

Slides from INF3331 - Python introduction

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2012



Basic Python Constructs

First encounter with Python

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

```
from math import sin  
import sys
```

```
x = float(sys.argv[1])  
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Running the Script

Code in file `hw.py`.

Run with command:

```
> python hw.py 0.5  
Hello world, sin(0.5) = 0.479426.
```

Linux alternative if file is executable (`chmod a+x hw.py`):

```
> ./hw.py 0.5  
Hello world, sin(0.5) = 0.479426.
```

Quick Run Through

On Unix; find out what kind of script language (interpreter) to use:

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

Access library functions:

```
from math import sin
import sys
```

Read command line argument and convert it to a floating point:

```
x = float(sys.argv[1])
```

Print out the result using a format string:

```
print "Hello world, sin(%g) = %g." % (x, sin(x))
```

Simple Assignments

```
a = 10 # a is a variable referencing an
      # integer object of value 10
```

```
b = True # b is a boolean variable
```

```
a = b # a is now a boolean as well
      # (referencing the same object as b)
```

```
b = increment(4) # b is the value returned by a function
```

```
is_equal = a == b # is_equal is True if a == b
```

Simple control structures

● Loops:

```
while condition:  
    <block of statements>
```

Here, `condition` must be a boolean expression (or have a boolean interpretation), for example: `i < 10` or `!found`

```
for element in somelist:  
    <block of statements>
```

Note that `element` is a copy of the list items, not a reference into the list!

● Conditionals:

```
if condition:  
    <block of statements>  
elif condition:  
    <block of statements>  
else:  
    <block of statements>
```

Ranges and Loops

- `range(start, stop, increment)` constructs a list. Typically, it is used in for loops:

```
for i in range(10):  
    print i
```

- `xrange(start, stop, increment)` is better for fat loops since it constructs an iterator:

```
for i in xrange(10000000):  
    sum += sin(i*pi*x)
```

- Looping over lists can be done in several ways:

```
names = ["Ola", "Per", "Kari"]  
surnames = ["Olsen", "Pettersen", "Bremnes"]  
for name, surname in zip(names, surnames):  
    print name, surname # join element by element
```

```
for i, name in enumerate(names):  
    print i, name # join list index and item
```


Lists and Tuples

```
mylist = ['a string', 2.5, 6, 'another string']
mytuple = ('a string', 2.5, 6, 'another string')
mylist[1] = -10
mylist.append('a third string')
mytuple[1] = -10 # illegal: cannot change a tuple
```

A tuple is a constant list (immutable)

List functionality

<code>a = []</code>	initialize an empty list
<code>a = [1, 4.4, 'run.py']</code>	initialize a list
<code>a.append(elem)</code>	add <code>elem</code> object to the end
<code>a + [1, 3]</code>	add two lists
<code>a[3]</code>	index a list element
<code>a[-1]</code>	get last list element
<code>a[1:3]</code>	slice: copy data to sublist (here: index 1, 2)
<code>del a[3]</code>	delete an element (index 3)
<code>a.remove(4.4)</code>	remove an element (with value 4.4)
<code>a.index('run.py')</code>	find index corresponding to an element's value
<code>'run.py' in a</code>	test if a value is contained in the list

More list functionality

<code>a.count(v)</code>	count how many elements that have the value <code>v</code>
<code>len(a)</code>	number of elements in list <code>a</code>
<code>min(a)</code>	the smallest element in <code>a</code>
<code>max(a)</code>	the largest element in <code>a</code>
<code>min(["001", 100])</code>	tricky!
<code>sum(a)</code>	add all elements in <code>a</code>
<code>a.sort()</code>	sort list <code>a</code> (changes <code>a</code>)
<code>as = sorted(a)</code>	sort list <code>a</code> (return new list)
<code>a.reverse()</code>	reverse list <code>a</code> (changes <code>a</code>)
<code>b[3][0][2]</code>	nested list indexing
<code>isinstance(a, list)</code>	is True if <code>a</code> is a list

Functions and arguments

- User-defined functions:

```
def split(string, char):  
    position = string.find(char)  
    if position > 0:  
        return string[:position+1], string[position+1:]  
    else:  
        return string, ""
```

```
# function call:  
message = "Heisann"  
print split(message, "i")
```

prints out ('Hei', 'sann').

- Positional arguments must appear before keyword arguments:

```
def split(message, char="i"):  
    [...]
```

How to find more Python information

- These slides are very brief
- The book covers more, but still only fragments of the Python language
- Therefore you will need to look up more Python information
- Primary reference: The official Python documentation at `docs.python.org`
- Very useful: The Python Library Reference, especially the index
- Example: what can I find in the `math` module? Go to the Python Library Reference index, find "math", click on the link and you get to a description of the module
- Alternative: `pydoc math` in the terminal window (briefer)
- Note: for a newbie it is difficult to read manuals (intended for experts) – you will need a lot of training; just browse, don't read everything, try to dig out the key info

eval and exec

- Evaluating string expressions with `eval`:

```
>>> x = 20
>>> r = eval('x + 1.1')
>>> r
21.1
>>> type(r)
<type 'float'>
```

- Executing strings with Python code, using `exec`:

