Introduction to Python

This paper was prepared for submittal to
4th International Python Workshop
Livermore, California
June 3-6, 1996

May 23, 1996

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.
Introduction to Python

Paul F. Dubois
Lawrence Livermore National Laboratory
dubois1@llnl.gov

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.
This tutorial will cover the basics of Python and its new Numerical extension.

Part I: The Object-oriented Approach
- A quick look at Python
- Basic terminology and philosophy of object-oriented programming
- Design by Contract
- Advantages of “steering” a compiled program

Part II: Basic Elements of Python
- Programs
- Modules
- Classes
- Procedures
- Inheritance and Dynamic Binding
- Statements
- Basic types
- Extension objects

Part III: Numerical Python

Part IV: Extending Python
Everything you need is at www.python.org

Sources
Binaries
Libraries of python code and python extensions
Documentation
  Tutorial
  Python Reference
  Python Library
  Extending Python
References to other tutorials, examples, etc.
Part I:
The Object-Oriented Approach
Python is an object-oriented interpreted language.

Let us observe tradition:

print 'Hello, World.'

You can:
- Put this in a file hello.py and execute: python hello.py
- (Unix) add #!/usr/local/bin/python as first line, make it executable.
- python -i hello.py -- says hello and then gives you a prompt
- python
  >>> print 'Hello, World'
- python
  >>> execfile('hello.py')
- python
  >>> import hello

For the most part I will show the script and not discuss the execution method.
Python is easy to learn.

Most statements look familiar to users of C, Matlab, IDL, etc.

```python
x = 1
y = 2
print x * y / 2.0e-5
z = [1., 2., 3.]
print "The answer is", z[1]

def dot (x, y):
    sum = 0.0
    for i in range (len (x)):
        sum = sum + x[i] * y[i]
    return sum

def sum_square (x):
    sum = 0.0
    for y in x:
        sum = sum + y**2
```

Paul F. Dubois
dubois1@lanl.gov
Creating meaningful objects is easy.

from Numeric import *
class Point2:
    "Points in two space"
    def __init__ (self, x, y):
        "Initialize a new point (x, y)"
        self.x = x
        self.y = y
    def norm (self):
        "Distance from this point to the origin"
        return sqrt(self.x**2 + self.y**2)
    def __add__ (self, other):
        "Sum of other point and self"
        return Point2(self.x + other.x, self.y + other.y)
    def __sub__ (self, other):
        "Difference of self and other"
        return Point2(self.x - other.x, self.y - other.y)
    def __repr__ (self):
        "Print self as (x, y)"
        # Note: back ticks convert any object to string representation
        return '(x + `self.x` + ', y + `self.y` + ')'
    __str__ = __repr__  # use same method if back-ticked ourselves.

Procedures are objects, too.
Now we can use the Point2 objects naturally.

The statements

\[
p1 = \text{Point2}(1., 1.) \\
p2 = \text{Point2}(2., 0.) \\
\text{print } p1, p2, p1 + p2, p1 - p2, p1.\text{norm()}\]

are executed after the definition of Point2, producing output:

\[
(1.0, 1.0) \quad (2.0, 0.0) \quad (3.0, 1.0) \quad (-1.0, 1.0) \quad 1.41421356237
\]

print is using the \_\_repr\_ method when asked to print each Point2 object.
Built-in and contributed modules provide powerful capabilities.

Strings
- strings
- regular expressions

Operating System
- os, signal, socket, thread, time, getopt, tempfile
- Many more which are unix specific

Numerical (under construction)
- Numerical extension for fast arrays of many types
- Matrix packages, FFT, ...

Graphics
- Gist, Narcisse, PLPLOT, GL

GUI
- Tkinter, WPY, Microsoft Foundation Classes

Platform specific
- MAC, Windows, SGI

Database
- PDB, Netcdf, another PDB, dbm, gdbm
Debugger / Profiler

Internet / WWW
- Grail browser
- CGI, URL, HTTP, FTP, NNTP, SGML, MIME
- Plugins, etc., all happening too fast for anyone to keep up.

