Using Python

Alessio Bechini
Dip. Ing. Informazione
Università di Pisa

Agenda

- Compilation vs interpretation and scripting languages
- Python architecture
- Data structures and control flow
- Functional features
- Basic input/output
- Object Oriented Programming
- Advanced features
- Scientific scripting: NumPy
- Extending and Embedding the Python Interpreter

What Python is said to be easy to…

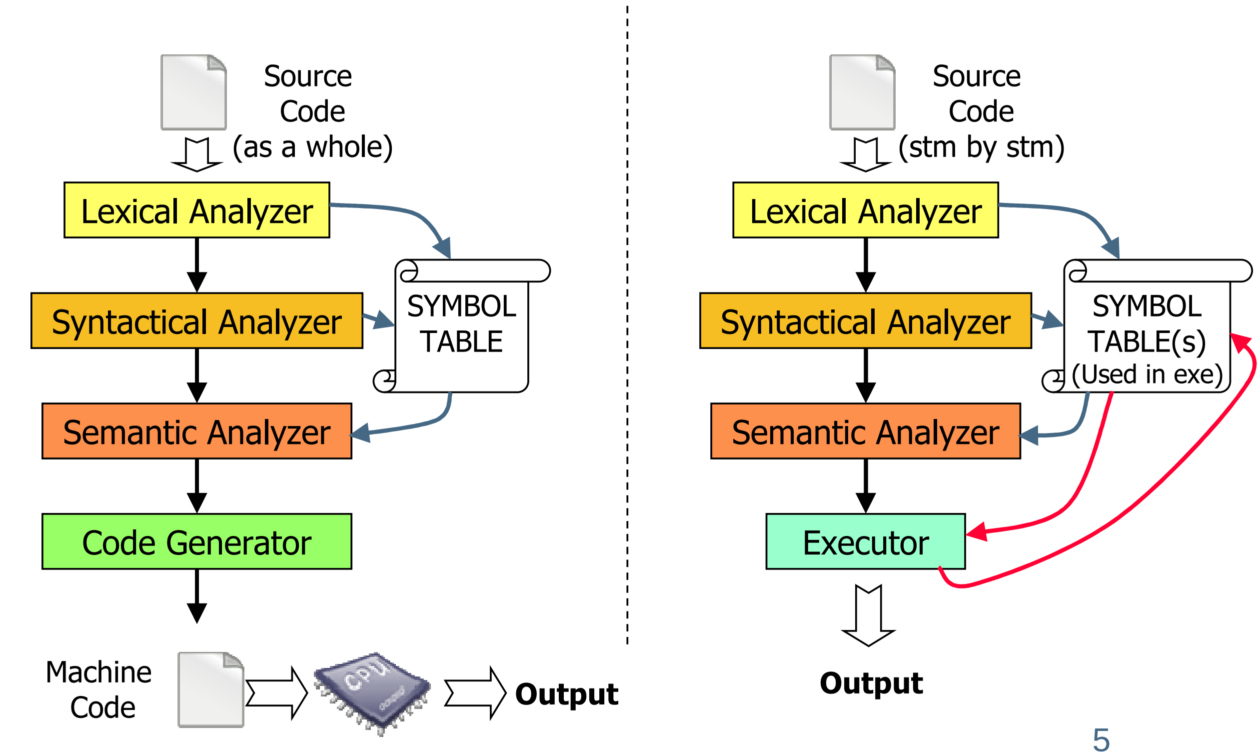
- Writing down "readable" code
  - Natural syntax
  - "Blocks by Indentation" forces proper code structuring & readability
- Code reuse
  - Straightforward, flexible way to use modules (libraries)
  - Massive amount of libraries freely available
- Object-oriented programming
  - OO structuring: effective in tackling complexity for large programs
- High performance (close ties to C…)
  - NumPy (numerical library) allows fast matrix algebra
  - Can dump time-intensive modules in C easily

Compilation and Interpretation

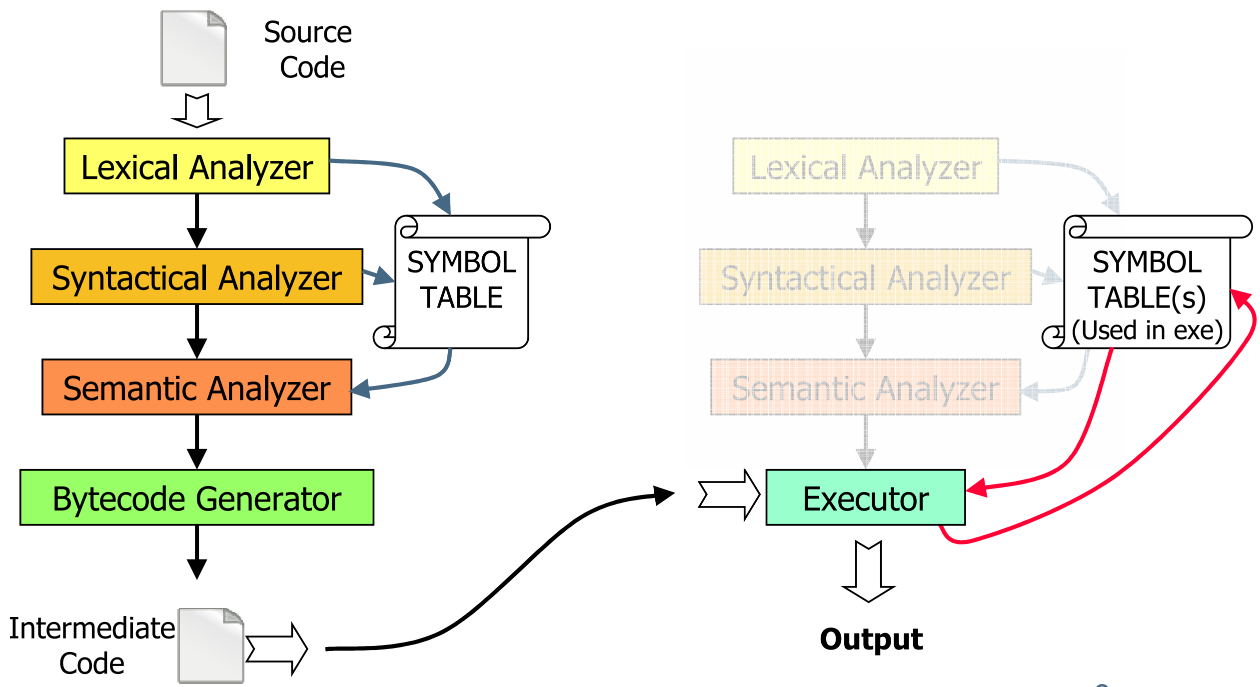
- Actual execution of programs
  written in high-level languages is usually obtained
  by two different alternative approaches:

- Compilation
  - the executable code for the target CPU is obtained
    by processing self-contained pieces of source code.
    This operation is carried out by a *compiler*.
- Interpretation
  - an *interpreter* program takes care of reading
    the high-level statements (from a source file or a console)
    and executing them one at a time

Compilation and Interpretation Steps



Use of Intermediate Languages



Scripting Languages

- Python is a ***scripting*** language: What does this mean?
  - Programming languages are aimed at *developing programs*
  - Scripting languages are aimed at *controlling applications*
- A scripting language for an operating system is called a *shell script*.
- Scripts are usually **interpreted** from the source code or "compiled" to intermediate representation (bytecode), which is interpreted.
- Some examples: shells (e.g. bash), Unix AWK, JavaScript/ActionScript (Flash), VBScript, Perl, Tcl, Python, Ruby, Boo, Groovy, MATLAB, MEL, PHP, …

Python's Features (I)

- Simple
  - Python is a *minimalistic* language.
    It allows you to concentrate on the solution of the problem.
- Easy to Learn
  - Python is easy to get started with (very simple syntax).
- Free and Open Source
  - Copies can be freely distributed, it can be changed at will and used in new free programs, etc.
- High-level
  - Programs are transparent to low-level details (e.g. memory management, etc.).
- Portable
  - Programs can work on several platforms with no changes at all (if you avoid any system-dependent features).

Python’s Features (II)

- Multi-paradigm
  - Programming paradigm: a particular approach to master program complexity by decomposing problems into simpler ones.
- Object Oriented
  - OO deals with proper combination of data and functionality.
- Extensible
  - If performance is required, a critical piece of code can be developed in C/C++ and then used from a Python program.
- Embeddable
  - Python can be embedded in C/C++ programs to provide them with ‘scripting’ capabilities.
- Extensive Libraries
  - The Python Standard Library is really huge and available on every Python installation:
    This is called the 'Batteries Included' philosophy of Python.

Interpretation in Python

- Interpreter:
  a computer program that executes instructions
  written in a given programming (scripting) language
- Python interpreter translates source code
  into an efficient intermediate representation (bytecode)
  and *immediately* executes it.
- The main benefits of interpretation are
  - flexibility
  - ease of use
  - development rapidity
- The main disadvantages are related
  to limited execution performance.

Interacting with the Interpreter

- Start Python by typing "python"
  - The actual installation directory must be in PATH …
- Other possibility: IDLE (GUI)
- ^D (control-D) exits

```
% python
>>> ^D
%
```

- Comments start with '#'

```
>>> 2+3  #Comment on the same line as text
5
>>> 7/3  #Numbers are integers by default, so…
2
>>> x=y=z=0 #Multiple assigns at once
>>> z
0
```


Running Python Programs

- A program is contained in a text file with extension .py
- To invoke the interpreter over the program:

```
% python myprogram.py
```

- How to create executable scripts (under Unix)
  - Make file executable:

```
% chmod +x myprogram.py
```

  - The first line is a kind of special comment, as it makes the OS know how to execute it:

```
#!/usr/bin/python
```

  - Then you can just type the script name to execute

```
% myprogram.py
```

  - or

```
% myprogram.py > myoutput.txt
```

Editors and Python

- A number of diverse text editors supports development of Python scripts, by means of syntax highlighting, etc.
- There is also a Python development environment in IDLE.

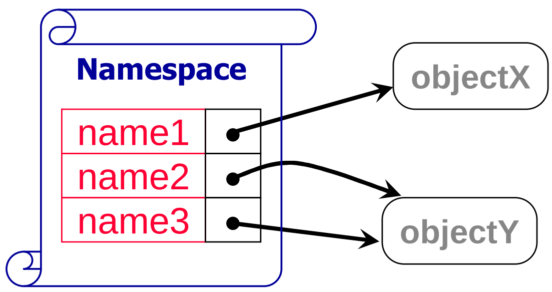
Getting Started

- Overview of language architecture; then…
- What we need to master in the first place:
  - Data types (very basic ones)
  - Literals
  - Variables
  - Control flow/conditionals
  - Functions
  - Modules

Python Structural Overview

Handled Entities

In Python, every *handled entity* is either
an *object* or a *name* or a *namespace*.



- **Objects** are management units aimed at storing data or functionality. All data is kept in objects, and modules, functions, classes, methods are objects as well.
- **Names** are used to refer objects. Multiple names can refer the same object. A non-referred object cannot be used and will be automatically garbage-collected.
- **Namespaces** are aimed at collecting names.

Python Objects

- Every object has an *identity*, a *type* and a *value*
- Identity: defined at creation (obj's address in memory)
  - Identity of object **`x`** can be known by invoking **`id(x)`**
  - Identity of two objects can be compared by '**`is`**'
- Type: defines possible values/operations for the obj
- Value: - trivial –
  - it can be MUTABLE or IMMUTABLE, depending on the fact it can be changed or not, according to the type
  - Changes to mutable objects can be done *in place*, i.e. without altering its identity (address)

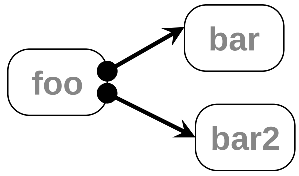
More on Python Objects

```
>>> x = 1
>>> y = ['hello']
>>> x is y
False
>>> z = x
>>> x is z
True
>>> type(x)
<type 'int'>
>>> type(y)
<type 'list'>
>>> id(x)
9852936
```

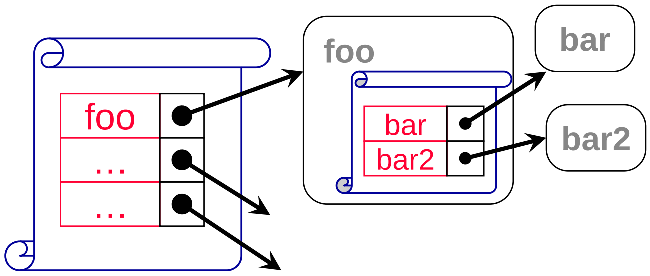
- Objects not referenced anymore are **garbage collected** by the system

Attributes and Local Variables

- An object can be supplied
other directly related objects (**attributes**).
Every object has an associated namespace
for its attributes.



- Attributes can be accessed
using the "dot notation".
E.g.: `foo.bar` refers to
the attribute named "bar"
of the object named "foo".



- A function attribute of a class
(or a class instance)
is ordinarily called "**method**"

- In addition, functions and methods provide **a temporary namespace** during execution to storage local variables.

What Namespace?

Rules that determine in which namespace a given name resides:

- Definitions within functions and methods are made in the temporary execution namespace.
Code in a function can also use (but not assign to) names in the surrounding module.
- Definitions in modules end up in the attribute namespace of the module object.
- Definitions in a class are in the attribute namespace of the class.
- Finally, in a *class instance*, when an attribute is requested that is not in the object namespace, it is searched for in the class namespace.
This is how methods are normally found.

Callable Objects

- Callable types are those whose instances support the function-call operation, denoted by the use of (), possibly with arguments.
- Among callables:
  - Functions
  - Built-in types like **`list`**, **`tuple`**, **`int`** (the call create an instance)

```
>>> a=list('ciao')
>>> a
['c', 'i', 'a', 'o']
```

  - Class objects (the call create an instance)
  - Methods (functions bound to class attributes)

Data Types:
- Basic
- Composite

Python's Basic Data Types

- *Integers* - equivalent to C longs
- *Floating-Point numbers* - equivalent to C doubles
- *Long integers*
- *Complex Numbers*
- *Strings*
- Some others, such as *type* and *function*

- Special value: **None**
  - just to refer to nothing (resembles ‘void’ in C)

Python's Composite Data Types

- aka “Container data structures”
- In other languages (e.g. Java, C++) Containers are “add-on” features, not part of the core language
- Python holds them as *fundamental data types*!

- *Sequences:*
  - *Lists*
  - *Tuples*
  - *Dictionaries*,
    aka Dicts, Hash Tables, Maps, or Associative Arrays
  - the built-in function *len(<seq>)* returns the length of a sequence

- **Arrays are not a built-in feature (!)**

Entering Values (Numbers)

- Literals for integers and long integers

```
n = 25
n = 034  #octal, prepending 0 (zero)
n = 0x4f #hex, prepending 0x
longn = 135454L
```

- Computations with short integers that overflow are automatically turned into long integers
- Floating point literals

```
f1 = 4.0
f2 = 4.2E-12
```

- Complex literals

```
cn = 10+4j
imunit = 1j
```


Entering Values (Strings)

- Strings: single or triple quoted
- Only triple quoted strings can span multiple lines
- Single quoting can be done using either ‘ or “, but:

```
s = ‘spam‘    #ok
s = “spam“    #ok
s = ‘spam“    #not correct
s = “spam‘‘   #correct as well!
```

- Triple quoting can be done repeating three times either “ or ‘:

```
s1 = “““foo“““   #three times “
s2 = ‘‘‘bar‘‘‘   #three times ‘
s3 = “‘foofoo“‘  #not correct!
```

Escape Sequences and Raw Strings

- We want to deal with the sentence “What’s your name?”

```
>>> “What’s your name?”
“What’s your name?”
>>> 'What\'s your name?'
"What's your name?"
>>> print 'What\'s \nyour name?'
What's
your name?
```

- ‘\’ can be used to break command lines
- Raw strings (escape seq not processed): pre-pend ‘r’

```
>>> print r"Newlines are indicated by \n"
Newlines are indicated by \n
```


print Statement

- The print command prints out to the standard output

```
>>> print "a", "b"
a b
>>> print "a"+"b"
ab
>>> print "%s    %s" % (a,b)      #we’ll see later…
a   b
```

- Notes
  - print automatically insert a new line
  - print(string) is equivalent to sys.stdout(string + '\n')
  - formatted print presents a similar syntax to its C-counterpart (printf())

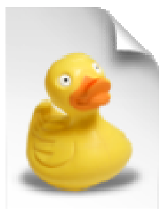
Variables

- Variables:
  - "places" to store "values", referenced by an identifier ("object name")
- Assignment: '='
  - binds names to objects stored in memory
- Python is not a strongly typed language

```
>>> x = 12.0
>>> x
12.0
>>> _        #_ refers to the last value
12.0

>>> y = 'Hello'
>>> y = y+x       # error! incompatible data types!!!
>>> y = 4.5j      # re-assignment
>>> x = x+y       # now x refer to a complex value
>>> x
(12+4.5j)
```

Variables and Data Types

- No type declaration is required in Python
- Type info is associated with objects,
  not with referencing variables!



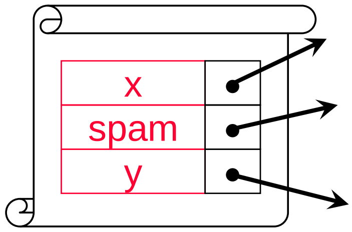
- The type corresponding to a referenced object
  is often inferred by the way it is used (!)
- This is "duck typing", widely used in scripting languages.

*"If it walks like a duck, and it quacks like a duck,
then we would call it a duck"*

- Duck typing is a form or dynamic typing
  that allows polymorphism without inheritance (we'll see)

Symbol Table(s)

- The binding (name → referenced object)
  is kept by the interpreter in a so-called “Symbol table”

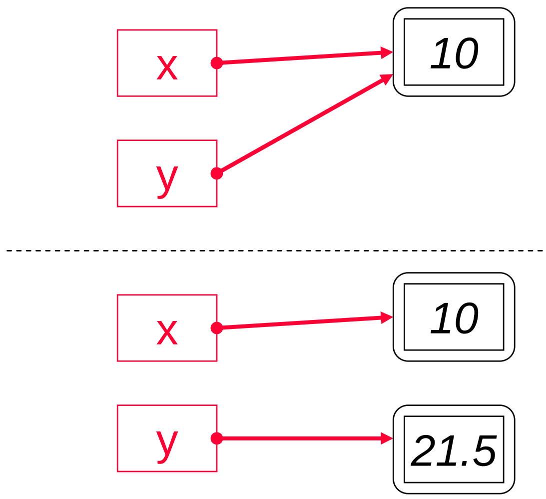


- Depending on the particular execution position,
  different symbol tables can be consulted by the interpreter
- Different symbol tables in the same execution are properly related
- A name “**x**” can be deleted from the symbol table by calling **`del(x)`**
  - It is illegal to del(x) in case the name x is referenced in an enclosing scope (we’ll see later)

Assigning Variables

- Whenever a variable references an *IMMUTABLE value*, a new object has to be created if a different value has to be stored

```
>>> x = 10
>>> x
10
>>> y = x
>>> y = y*2.15
>>> y
21.5
>>> x
10
```



Lists

- Lists are *general sequences of items*
- Lists are MUTABLE
  - items can be added, removed, searched for

```
>>> mylist = ['bye', 3.1415, 2+1j]
>>> mylist
['bye', 3.1415000000000002, (2+1j)]
>>> mylist[0]   # starts from 0, as in C arrays
'bye'
>>> mylist[-2]  # neg. index: from the end
3.1415000000000002
>>> mylist[0:2] # ‘:’ denotes a range
['bye', 3.1415000000000002]
>>> mylist[:3]  # up to index 3 (excluded)
['bye', 3.1415000000000002, (2+1j)]
>>> mylist[2:]  # from index 2 (included) on:
[(2+1j)]
```


