

Python and Scripting for Games

Simon Kågström

Department of Systems and Software Engineering
Blekinge Institute of Technology
Ronneby, Sweden

<http://www.ipd.bth.se/ska>



Outline of Part I

- 1 Introduction
 - Motivation
 - Python Introduction
- 2 Types and Data Structures
 - Built-in Types and Type Handling
 - Data structures
- 3 Control Flow
 - If/While
 - For-loops
- 4 Functions and classes
 - Modules
 - Functions
 - Classes
 - Operator Overloading

5 Interfacing Python with C/C++

- Overview
- SWIG
- Raw Interface

6 Low-level Details

- Python Bytecode
- Memory Management

7 Motivation

- Why do we Want Scripting?
- What do we Need from Scripting?

8 Structuring

- Case studies

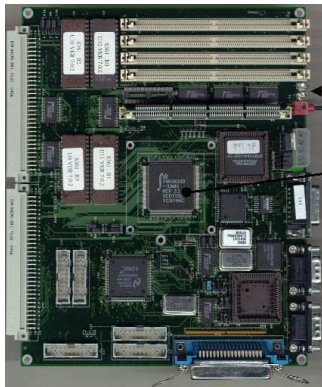
Part I

Python Introduction

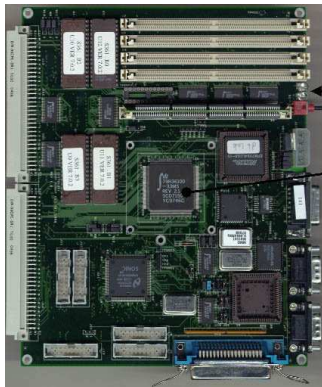
- 1 Introduction
 - Motivation
 - Python Introduction
- 2 Types and Data Structures
 - Built-in Types and Type Handling
 - Data structures
- 3 Control Flow
 - If/While
 - For-loops
- 4 Functions and classes
 - Modules
 - Functions
 - Classes
 - Operator Overloading

- Maybe C/C++ are not always the best languages?
- Example: we want a program that removes the old music on the MP3 player (pretend iTunes etc does not exist) and fill it with random new music (OGG, MP3)
- Tasks:
 - Mount the MP3 player
 - Remove old music
 - While space left:
 - Pick one random song (not repeated!), copy it to the MP3 player
 - Unmount the MP3 player again
- How many lines in C/C++? Maybe 200-400?
- How many lines in bash/awk?

- Maybe C/C++ are not always the best languages?
- Example: we want a program that removes the old music on the MP3 player (pretend iTunes etc does not exist) and fill it with random new music (OGG, MP3)
- Tasks:
 - Mount the MP3 player
 - Remove old music
 - While space left:
 - Pick one random song (not repeated!), copy it to the MP3 player
 - Unmount the MP3 player again
- How many lines in C/C++? Maybe 200-400?
- How many lines in bash/awk?
 - **40 lines**, no segfaults, no recompiles



- Another task: convert symbols to the MIPS-board format
- Tasks:
 - Read symbols and symbol names from the ELF-file
 - Write number of symbols/string table length to the outfile
 - For each symbol:
 - Write symbol address and strtab offset
 - For each string, write the string plus NULL-termination
- Not that suitable for shell-scripts: need to keep track of symbols, binary output
- C/C++: 300-400 lines?



- Another task: convert symbols to the MIPS-board format
- Tasks:
 - Read symbols and symbol names from the ELF-file
 - Write number of symbols/string table length to the outfile
 - For each symbol:
 - Write symbol address and strtab offset
 - For each string, write the string plus NULL-termination
- Not that suitable for shell-scripts: need to keep track of symbols, binary output
- C/C++: 300-400 lines?
- Python: **26 lines** (excluding usage() and option parsing), no segfaults, no recompiles

Motivation, III

```
f = os.popen("nm %s" % (infile))
for line in f: # Read all symbols
    sym = Symbol(int( line.split()[0], 16 ), line.split()[1], line.split()[2])
    strtab_len = strtab_len + len(sym.symbol) + 1
    symbol_table.append(sym)
f.close()
```

```
f = open(outfile, "wb") # Write out to file
f.write(struct.pack(">L",len(symbol_table)))
f.write(struct.pack(">L",strtab_len))
cur_offset = 0
for sym in symbol_table: # Write the symbols
    sym.strtab_offset = cur_offset
    f.write(struct.pack(">L",sym.strtab_offset))
    f.write(struct.pack(">L",sym.address))
    cur_offset = cur_offset + len(sym.symbol) + 1
for sym in symbol_table: # Write 8 unknown bytes for each symbol
    f.write(struct.pack(">L",0))
    f.write(struct.pack(">L",0))
for sym in symbol_table: # Write the strings
    f.write(sym.symbol)
    f.write(struct.pack("b", 0))
f.close()
```

Motivation, III

```
f = os.popen("nm %s" % (infile))
for line in f: # Read all symbols
    sym = Symbol(int( line.split()[0], 16 ), line.split()[1], line.split()[2])
    strtab_len = strtab_len + len(sym.symbol) + 1
    symbol_table.append(sym)
f.close()
```

```
f = open(outfile, "wb") # Write out to file
f.write(struct.pack(">L",len(symbol_table)))
f.write(struct.pack(">L",strtab_len))
cur_offset = 0
for sym in symbol_table: # Write the symbols
    sym.strtab_offset = cur_offset
    f.write(struct.pack(">L",sym.strtab_offset))
    f.write(struct.pack(">L",sym.address))
    cur_offset = cur_offset + len(sym.symbol) + 1
for sym in symbol_table: # Write 8 unknown bytes for each symbol
    f.write(struct.pack(">L",0))
    f.write(struct.pack(">L",0))
for sym in symbol_table: # Write the strings
    f.write(sym.symbol)
    f.write(struct.pack("b", 0))
f.close()
```

Motivation, III

```
f = os.popen("nm %s" % (infile))
for line in f: # Read all symbols
    sym = Symbol(int( line.split()[0], 16 ), line.split()[1], line.split()[2])
    strtab_len = strtab_len + len(sym.symbol) + 1
    symbol_table.append(sym)
f.close()
```

```
f = open(outfile, "wb") # Write out to file
f.write(struct.pack(">L",len(symbol_table)))
f.write(struct.pack(">L",strtab_len))
cur_offset = 0
for sym in symbol_table: # Write the symbols
    sym.strtab_offset = cur_offset
    f.write(struct.pack(">L",sym.strtab_offset))
    f.write(struct.pack(">L",sym.address))
    cur_offset = cur_offset + len(sym.symbol) + 1
for sym in symbol_table: # Write 8 unknown bytes for each symbol
    f.write(struct.pack(">L",0))
    f.write(struct.pack(">L",0))
for sym in symbol_table: # Write the strings
    f.write(sym.symbol)
    f.write(struct.pack("b", 0))
f.close()
```

Motivation, III

```
f = os.popen("nm %s" % (infile))
for line in f: # Read all symbols
    sym = Symbol(int( line.split()[0], 16 ), line.split()[1], line.split()[2])
    strtab_len = strtab_len + len(sym.symbol) + 1
    symbol_table.append(sym)
f.close()
```

```
f = open(outfile, "wb") # Write out to file
f.write(struct.pack(">L",len(symbol_table)))
f.write(struct.pack(">L",strtab_len))
cur_offset = 0
for sym in symbol_table: # Write the symbols
    sym.strtab_offset = cur_offset
    f.write(struct.pack(">L",sym.strtab_offset))
    f.write(struct.pack(">L",sym.address))
    cur_offset = cur_offset + len(sym.symbol) + 1
for sym in symbol_table: # Write 8 unknown bytes for each symbol
    f.write(struct.pack(">L",0))
    f.write(struct.pack(">L",0))
for sym in symbol_table: # Write the strings
    f.write(sym.symbol)
    f.write(struct.pack("b", 0))
f.close()
```

Motivation, III

```
f = os.popen("nm %s" % (infile))
for line in f: # Read all symbols
    sym = Symbol(int( line.split()[0], 16 ), line.split()[1], line.split()[2])
    strtab_len = strtab_len + len(sym.symbol) + 1
    symbol_table.append(sym)
f.close()
```

```
f = open(outfile, "wb") # Write out to file
f.write(struct.pack(">L",len(symbol_table)))
f.write(struct.pack(">L",strtab_len))
cur_offset = 0
for sym in symbol_table: # Write the symbols
    sym.strtab_offset = cur_offset
    f.write(struct.pack(">L",sym.strtab_offset))
    f.write(struct.pack(">L",sym.address))
    cur_offset = cur_offset + len(sym.symbol) + 1
for sym in symbol_table: # Write 8 unknown bytes for each symbol
    f.write(struct.pack(">L",0))
    f.write(struct.pack(">L",0))
for sym in symbol_table: # Write the strings
    f.write(sym.symbol)
    f.write(struct.pack("b", 0))
f.close()
```

- Python was invented by Guido van Rossum (Netherlands) in the early 1990s
- Interpreted, dynamically typed, object-oriented very high-level language, compiles to bytecode
- Close to pseudo code, indentation marks block structure
- Popular for application development on its own
- Used as scripting language in many applications and games
 - Gimp
 - Eve Online
 - Battlefield 2
 - ...
- Resources
 - <http://diveintopython.org> - free book
 - <http://www.thinkpython.com> - how to think like a computer scientist

- 1 Introduction
 - Motivation
 - Python Introduction
- 2 **Types and Data Structures**
 - **Built-in Types and Type Handling**
 - **Data structures**
- 3 Control Flow
 - If/While
 - For-loops
- 4 Functions and classes
 - Modules
 - Functions
 - Classes
 - Operator Overloading

Description

- Variables do not need declaring in Python
- Strings, integers, boolean and floating point types are builtin
- Multiple assignment is possible
- `print` is used to printout

Examples

```
>>> var = 128 # assignment, var is an int
>>> var = "var is now a string" # assignment, var is now a string
>>> print var
var is now a string
>>> a = 64 * 48.0 # arithmetic, as expected
>>> b = 5**2.2 # Python supports power-of directly
>>> c = "chico " + "chica" # Arithmetic with strings
>>> a,b = 1040, 130 # multiple assignment
>>> print "Numbers are %.3f, %d" % (a,b) # printf style formatting
Numbers are 1040.000, 130
```

Description

- Variables do not need declaring in Python
- Strings, integers, boolean and floating point types are builtin
- Multiple assignment is possible
- print is used to printout

Examples

```
>>> var = 128 # assignment, var is an int
>>> var = "var is now a string" # assignment, var is now a string
>>> print var
var is now a string
>>> a = 64 * 48.0 # arithmetic, as expected
>>> b = 5**2.2 # Python supports power-of directly
>>> c = "chico " + "chica" # Arithmetic with strings
>>> a,b = 1040, 130 # multiple assignment
>>> print "Numbers are %.3f, %d" % (a,b) # printf style formatting
Numbers are 1040.000, 130
```

Description

- Variables do not need declaring in Python
- Strings, integers, boolean and floating point types are builtin
- Multiple assignment is possible
- `print` is used to printout

