', 'Tina', 'Sam']
M = ['Mary', 'Tina', 'Tom']
for x in M :
  for y in L :
    if x == y :
      print "%s found" % x
      break
  else :
    print "%s is not in %s" % (y, M)
```

RANGE Function

- General formats, all returning a list:

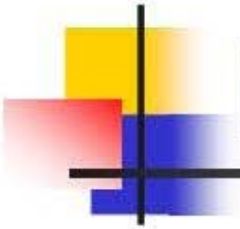
```
range(hi)                  # 0 to hi-1
range(lo, hi)              # lo to hi-1
range(lo, hi , incr)       # lo to hi-1 by incr
```

```
>>> range(3), range(2,5), range(0,5,2)
([0, 1, 2], [2, 3, 4], [0, 2, 4])
>>> for I in range(1,5): print I,
1 2 3 4
```



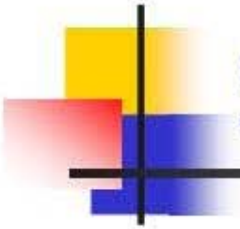
Named Functions

- General format:
  - `def name(arg0, … , argN) : # header`
  - `    <statements>      # optional body`
  - `    return <object> # optional return`
- **def** is an executable statement that creates a function object and assigns *name* to it.
- Arguments are passed by reference, not value. (i.e., as with assignment)
- Arguments, return values, and variables are not declared.



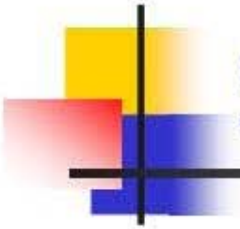
Named Function Example

- Get intersection of a set of sequences
  - `def intersect(seq1, seq2) :`
  - `  res = []`
  - `  for x in seq1 :`
  - `  if x in seq2 :`
  - `  res.append(x)`
  - `  return res`
  - `>>> intersect("Summer's", 'Blues')`
  - `['u', 'e', 's']`



Scope Rules

- The enclosing module is the global scope.
- Each function call defines a new local scope.
- Assigned names are local unless declared **global**. All other names are global or built-in.
- **LGB** rule – **L**ocal, **G**lobal, **B**uilt-in:
  - Names are looked up first in the local function, then the global (i.e., module) scope, and then in the list of Built-in names.
  - For name lookup purposes, enclosing function names and the function's own name are *ignored*.



Scope Rules Example (1)

- The following will not run successfully because of the name lookup error.

```
def outer(n) :
   def inner(n) :
      if n > 1 :
         return n * inner(n-1) # err - does not
      else:                    #  know own name
         return 1
   return inner(n)
```

Scope Rules Example (2)

- The following quick and dirty fix works, but what is wrong with it?

```
def outer(n) :
  global inner    # put name in global scope
  def inner(n) :
    if n > 1 :
      return n * inner(n-1) # finds name by
    else:                   #  LGB rule
      return 1
  return inner(n)
```

GLOBAL Statement

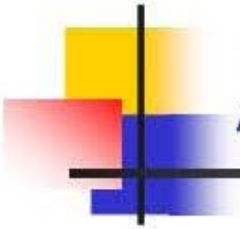
- Global names must be declared only if they are assigned in a function. This does not apply to sub-objects.
- Global names may be referenced without being declared.

```
A = [1, 2]; B = []
C = {'Ann':'M'}
def F(X) :
  print "Before: X=%s C=%s" % (X, C)
  X.append(A)
  C['Ann'] = 'F' # allowed to change sub-object
  global C       # needed to change global C
  C = {}         # illegal without global stmt
  print "After:  X=%s C=%s" % (X, C)
F(B)             # changes B to [1, 2]
```

RETURN Statement

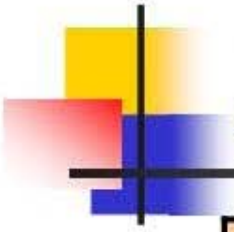
- **return** statements can return any type of object.

