Lectures 2

Python Basics

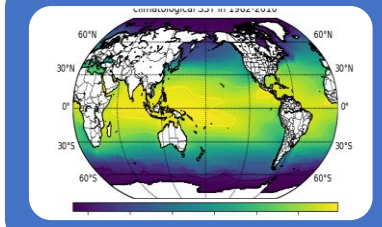
- Background
- Installation & setup
- Python Language

Python Advanced

- I/O & Exceptions Handling
- Modules & Packages
- Object-Oriented Programming in Python

Python for Scientific Computation

- