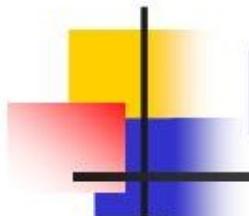


# Getting Python

---

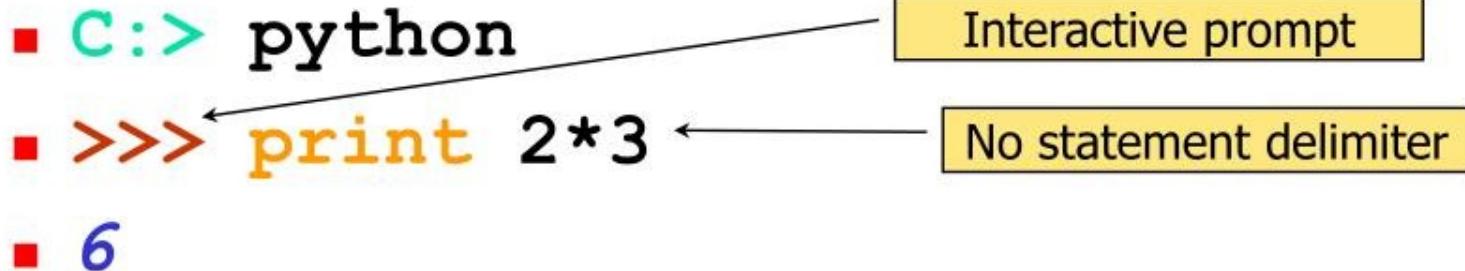
- On the Web:

**[www.python.org](http://www.python.org)**



# Running Python (1)

- Interactively from console:

- `C:> python`
  - `>>> print 2*3`
  - `6`
- 

Interactive prompt

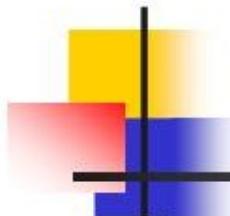
No statement delimiter

- As Python module files:

- `C:> python mypgm.py`



Python modules are text files with `.py` extensions



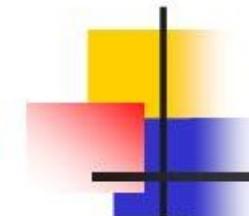
# Running Python (2)

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

Or

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

Python defined as an environment variable



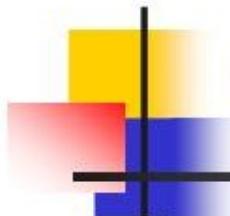
# Running Python (3)

- Embedded in another system

- `#include <Python.h>`
  - `// . . .`
  - `Py_Initialize();`
  - `PyRun_SimpleString("x=px+root+sfx");`
  - `// . . .`

- Platform-specific invocation

- E.g., Double clicking `.py` files



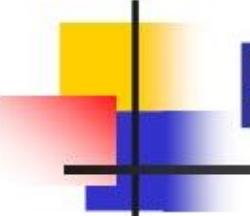
# Simple examples

- 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.

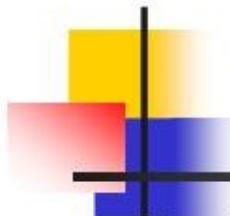
print statement removes quotes



# 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

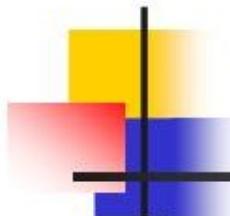
Can be indexed/sliced?

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

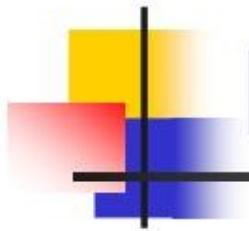
	Operators	Description
Low ↓	<code>x or y, lambda</code> arguments: expression	Logical OR (y evaluated only if x false), anonymous function
	<code>x and y</code>	Logical AND (y evaluated only if x is true)
	<code>not x</code>	Logical negation
	<code>&lt;, &lt;=, &gt;, &gt;=, ==, &lt;&gt;, !=, is, is not,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



# Basic Operations (1)

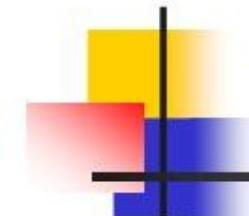
- Assignment creates names
  - `s = 'A string' # s is created`
- Names can be any length
- Names are case sensitive
  - `>>> A = 1; a = 2; A+a`
  - 3

Semicolons separates statements on the same line



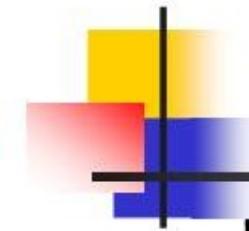
# Basic Operations (2)

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



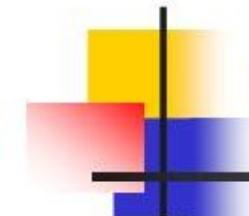
# Basic Operations (3)

- 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

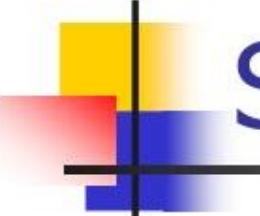
Expression	Result	Description
<code>1 / 2.0</code>	<code>1.0 / 2.0 = 0.5</code>	Mixing types promotes operands to most inclusive type.
<code>x = 1</code> <code>x &lt;&lt; 2, x   2</code>	<code>1</code> <code>(4, 3)</code>	Assigns built-in long variable x value 1 Bit shifts left 2 bits, Bitwise OR
<code>99999999+1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

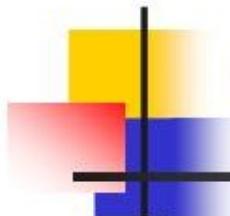


Range includes lower bound and excludes upper bound

# String Operations

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

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)

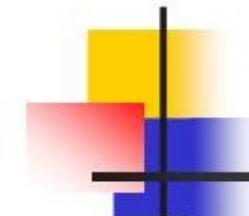
Concatenation of similar object types

Append is only way of growing list

Only way of deleting an element

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

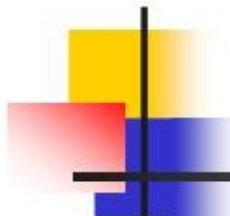
[1, "a", [3, 4]]	[1, 'a', [3, 4]]
[1, 2, 3][1:2]	[2]
[1] + list('ab' + '76')	[1, 'a', 'b', '7', '6']
L = [1, 2, 3]; L[1] = 5; L	[1, 5, 3]
L = [1, 2, 3]; del L[1]; L	[1, 3]
L.append(7); L	[1, 3, 7]



## 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.

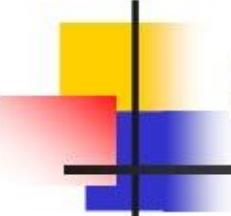
<code>D={'a':1, 'b':[2,3]}</code>	<code>{'b': [2, 3], 'a': 1}</code>
<code>D['a'], D['b']</code>	<code>(1, [2, 3])</code>
<code>D.keys(), len(D)</code>	<code>(['b', 'a'], 2)</code>
<code>D.has_key('a')</code>	<code>1</code>
<code>D['c']=list('xy'); D</code>	<code>{'b': [2, 3], c: ['x', 'y'], 'a': 1}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

# Tuples

Used to distinguish tuple from expression

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

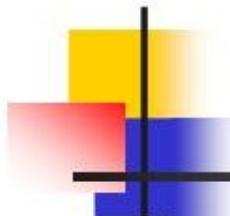
<code>t = ('a', {'b': 2}); t</code>	<code>('a', {'b': 2})</code>
<code>t[1]['b'], len(t)+1</code>	<code>(2, 3)</code>
<code>tuple(t[0]) + t</code>	<code>('a', 'a', {'b': 2})</code>
<code>u = ('c',), u</code>	<code>('c')</code>
<code>for i in t: print i,</code>	<code>a {'b': 2}</code>



# Comparisons, Equality

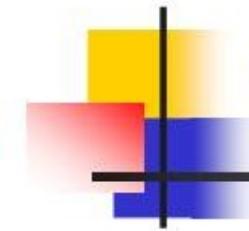
- In comparisons, Python automatically traverses data structures checking all objects
- Equivalence (`==`) tests value equality
- Identity (`is`) compares objects addresses

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



# 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

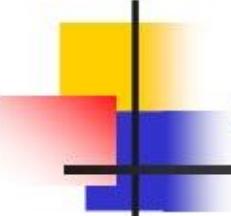
```
b = 'a' \  
    'b'
```

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

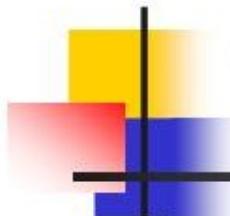
```
d = "abc"; print d
```



# 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.

<code>a = "Normal assign"; a</code>	<i>'Normal assign'</i>
<code>[a, b] = [1, 2]; a, b</code>	<i>(1, 2)</i>
<code>[a, b] = [b, a]; a, b</code>	<i>(2, 1)</i>
<code>a = b = "men"; b = "mice"; a, b</code>	<i>('men', 'mice')</i>
<code>for c in "abc": print c,</code>	<i>a b c</i>



# 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'
```

Required after  
conditional and **else**

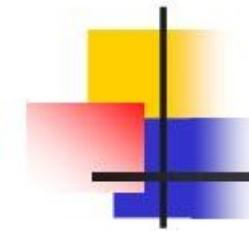
# Blocks (a.k.a. Suites)

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

Suite 1:

```
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"
```

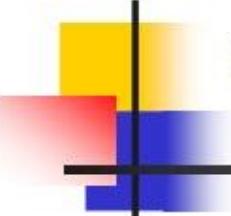
Suite 2:



# 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.

<code>2 &gt; 32, 4 &lt; 6, 31 == 31</code>	<code>(0, 1, 1)</code>
<code>3 and 4, [3, 4] and []</code>	<code>(4, [])</code>
<code>[] and {}</code>	<code>[]</code>
<code>(3 &lt; 2) or (0,), [] or {}</code>	<code>((0,), {})</code>

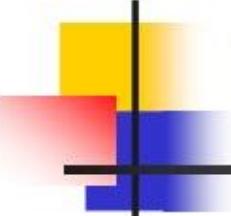


# WHILE Statement

- General format:

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

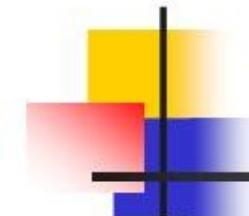
```
a = 0; b = 5  
  
while a < b :  
  
    print a,      # outputs 0 1 2 3 4  
  
    a = a + 1
```



# 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.

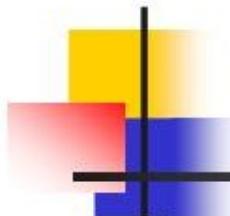
```
■ while <test0> : # loop header
  ■   <stmts1>      # run if test0 true
  ■   if <test1> : break    # exit, skip else
  ■   if <test2> : continue # go to loop header
  ■   <stmts2>      # not run if test2 true
  ■   else :
  ■     <stmts3>      # run if didn't hit break
```



# 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
```

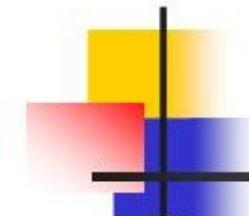


# FOR Statement

- General format:

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

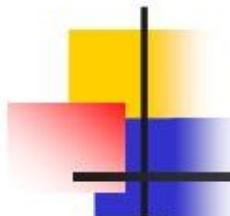
```
sum = 0  
  
for x in [1, 2, 3, 5] : sum = sum + x  
  
sum          # outputs 11
```



# 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
```

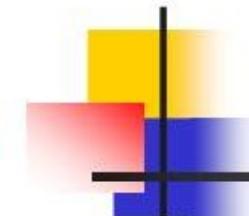
```
range(lo, hi , incr) # lo to hi-1 by incr
```

```
>>> range(3), range(2,5), range(0,5,2)
```

```
([0, 1, 2], [2, 3, 4], [0, 2, 4])
```

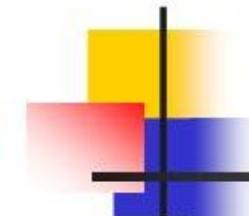
```
>>> for I in range(1,5): print I,
```

```
1 2 3 4
```



# Named Functions

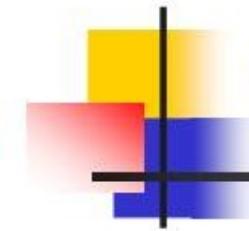
- General format:
  - `def name(arg0, ... , argN) : # header`
  - `<statements> # optional body`
  - `return <object> # optional return`
- **def** is an executable statement that creates a function object and assigns *name* to it.
- Arguments are passed by reference, not value. (i.e., as with assignment)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



# Named Function Example

- Get intersection of a set of sequences

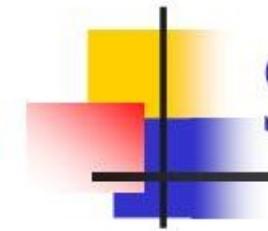
- ```
def intersect(seq1, seq2) :  
    res = []  
    for x in seq1 :  
        if x in seq2 :  
            res.append(x)  
    return res  
>>> intersect("Summer's", 'Blues')  
['u', 'e', 's']
```



# Scope Rules

---

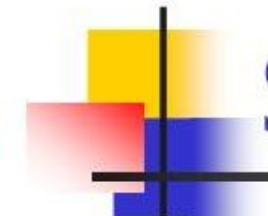
- The enclosing module is the global scope.
- Each function call defines a new local scope.
- Assigned names are local unless declared **global**. All other names are global or built-in.
- **LGB** rule – **L**ocal, **G**lobal, **B**uilt-in:
  - Names are looked up first in the local function, then the 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.

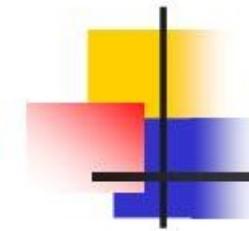
- ```
■ def outer(n) :  
■     def inner(n) :  
■         if n > 1 :  
■             return n * inner(n-1) # err - does not  
■         else:                      # know own name  
■             return 1  
■     return inner(n)
```



# Scope Rules Example (2)

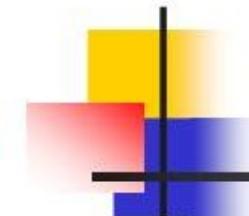
- The following quick and dirty fix works, but what is wrong with it?

- ```
■ 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

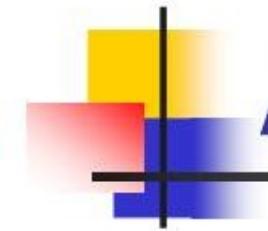
- Global names must be declared only if they are assigned in a function. This does not apply to sub-objects.
- Global names may be referenced without being declared.
  - `A = [1, 2]; B = []`
  - `C = {'Ann': 'M'}`
  - `def F(X) :`
    - `print "Before: X=%s C=%s" % (X, C)`
    - `X.append(A)`
    - `C['Ann'] = '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

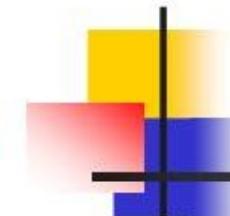
- **return** statements can return any type of object.