Examples

```
>>> var = 128                                # assignment, var is an int
>>> var = "var is now a string"              # assignment, var is now a string
>>> print var
var is now a string
>>> a = 64 * 48.0                             # arithmetic, as expected
>>> b = 5**2.2                                # Python supports power-of directly
>>> c = "chico " + "chica"                   # Arithmetic with strings
>>> a,b = 1040, 130                           # multiple assignment
>>> print "Numbers are %.3f, %d" % (a,b)     # printf style formatting
Numbers are 1040.000, 130
```

Description

- Variables do not need declaring in Python
- Strings, integers, boolean and floating point types are builtin
- **Multiple assignment is possible**
- `print` is used to printout

Examples

```
>>> var = 128 # assignment, var is an int
>>> var = "var is now a string" # assignment, var is now a string
>>> print var
var is now a string
>>> a = 64 * 48.0 # arithmetic, as expected
>>> b = 5**2.2 # Python supports power-of directly
>>> c = "chico " + "chica" # Arithmetic with strings
>>> a,b = 1040, 130 # multiple assignment
>>> print "Numbers are %.3f, %d" % (a,b) # printf style formatting
Numbers are 1040.000, 130
```

Description

- `int(val[, base])` converts to an integer (of base *base*)
- `float(val)`: Corresponding to floats
- `str(val)`: Corresponding to strings (classes allow this automatically)
- `type(x)` returns the type of a variable (also classes)

Examples

```
>>> b = int("15") + int("0xf", 16) # to int, base 10 and base 16
>>> s = "hej is " + str(True)
>>> type(s)
<type 'str'>
if type(s) == str:
    print "s is a string!"
```

Description

- `int(val[, base])` converts to an integer (of base *base*)
- `float(val)`: Corresponding to floats
- `str(val)`: Corresponding to strings (classes allow this automatically)
- `type(x)` returns the type of a variable (also classes)

Examples

```
>>> b = int("15") + int("0xf", 16) # to int, base 10 and base 16
>>> s = "hej is " + str(True)
>>> type(s)
<type 'str'>
if type(s) == str:
    print "s is a string!"
```

- Python also has built-in support for data structures such as lists, dictionaries, tuples and sets
 - Lists: can be used as lists, stacks and queues
 - Dictionaries: associative memories (e.g., hash tables)
 - Tuples: e.g., coordinate-pairs
 - Sets: unordered collection of unique items
- All these can store arbitrary types
- These helps tremendously when writing more complex things

Description

- Lists are enclosed in brackets
- The lists are classes, which provide methods such as `append(obj)`, `extend(list)`, `insert(idx, obj)` and `pop()`
- The `len(list)` builtin returns the number of list items
- “Slicing” is used to get parts of lists
- The `in` operator tests if an element is in a list

Example

```
a = [64, "hej", 130, 20] # Declare a list with some elements
a.append(48)             # Append an item
print a[1:2]            # print a sliced list with item 1-2
a.sort()                # Sort the list
print 64 in a           # Check if 64 is a member of a
```

Description

- Lists are enclosed in brackets
- The lists are classes, which provide methods such as `append(obj)`, `extend(list)`, `insert(idx, obj)` and `pop()`
- The `len(list)` builtin returns the number of list items
- “Slicing” is used to get parts of lists
- The `in` operator tests if an element is in a list

Example

```
a = [64, "hej", 130, 20] # Declare a list with some elements
a.append(48)             # Append an item
print a[1:2]            # print a sliced list with item 1-2
a.sort()                # Sort the list
print 64 in a           # Check if 64 is a member of a
```

Description

- A dictionary stores key:value pairs
- Dictionaries provide fast lookup of values

Example

```
>>> d = {"a":19}           # Create a dictionary with a:19
>>> d[1040] = "Cubase"    # Key 1040 with value "Cubase"
>>> print d.values()      # Print a list of values in the dictionary
[19, 'Cubase']
>>> print d.keys()        # Print a list of keys in the dictionary
['a', 1040]
```

- 1 Introduction
 - Motivation
 - Python Introduction
- 2 Types and Data Structures
 - Built-in Types and Type Handling
 - Data structures
- 3 Control Flow**
 - If/While
 - For-loops
- 4 Functions and classes
 - Modules
 - Functions
 - Classes
 - Operator Overloading

Description

- Python supports the usual control flow primitives, **if**, **for** and **while**
- Block structure is marked by indentation
- **if** and **while** works the same way as in C

Definition

```
if CONDITION:
    STATEMENTS
elif CONDITION: # to avoid excessive indentation with else, if
    STATEMENTS
else:
    STATEMENTS

while CONDITION:
    STATEMENTS
else:
    STATEMENTS # Run once when the condition is no longer true
```

Description

- Python supports the usual control flow primitives, **if**, **for** and **while**
- Block structure is marked by indentation
- **if** and **while** works the same way as in C

Definition

```
if CONDITION:
    STATEMENTS
elif CONDITION: # to avoid excessive indentation with else, if
    STATEMENTS
else:
    STATEMENTS

while CONDITION:
    STATEMENTS
else:
    STATEMENTS # Run once when the condition is no longer true
```

Description

- **for**-loops work differently in Python compared to C
- Instead, the loop iterates over items in a list

Definition

```
for ITEM in LIST:      # Each ITEM in LIST
    STATEMENTS        # Perform something on ITEM
else:                  # Run once after the iterations, except on break
    STATEMENTS
```

Example

```
b = [48,128,130,500]
for item in b:         # Iterate over items in b
    print "Cur item:", item
for num in range(0,3): # Range generates the list [0,1,2]
    print "Nr: ", num
```