Manipulating Lists (I)

```
>>> a = mylist   # from the previous example
>>> a
['bye', 3.1415000000000002, (2+1j)]
>>> a = a + ['hello']     # append one element
>>> a
['bye', 3.1415000000000002, (2+1j), 'hello']
>>> mylist
['bye', 3.1415000000000002, (2+1j)]  # why???
>>> a
['bye', 3.1415000000000002, (2+1j), 'hello']
>>> a.append(2L)    # another way to append (a “method”!)
>>> a
['bye', 3.1415000000000002, (2+1j), 'hello', 2L]
>>> del mylist[1]  # delete element 1: let’s check!
>>> mylist
['bye', (2+1j)]
```

Manipulating Lists (II)

```
>>> list1 = ['a', 'b', 'c']
>>> list2 = list1    #list2 -> the same obj as list1
>>> list2
['a', 'b', 'c']

>>> list1.append(1j)  # in-place append!
>>> list2
['a', 'b', 'c', 1j]

>>> list1 = list1 + [2j]  # list-copy append!
>>> list2    #list2 still -> the “old” list1
['a', 'b', 'c', 1j]

>>> list1
['a', 'b', 'c', 1j, 2j]
```

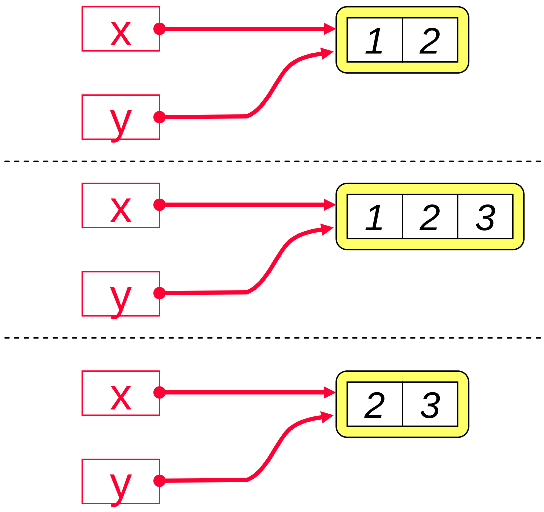

Manipulating Lists (III)

```
>>> a = [4, 5, -2, 6, 1.2]
>>> a
[4, 5, -2, 6, 1.2]
>>> a.sort()     method     #this modify the list in place!
>>> a
[-2, 1.2, 4, 5, 6]
>>> b = [0, 1]
>>> a[1] = b
>>> b.append(2)
>>> a
[-2, [0, 1, 2], 4, 5, 6]
>>> b
[0, 1, 2]
```

Referencing Lists (I)

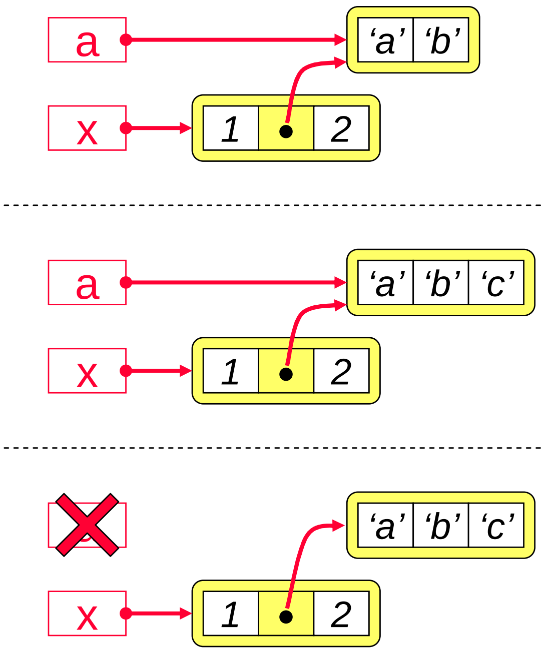
– Whenever a new variable is assigned another variable *that references a MUTABLE value* (as a list), the new variable will reference the same obj

```
>>> x = [1, 2]
>>> y = x
>>> y
[1, 2]
>>> x.append(3)
>>> y
[1, 2, 3]

>>> del x[0]
>>> x
[2, 3]
```



Referencing Lists (II)

```
>>> a = ['a', 'b']
>>> x = [1, a, 2]
>>> x
[1, ['a', 'b'], 2]

>>> a.append('c')
>>> x
[1, ['a', 'b', 'c'], 2]

>>> del a  #delete the var!
>>> x
[1, ['a', 'b', 'c'], 2]
>>> a   #error!
```



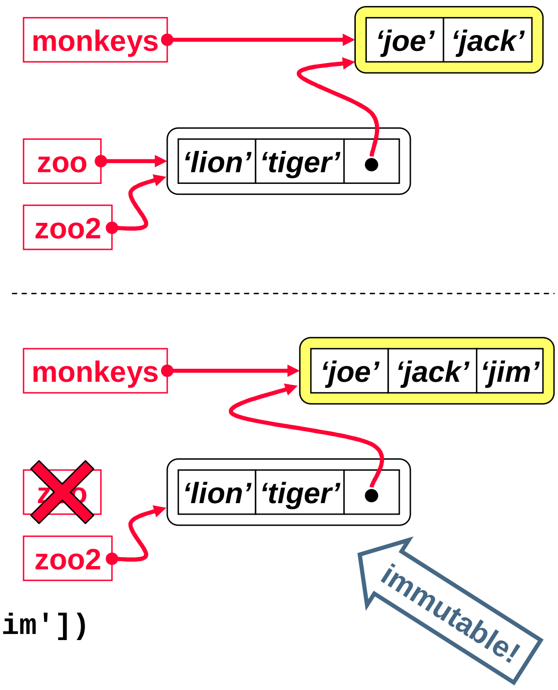
Tuples

- Tuples, like strings, are IMMUTABLE sequences
- The items of a tuple are arbitrary Python objects (either mutable or immutable)
  - Used to handle collections that are not expected to change over time (but single objects in them could…)

```
>>> firstprimes = (2,3,5,7)
>>> firstprimes[1]
3
>>> firstprimes
(2, 3, 5, 7)
>>> firstprimes.append(9) #error!
>>> del firstprimes[0]    #error!
```