```
exec("""
def f(x):
    return %s
""") % sys.argv[1])
```

File reading and writing

● Reading a file:

```
infile = open(filename, 'r')
for line in infile:
    # process line

lines = infile.readlines()
for line in lines:
    # process line

for i in xrange(len(lines)):
    # process lines[i] and perhaps next line lines[i+1]

fstr = infile.read()
# process the whole file as a string fstr

infile.close()
```

● Writing a file:

```
outfile = open(filename, 'w') # new file or overwrite
outfile = open(filename, 'a') # append to existing file
outfile.write("""Some string
...
""")
```

Dictionary functionality

```
a = {}
```

initialize an empty dictionary

```
a = {'point': [2, 7], 'value': 3}
```

initialize a dictionary

```
a = dict(point=[2, 7], value=3)
```

initialize a dictionary

```
a['hide'] = True
```

add new key-value pair to a dictionary

```
a['point']
```

get value corresponding to key `point`

```
'value' in a
```

True if `value` is a key in the dictionary

```
del a['point']
```

delete a key-value pair from the dictionary

```
a.keys()
```

list of keys

```
a.values()
```

list of values

```
len(a)
```

number of key-value pairs in dictionary `a`

```
for key in a:
```

loop over keys in unknown order

```
for key in sorted(a.keys()):
```

loop over keys in alphabetic order

```
isinstance(a, dict)
```

is True if `a` is a dictionary

String operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]           # extract substring
s.find(':')       # index where first ':' is found
s.split(':')      # split into substrings
s.split()         # split wrt whitespace
'Berlin' in s     # test if substring is in s
s.replace('18.4', '20')
s.lower()         # lower case letters only
s.upper()        # upper case letters only
s.split()[4].isdigit()
s.strip()        # remove leading/trailing blanks
', '.join(list_of_words)
```

Using modules

Import module as namespace:

```
import sys
x = float(sys.argv[1])
```

Import module member `argv` into current namespace:

```
from sys import argv
x = float(argv[1])
```

Import everything from `sys` into current namespace (evil)

```
from sys import *
x = float(argv[1])
```

Import `argv` into current namespace under an alias

```
from sys import argv as a
x = float(a[1])
```

Exceptions

● Handling exceptions:

```
try:
    <statements>
except ExceptionType1:
    <provide a remedy for ExceptionType1 errors>
except ExceptionType2, ExceptionType3, ExceptionType4:
    <provide a remedy for three other types of errors>
except:
    <provide a remedy for any other errors>
...
```

● Raising exceptions:

```
if z < 0:
    raise ValueError\
        ('z=%s is negative - cannot do log(z)' % z)
a = math.log(z)
```

Class programming in Python

Contents

- Intro to the class syntax
- Special attributes
- Special methods
- Classic classes, new-style classes
- Static data, static functions
- Properties
- About scope

More info

- Ch. 8.6 in the course book
- Python Tutorial
- Python Reference Manual (special methods in 3.3)
- Python in a Nutshell (OOP chapter - recommended!)

Classes in Python

- Similar class concept as in Java and C++
- All functions are virtual
- No private/protected variables (the effect can be "simulated")
- Single and multiple inheritance
- Everything in Python is an object, even the source code
- Class programming is easier and faster than in C++ and Java (?)

The basics of Python classes

- Declare a base class `MyBase`:

```
class MyBase:

    def __init__(self, i, j):    # constructor
        self.i = i; self.j = j

    def write(self):           # member function
        print 'MyBase: i=', self.i, ' j=', self.j
```

- `self` is a reference to this object

- Data members are prefixed by `self`:

```
self.i, self.j
```

- All functions take `self` as first argument in the declaration, but not in the call

```
inst1 = MyBase(6, 9); inst1.write()
```


Implementing a subclass

- **Class MySub is a subclass of MyBase:**

```
class MySub(MyBase):  
  
    def __init__(self, i, j, k): # constructor  
        MyBase.__init__(self, i, j)  
        self.k = k;  
  
    def write(self):  
        print 'MySub: i=', self.i, ' j=', self.j, ' k=', self.k
```

- **Example:**

```
# this function works with any object that has a write func:  
def write(v): v.write()  
  
# make a MySub instance  
i = MySub(7, 8, 9)  
  
write(i) # will call MySub's write
```

Comment on object-orientation

- Consider

```
def write(v):  
    v.write()
```

```
write(i)    # i is MySub instance
```

- In C++/Java we would declare `v` as a `MyBase` reference and rely on `i.write()` as calling the virtual function `write` in `MySub`
- The same works in Python, but we do not need inheritance and virtual functions here: `v.write()` will work for *any* object `v` that has a callable attribute `write` that takes no arguments
- Object-orientation in C++/Java for parameterizing types is not needed in Python since variables are not declared with types

Private/non-public data

- There is no technical way of preventing users from manipulating data and methods in an object
- Convention: attributes and methods starting with an underscore are treated as non-public (“protected”)
- Names starting with a double underscore are considered strictly private (Python mangles class name with method name in this case: `obj.__some` has actually the name `_classname__some`)

```
class MyClass:
    def __init__(self):
        self._a = False      # non-public
        self.b = 0           # public
        self.__c = 0        # private
```

Special attributes

`i1` is `MyBase`, `i2` is `MySub`

• Dictionary of user-defined attributes:

```
>>> i1.__dict__ # dictionary of user-defined attributes
{'i': 5, 'j': 7}
>>> i2.__dict__
{'i': 7, 'k': 9, 'j': 8}
```

• Name of class, name of method:

```
>>> i2.__class__.__name__ # name of class
'MySub'
>>> i2.write.__name__ # name of method
'write'
```

• List names of all methods and attributes:

```
>>> dir(i2)
['__doc__', '__init__', '__module__', 'i', 'j', 'k', 'write']
```

Testing on the class type

- Use `isinstance` for testing class type:

```
if isinstance(i2, MySub):  
    # treat i2 as a MySub instance
```

- Can test if a class is a subclass of another:

```
if issubclass(MySub, MyBase):  
    ...
```

- Can test if two objects are of the same class:

```
if inst1.__class__ is inst2.__class__
```

(`is` checks object identity, `==` checks for equal contents)

- `a.__class__` refers the class object of instance `a`

Creating attributes on the fly

● Attributes can be added at run time (!)

```
>>> class G: pass

>>> g = G()
>>> dir(g)
['__doc__', '__module__'] # no user-defined attributes

>>> # add instance attributes:
>>> g.xmin=0; g.xmax=4; g.ymin=0; g.ymax=1
>>> dir(g)
['__doc__', '__module__', 'xmax', 'xmin', 'ymax', 'ymin']
>>> g.xmin, g.xmax, g.ymin, g.ymax
(0, 4, 0, 1)

>>> # add static variables:
>>> G.xmin=0; G.xmax=2; G.ymin=-1; G.ymax=1
>>> g2 = G()
>>> g2.xmin, g2.xmax, g2.ymin, g2.ymax # static variables
(0, 2, -1, 1)
```

Another way of adding new attributes

- Can work with `__dict__` directly:

```
>>> i2.__dict__['q'] = 'some string'
>>> i2.q
'some string'
>>> dir(i2)
['__doc__', '__init__', '__module__',
'i', 'j', 'k', 'q', 'write']
```

Special methods

- Special methods have leading and trailing double underscores (e.g. `__str__`)
- Here are some operations defined by special methods:

```
len(a)           # a.__len__()
c = a*b          # c = a.__mul__(b)
a = a+b         # a = a.__add__(b)
a += c          # a.__iadd__(c)
d = a[3]        # d = a.__getitem__(3)
a[3] = 0        # a.__setitem__(3, 0)
f = a(1.2, True) # f = a.__call__(1.2, True)
if a:           # if a.__len__()>0: or if a.__nonzero__():
```


Example: functions with extra parameters

- Suppose we need a function of x and y with three additional parameters a , b , and c :

```
def f(x, y, a, b, c):  
    return a + b*x + c*y*y
```

- Suppose we need to send this function to another function

```
def gridvalues(func, xcoor, ycoor, file):  
    for i in range(len(xcoor)):  
        for j in range(len(ycoor)):  
            f = func(xcoor[i], ycoor[j])  
            file.write('%g %g %g\n' % (xcoor[i], ycoor[j], f))
```

`func` is expected to be a function of x and y only (many libraries need to make such assumptions!)

- How can we send our `f` function to `gridvalues`?

Possible (inferior) solutions

- Bad solution 1: global parameters

```
global a, b, c
...
def f(x, y):
    return a + b*x + c*y*y

...
a = 0.5; b = 1; c = 0.01
gridvalues(f, xcoor, ycoor, somefile)
```

Global variables are usually considered evil

- Bad solution 2: keyword arguments for parameters

```
def f(x, y, a=0.5, b=1, c=0.01):
    return a + b*x + c*y*y

...
gridvalues(f, xcoor, ycoor, somefile)
```

useless for other values of a, b, c

Solution: class with call operator

- Make a class with function behavior instead of a pure function
- The parameters are class attributes
- Class instances can be called as ordinary functions, now with x and y as the only formal arguments

```
class F:
    def __init__(self, a=1, b=1, c=1):
        self.a = a; self.b = b; self.c = c

    def __call__(self, x, y):    # special method!
        return self.a + self.b*x + self.c*y*y

f = F(a=0.5, c=0.01)
# can now call f as
v = f(0.1, 2)
...
gridvalues(f, xcoor, ycoor, somefile)
```

Alternative solution: Closure

- Make a function that locks the namespace and constructs and returns a tailor made function

```
def F(a=1,b=1,c=1):  
    def f(x, y):  
        return a + b*x + c*y*y  
    return f  
  
f = F(a=0.5, c=0.01)  
# can now call f as  
v = f(0.1, 2)  
...  
gridvalues(f, xcoor, ycoor, somefile)
```

Some special methods

- `__init__(self [, args])`: constructor
- `__del__(self)`: destructor (seldom needed since Python offers automatic garbage collection)
- `__str__(self)`: string representation for pretty printing of the object (called by `print` or `str`)
- `__repr__(self)`: string representation for initialization (`a==eval(repr(a))` is true)

Comparison, length, call

- `__eq__(self, x)`: for equality (`a==b`), should return `True` or `False`
- `__cmp__(self, x)`: for comparison (`<`, `<=`, `>`, `>=`, `==`, `!=`); return negative integer, zero or positive integer if `self` is less than, equal or greater than `x` (resp.)
- `__len__(self)`: length of object (called by `len(x)`)
- `__call__(self [, args])`: calls like `a(x, y)` implies `a.__call__(x, y)`

Indexing and slicing

- `__getitem__(self, i)`: used for subscripting:
`b = a[i]`
- `__setitem__(self, i, v)`: used for subscripting: `a[i] = v`
- `__delitem__(self, i)`: used for deleting: `del a[i]`
- These three functions are also used for slices:
`a[p:q:r]` implies that `i` is a slice object with attributes `start (p)`, `stop (q)` and `step (r)`

```
b = a[: -1]
# implies
b = a.__getitem__(i)
isinstance(i, slice) is True
i.start is None
i.stop  is -1
i.step  is None
```

Arithmetic operations

- `__add__(self, b)`: used for `self+b`, i.e., `x+y` implies `x.__add__(y)`
- `__sub__(self, b)`: `self-b`
- `__mul__(self, b)`: `self*b`
- `__div__(self, b)`: `self/b`
- `__pow__(self, b)`: `self**b` or `pow(self, b)`

In-place arithmetic operations

- `__iadd__(self, b): self += b`
- `__isub__(self, b): self -= b`
- `__imul__(self, b): self *= b`
- `__idiv__(self, b): self /= b`

Right-operand arithmetics

- `__radd__(self, b)`: This method defines `b+self`, while `__add__(self, b)` defines `self+b`. If `a+b` is encountered and `a` does not have an `__add__` method, `b.__radd__(a)` is called if it exists (otherwise `a+b` is not defined).
- Similar methods: `__rsub__`, `__rmul__`, `__rdiv__`

Type conversions

- `__int__(self)`: conversion to integer
(`int(a)` makes an `a.__int__()` call)
- `__float__(self)`: conversion to float
- `__hex__(self)`: conversion to hexadecimal number

Documentation of special methods: see the *Python Reference Manual* (not the Python Library Reference!), follow link from index “overloading - operator”

Boolean evaluations

- `if a:`
when is `a` evaluated as true?
- If `a` has `__len__` or `__nonzero__` and the return value is 0 or `False`, `a` evaluates to false
- Otherwise: `a` evaluates to true
- Implication: no implementation of `__len__` or `__nonzero__` implies that `a` evaluates to true!!
- `while a` follows (naturally) the same set-up

Example on call operator: StringFunction

- Matlab has a nice feature: mathematical formulas, written as text, can be turned into callable functions

- A similar feature in Python would be like

```
f = StringFunction_v1('1+sin(2*x)')  
print f(1.2) # evaluates f(x) for x=1.2
```

- $f(x)$ implies `f.__call__(x)`

- Implementation of class `StringFunction_v1` is compact! (see next slide)

Implementation of StringFunction classes

● Simple implementation:

```
class StringFunction_v1:
    def __init__(self, expression):
        self._f = expression

    def __call__(self, x):
        return eval(self._f) # evaluate function expression
```

● Problem: `eval(string)` is slow; should pre-compile expression

```
class StringFunction_v2:
    def __init__(self, expression):
        self._f_compiled = compile(expression,
                                   '<string>', 'eval')

    def __call__(self, x):
        return eval(self._f_compiled)
```

New-style classes

- The class concept was redesigned in Python v2.2
- We have *new-style* (v2.2) and *classic* classes
- New-style classes add some convenient functionality to classic classes
- New-style classes must be derived from the `object` base class:

```
class MyBase(object):  
    # the rest of MyBase is as before
```

Static data

- Static data (or class variables) are common to all instances

```
>>> class Point:
    counter = 0 # static variable, counts no of instances
    def __init__(self, x, y):
        self.x = x; self.y = y;
        Point.counter += 1

>>> for i in range(1000):
    p = Point(i*0.01, i*0.001)

>>> Point.counter      # access without instance
1000
>>> p.counter          # access through instance
1000
```


Static methods

- Python classes allow static methods
(methods that can be called without having an instance)

```
class Point:
    _counter = 0
    def __init__(self, x, y):
        self.x = x; self.y = y; Point._counter += 1
    def ncopies(): return Point._counter
    ncopies = staticmethod(ncopies)
```

- Calls:

```
>>> Point.ncopies()
0
>>> p = Point(0, 0)
>>> p.ncopies()
1
>>> Point.ncopies()
1
```

- Cannot access `self` or instance attributes in static methods

Properties

- Python 2.3 introduced “intelligent” assignment operators, known as *properties*
- That is, assignment may imply a function call:

```
x.data = mydata;      yourdata = x.data
# can be made equivalent to
x.set_data(mydata);  yourdata = x.get_data()
```

- **Construction:**

```
class MyClass(object):    # new-style class required!
    ...
    def set_data(self, d):
        self._data = d
        <update other data structures if necessary...>

    def get_data(self):
        <perform actions if necessary...>
        return self._data

    data = property(fget=get_data, fset=set_data)
```

Attribute access; traditional

- Direct access:

```
my_object.attr1 = True  
a = my_object.attr1
```

- get/set functions:

```
class A:  
    def set_attr1(attr1):  
        self._attr1 = attr # underscore => non-public variable  
        self._update(self._attr1) # update internal data too  
    ...
```

```
my_object.set_attr1(True)
```

```
a = my_object.get_attr1()
```

Tedious to write! Properties are simpler...

Attribute access; recommended style

- Use direct access if user is allowed to read *and* assign values to the attribute
- Use properties to restrict access, with a corresponding underlying non-public class attribute
- Use properties when assignment or reading requires a set of associated operations
- Never use get/set functions explicitly
- Attributes and functions are somewhat interchanged in this scheme
⇒ that's why we use the same naming convention

```
myobj.compute_something()  
myobj.my_special_variable = yourobj.find_values(x,y)
```

More about scope

- Example: a is global, local, and class attribute

```
a = 1                # global variable

def f(x):
    a = 2            # local variable

class B:
    def __init__(self):
        self.a = 3    # class attribute

    def scopes(self):
        a = 4         # local (method) variable
```

- Dictionaries with variable names as keys and variables as values:

```
locals()      : local variables
globals()    : global variables
vars()       : local variables
vars(self)   : class attributes
```

Demonstration of scopes (1)

● Function scope:

```
>>> a = 1
>>> def f(x):
    a = 2                # local variable
    print 'locals:', locals(), 'local a:', a
    print 'global a:', globals()['a']
```

```
>>> f(10)
locals: {'a': 2, 'x': 10} local a: 2
global a: 1
```

a refers to local variable

Demonstration of scopes (2)

● Class:

```
class B:
    def __init__(self):
        self.a = 3      # class attribute

    def scopes(self):
        a = 4          # local (method) variable
        print 'locals:', locals()
        print 'vars(self):', vars(self)
        print 'self.a:', self.a
        print 'local a:', a, 'global a:', globals()['a']
```

● Interactive test:

```
>>> b=B()
>>> b.scopes()
locals: {'a': 4, 'self': <scope.B instance at 0x4076fb4c>}
vars(self): {'a': 3}
self.a: 3
local a: 4 global a: 1
```

Namespaces for exec and eval

- `exec` and `eval` may take dictionaries for the global and local namespace:

```
exec code in globals, locals
eval(expr, globals, locals)
```

- Example:

```
a = 8; b = 9
d = {'a':1, 'b':2}
eval('a + b', d) # yields 3
```

and

```
from math import *
d['b'] = pi
eval('a+sin(b)', globals(), d) # yields 1
```

- Creating such dictionaries can be handy

Generalized StringFunction class (1)

- Recall the StringFunction-classes for turning string formulas into callable objects

```
f = StringFunction('1+sin(2*x)')
print f(1.2)
```

- We would like:

- an arbitrary name of the independent variable
- parameters in the formula

```
f = StringFunction_v3('1+A*sin(w*t)',
                    independent_variable='t',
                    set_parameters='A=0.1; w=3.14159')

print f(1.2)
f.set_parameters('A=0.2; w=3.14159')
print f(1.2)
```

First implementation

- Idea: hold independent variable and “set parameters” code as strings
- Exec these strings (to bring the variables into play) right before the formula is evaluated

```
class StringFunction_v3:
    def __init__(self, expression, independent_variable='x',
                 set_parameters=''):
        self._f_compiled = compile(expression,
                                    '<string>', 'eval')
        self._var = independent_variable # 'x', 't' etc.
        self._code = set_parameters

    def set_parameters(self, code):
        self._code = code

    def __call__(self, x):
        exec '%s = %g' % (self._var, x) # assign indep. var.
        if self._code: exec(self._code) # parameters?
        return eval(self._f_compiled)
```

Efficiency tests

- The exec used in the `__call__` method is slow!
- Think of a hardcoded function,

```
def f1(x):  
    return sin(x) + x**3 + 2*x
```

and the corresponding `StringFunction`-like objects

- Efficiency test (time units to the right):

```
f1 : 1  
StringFunction_v1: 13  
StringFunction_v2: 2.3  
StringFunction_v3: 22
```

Why?

- eval w/compile is important; exec is very slow

A more efficient StringFunction (1)

- Ideas: hold parameters in a dictionary, set the independent variable into this dictionary, run eval with this dictionary as local namespace
- Usage:

```
f = StringFunction_v4('1+A*sin(w*t)', A=0.1, w=3.14159)
f.set_parameters(A=2)    # can be done later
```

A more efficient StringFunction (2)

● Code:

```
class StringFunction_v4:
    def __init__(self, expression, **kwargs):
        self._f_compiled = compile(expression,
                                    '<string>', 'eval')
        self._var = kwargs.get('independent_variable', 'x')
        self._prms = kwargs
        try:    del self._prms['independent_variable']
        except: pass

    def set_parameters(self, **kwargs):
        self._prms.update(kwargs)

    def __call__(self, x):
        self._prms[self._var] = x
        return eval(self._f_compiled, globals(), self._prms)
```

Extension to many independent variables

- We would like arbitrary functions of arbitrary parameters and independent variables:

```
f = StringFunction_v5('A*sin(x)*exp(-b*t)', A=0.1, b=1,
                    independent_variables=('x', 't'))
print f(1.5, 0.01) # x=1.5, t=0.01
```

- Idea: add functionality in subclass

```
class StringFunction_v5(StringFunction_v4):
    def __init__(self, expression, **kwargs):
        StringFunction_v4.__init__(self, expression, **kwargs)
        self._var = tuple(kwargs.get('independent_variables',
                                     'x'))
        try: del self._prms['independent_variables']
        except: pass

    def __call__(self, *args):
        for name, value in zip(self._var, args):
            self._prms[name] = value # add indep. variable
        return eval(self._f_compiled,
                    globals(), self._prms)
```

Efficiency tests

● Test function: `sin(x) + x**3 + 2*x`

```
f1 : 1
StringFunction_v1: 13      (because of uncompiled eval)
StringFunction_v2: 2.3
StringFunction_v3: 22     (because of exec in __call__)
StringFunction_v4: 2.3
StringFunction_v5: 3.1    (because of loop in __call__)
```

Removing all overhead

- Instead of `eval` in `__call__` we may build a (lambda) function

```
class StringFunction:
    def _build_lambda(self):
        s = 'lambda ' + ', '.join(self._var)
        # add parameters as keyword arguments:
        if self._prms:
            s += ', ' + ', '.join(['%s=%s' % (k, self._prms[k]) \
                                   for k in self._prms])
        s += ': ' + self._f
        self.__call__ = eval(s, globals())
```

- For a call

```
f = StringFunction('A*sin(x)*exp(-b*t)', A=0.1, b=1,
                  independent_variables=('x', 't'))
```

the `s` looks like

```
lambda x, t, A=0.1, b=1: return A*sin(x)*exp(-b*t)
```


Final efficiency test

- StringFunction objects are as efficient as similar hardcoded objects, i.e.,

```
class F:  
    def __call__(self, x, y):  
        return sin(x)*cos(y)
```

but there is some overhead associated with the `__call__` op.

- Trick: extract the underlying method and call it directly

```
f1 = F()  
f2 = f1.__call__  
# f2(x,y) is faster than f1(x,y)
```

Can typically reduce CPU time from 1.3 to 1.0

- Conclusion: now we can grab formulas from command-line, GUI, Web, anywhere, and turn them into callable Python functions *without any overhead*

Adding pretty print and reconstruction

● “Pretty print”:

```
class StringFunction:
    ...
    def __str__(self):
        return self._f # just the string formula
```

● Reconstruction: `a = eval(repr(a))`

```
# StringFunction('1+x+a*y',
                 independent_variables=('x', 'y'),
                 a=1)

def __repr__(self):
    kwargs = ', '.join(['%s=%s' % (key, repr(value)) \
                        for key, value in self._prms.items()])
    return "StringFunction1(%s, independent_variable=%s"
           ", %s)" % (repr(self._f), repr(self._var), kwargs)
```

Examples on StringFunction functionality (1)

```
>>> from scitools.StringFunction import StringFunction
>>> f = StringFunction('1+sin(2*x)')
>>> f(1.2)
1.6754631805511511

>>> f = StringFunction('1+sin(2*t)', independent_variables='t')
>>> f(1.2)
1.6754631805511511

>>> f = StringFunction('1+A*sin(w*t)', independent_variables='t', \
                        A=0.1, w=3.14159)
>>> f(1.2)
0.94122173238695939
>>> f.set_parameters(A=1, w=1)
>>> f(1.2)
1.9320390859672263

>>> f(1.2, A=2, w=1)    # can also set parameters in the call
2.8640781719344526
```

Examples on StringFunction functionality (2)

```
>>> # function of two variables:
>>> f = StringFunction('1+sin(2*x)*cos(y)', \
                       independent_variables=('x','y'))
>>> f(1.2,-1.1)
1.3063874788637866
>>> f = StringFunction('1+V*sin(w*x)*exp(-b*t)', \
                       independent_variables=('x','t'))
>>> f.set_parameters(V=0.1, w=1, b=0.1)
>>> f(1.0,0.1)
1.0833098208613807
>>> str(f) # print formula with parameters substituted by values
'1+0.1*sin(1*x)*exp(-0.1*t)'
>>> repr(f)
"StringFunction('1+V*sin(w*x)*exp(-b*t)',
independent_variables=('x','t'), b=0.10000000000000000001,
w=1, V=0.10000000000000000001)"
>>> # vector field of x and y:
>>> f = StringFunction('[a+b*x,y]', \
                       independent_variables=('x','y'))
>>> f.set_parameters(a=1, b=2)
>>> f(2,1) # [1+2*2, 1]
[5, 1]
```