Multimedia

Cryptography
A community effort of cooperation and donation of effort to the common good is a hallmark of the Python community.

LLNL is pleased to announce the release of these modules, with more to follow:

- GIST -- Interface to a no-cost package for scientific graphics.
- PDB -- Interface to a no-cost package for self-describing data files.
- HELP -- Module to provide simple help packages for Python.
- URNG -- A independent random number stream package (Unix, at least).
The three main features of an object-oriented approach to programming are encapsulation, inheritance, and dynamic binding.

Encapsulation

Data and the functions that operate upon it are bound together into a class, which is a representation of an abstract data type. The class is instantiated one or more times to create objects.

Inheritance

New classes can be created which are modifications of previously defined classes.

Dynamic Binding

If an operation is requested on an object the version of the operation defined in the most specific class to which the object belongs is actually executed.
These three concepts work together to allow the production of reusable, reliable, maintainable, and modifiable software.

**Encapsulation**

*Clients* of a class (that is, other classes which use the given class) can rely only on the class interface. Details of the implementation are hidden, and therefore changeable without consequences to clients.

**Inheritance**

Classification and abstraction are directly expressed in the software. When a class is needed that is the same as an existing class but with some additions and changes, it can be constructed without disturbing the original.

**Dynamic Binding**

*Dynamic binding* is like delegation in an organization: it allows the correct specialist to do the job, while allowing most of the program to be unaware that this is happening.
The interface to class “Particle” from a particle in cell code. (Eiffel)

defered class interface

    PARTICLE

    feature -- Conversion
        out: STRING

    feature -- Status report
        charge: REAL
            -- Particle charge
        mass: REAL
            -- Particle mass
        momentum: MOMENTUM
            -- Particle momentum
        position: POSITION
            -- Particle position
        ux: REAL
            -- X coordinate of momentum
        uy: REAL
            -- X coordinate of momentum
        uz: REAL
            -- X coordinate of momentum
        x: REAL
            -- X coordinate of position
        y: REAL
            -- Y coordinate of position
        z: REAL
            -- Z coordinate of position
feature -- Status setting
    set_momentum (m: like momentum)
        -- Set momentum to m.
        require
            not_void: m /= void
        ensure
            x_set: momentum.x = m.x;
            y_set: momentum.y = m.y;
            z_set: momentum.z = m.z
    set_position (p: like position)
        -- Set position to p
        require
            not_void: p /= void
        ensure
            x_set: position.x = p.x;
            y_set: position.y = p.y;
            z_set: position.z = p.z

invariant
    mass_positive: mass > 0.0;
    position_x: x = position.x;
    position_y: y = position.y;
    position_z: z = position.z;
    momentum_x: ux = momentum.x;
    momentum_y: uy = momentum.y;
    momentum_z: uz = momentum.z;

end -- class PARTICLE

Things to note:
The quantity "position" has features x, y, and z and the particle has features x, y, and z, and we are sure they are equal.

This class is abstract (deferred in the Eiffel terminology, virtual in C++). That is, you cannot create a PARTICLE. You must create a particular descendent of PARTICLE that has been completely implemented.

But, you can use PARTICLE in most of your program, without knowing the actual form of the representation.
Note that an interface constitutes a contract between the user of a class and its author.

The require clauses state what must be true before a procedure executes.

The ensure clauses state what must be true after it executes.

The class invariants state what must be true every time the object is available to be used, i.e., after it is created and after executing any member of its public interface.
Design by Contract views the supplier / client relationship as a contract with mutual obligations and benefits.

The supplier's `require` clause or `precondition` states conditions under which the procedure is willing to work.

The supplier's `ensure` clause or `postcondition` states what the procedure promises to accomplish.

The obligation of the client is to make sure the precondition is satisfied before making the call. The benefit to the client is not having to check that the supplier did the job promised.

The obligation of the supplier is to do the job promised. The benefit is not having to check the preconditions or deal with cases not conforming to them.
Design by Contract has implications for inheritance.

If you specialize the supplier class is should be allowed to be:

- *Or* a new precondition with the parent’s precondition, that is, be more liberal as to acceptable job conditions.
- *And* a new postcondition with the parent’s postcondition, that is, be more generous as to what will be accomplished.
Constructing the contracts between your components is often part and parcel of making decisions about who is responsible for modeling what.

You in fact are constructing a set of axiom-like relationships between the components. These relationships are closely related to the models and approximations you are making.

The collection of classes you develop becomes a long-term resource that models your application domain.

The typical scientific skills of abstraction, specialization, classification, and logical rigor are brought to bear upon the software construction process.
Reuse should be part of a long-term strategy.

Reuse has many meanings:
- Reuse in a new project
- Reuse in another part of the first project
- Reuse when the component survives a major upgrade or serves as a parent class to a new component.

Reuse increases reliability and efficiency.
- When you reuse a part you increase reliability. You are using a part that is known to function correctly.
- If a component is reused it becomes a more attractive target for optimization.
Python supports object-oriented programming but doesn’t enforce it.

class Particle3:

    def __init__(self):
        "Set position and momentum to zero."
        self.x = 0.0
        self.y = 0.0
        self.z = 0.0
        self.ux = 0.0
        self.uy = 0.0
        self.uz = 0.0

    def position (self):
        "The current location as a 3-tuple"
        return (self.x, self.y, self.z)

    def momentum (self):
        "The current momentum as a 3-tuple"
        return (self.ux, self.uy, self.uz)

    def set_position (self, p):
        "Set the position to p"
        self.x = p[0]
        self.y = p[1]
        self.z = p[2]
def set_momentum(self, m):
    "Set the momentum to m"
    self.ux = m[0]
    self.uy = m[1]
    self.uz = m[2]

I chose Python "tuples" to represent position and momentum for simplicity.
How Python stacks up as an object-oriented language.

Supports object-oriented programming?
- Supports encapsulation, dynamic binding, and inheritance. Encapsulation weak, but strengthened by additional module concept.

Single or multiple inheritance?
- Multiple, but no adaptation.

Compiled or interpreted or both?
- Interpreted but you can add compiled object extensions.

Statically type-checked or not?
- Not.

Type-safe or not?
- Not.

Garbage collected or not?
- Reference counted GC.

Exception handling?
- Yes.

Support for programming by contract?
- No.
Portability, Standards, Vendors

- Yes, no, free but only one source.

Libraries

- Yes, yes, yes.

Development environment

- python.el for Emacs, many useful libraries including MFC.
Steerable applications are powerful and able to be adapted to new opportunities.

- An application is called *programmable* or *steerable* if the user interface has the power of a full programming language so that the user can control and query the major structures in the application.
- At LLNL, four systems for creating steerable applications have been used quite successfully:
  - Basis -- interface language is Basis Language
  - Panacea -- interface language Scheme
  - Yorick -- interface language Yorick
  - Supercode -- interface language similar to C++
- All these systems have in common the coupling of fast compiled worker routines with flexible and powerful interpreted languages.
- Even though the languages are interpreted the array operations in particular must be fast

```
plot sqrt (x**2 + y**2), ln (z) color=red
```

Statements like this one may involve thousands of arithmetic operations.
Python is an excellent choice for a steering language.

- Python is particularly designed to have "C extensions", easy-to-write components which can add new object types and new modules and functions to Python.
- Python as a language is object-oriented, so we can "shadow" the class structures in object-oriented compiled components with similarly behaving Python objects.
- Object types can be created in C which are efficient.
- The Numerical extension has all the speed and power of the best of our current interpreted languages.
Part II

Basic elements of Python
Python programs are made up objects which are bound to names in one or more namespaces.

At any given time there are exactly three namespaces active.

```python
x = 2.
def doit:
    x = 3.; y = [1., 2., 3.]
doit()
```

![Diagram showing namespaces and objects](image)
Each object has an identity, a type, and a value.

- The identity never changes once the object is created. You can think of it as the object’s address.
- The type never changes. The type determines the operations that an object supports and determines the possible values for objects of that type.
- The value of an object is its contents. The value may be changeable or not changeable.
Types are classified as either mutable or immutable.

Mutable -> value can change

Immutable -> value can not change

Immutable

- None
  None is what is returned by a routine that doesn’t return anything

- Numbers: int, long, float
  3, 4L (unlimited range), 4.0, 4.0e-6

- Strings
  "Hello", 'Hello world', "Fireman’s Ball", ' "Hello", he said.', """"This is a string which can last a long while before I finally get to the triple double quote at the end"

- Tuples
  (1, 2, [2,3,4L])
  Tuples are immutable sequences of other objects.
  An argument list is a tuple, for example.
Modules are encapsulated name spaces.

There are two basic kinds of module:

- Extension written in C, whether built in to the basic Python or not, and
- Python source file foo.py

When a module is initialized, a namespace is created to be global namespace for the module.

Objects created in the module are bound to names in the module namespace.

Later, other names in other namespaces may be bound to the same objects.
The names in the module are accessed from outside using dot notation.

```python
# file my_team.py
names = ['Zane', 'Lee', 'Sharon', 'Brian']

>>> import my_team
>>> print my_team.names[0]
Zane
>>> team_names = my_team.names
>>> print names[0]
Zane
>>> team_names.append ('Fred')
>>> print team_names[4]
Fred
>>> print my_team.names[4]
Fred
```

![Diagram showing module my_team with names and list object](image)
Functions are objects, too.

```python
>>> def square(x):
    return x * x
>>> s = square
>>> print s(5), s(5.)
25 25.0
```

Shortcut names for functions in modules are easy.

```python
import os  # Module has os.getcwd() to get current directory name
cwd = os.getcwd
print cwd()
```
Classes in Python are a type of object.

class Peanut:
    salted = 'yes'
    has_shell = 'yes'

    def remove_shell (self):
        self.has_shell = 'no'
The class object supports the call operator (), which creates a new instance of the class.

```
x = Peanut ()
y = Peanut ()
y.remove_shell ()

>>> print type (x)
<type 'instance'>
>>> print x.__class__
<class Peanut at 7bcc98>
>>> print type (Peanut)
<class Peanut>
```
A creation routine can be supplied by supplying a class method with the special name \texttt{\_init\_}.

```python
class Peanut:
    def \_init\_ (self, salted = 'yes', has\_shell = 'no'):
        self.salted = salted
        self.has\_shell = has\_shell

Now we can create Peanuts in different ways:
zz = Peanut () #Salted, shelled
zz = Peanut ("no", "yes") #Unsalted, unshelled
zz = Peanut (salted = 'no') #Unsalted, shelled
```
The creation routine can be made to handle different argument signatures.

```python
import types  # A module built in to Python that has standard type objects

class Circle:
    error = "Circle creation error"
    center = (0., 0.)
    radius = 1.0
    def __init__(self, *args):
        if len(args) == 1:
            if len(args) == 2:
                self.center = (float(args[0]), float(args[1]))
            elif len(args) == 1:
                self.radius = float(args[0])
            else:
                throw error, "Bad number of arguments"

Positional and keyword arguments can also be combined.
```
Classes define methods with the first argument representing the instance on which the method is called.

This argument is called *self* by convention.

class Circle:
    center = (0., 0.)
    def translate (self, x, y):
        self.center = (self.center[0] + x, self.center[1] + y)

c = Circle()
c.translate (1., 2.)
Circle.translate (c, 1., 2.)  #same thing

Frequent beginner errors:

- Forget the self argument in the definition, get confusing error message about wrong number of arguments.
- Forget to use self as a target within its own methods to get at its attributes.

center = (center[0] + x, center[1] + y)

This would have failed because there is no name called *center* when evaluating the right hand side, and would have created a local variable named *center* in the namespace of *translate*, which would then have gone out of scope.
More about the namespace rules...

The import statement has several forms.

- **import foo**
  Tries to find a built-in module foo, a Python source file foo.py or a C extension module foomodule.so.
  Places the name foo in the current global symbol table, initializes the module if it hasn’t been already initialized.

- **from foo import name1, name2, ...**
  Puts the names listed in the current global symbol table. Does not put the name of the module in the current global symbol table.

- **from foo import ***
  Imports all names except those beginning with an underscore

Scoping rules

- There is a module in effect when Python starts named "__main__". The built-in functions have a namespace named "__builtin__".
- The local name space for a function is created when the function is called and forgotten when is returns or raises an exception.
- Assignments always go into the innermost scope.

At any time during execution, exactly three name spaces are in use.

- Local
- global
- builtin

Class definitions place yet another name space in the local scope.
Class definitions are executable statements that create a class object.

As such, they can go in an if statement, or inside a function, for example.

When a class definition is entered, a new name space is created, and used as the local scope.

All assignments and function definitions in the body of the class definition therefore go into the class’ name space.

When the class definition is left normally via the end, a class object is created. The original local scope is reinstated, and the class object is bound to the class name.
Class objects support two operations, instantiation and attribute references. Instance objects support attribute references.

class MyBin:
    i = 55555
    def f(x):
        return i - 7 + x

Compare:

myb = MyBin()
print myb.i  #prints 5
MyBin.i = 7
print myb.i  #prints 7

But:
myc = MyBin()
myc.i = 9
myc.i = 9
print myb.i, myc.i #prints 7 9
Here's a model for thinking about classes.

Data hides functions

Instance data hides class data
Usually you want instance data, leaving class data for shared data structures and argument defaults.

```python
class Point:
    origin = (0., 0., 0.)
    def __init__(self, where = origin)
        self.location = where

b = Point ()  # b.location == origin
b = Point ((1., 2., 1.))  # c.location == (1., 2., 1.)

Note that origin can be seen as the default object in the def __init__ statement, but not from inside the function __init__ when it executes, except using its full name Point.origin.

Frequent error:

class Something:
    my_list = []
    def foo (self, z, y = my_list):
        y.append(z)
        return len(y)

The second time foo is called with only one argument, my_list isn’t empty any more. Should be y = my_list[:], a copy.
```
Classes can implement basic operations by defining routines with certain names.

Numeric behavior:

class A:
    def __init__ (self, value):
        self.value = value
    def __add__ (self, other):
        return A (self.value + other.value)
    def __repr__(self):
        return `self.value`

x = A(1.)
y = A(2.)
print x + y  # prints 3.0
In similar ways we can control operations such as

- x[i]
- x[i:j]
- x['hello']
- del x[i]
- x about to be garbage collected
- x about to be compared to y
- x (arglist)
- x.something
- x.something = something
- del x.something
- len (x)
- x - y, x**2, x << y, -x, +x, float (x), int (x), hex (x)
The global declaration can be used to change the rule about assignments being to innermost scope.

```python
x = 1
def f(y):
    x = y

f(12)
print x  # prints 1 still...the 12 was stored in f's local data.

def g(y):
    global x
    x = y

g(12)
print x  # prints 12
```
Simple inheritance is simple.

class Rectangle:
    def __init__(self, l, w):
        self._length = l
        self._width = w
    def length(self):
        return self._length
    def width(self):
        return self._width
    def perimeter(self):
        return 2.0 * (self._length + self._width)

class Square(Rectangle):
    def __init__(self, side):
        self._length = side
    def width(self):
        return self._length
    def perimeter(self):
        return 4.0 * self._length
Inheritance is worthless unless the most specific version of a function for a given type is called. In Python, it is.

```python
a_rectangle = Rectangle (4.0, 5.0)
a_square = Square (6.0)
print a_rectangle.perimeter(), a_square.perimeter (), a_square.length()
```

- Dynamic binding ensures that `a_square.perimeter()` calls the version in class `Square`.
- `a_square.length()` uses the parent version in `Rectangle` which was not redefined.
Python has a small number of statement types.

Simple statements

- expression
  \( f.g() \)
- assignment
  \( x = y + 1 \)
- pass -- do nothing
- del -- unbind a name from an object or remove from list
  \( \text{del } x[i], y \)
- print
- return
- raise -- raise an exception
- break -- leave a loop
- continue -- next iteration of a loop
- import -- module importation
- global -- scoping command
- exec -- recursive parsing
## Introduction to Python

Statements are line-oriented.

Line continuation with backslash (\) but not required if inside () or [].

Semicolon can be used for separate statements per line.

Indentation of the first line of the statement must be correct.

```python
maze = [
    [1,1,1,1,1,1,1,1,1,1],
    [1,0,0,1,1,0,0,0,1,1,1],
    [1,0,1,1,1,1,1,0,1,0,1],
    [1,0,0,0,0,1,1,0,1,0,1],
    [1,0,1,1,0,1,1,0,0,0,1],
    [1,0,1,1,0,0,1,0,1,1,1],
    [1,0,1,0,0,1,1,0,1,0,1],
    [1,0,1,0,1,1,1,0,1,0,1],
    [1,0,1,0,0,0,0,0,0,0,1],
    [1,0,0,1,1,1,1,0,0,0,1],
    [1,0,0,0,1,1,1,1,0,1,1],
    [1,0,0,0,0,0,0,0,1,0,1],
    [1,1,1,1,1,1,1,1,1,1,1]
]
entrance = (11, 0); goal = (2,10); initial_direction = East
monsters = yes  # Error, indented statement at top level
```
Assignments bind names to objects.

Binding a name to an object is not the same as assigning a variable a value.

\[ x = [1., 2., 3.] \]
\[ y = x \quad \# \text{Binds } y \text{ to the same object as } x \text{ is bound to, not a copy!} \]
\[ y[2] = 9. \quad \# \text{Changing the object both } x \text{ and } y \text{ point to.} \]
\[ \text{print } x \quad \# \text{Prints } [1., 2., 9.] \]

Multiple left-hand sides are allowed when the right side is a tuple.

\[ \text{apex} = (4., 5.) \]
\[ (x\_\text{apex}, y\_\text{apex}) = \text{apex} \]
\[ x\_\text{apex}, y\_\text{apex} = \text{apex} \quad \# \text{You can often omit the parens in a tuple} \]
\[ w, z = z, w \]

Same trick allowed for lists on both left and right.

\[ [x, y, z] = [1, 2, 3] \]
del is used to unbind names from objects and to remove attributes, list members.

When the name is unbound from the object, the object’s reference count goes down by 1. When the reference count reaches zero, the object is reclaimed by the "garbage collector".

```python
>>> x = [1., 2., 3]
>>> del x[1]  #
>>> print x
[1., 3.]
```

Example of deleting an attribute:
```python
y = [1., 2., 3.]
class A:
    pass
a = A()
a.x = y
del y
```

There is now no more "y", but the list object we defined is still bound to a.x.
The print statement has several variations.

- Each item is blank delimited
- Statement adds a new-line unless it ends in a comma
- C-like formatted output available using % operator:
- You can change the file to which the output goes by assigning to sys.stdout or use open/write

```python
>>> print "The answer is %d, I think. No, maybe it is %d!", (3, 4)
The answer is 3, I think. No, maybe it is 4!
```

This is actually a trick of the string class overloading the percent operator.

```python
f = open("output_1", "w")
s = "(%e, %e)\n" % (x, y)
f.write(s)
f.close()
```
The return statement can return any kind of object.

def swap (x, y):
    return y, x
w, z = swap (1, 2)

If a procedure exits without doing a return it returns an object None, which is unique in the system.
Compound statements are delimited by indentation.

- if statement
- while statement
- for statement
- try statement
- function definition
- class definition
If statements can have optional elif and else clauses.

```python
# dogyears.py by Joseph Strout
### get the original age
age = input("Enter your age (in human years): ")
print # print a blank line

### do some range checking, then print result
if age < 0:
    print "Negative age?!? I don't think so."
elif age < 3 or age > 110:
    print "Frankly, I don't believe you."
else:
    print "That's", age*7, "in dog years."
```
There are two looping clauses, for and while.

The while statement executes until its condition is false.

```python
while abs(x * x - 2.0) > 1.e-3:
    x = (x / 2.0 + 1.0 / x)
```

```python
while (1):
    z = f.readline()
    if not z:
        break
...
```

The for statement needs a list of values:

```python
for x in a_list:
    print x
```

```python
for x in range(10):
    print x  # range (10) = [0, 1, ..., 9]
```
Both loops support an optional else clause.

The else clause is executed unless the loop is exited by a break statement.

This is useful when you are looking for things:

```python
error = "Friend not found"
name = "Joe"
for n in friends:
    if name == n:
        break
else:
    raise error, name + " is not a friend."
print name, "is a friend."
```
Function definitions can use variable numbers of arguments and keyword arguments.

```python
def sundae (flavor, use_nuts, use_whipped_cream):
    ...
    sundae ("chocolate", 1, 1)

def add_to_list (list, *args):
    # args becomes a tuple containing the arguments
    for x in args:
        list.append (x)

list = []
add_to_list (list, 1)
add_to_list (list, 2, 3, 4)

def sundae (flavor, use_nuts = 1, use_whipped_cream = 1):
    ...
    sundae ("vanilla")
```

Paul F. Dubois
dubois1@lnl.gov
Each type supports its own set of operations.

lists
- `alist.append (object), len (alist), subscripting, slicing, searching`
- `x [i: j] = [x[i], x[i+1], ..., x[j-1]], lowest subscript is zero.`
- `---------------------------^^^^ surprise! It is the holes that are numbered, not the pickets.`

dictionaries
- Literally written this way:

```python
phonebook = {
    "joe": "2-4444",
    "bill": "2-4567",
    "jennifer": "2-1234"
}
dial (phonebook["jennifer"])`
- `phonebook.keys() => list of names, random order`
- `phonebook.values() => list of numbers, random order`

names = phonebook.keys()
names.sort()
for name in names:
    print name, phonebook[name]
```
Extension objects are objects, too.

Creating extensions to Python in C, we can create objects of new types. We can cause them to behave like sequences, or like dictionaries, or as objects that have attributes.

- C extensions can enforce ironclad encapsulation while classes cannot.

```python
class A:
    ...
    a = A()
    a.a_new_member_you_never_expected = 22.0
    del a.vital_piece
```

Python classes depend on good behavior and naming conventions, such as attributes whose names begin with underscores are private.

- C extensions can detect any attempt to mess with the object and refuse to support it or decide to intercept it.

- C extensions can also be functional, adding new functions but not new object types, or both.
- C extensions are often paired with Python classes that make the interface more powerful and easy to use.

```python
f = PR("pdbfile.b")
print f.x  #print object named x from the database pdbfile.b
f.close() #...if you forget this, ok, done when f is garbage collected.
```
The available built-in functions and modules represent a vast resource. Look before you build anything new.

Change prompt
Open a file
Change where it looks when it opens a file
Time things
Directories, file queries and operations
Threads, sockets, pipes, persistence, parsing, ...
Web stuff like CGI, HTML, ...
Math support

The kernel is light, the library is heavy.
The Numerical extension adds arrays stored as contiguous objects with C-speed operations.

Extension implement mostly by James Hugunin of MIT with design help from Konrad Hinsenk, Jim Fulton, Paul Dubois, and many other members of the Python Matrix-SIG.

Most arrays will arise from compiled code that is creating them, but there are some functions to create them in Python.