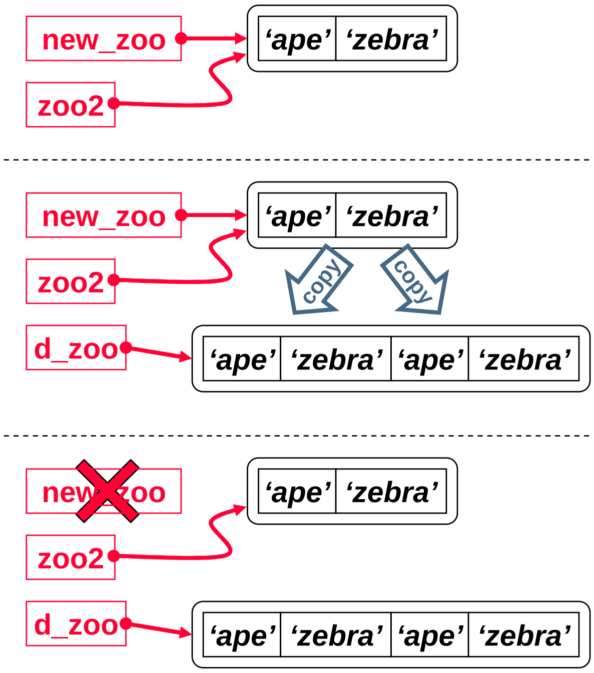

Manipulating Tuples (I)

```
>>> monkeys=['joe','jack']
>>> zoo=('lion', 'tiger', monkeys)
>>> zoo
('lion', 'tiger', ['joe', 'jack'])
>>> zoo[1]
'tiger'
>>> zoo2=zoo
>>> zoo2
('lion', 'tiger', ['joe', 'jack'])
>>> del zoo
>>> zoo2
('lion', 'tiger', ['joe', 'jack'])
>>> zoo  #error!
>>> monkeys.append('jim')
>>> zoo2
('lion', 'tiger', ['joe', 'jack', 'jim'])
>>> del zoo2[2]  #error!
>>> zoo2[2][1]
'jack'
```

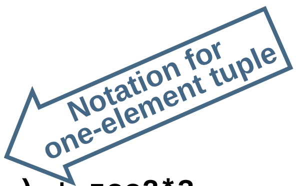


Manipulating Tuples (II)

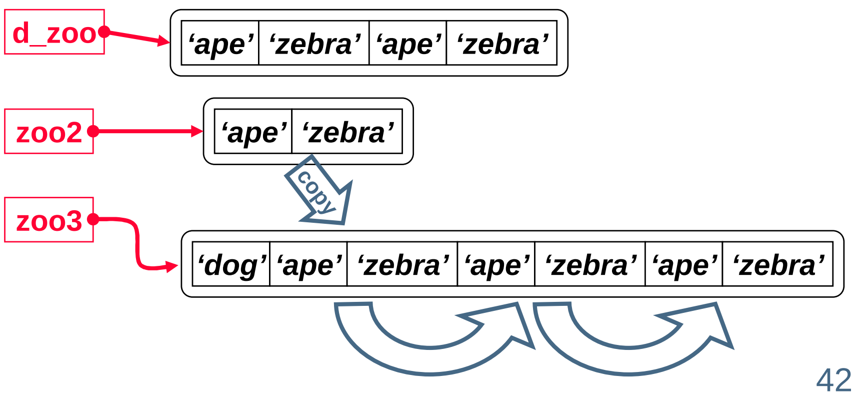
```
>>> new_zoo = ('ape','zebra')
>>> zoo2 = new_zoo

>>> d_zoo = new_zoo + new_zoo
>>> d_zoo
('ape', 'zebra', 'ape', 'zebra')

>>> del new_zoo
>>> zoo2
('ape', 'zebra')
>>> d_zoo
('ape', 'zebra', 'ape', 'zebra')
```



Manipulating Tuples (III)




```
>>> zoo3 = (‘dog’,) + zoo2*3

>>> zoo3
('dog', 'ape', 'zebra', 'ape', 'zebra', 'ape', 'zebra')
```



Tuples & print

- Tuples are commonly used
  in the formatted print statement
- print takes a “format model string”, followed by a tuple with values to be substituted in the model string:

```
>>> n= ‘Alex’
>>> print ‘My name is %s and I am %d years old’ \
 % (n, 21)
```

Tuple with params

```
My name is Alex and I am 21 years old
```


Dictionaries: Python’s Associative Arrays

- Dictionary: container of objects,
  referred through an index set (“keys”)
- The notation a[k] selects the item indexed by k from the mapping a
  - used in: expressions, as target of assignment/del statements

```
>>> atomic_num = {'Dummy' : 0,'H' : 1,'He' : 2}
>>> atomic_num['He']
2
>>> atomic_num['C'] = 6  #add a new element…
>>> atomic_num['Dummy'] = -1  #overwrite…
>>> del atomic_num['H']
>>> atomic_num
{'Dummy': -1, 'He': 2, 'C': 6}
```

Using Sequences: Indexing

- Lists, tuples and strings are examples of sequences.
- *Indexing* allows to fetch a particular item in the sequence
  - on strings:

```
>>> s='London bridge is falling down'
>>> s[1]
'o'
>>> s[-2]
'w'
```

  - on lists and tuples, the same syntax applies.
  - on dictionaries, indexing is possible through a value for the key:

```
>>> t = {3:'jim', 'foo':'jack', 'num0':7.1, 'bar': 'john'}
>>> t['num0']
7.0999999999999996
>>> t[3]
'jim'
>>> t[1]  #error!
```

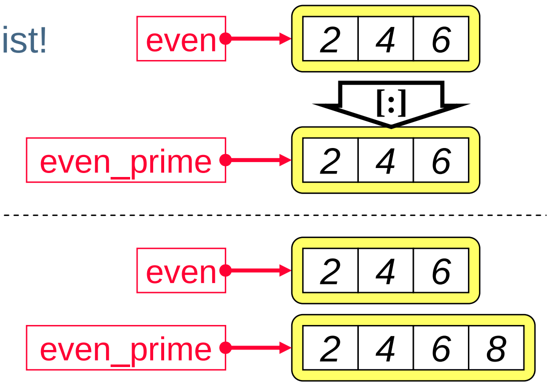

Using Sequences: Slicing

- Slicing allows to retrieve a part of the sequence

```
>>> s='London bridge is falling down'
>>> s[13:-3]
' is falling d'
>>> s[15:]  #from 15 to the end
's falling down'
>>> s[:]    #from the beginning to the end
'London bridge is falling down'
```

- Slicing can be used to get a copy of a list!

```
>>> even = [2,4,6]
>>> even_prime = even[:]
>>> even_prime.append(8)
>>> even
[2, 4, 6]
>>> even_prime
[2, 4, 6, 8]
```



Assignment Operation

…Just a Few Words

- Assignement can be either *plain* or *augmented*

- Plain assignment: in the form *target = expression*
- Plain assignment to a variable (name = value)
is the way to create a new variable
or (if already existing) to rebind it to a new value.

- Similar semantics holds whether the target
is an object attribute or a container item(s);
  - in these cases, the operation must be intended
  as ***a request of binding*** issued to the involved object,
  and such request could be either accepted or not.

Augmented Assignment

- In these cases, an augmented operator is used instead:

```
+=     -=     *=     /=     //=    %=
   **=    |=     >>=    <<=    &=     ^=
```

- Augmented assignment is allowed only with already existing (and thus bounded) targets
  - If the target refers an object that has a corresponding *in-place* method for the operator, it will be called with the right-side expression as argument;
  - otherwise, the corresponding binary operator will be applied to both the left and right sides, and the target will be bound to the result

Control Flow:
- Conditionals
- Iteration

Identifying Code Blocks

- Instructions can be grouped in blocks;
  syntactically, a block is defined by its *indentation depth*
- Statements in the same block are consecutive,
  with the same indentation for each logical line.
- New blocks of statements cannot be arbitrarily started,
  so pay attention to initial blanks

```
x = 1
 print 'x: ', x #  Error! a single space…
```

- How to indent:
  - Do not mix tabs/ spaces
  - Follow a precise indentation style, e.g.
    a single tab or a fixed number of spaces for each indentation level

Conditionals: the *if* Statement

- The *if* statement is used to check a condition;
  if the condition is true, a block (the *if-block*) is run,
  else another block (the *else-block*) is executed.
  The else clause is optional.

textual input

```
n = 51
guess = int(raw_input('Type an integer: '))
if guess == n:
    print 'You guessed it!'     # block: first line
    print "Congratulations!"   # block: last line
elif guess < n:
    print 'Too low!'    # block
else:
    print 'Too high!'   # block
print 'Program finished' # Always executed
```

Conditions as Expressions

- Conditions (e.g. as used in the *if* statement) are implemented in Python as expressions.
- “In a Boolean context”, in case an expression returns a nonzero value or a nonempty container, such outcome is taken as True; zero, **None** or empty containers are taken as False.
- The most elegant way (in Pythonic sense) to test a value x is:

```
if x:
```

- Other (less elegant) ways:

```
if x is True:

if x == True:

if bool(x):
```


Iterating with *while*

- In Python, the *while* statement can possibly present an *else* clause

```
n = 51
guessed = False
while not guessed:
   guess = int(raw_input('Type an integer: '))
   if guess == n:
      print 'You guessed it! '
      guessed = True
   elif guess < n:
      print 'Too low!'
   else:
      print 'Too high!'
else:
   print 'Finally out of the loop!'
print 'Program finished'
```

Simple Iterations with *for ... in*

- The for..in statement is used to iterate **over a sequence of objects**.

```
for x in [0, 2, -1, ‘foo’, 18, ‘bar’]:
    print x
else:  #possible else clause here as well!
    print ‘End of program’
```

- The built-in function *range()* returns arithmetic progressions

```
>>> range(10)       #from 0 (incl.) to 10 (excl.)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,12,2)   #from 1 to 12 step 2
[1, 3, 5, 7, 9, 11]
```

  – Example:

```
for x in range(1, 5):
    print x
```

  – Example:

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for x in range(len(a)):
    print x, a[x]
```


Use of *for* Loops in Other Languages

- Please note that *for* loops in Python are very different from their C/C++ counterparts.
- *for* loops in Python resemble *foreach* loops in C#.
- Java, since v. 1.5, provides a similar construct:

```
for (int i : IntArray)
```

- In C/C++:

```
for (int i = 0; i < 100; i++) { … }
```

- In Python:

```
for i in range(0,100): …
```

Use of the *break* Statement

- *break* is used to stop the execution of a looping statement, regardless of the usual loop control.
- If you break out of a loop, any corresponding *else block* is not executed.

```
while True:
    s = raw_input('Enter a string: ')
    if s == 'exit':
        break
    print 'The length is', len(s)
print 'End of program'
```


Use of the *continue* Statement

- *continue* is used to skip the rest of the statements in the current iteration

```
while True:
    s = raw_input('Enter a string: ')
    if s == 'exit':
        break
    if len(s) < 4:
        continue
    print 'The length is sufficient:', len(s)
print 'End of program'
```

range() and *xrange()*

- Very large lists to loop over in the for statement may lead to unreasonable waste of memory.
- To cope with this problem, xrange() can be used instead of range().
- xrange() returns a special-purpose read-only object that consumes much less memory (with a moderately higher overhead).

```
for x in range(1,2000,2): […]

for x in xrange(1,200,2):  […]

>>> range(1,10,2)
[1, 3, 5, 7, 9]
>>> xrange(1,10,2)
xrange(1, 11, 2)
```

Special output format

Iterators

- A more general way to loop over items in a data structure makes use of ***Iterators***.
- An Iterator is an object **`i`** such that it is possible to call **`i.next()`**, getting back the next item of iterator **`i`**.
- In case all items have been already obtained, a **`StopException`** is raised.
- An iterator over an "iterable" object **`obj`** can be obtained by calling **`iter(obj)`**

```
for x in it_obj:
    statement(s)
```

```
temp = iter(it_obj)
while True:
    try: temp.next()
    except StopIteration:
        break
    statement(s)
```

Functions

Functions: How to Define and Call

- Functions are reusable portions of programs.
- Functions have to be *defined*,
  and later they can be *called*
  (function call corresponds to an expression evaluation).
- Defined using the **`def`** keyword, followed by a *function name* and possible *parameters* (in parentheses);
  a block of statements implements the function body.
  – Example:

no params

```
def printHW():
   print 'Hello World!' # function body

for x in range(10):
   printHW()   # function call
```

Functions are Objects

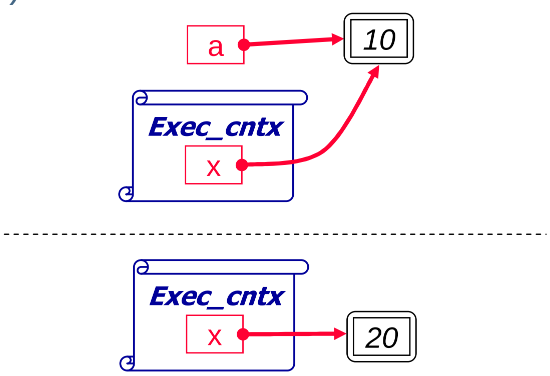
- In Python functions are objects as well.
- Thus, any variable (name) can be bound to a function