Exercise

- Implement a class for vectors in 3D

- Application example:

```
>>> from Vec3D import Vec3D
>>> u = Vec3D(1, 0, 0) # (1,0,0) vector
>>> v = Vec3D(0, 1, 0)
>>> print u**v # cross product
(0, 0, 1)
>>> u[1] # subscripting
0
>>> v[2]=2.5 # subscripting w/assignment
>>> u+v # vector addition
(1, 1, 2.5)
>>> u-v # vector subtraction
(1, -1, -2.5)
>>> u*v # inner (scalar, dot) product
0
>>> str(u) # pretty print
'(1, 0, 0)'
>>> repr(u) # u = eval(repr(u))
'Vec3D(1, 0, 0)'
```

Exercise, 2nd part

- Make the arithmetic operators $+$, $-$ and $*$ more intelligent:

```
u = Vec3D(1, 0, 0)
v = Vec3D(0, -0.2, 8)
a = 1.2
u+v # vector addition
a+v # scalar plus vector, yields (1.2, 1, 9.2)
v+a # vector plus scalar, yields (1.2, 1, 9.2)
a-v # scalar minus vector
v-a # scalar minus vector
a*v # scalar times vector
v*a # vector times scalar
```

Python modules

Contents

- Making a module
- Making Python aware of modules
- Packages
- Distributing and installing modules

More info

- Appendix B.1 in the course book
- Python electronic documentation:
Distributing Python Modules, Installing Python Modules

Make your own Python modules!

- Reuse scripts by wrapping them in classes or functions
- Collect classes and functions in library modules
- How? just put classes and functions in a file MyMod.py
- Put MyMod.py in one of the directories where Python can find it (see next slide)
- Say

```
import MyMod
# or
import MyMod as M    # M is a short form
# or
from MyMod import *
# or
from MyMod import myspecialfunction, myotherspecialfunction
```

in any script

How Python can find your modules

- Python has some 'official' module directories, typically

```
/usr/lib/python2.7  
/usr/lib/python2.7/site-packages
```

+ current working directory

- The environment variable `PYTHONPATH` may contain additional directories with modules

```
unix> echo $PYTHONPATH  
/home/me/python/mymodules:/usr/lib/python2.2:/home/you/yourlibs
```

- Python's `sys.path` list contains the directories where Python searches for modules

- `sys.path` contains 'official' directories, plus those in `PYTHONPATH`)

Setting PYTHONPATH

- In a Unix Bash environment environment variables are normally set in `.bashrc`:

```
export PYTHONPATH=$HOME/pylib:$scripting/src/tools
```

- Check the contents:

```
unix> echo $PYTHONPATH
```

- In a Windows environment one can do the same in `autoexec.bat`:

```
set PYTHONPATH=C:\pylib;%scripting%\src\tools
```

- Check the contents:

```
dos> echo %PYTHONPATH%
```

- Note: it is easy to make mistakes; `PYTHONPATH` may be different from what you think, so check `sys.path`

Summary of finding modules

- Copy your module file(s) to a directory already contained in `sys.path`

```
unix or dos> python -c 'import sys; print sys.path'
```

- Can extend `PYTHONPATH`

```
# Bash syntax:  
export PYTHONPATH=$PYTHONPATH:/home/me/python/mymodules
```

- Can extend `sys.path` in the script:

```
sys.path.insert(0, '/home/me/python/mynewmodules')
```

(insert first in the list)

Packages (1)

- A class of modules can be collected in a *package*
- Normally, a package is organized as module files in a directory tree
- Each subdirectory has a file `__init__.py` (can be empty)
- Packages allow “dotted modules names” like

`MyMod.numerics.pde.grids`

reflecting a file `MyMod/numerics/pde/grids.py`

Packages (2)

- Can import modules in the tree like this:

```
from MyMod.numerics.pde.grids import fdm_grids

grid = fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
...
```

Here, class `fdm_grids` is in module `grids` (file `grids.py`) in the directory `MyMod/numerics/pde`

- Or

```
import MyMod.numerics.pde.grids
grid = MyMod.numerics.pde.grids.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
#or
import MyMod.numerics.pde.grids as Grid
grid = Grid.fdm_grids()
grid.domain(xmin=0, xmax=1, ymin=0, ymax=1)
```

- See ch. 6 of the Python Tutorial (part of the electronic doc)

Test/doc part of a module

- Module files can have a test/demo script at the end:

```
if __name__ == '__main__':  
    infile = sys.argv[1]; outfile = sys.argv[2]  
    for i in sys.argv[3:]:  
        create(infile, outfile, i)
```

- The block is executed if the module file is run as a script
- The tests at the end of a module often serve as good examples on the usage of the module

Public/non-public module variables

- Python convention: add a leading underscore to non-public functions and (module) variables

```
_counter = 0

def _filename():
    """Generate a random filename."""
    ...
```

- After a standard `import MyMod`, we may access

```
MyMod._counter
n = MyMod._filename()
```

but after a `from MyMod import *` the names with leading underscore are *not* available