The most basic one is array (a_list), which turns a list into an array.

```python
from Numeric import *
x = array ([[1., 2., 3., 4.]])
f = arange(10)
z = ones(200)
w = zeros(50)
```
Each array has a shape, available as an attribute of the array.

```python
>>> x=arange(10)
>>> print x
0 1 2 3 4 5 6 7 8 9
>>> x.shape
(10,)
```

The shape can be changed by assigning to this attribute:

```python
>>> x.shape = (2,5)
>>> print x
0 1 2 3 4
5 6 7 8 9
```
Array expressions are carried out by producing a result whose components are the result of doing the operation on the corresponding elements of each operand.

This element-wise idea is extended to various cases where the shapes of the operands are not identical, and a collection of common functions that operate element-wise on arrays are provided.

```python
>>> x
10 11 12
13 14 15
16 17 18
19 20 21

>>> y
0 1 2

>>> x + y
10 12 14
13 15 17
16 18 20
19 21 23
```
A collection of common functions that operate element-wise on arrays are provided.

Thus, if two real arrays a and b have the same shape,

\[ x = \frac{a + b}{a - b} + \sin(a)^2 + \cos(b)^2 \]

is a valid expression which will be carried out at C-like speed.
Subscripting can yield individual elements or references to rows and columns

```python
>>> y = arange(12)
>>> y.shape = (4, 3)
>>> print y
0 1 2
3 4 5
6 7 8
9 10 11
>>> print y[0, 0]
0
>>> print y[1, 0]
3
>>> print y[0, 1]
1
>>> print y[0][1]
1
>>> print y[1]
3 4 5
```
The subscript $i: j$ means all indices from $i$ up to but not including $j$. An optional $:k$ can be added to indicate a stride count.

```python
>>> print y[1:3]
3 4 5
6 7 8
>>> print y[:, 1]
1 4 7 10
>>> y[:, 1] = [5, 6, 7, 8]
>>> print y
0 5 2
3 6 5
6 7 8
9 8 11
```
Broadcasting rules and special indexing operations add great power.

Boring and slow outer product:

```python
>>> a = array([1, 2, 3])
>>> b = array([10, 20])
>>> c = zeros(len(b), len(a), Integer())
>>> for i in range(len(b)):
...     for j in range(len(a)):
...         c[i, j] = b[i] * a[j]
...
>>> print c
10 20 30
20 40 60
```
Broadcasting rules provide the key

```python
>>> b.shape=(2,1)
>>> print b
10
20
>>> c = a * b
>>> print c
10 20 30
20 40 60
```

This is efficient but clumsy. The special index NewAxis is a shorthand for adding a new axis of length 1 in a given position.

```
b [:, NewAxis]
```

will be just what we need, and the outer product can be written as

```
c = a * b [:, NewAxis]
```

You can also subscript at one end even if you don’t know how many dimensions there are: `b[... , NewAxis]`
There are operations that reduce the rank

```python
>>> z
1.00000000 2.00000000 3.00000000
4.00000000 5.00000000 6.00000000
>>> add.reduce(z)  # sum the columns
5.00000000 7.00000000 9.00000000
>>> add.reduce(z,1)  # sum the rows
6.00000000 15.00000000
```
Each array object has a set of methods that can be applied to it.

These include:

- `x.equal(y)` returns an array of 1’s and 0’s of the same shape as x, indicating whether or not the corresponding elements are equal. `equal` has brothers `notEqual`, `greater`, `greaterEqual`, `less`, and `lessEqual`, and cousins `andLogical`, `orLogical`, and `notLogical` for carrying out logical operations. (Caution, Python’s normal scalar comparison operators do not work on arrays).

- `x.matrixMultiply(y)` returns the mathematical matrix product of x and y.

- There are a set of methods for producing arrays derived from the given array, such as transposes, complex conjugate, copies, concatenations, and so on.

- `x.choose(list)`, `x.take(list)`, and `x.repeat(list)` are available for more complicated needs in choosing portions of an array.

Some names and other details may change, since this is a beta release, but the basic ideas are firm.
Extending Python in C is easy.

In essence, most of the rest of the conference is about extensions.

There are tools for helping you write extensions:
- Modulator
- SWIG
- FIDL (Fortran)