```
>>> def printHW():
        print 'Hello World!' # function body

>>> phw = printHW
>>> for x in range(3):
        printHW()
        phw()

Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```


Argument Passing: *By Value*

- At a function call, each parameter becomes a local variable in the execution context (i.e. in the temporary execution namespace), bound to the object passed as actual parameter.
- Modifications performed in the function body to the object bounded to an argument would be seen after the function execution only in case *all* such modifications have been done *in-place* (so, only for instances of mutable types)

```
>>> def double(x):
        print 'as passed: ', x
        x = x*2
        print 'as modified:', x

>>> a = 10
>>> double(a)
as passed:  10
as modified: 20
>>> a
10
```



More on Argument Passing

- Exclusive or not-exclusive use of in-place operations on mutable parameters may affect (or not) the original object

in-place op.

```
>>> def append_one(x):
      x.append(1)
      print 'as modified: ', x

>>> a=[3, 5]
>>> id(a)
13381584
>>> append_one(a)
as modified:  [3, 5, 1]
>>> a
[3, 5, 1]
>>> id(a)
13381584
```

NOT in-place op.

```
>>> def append_two(x):
        x = x+[1]
        print 'as mod.: ', x, 'at ', id(x)

>>> a=[3, 5]
>>> id(a)
13398144
>>> append_two(a)
as modified:  [3, 5, 1] at  13042232
>>> a
[3, 5]
>>> id(a)
13398144
```


Default & Keywords Arguments

- A default value for a parameter can be specified

```
>>> def myprint(msg1='Hello ', msg2='World !', times=1):
    print (msg1+msg2) * times

>>> myprint('Good', ' Bye! ', 2)
Good Bye! Good Bye!
>>> myprint()
Hello World !
>>> myprint(msg2='Alice! ', times = 3)
Hello Alice! Hello Alice! Hello Alice!
```

keyword args

- Keyword args are used to specify parameter-value bindings whenever needed

Functions: Returned Values

- Functions can return back a result from their invocation by means of the *return* statement

```
>>> def mymax(x,y):
       if x>=y:
          return x
       return y

>>> mymax(2,-5)
2
>>> c = mymax('foo', 'bar')
>>> c
'foo'
```


Documentation Strings in Functions

- To provide documentation for a function,
  a string can be placed on the first logical line of the function.
- DocStrings: also in other contexts (Modules and Classes)
- By convention, a docstring is a multi-line string composed this way:
  – The first line starts with a capital letter and ends with a dot
  – The second line is blank
  – Any detailed explanation starts from the third line.

```
def myMax(x, y):
   '''Returns the maximum of two numbers.

   The two values are supposed to be integers.'''
   if x > y:
      return x
   return y

myMax(2, 22)
print myMax.__doc__  #docstring is accessible via __doc__
```

Exceptions

What Exceptions are?

- An ***exception*** in Python is an object that represents an error or an anomalous/unexpected/special condition.
- Whenever an error/anomalous condition takes place, an exception is ***raised*** and passed to the exception-propagation mechanism.
- The raised exception can be caught from the propagation mechanism and some specific code can be executed in response to this event (***Exception Handling***)

A Motivating Example

```
>>> def show_n(list, n):
  print(list[n])


>>> show_n([1,3,5,7], 2)
5
>>> show_n([1,3,5,7], 4)

Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    show_n([1,3,5,7], 4)
  File "<pyshell#2>", line 2, in show_n
    print(list[n])
IndexError: list index out of range
```


The *try* Statement

- The **`try`** statement is aimed at delimiting a block where exceptions may occur.
- If some exceptions actually occur there, it's possible to specify what to do in the **`except`**/**`else`** clauses
- One try block may have multiple **`except`** clauses

```
try:
    statement(s)
except [expression [, target]]:
    statement(s)
[else:
    statement(s)]
```

Exception Handler

Opt. "else" clause

- The exception handler is executed in case the expression in the except clause would match the raised exception object.

The Example Revisited

```
>>> def show_n(list, n):
        print(list[n])


>>> show_n([1,3,5,7], 2)
5
>>> show_n([1,3,5,7], 4)
```

```
>>> def show_n(list, n):
        try:
                print(list[n])
        except IndexError:
                print('OUT!')


>>> show_n([1,3,5,7], 2)
5
>>> show_n([1,3,5,7], 4)
OUT!
>>>
```


The Exception Propagation Mechanism

- If no expression in the except clauses matches the raised exception object, this will be propagated up to the calling function.
- The same applies in case the exception occurs out of a try block.
- This back-propagation mechanism comes to an end as one applicable handler is found; if it is not found, the program terminates.
- The program resumes just after the executed handler.

The *try/except/finally* Statement

- A more sophisticated version of the **try** statement is the following (available in this form from Python 2.5):

```
try:
    statement(s)
except [expression [, target]]:
    statement(s)
[else:
    statement(s)]
[finally:
    statement(s)]
```

Clean-up Handler

- The finally block is executed **anyway**, regardless of the occurrence of an exception.
- In case the exception is propagated, the finally block is executed before the actual propagation.

Modules

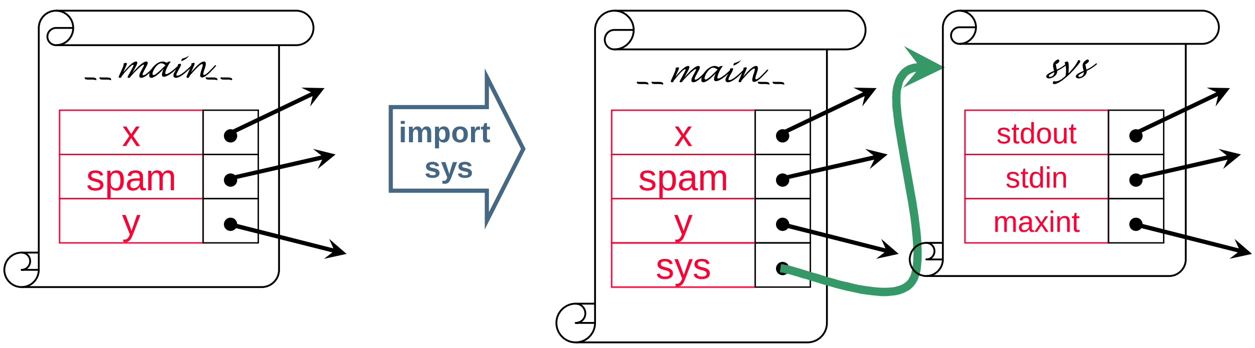
Modules: Generalities

- As functions allow reuse of code within programs,
  modules allow reuse of functions (and vars) across programs.
- **Module**: file containing all required functions/variables.
  Its filename extension must be .py
- Each module is associated with the corresponding symbol table
- Within a module, its name is accessible via the **`__name__`** global variable
- A module can contain also executable statements, that are intended to initialize the module.
  They are executed only the first time the module is imported.