- 1 Introduction
 - Motivation
 - Python Introduction
- 2 Types and Data Structures
 - Built-in Types and Type Handling
 - Data structures
- 3 Control Flow
 - If/While
 - For-loops
- 4 Functions and classes
 - Modules
 - Functions
 - Classes
 - Operator Overloading

Description

- Python modules are libraries
- **import** is similar to **#include** in C/C++
- `sys.path` (in the `sys` module) defines where to look for modules

Definition

```
import MODULE
from MODULE import ITEM, ITEM2 # Import
```

Example

```
import math # Import the math library
print math.pi # Access "pi" in the math module
from math import pi, cos # Specific functions
print pi, cos(pi)
from math import * # Everything from math (careful!)
```

Description

- Python modules are libraries
- **import** is similar to **#include** in C/C++
- `sys.path` (in the `sys` module) defines where to look for modules

Definition

```
import MODULE
from MODULE import ITEM, ITEM2 # Import
```

Example

```
import math # Import the math library
print math.pi # Access "pi" in the math module
from math import pi, cos # Specific functions
print pi, cos(pi)
from math import * # Everything from math (careful!)
```

Description

- Python functions are defined with **def**
- Parameters can have default values
- **return** is used to return from a function

Definition

```
def NAME( LIST OF PARAMETERS ):
    "Documentation string"      # Optional
    STATEMENTS
```

Example

```
def func(x, y = 64):           # y has a default value
    return x + y, x / y       # several return values is possible

a,b = func(48)
c,d = func(48,128)
x   = func(64)                # What will x be?
```

Description

- Python functions are defined with **def**
- Parameters can have default values
- **return** is used to return from a function

Definition

```
def NAME( LIST OF PARAMETERS ):
    "Documentation string"      # Optional
    STATEMENTS
```

Example

```
def func(x, y = 64):           # y has a default value
    return x + y, x / y       # several return values is possible

a,b = func(48)
c,d = func(48,128)
x   = func(64)                # What will x be?
```

Definition

- Python classes support multiple inheritance, polymorphism etc.
- **All** member functions are virtual (in C++-terms)
- There is no data protection, members do not need to be defined
- How does Python handle structs?

Definition

- Python classes support multiple inheritance, polymorphism etc.
- **All** member functions are virtual (in C++-terms)
- There is no data protection, members do not need to be defined
- How does Python handle structs?

Example

```
class Coord:                                # Define the Coord class
    pass                                    # ... which is empty

p = Coord()                                 # New instance of the Coord class
p.x = 128                                   # Give the members some values
p.y = 48
```

Definition

- The `__init__` function is the constructor in Python
- `self` corresponds to `this` in C++
- Derived classes are specified after the class name
- Note the call of the superclass constructor

Example

```
class Entity:
    """Docstring for the class."""          # """ for multi-line strings
    def __init__(self, sprite, x=0, y=0):
        self.pos = Vector(x,y)
        ...

class MovableEntity(Entity):
    def __init__(self, sprite, vel, max_speed, x=0, y=0):
        Entity.__init__(self, sprite, x, y)
```

Definition

- The `__init__` function is the constructor in Python
- `self` corresponds to `this` in C++
- **Derived classes** are specified after the class name
- Note the call of the superclass constructor

Example

```
class Entity:
    """Docstring for the class."""          # """ for multi-line strings
    def __init__(self, sprite, x=0, y=0):
        self.pos = Vector(x,y)
        ...

class MovableEntity(Entity):
    def __init__(self, sprite, vel, max_speed, x=0, y=0):
        Entity.__init__(self, sprite, x, y)
```

Definition

- The `__init__` function is the constructor in Python
- `self` corresponds to `this` in C++
- Derived classes are specified after the class name
- **Note the call of the superclass constructor**

Example

```
class Entity:
    """Docstring for the class."""          # """ for multi-line strings
    def __init__(self, sprite, x=0, y=0):
        self.pos = Vector(x,y)
        ...

class MovableEntity(Entity):
    def __init__(self, sprite, vel, max_speed, x=0, y=0):
        Entity.__init__(self, sprite, x, y)
```

Definition

- Overriding operators is done with the `__add__`, `__mul__` (...) functions
- `__str__` is used to get the string representation of a class
- `__getitem__` is used for accesses with brackets (`[]`)

Example

```
class Vector:
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __str__(self):
        return str(self.x) + " " + str(self.y)
    def __getitem__(self, obj):
        if type(obj) != int:
            return None
            # Should raise an exception
        if obj == 0:
            return self.x
        elif obj == 1:
            return self.y
        return None
        # Should raise an exception
```

Definition

- Overriding operators is done with the `__add__`, `__mul__` (...) functions
- `__str__` is used to get the string representation of a class
- `__getitem__` is used for accesses with brackets (`[]`)

Example

```
class Vector:
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __str__(self):
        return str(self.x) + " " + str(self.y)
    def __getitem__(self, obj):
        if type(obj) != int:
            return None
            # Should raise an exception
        if obj == 0:
            return self.x
        elif obj == 1:
            return self.y
        return None
        # Should raise an exception
```

Definition

- Overriding operators is done with the `__add__`, `__mul__` (...) functions
- `__str__` is used to get the string representation of a class
- `__getitem__` is used for accesses with brackets (`[]`)

Example

```
class Vector:
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __str__(self):
        return str(self.x) + " " + str(self.y)
    def __getitem__(self, obj):
        if type(obj) != int:
            return None
            # Should raise an exception
        if obj == 0:
            return self.x
        elif obj == 1:
            return self.y
        return None
        # Should raise an exception
```

Definition

- Overriding operators is done with the `__add__`, `__mul__` (...) functions
- `__str__` is used to get the string representation of a class
- `__getitem__` is used for accesses with brackets (`[]`)

Example

```
class Vector:
    def __init__(self, x,y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)
    def __str__(self):
        return str(self.x) + " " + str(self.y)
    def __getitem__(self, obj):
        if type(obj) != int:
            return None
            # Should raise an exception
        if obj == 0:
            return self.x
        elif obj == 1:
            return self.y
        return None
        # Should raise an exception
```

Description

- Exceptions are caught with the `try/except` pair in Python
- Exceptions are classes and can be inherited from just like other classes
- `raise` raises an exception

Example

```
try:
    16 / 0
except ZeroDivisionError, detail:
    print "Ouch! Got an exception:", detail
class CorrectNumberException(Exception):
    def __str__(self):
        return "The supplied number is not one of the allowed numbers"

def testNum(num):
    if num not in [16,20,48,64,128,130,500,520,1000,1040,2000]:
        raise CorrectNumberException
```

Description

- **for** calls the `iter` function on a class to get an iterator object and then `next` to get the next item
- The `StopIteration` exception is raised when the items are up
- `__iter__` returns an iterator

Example

```
class Reverse:
    "Iterator for looping over a sequence backwards"
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

Example

```
class Entity:
    obj_id=0 # Class-global variable (static member in C++)
    def __init__(self, x=0, y=0):
        self.pos = Vector(x,y)
        self.obj_id = Entity.obj_id
        Entity.obj_id = Entity.obj_id + 1
    def draw(self):
        pass

class MovableEntity(Entity):
    def __init__(self, vel, max_speed, x=0, y=0):
        Entity.__init__(self, x, y) # Call the constructor of the base class
        self.vel = vel
        self.max_speed = max_speed
        self.mass = 10 # How heavy this object is
    def seek(self):
        if self.aim_for_target == self.pos:
            return Vector(0,0)
        desired_vel = (self.aim_for_target - self.pos).norm() * self.max_speed
        return desired_vel - self.vel
    def update(self, frame_time): # Time in fractions of a second
        accel = (self.seek() / self.mass) * frame_time
        self.vel = self.vel + accel * frame_time
        self.pos = self.pos + self.vel * frame_time

while True: # The would-be game loop
    for e in entity_list:
        e.update(frame_time)
        e.draw()
```

Example

```
class Entity:
    obj_id=0 # Class-global variable (static member in C++)
    def __init__(self, x=0, y=0):
        self.pos = Vector(x,y)
        self.obj_id = Entity.obj_id
        Entity.obj_id = Entity.obj_id + 1
    def draw(self):
        pass

class MovableEntity(Entity):
    def __init__(self, vel, max_speed, x=0, y=0):
        Entity.__init__(self, x, y) # Call the constructor of the base class
        self.vel = vel
        self.max_speed = max_speed
        self.mass = 10 # How heavy this object is
    def seek(self):
        if self.aim_for_target == self.pos:
            return Vector(0,0)
        desired_vel = (self.aim_for_target - self.pos).norm() * self.max_speed
        return desired_vel - self.vel
    def update(self, frame_time): # Time in fractions of a second
        accel = (self.seek() / self.mass) * frame_time
        self.vel = self.vel + accel * frame_time
        self.pos = self.pos + self.vel * frame_time

while True: # The would-be game loop
    for e in entity_list:
        e.update(frame_time)
        e.draw()
```

Example

```
class Entity:
    obj_id=0 # Class-global variable (static member in C++)
    def __init__(self, x=0, y=0):
        self.pos = Vector(x,y)
        self.obj_id = Entity.obj_id
        Entity.obj_id = Entity.obj_id + 1
    def draw(self):
        pass

class MovableEntity(Entity):
    def __init__(self, vel, max_speed, x=0, y=0):
        Entity.__init__(self, x, y) # Call the constructor of the base class
        self.vel = vel
        self.max_speed = max_speed
        self.mass = 10 # How heavy this object is
    def seek(self):
        if self.aim_for_target == self.pos:
            return Vector(0,0)
        desired_vel = (self.aim_for_target - self.pos).norm() * self.max_speed
        return desired_vel - self.vel
    def update(self, frame_time): # Time in fractions of a second
        accel = (self.seek() / self.mass) * frame_time
        self.vel = self.vel + accel * frame_time
        self.pos = self.pos + self.vel * frame_time

while True: # The would-be game loop
    for e in entity_list:
        e.update(frame_time)
        e.draw()
```

Example

```

class Entity:
    obj_id=0 # Class-global variable (static member in C++)
    def __init__(self, x=0, y=0):
        self.pos = Vector(x,y)
        self.obj_id = Entity.obj_id
        Entity.obj_id = Entity.obj_id + 1
    def draw(self):
        pass

class MovableEntity(Entity):
    def __init__(self, vel, max_speed, x=0, y=0):
        Entity.__init__(self, x, y) # Call the constructor of the base class
        self.vel = vel
        self.max_speed = max_speed
        self.mass = 10 # How heavy this object is
    def seek(self):
        if self.aim_for_target == self.pos:
            return Vector(0,0)
        desired_vel = (self.aim_for_target - self.pos).norm() * self.max_speed
        return desired_vel - self.vel
    def update(self, frame_time): # Time in fractions of a second
        accel = (self.seek() / self.mass) * frame_time
        self.vel = self.vel + accel * frame_time
        self.pos = self.pos + self.vel * frame_time

while True: # The would-be game loop
    for e in entity_list:
        e.update(frame_time)
        e.draw()

```

Example

```
class Entity:
    obj_id=0 # Class-global variable (static member in C++)
    def __init__(self, x=0, y=0):
        self.pos = Vector(x,y)
        self.obj_id = Entity.obj_id
        Entity.obj_id = Entity.obj_id + 1
    def draw(self):
        pass

class MovableEntity(Entity):
    def __init__(self, vel, max_speed, x=0, y=0):
        Entity.__init__(self, x, y) # Call the constructor of the base class
        self.vel = vel
        self.max_speed = max_speed
        self.mass = 10 # How heavy this object is
    def seek(self):
        if self.aim_for_target == self.pos:
            return Vector(0,0)
        desired_vel = (self.aim_for_target - self.pos).norm() * self.max_speed
        return desired_vel - self.vel
    def update(self, frame_time): # Time in fractions of a second
        accel = (self.seek() / self.mass) * frame_time
        self.vel = self.vel + accel * frame_time
        self.pos = self.pos + self.vel * frame_time

while True: # The would-be game loop
    for e in entity_list:
        e.update(frame_time)
        e.draw()
```

Description

- Opening and closing files are similar to C
- There are more convenience functions for reading and writing
- Pickling allows saving and loading objects and classes automatically
 - Loading and saving objects becomes *much* easier!

Example

```
f = open("files.py", "r")
for line in f.readlines():
    print line
f.close()
```

```
f = open("pickle", "w")
e = Entity("meboo", 16, 0)
pickle.dump(e, f)
f.close()
...
e = pickle.load(f)
```

Description

- Opening and closing files are similar to C
- There are more convenience functions for reading and writing
- Pickling allows saving and loading objects and classes automatically
 - Loading and saving objects becomes *much* easier!

Example

```
f = open("files.py", "r")
for line in f.readlines():
    print line
f.close()
```

```
f = open("pickle", "w")
e = Entity("meboo", 16, 0)
pickle.dump(e, f)
f.close()
...
e = pickle.load(f)
```

Description

- Opening and closing files are similar to C
- There are more convenience functions for reading and writing
- Pickling allows saving and loading objects and classes automatically
 - Loading and saving objects becomes *much* easier!

Example

```
f = open("files.py", "r")
for line in f.readlines():
    print line
f.close()
```

```
f = open("pickle", "w")
e = Entity("meboo", 16, 0)
pickle.dump(e, f)
f.close()
...
e = pickle.load(f)
```

Part II

Python and C/C++

5 Interfacing Python with C/C++

- Overview
- SWIG
- Raw Interface

6 Low-level Details

- Python Bytecode
- Memory Management

Interfacing Python with C

- There are two ways of interfacing Python with C/C++: **manually** or through an **interface-generator**
- Manual implementation gives you more control, but is much more tedious
 - Handling of reference counting etc., is manual
- There are many interface generators for Python: SWIG, boost, CXX etc.
- Boost: C++-only, Python only, easier to call Python from C++
 - Used by Civilization IV
- SWIG: Supports C and C++, bindings for many languages (Python, Lua, ...). Calling Python from C/C++ is slightly harder.
 - Used by, e.g., the Ogre 3D engine

Description

- `%module NAME` defines a new module (namespace)
- `{` and `}` contains things included verbatim in the wrapper file
- The rest of the file defines the interface to export
- Running **swig -python** on this file generates a `testing_wrap.c` and a `testing.py` file
- SWIG maps basic types (ints, strings, floats) to corresponding types in Python - everything else is handled through pointers (references)

Example (example.i)

```
%module example
%{
#include "game.h"                /* This is included in the generated wrapper */
%}
int testing(int a, int b);
```

Description

- `%module NAME` defines a new module (namespace)
- `{` and `}` contains things included verbatim in the wrapper file
- The rest of the file defines the interface to export
- Running `swig -python` on this file generates a `testing_wrap.c` and a `testing.py` file
- SWIG maps basic types (ints, strings, floats) to corresponding types in Python - everything else is handled through pointers (references)

Example (example.i)

```
%module example
%{
#include "game.h"                /* This is included in the generated wrapper */
%}
int testing(int a, int b);
```

Description

- `%module NAME` defines a new module (namespace)
- `{` and `}` contains things included verbatim in the wrapper file
- The rest of the file defines the interface to export
- Running `swig -python` on this file generates a `testing_wrap.c` and a `testing.py` file
- SWIG maps basic types (ints, strings, floats) to corresponding types in Python - everything else is handled through pointers (references)

Example (example.i)

```
%module example
%{
#include "game.h"                /* This is included in the generated wrapper */
%}
int testing(int a, int b);
```

Description

- `%module NAME` defines a new module (namespace)
- `{` and `}` contains things included verbatim in the wrapper file
- The rest of the file **defines the interface to export**
- Running **swig -python** on this file generates a `testing_wrap.c` and a `testing.py` file
- SWIG maps basic types (ints, strings, floats) to corresponding types in Python - everything else is handled through pointers (references)

Example (example.i)

```
%module example
%{
#include "game.h"                /* This is included in the generated wrapper */
%}
int testing(int a, int b);
```

Description

- C++ classes can be wrapped by SWIG
- Most functionality “just works” with the wrapped class
- You can derive Python classes from C++ classes
- Overriding member functions is also possible

Example (C)

```
class Entity
{
public:
    Entity(char *sprite, int x, int y);
    ~Entity();

    void draw(void);
    void update(double seconds);

    double x,y;
};
```

Example (Python)

```
class PythonEntity(Entity):
    def draw(self):
        print "draw: Python: ", self.x, self.y

e = Entity("bruce", 0, 0)
b = PythonEntity("monty", 64, 64) # Uses the C++ __init__

e.update(0.3) # Will call C++
e.draw()
b.update(0.3) # Will call C++
b.draw() # Will call Python
```

Description

- C++ classes can be wrapped by SWIG
- Most functionality “just works” with the wrapped class
- You can derive Python classes from C++ classes
- Overriding member functions is also possible

Example (C)

```
class Entity
{
public:
    Entity(char *sprite, int x, int y);
    ~Entity();

    void draw(void);
    void update(double seconds);

    double x,y;
};
```

Example (Python)

```
class PythonEntity(Entity):
    def draw(self):
        print "draw: Python: ", self.x, self.y

e = Entity("bruce", 0, 0)
b = PythonEntity("monty", 64, 64) # Uses the C++ __init__

e.update(0.3) # Will call C++
e.draw()
b.update(0.3) # Will call C++
b.draw() # Will call Python
```

Description

- C++ classes can be wrapped by SWIG
- Most functionality “just works” with the wrapped class
- You can derive Python classes from C++ classes
- **Overriding member functions** is also possible

Example (C)

```
class Entity
{
public:
    Entity(char *sprite, int x, int y);
    ~Entity();

    void draw(void);
    void update(double seconds);

    double x,y;
};
```

Example (Python)

```
class PythonEntity(Entity):
    def draw(self):
        print "draw: Python: ", self.x, self.y

e = Entity("bruce", 0, 0)
b = PythonEntity("monty", 64, 64) # Uses the C++ __init__

e.update(0.3) # Will call C++
e.draw()
b.update(0.3) # Will call C++
b.draw() # Will call Python
```

Description

- SWIG generates two files on compilation of an interface
 - C/C++ wrapper file - proxy class
 - Python wrapper file - Python to C/C++

Example (example_wrap.cc)

```
static PyObject *_wrap_Entity_draw(PyObject *, PyObject *args) {
    PyObject *resultobj;
    Entity *arg1 = (Entity *) 0 ;
    PyObject * obj0 = 0 ;

    if(!PyArg_ParseTuple(args,(char *)"O:Entity_draw",&obj0)) goto fail;
    SWIG_Python_ConvertPtr(obj0, (void **)&arg1, SWIGTYPE_p_Entity, SWIG_POINTER_EXCEPTION | 0);
    if (SWIG_arg_fail(1)) SWIG_fail;
    (arg1)->draw();

    Py_INCREF(Py_None); resultobj = Py_None;
    return resultobj;
fail:
    return NULL;
}
```

Description

- Including `Python.h` and linking with Python libraries is necessary
- You might need to set the include path
- Almost all functions in the Python/C API deals with `PyObject` pointers
- Reference counting must be handled from C manually

Example

```
Py_Initialize();
PyRun_SimpleString("import sys");
PyRun_SimpleString("sys.path.append('')");
...
Py_Finalize();
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if (!p_args)
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if (!p_args)
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

Python Code

```
def adder(a,b):
    return a + b
```

Calling through wrapper

```
/* Open the module */
p_module = PyImport_ImportModule("script");

if (!(p_dict = PyModule_GetDict(p_module)))
    python_error("Dictionary");

/* Call the Python function */
p_res = call_adder(p_dict, 5, 6);
if (p_res) {
    printf("res: %ld\n", PyInt_AsLong(p_res));
    Py_DECREF(p_res);
}
Py_DECREF(p_dict);
Py_DECREF(p_module);
```

C implementation

```
PyObject *call_adder(PyObject *p_dict, int a, int b) {
    PyObject *p_func = PyDict_GetItemString(p_dict, "adder");
    /* Borrowed reference */

    if (p_func && PyCallable_Check(p_func)) {
        PyObject *p_args;
        PyObject *p_value;

        /* Construct objects from the format string below */
        p_args = Py_BuildValue("ii", a, b);
        if ( !p_args )
            python_error("Cannot convert argument");

        p_value = PyObject_CallObject(p_func, p_args);
        Py_DECREF(p_args); /* We are done with Args */

        return p_value;
    }

    return NULL;
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsubmodule(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsubmodule();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "..." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsubber(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsubber();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsubber(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsubber();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsmodule(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsmodule();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsubber(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsubber();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

C, function

```
static int subber(int a, int b) {
    return a - b;
}
```

Python Code

```
import seemodule

print seemodule.subber(20,16)
```

C, setup

```
static PyMethodDef module_methods[] = {
    { "subber", HAPI_subber, METH_VARARGS, "." },
    { NULL, NULL, 0, NULL }
};

/* This function MUST be named like this */
PyMODINIT_FUNC initsubber(void) {
    Py_InitModule("seemodule", module_methods);
}
```

C wrapper

```
PyObject *HAPI_subber(PyObject *self, PyObject *args) {
    int ret, a, b;

    /* Parse the tuple we got */
    if ( !PyArg_ParseTuple(args, "ii", &a, &b) )
        return NULL;
    ret = subber(a, b);

    return Py_BuildValue("i", ret);
}
```

C, main

```
int main(int argc, char *argv[]) {
    Py_Initialize();

    initsubber();

    PyRun_SimpleString("import sys");
    PyRun_SimpleString("sys.path.append('.')");
    PyRun_SimpleString("import main");

    Py_Finalize();
}
```

5 Interfacing Python with C/C++

- Overview
- SWIG
- Raw Interface

6 Low-level Details

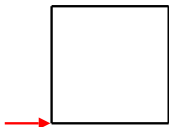
- Python Bytecode
- Memory Management

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

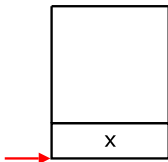
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

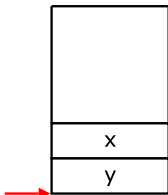
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

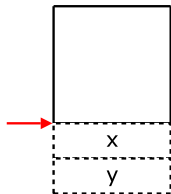
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

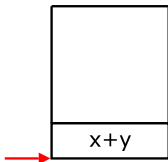
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

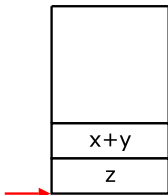
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

```
0 LOAD_FAST           0 (x)  
3 LOAD_FAST           1 (y)  
6 BINARY_ADD  
7 LOAD_FAST           2 (z)  
10 LOAD_FAST          0 (x)  
13 BINARY_ADD  
14 BUILD_TUPLE        2  
17 RETURN_VALUE  
18 LOAD_CONST         0 (None)  
21 RETURN_VALUE
```

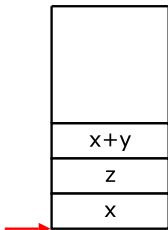
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

```
0 LOAD_FAST          0 (x)  
3 LOAD_FAST          1 (y)  
6 BINARY_ADD  
7 LOAD_FAST          2 (z)  
10 LOAD_FAST         0 (x)  
13 BINARY_ADD  
14 BUILD_TUPLE       2  
17 RETURN_VALUE  
18 LOAD_CONST         0 (None)  
21 RETURN_VALUE
```

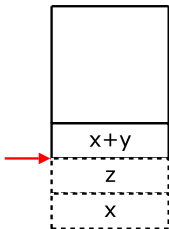
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

```
0 LOAD_FAST          0 (x)  
3 LOAD_FAST          1 (y)  
6 BINARY_ADD  
7 LOAD_FAST          2 (z)  
10 LOAD_FAST         0 (x)  
13 BINARY_ADD  
14 BUILD_TUPLE       2  
17 RETURN_VALUE  
18 LOAD_CONST        0 (None)  
21 RETURN_VALUE
```

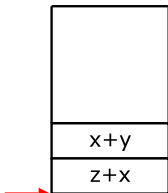
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

```
0 LOAD_FAST           0 (x)  
3 LOAD_FAST           1 (y)  
6 BINARY_ADD  
7 LOAD_FAST           2 (z)  
10 LOAD_FAST          0 (x)  
13 BINARY_ADD  
14 BUILD_TUPLE        2  
17 RETURN_VALUE  
18 LOAD_CONST         0 (None)  
21 RETURN_VALUE
```

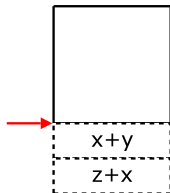
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

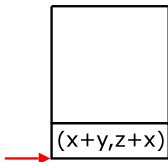
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

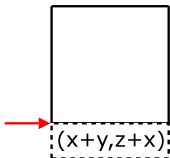
Python Bytecode

Description

- Python bytecode is stack-based, references pushed on the stack

Python function

```
def func3(x, y = 64, z = 128):  
    return x + y, z + x
```



Bytecode

0	LOAD_FAST	0 (x)
3	LOAD_FAST	1 (y)
6	BINARY_ADD	
7	LOAD_FAST	2 (z)
10	LOAD_FAST	0 (x)
13	BINARY_ADD	
14	BUILD_TUPLE	2
17	RETURN_VALUE	
18	LOAD_CONST	0 (None)
21	RETURN_VALUE	

Garbage collection and reference counting

- Python uses reference counting
 - A count is increased for each new reference to an object, decreased when it is released
 - However, with cycles, reference counting is not enough
- Garbage collection is used for cycles in references
- Garbage collection is known to be a problem for real-time performance
 - Python therefore provides a means to control the garbage collection in the gc module

Part III

Scripting for Games

7 Motivation

- Why do we Want Scripting?
- What do we Need from Scripting?

8 Structuring

- Case studies

Why Game Scripting?

- Martin Bronlow, “Game Programming Golden Rules”:
Non-programmers should be able to add content to a game
- Scripting allows the game engine to become more extensible
- ... but performance can be a problem with interpreted languages

Martin Bronlow, “Game Programming Golden Rules”:

- Easy to understand and create
- Dynamically loaded
- Safe, not crash the game
- Multitasking
- Debuggable
- Expandable
- Not be able to break the game engine

- 7 Motivation
 - Why do we Want Scripting?
 - What do we Need from Scripting?
- 8 Structuring
 - Case studies

- A game implemented with scripting can be implemented in different ways
- One way is to use the scripting system as an add-on to the game engine
 - The game is still centered around the (probably) C++ implementation
 - Scripting defines behavioral patterns etc.
- Another way is to center the implementation around the scripting implementation
 - The C++ implementation is then a “library”
 - Only performance-critical parts are implemented in C++

Description

- Harry Kalogirou, Sylphis3D Game Engine (uses *Stackless Python*)
 - <http://harkal.sylphis3d.com/2005/08/10/multithreaded-game-scripting-with-stackless-python/>
- “The game engine is the operating system of the game”
- Actions / events

Example

```
def close(self):
    try:
        self.setVelocity(self.mMoveVelocity)
        self.sleep(self.mCloseMoveTime)
    except CCollision, col:
        print "Oups.. sorry", col.name
    self.setVelocity(CVector3.ZERO)
```

Humongous Entertainment (now Atari)

- Bruce Dawson (GDC 2002)
- Humongous used SCUMM before
 - Well-suited to adventure games
 - Not very general
- Humongous uses Python for the main game engine, using C++ for performance-critical tasks
- They use a custom interface generator, similar to SWIG and also wrote an in-house debugger
- Garbage collection is done at the end of each level and they use a custom memory allocator for Python
- Humongous found that Python increased productivity with no noticeable performance loss

Quiz: Can you find the hidden code in the Python examples?

Questions?