- Use the underscore to tell users what is public and what is not
- Note: non-public parts can be changed in future releases

Installation of modules/packages

- Python has its own build/installation system: Distutils
- Build: compile (Fortran, C, C++) into module
(only needed when modules employ compiled code)
- Installation: copy module files to “install” directories
- Publish: make module available for others through PyPi
- Default installation directory:

```
os.path.join(sys.prefix, 'lib', 'python' + sys.version[0:3],  
             'site-packages')  
# e.g. /usr/lib/python2.3/site-packages
```

- Distutils relies on a `setup.py` script

A simple setup.py script

- Say we want to distribute two modules in two files

```
MyMod.py  mymodcore.py
```

- Typical setup.py script for this case:

```
#!/usr/bin/env python
from distutils.core import setup

setup(name='MyMod',
      version='1.0',
      description='Python module example',
      author='Hans Petter Langtangen',
      author_email='hpl@ifi.uio.no',
      url='http://www.simula.no/pymod/MyMod',
      py_modules=['MyMod', 'mymodcore'],
    )
```

setup.py with compiled code

- Modules can also make use of Fortran, C, C++ code
- `setup.py` can also list C and C++ files; these will be compiled with the same options/compiler as used for Python itself
- SciPy has an extension of Distutils for “intelligent” compilation of Fortran files
- Note: `setup.py` eliminates the need for makefiles
- Examples of such `setup.py` files are provided in the section on mixing Python with Fortran, C and C++

Installing modules

- Standard command:

```
python setup.py install
```

- If the module contains files to be compiled, a two-step procedure can be invoked

```
python setup.py build  
# compiled files and modules are made in subdir. build/  
python setup.py install
```

Controlling the installation destination

- `setup.py` has many options

- Control the destination directory for installation:

```
python setup.py install --prefix=$HOME/install  
# copies modules to /home/hpl/install/lib/python
```

- Make sure that `/home/hpl/install/lib/python` is registered in your `PYTHONPATH`

How to learn more about Distutils

- Go to the official electronic Python documentation
- Look up “Distributing Python Modules”
(for packing modules in `setup.py` scripts)
- Look up “Installing Python Modules”
(for running `setup.py` with various options)

Doc strings

Contents

- How to document *usage* of Python functions, classes, modules
- Automatic testing of code (through doc strings)

More info

- App. B.1/B.2 in the course book
- HappyDoc, Pydoc, Epydoc manuals
- Style guide for doc strings (see `doc.html`)

Doc strings (1)

- Doc strings = first string in functions, classes, files

- Put user information in doc strings:

```
def ignorecase_sort(a, b):  
    """Compare strings a and b, ignoring case."""  
    ...
```

- The doc string is available at run time and explains the purpose and usage of the function:

```
>>> print ignorecase_sort.__doc__  
'Compare strings a and b, ignoring case.'
```

Doc strings (2)

- Doc string in a class:

```
class MyClass:  
    """Fake class just for exemplifying doc strings."""  
  
    def __init__(self):  
        ...
```

- Doc strings in modules are a (often multi-line) string starting in the top of the file

```
"""  
This module is a fake module  
for exemplifying multi-line  
doc strings.  
"""
```

Doc strings (3)

- The doc string serves two purposes:
 - documentation in the source code
 - on-line documentation through the attribute `__doc__`
 - documentation generated by, e.g., HappyDoc
- HappyDoc: Tool that can extract doc strings and automatically produce overview of Python classes, functions etc.
- Doc strings can, e.g., be used as balloon help in sophisticated GUIs (cf. IDLE)
- Providing doc strings is a good habit!

Doc strings (4)

There is an official style guide for doc strings:

- PEP 257 "Docstring Conventions" from <http://www.python.org/dev/peps/>
- Use triple double quoted strings as doc strings
- Use complete sentences, ending in a period

```
def somefunc(a, b):  
    """Compare a and b."""
```

Automatic doc string testing (1)

- The `doctest` module enables automatic testing of interactive Python sessions embedded in doc strings

```
class StringFunction:
    """
    Make a string expression behave as a Python function
    of one variable.
    Examples on usage:
    >>> from StringFunction import StringFunction
    >>> f = StringFunction('sin(3*x) + log(1+x)')
    >>> p = 2.0; v = f(p) # evaluate function
    >>> p, v
    (2.0, 0.81919679046918392)
    >>> f = StringFunction('1+t', independent_variables='t')
    >>> v = f(1.2) # evaluate function of t=1.2
    >>> print "%.2f" % v
    2.20
    >>> f = StringFunction('sin(t)')
    >>> v = f(1.2) # evaluate function of t=1.2
    Traceback (most recent call last):
      v = f(1.2)
    NameError: name 't' is not defined
    """
```

Automatic doc string testing (2)

- Class `StringFunction` is contained in the module `StringFunction`
- Let `StringFunction.py` execute two statements when run as a script:

```
def _test():
    import doctest
    return doctest.testmod(StringFunction)

if __name__ == '__main__':
    _test()
```

- Run the test:

```
python StringFunction.py          # no output: all tests passed
python StringFunction.py -v       # verbose output
```