- `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')`



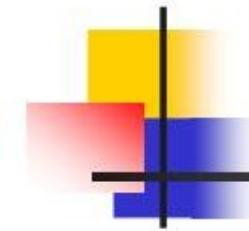
# Argument Matching (1)

- 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



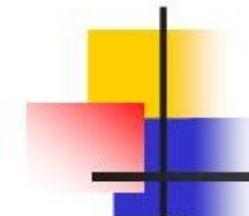
# Argument Matching Forms

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



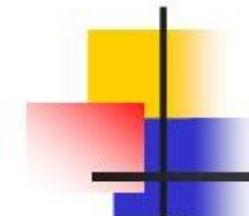
# Argument Matching Example

- `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)`



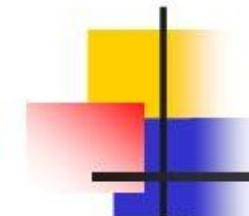
# LAMBDA Expressions

- **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

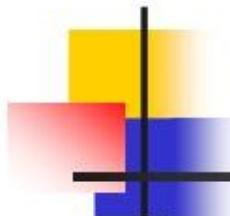
- The **apply** function allows arbitrary functions to be invoked with equally arbitrary arguments.
- **apply** has the form:
  - `apply(fcn, args)`
- Example:
  - `def generic(arg1, arg2=0, arg3=0) :`
  - `if arg2 is arg3 :`
  - `f, a = f1, (arg1, )`
  - `else :`
  - `f, a = f2, (arg2, arg3)`
  - `return apply(f, a)`



# MAP Built-in

---

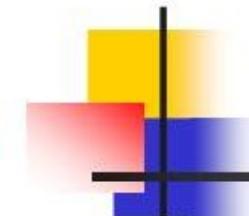
- The **map** function applies the same operation to each element in a sequence.
- **map** has the form:
  - `map(fcn, sequence)`
- Example:
  - `>>> map(lambda arg : arg / 2, (1, 2, 3))`
  - `[0, 1, 1]`



# 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
<code>import mymod</code>	Loads <b>mymod</b> module. Executes module <i>only the first time</i> it is loaded.
<code>from mymod import a, b</code>	Loads <b>mymod</b> module and creates local names <b>a</b> and <b>b</b> referencing objects with the same name inside the module <b>mymod</b> .
<code>Reload(mymod)</code>	Reload function loads module <b>mymod</b> , re-executing <b>mymod</b> each time it is reloaded.

# Import Statement (1)

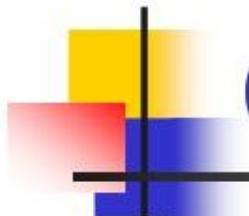
1 Qualified names

## Using the **import** statement:

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

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

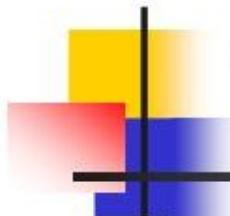
print **not** executed and counter  
**not** reset on second import



# Qualified Names

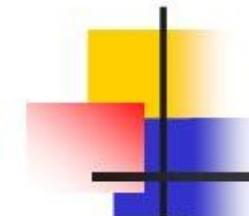
---

- 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.



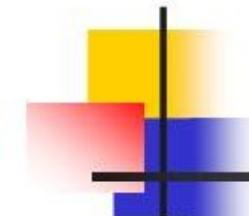
# IMPORT Statement (2)

- 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'>`



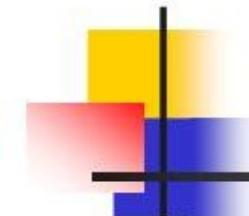
# FROM Statement (1)

- 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).



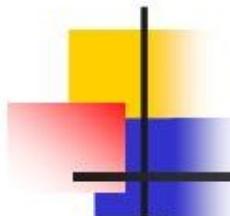
# FROM Statement (2)

- **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`



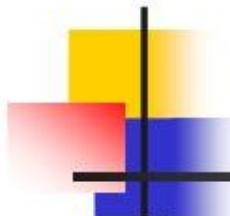
# FROM Statement (3)

- **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 **imports** 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.