```
def wow(x, y) :
    x = 2 * x
    y = 3 * y
    return x, y
X = ['Hi']
Y = ('a')
A, B = wow(X, Y)
>>> A, B
(['Hi', 'Hi'], 'aaa')
```



Argument Matching (1)

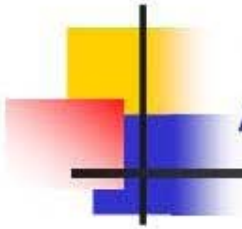
- Python supports the following types of argument matching:
  - Positional – normal left to right matching
  - Keywords – matched by argument name
  - Varargs – what remains after positional and keyword arguments matched
  - Defaults – specified values for missing arguments



Argument Matching Forms

Form	Where	Description
F(val)	Caller	Matched by position.
F(name=val)	Caller	Matched by name.
def F(name) :	Definition	Position parameter.
def F(name=val) :	Definition	Default value for named parameter, if parameter not used by caller.
def F(*name) :	Definition	Matches remaining positional parameters by forming a tuple. Must appear after all positional parameters.
def F(**name) :	Definition	Matches remaining keyword parameters by forming a dictionary. Must appear after all positional parameters and *name parameter, if any.

Argument Matching Example

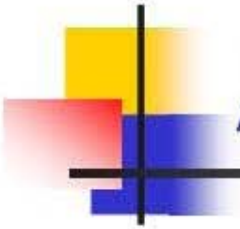


```
def w(p1='defval1', p2='defval2', *pa, **na):
  print [p1, p2, pa, na]
>>> w(5, unknown=4)
[5, 'defval2', (), {'unknown': 4}]
>>> w(5, 6, 7, unknown=4)
[5, 6, (7,), {'unknown': 4}]
```

- Note: Positional arguments must appear before keyword arguments in call to function. Thus, the following is illegal:
  - `>>> w(unknown='a', 5)`

LAMBDA Expressions

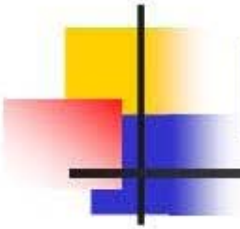
- **lambda** expressions define anonymous functions.
- They can appear anywhere an expression can appear, unlike statements that are limited.
- They return a value.
- They have the form:
  - `lambda arg1, arg2, … , argN : <expression>`
- Example:
  - `>>> F = lambda a1=3, a2=4 : a1 * a2`
  - `>>> F(3) # keyword & default args allowed`
  - `12`



APPLY Built-in

- The **apply** function allows arbitrary functions to be invoked with equally arbitrary arguments.
- **apply** has the form:
  - `apply(fcn, args)`
- Example:

```
def generic(arg1, arg2=0, arg3=0) :
  if arg2 is arg3 :
    f, a = f1, (arg1, )
  else :
    f, a = f2, (arg2, arg3)
  return apply(f, a)
```

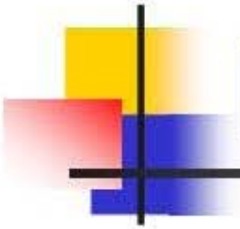


MAP Built-in

- The **map** function applies the same operation to each element in a sequence.
- **map** has the form:
  - `map(fcn, sequence)`
- Example:
  - `>>> map(lambda arg : arg / 2, (1, 2, 3))`
  - `[0, 1, 1]`

Function Gotchas

- Local names detected statically.
  - `def f():`
  - `    print B      # error - B not yet defined`
  - `    B = 2;`
- Nested functions are not nested scopes.
- Default values are saved when **def** is run, not when the function is called.



Loading Modules

- There are 3 ways to load a module:

Statement	Description
`import mymod`	Loads **mymod** module. Executes module *only the first time* it is loaded.
`from mymod import a, b`	Loads **mymod** module and creates local names **a** and **b** referencing objects with the same name inside the module **mymod**.
`Reload(mymod)`	Reload function loads module **mymod**, re-executing **mymod** each time it is reloaded.

Import Statement (1)

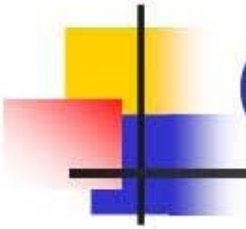
Qualified names

- Using the **import** statement:

```
>>> import sigma1
Loaded module sigma1
>>> sigma1.counter
1
>>> sigma1.Sigma([1, 2, 3])
6
>>> sigma1.counter = 2
>>> import sigma1
>>> sigma1.counter
2
```

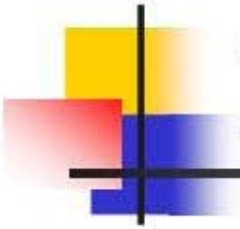
```
# sigma1.py - test module
counter = 1
def Sigma(L) :
  sum = 0
  for x in L : sum = sum + x
  return sum
print "Loaded module sigma1"
```

print **not** executed and counter **not** reset on second import



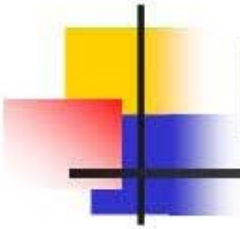
Qualified Names

- Qualified names have form: `a.b.….z`
- Qualification can be used with anything that has attributes.
- Unqualified names use the LGB rule.
- `a.b.c` means first find attribute `b` in object `a` and then find attribute `c` in `a.b`. Qualification ignores the LGB rule.



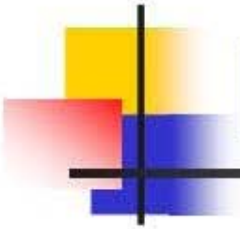
IMPORT Statement (2)

- Both **import** and **from** are forms of assignment statements
- **import** assigns a name to the module object.
  - `>>> import mymod`
  - `>>> mymod`
  - `<module 'mymod' from 'mymod.py'>`



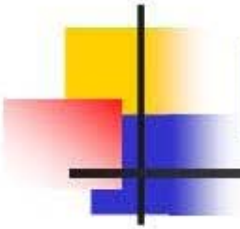
FROM Statement (1)

- Assume module ModA contains:
  - `A = 1; C = 2; D = 4;  # no B defined`
- If the following is entered:
  - `>>> A = 99; B = 98; C = 97; D = 96`
  - `>>> from ModA import A, C`
  - `>>> print A, B, C, D`
  - *`1 98 2 96`*
- A **from** imported name replaces any previously defined local variable having the same name (see variables A and C).



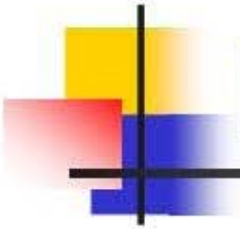
FROM Statement (2)

- **from** does not assign the module name.
- **from** is equivalent to:
  - `from mymod import name1, name2, . . .`
- Which is the same as:
  - `import mymod` `# load module and name`
  - `name1 = mymod.name1` `# copy name1 by assign`
  - `name2 = mymod.name2` `# copy name2 by assign`
  - `. . .`
  - `del mymod` `# delete module name`



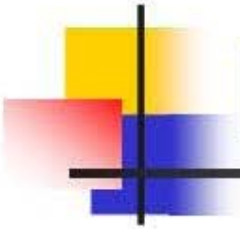
FROM Statement (3)

- **from** <module> **import** *
  - Imports **all** top level names from <module> into the current module's namespace, except names starting with an underscore (_).
  - This has grave potential for name conflicts
    - `>>> A = 99; B = 98; C = 97; D = 96`
    - `>>> from ModA import *`
    - `>>> print A, B, C, D`
    - *1 2 3 4*
    - `>>> A = 99; B = 98; C = 97`
    - `>>> import ModA`
    - `>>> print A, B, C, ModA.A, ModA.B, ModA.C`
    - *99 98 97 1 2 3*



RELOAD Function (1)

- **import** runs a module only the first time it is loaded. Subsequent **import**s of the same module uses the existing code without rerunning it.
- **reload** is a built-in function that forces an already loaded module to be reloaded and rerun. The module must already exist.
  - `import mymod`
  - `. . .`
  - `reload(mymod)`



RELOAD Function (2)

- **reload** rereads the module's source code and reruns its top-level code.
- It changes a module object in-place so all references to the module are updated.
  - **reload** runs the module file's new code in same namespace as before.
  - Top-level assignments replace existing names with new values.
  - **reload** impacts clients using imported names.
  - **reload** impacts only future use of old objects.