Modules: *import*

- A module can be used in a program by previously *importing* it:
  this also applies to the standard library.

```
import sys  # a module of the std.lib.function body

#here we can access names in module sys
#using the notation sys.name

import sys as mysys  #import with rename
```



The *from .. import* Statement

- If you want to directly import the name *name1* defined in module *mod1* into your program, then you can use:

```
from mod1 import name1
```

- From now on, it can be referred to as *name1* instead of *mod1.name1*
- To directly import all the names in *mod1*:

```
from mod1 import *
```

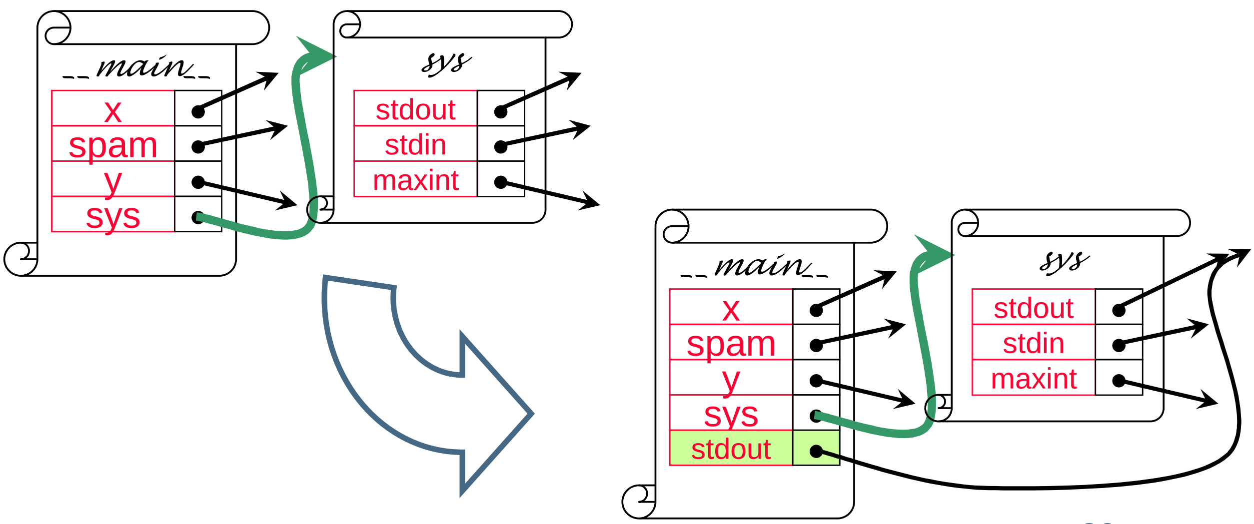
- Attention must be paid to avoid name conflicts:
  Generally, the plain *import* statement is preferable instead of *from .. import*

- All the names defined in a module **`x`** can be obtained by **`dir(x)`**

from ... import & Symbol Tables

- What happens to symbol tables in the following case?

```
import sys
from sys import stdout
```



Modules and .pyc Files

- To make module importing more efficient, Python usually creates pre-compiled files with the extension .pyc
- A .pyc file is related to the bytecode for the module content.
- The .pyc byte-compiled files are platform-independent, and can be used instead of the original module code.
- When the interpreter is asked to import a module, if the corresponding .pyc is present, part of its work has already been done.

How the Interpreter Looks for Modules

- When a module is imported, the interpreter searches for the corresponding .py file
  - first in the current directory
  - then in the list of directories specified by the environment variable PYTHONPATH.
- Actually, modules are searched in the list of directories given by the variable sys.path
- sys.path is initialized from:
  - the directory containing the input script (or the current directory)
  - PYTHONPATH
  - the installation-dependent default

Standard Modules

- Python comes with a library of standard modules, described in the “Python Library Reference”
- Some modules are built into the interpreter.
- Some commonly-used modules:
  - **sys** (system interfaces)
  - **os** (operating system interfaces)
  - **shutil** (files/directories ordinary management)
  - **string** (basic string operations)
  - **re** (regular expressions)
  - **math** (mathematical functions), **random** (random #s generation)
  - **zlib**, **gzip**, **bz2**, **zipfile**, **tarfile** (data compression)
  - **datetime** (managing dates and time data)
  - …

Packages (I)

- Packages are a way of structuring Python's module namespace by using “dotted module names”.
  - E.g., the name A.B designates the "B" submodule in the "A“ package.
- A possible structure for the “Sound” package:

```
Sound/                  Top-level package
  __init__.py           Initialize the sound package
  Formats/              Subp. for format conversions
        __init__.py
        wavread.py
        wavwrite.py ...
  Effects/              Subp. for sound effects
        __init__.py
        echo.py ...
  Filters/ Subpackage for filters
        __init__.py
        equalizer.py ...
```

Packages (II)

- Users of the package can import individual modules from the package, e.g.:

```
import Sound.Effects.echo
```

- Now the submodule Sound.Effects.echo must be referenced by its full name:

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

- An alternative way of importing the submodule is:

```
from Sound.Effects import echo
```

- Yet another variation is to import the desired function or variable directly:

```
from Sound.Effects.echo import echofilter
```

- This loads the submodule echo and makes its function echofilter() directly available:

```
echofilter(input, output, delay=0.7, atten=4)
```


Blocks

What are Blocks?

- *Block:* piece of Python program text **executed as a unit**.
- Examples of blocks:
  - a module
  - a function body
  - a class definition
  - each command typed interactively
  - a script file
  - etc.
- A code block is executed in an *execution frame*.
  - A frame contains administrative info and determines where and how execution continues after the code block's execution has completed

Blocks and Scope

- A ***scope*** defines the visibility of a name within a block.
  - If a local variable is defined in a block, its scope includes that block.
  - If the definition occurs in a function block, the scope extends to any blocks contained within the defining one (unless a contained block introduces a different binding for the name).
- The scope of a name corresponds to the set of related symbol tables that is searched for to resolve such a name.
- When a name is used in a code block, it is resolved using the nearest enclosing scope.
  The set of all such scopes visible to a code block is called the block's *environment*.
- The scope of names defined in a class block is limited to the class block; it does not extend to the code blocks of methods.

Global & Local Variables

- If a name is bound in a block,
  it is a *local variable* of that block.
- If a name is bound at the module level,
  it is a *global variable*.
- The variables of the module code block
  are local and global.
- If a variable is used in a code block but not defined there,
  it is a *free variable*.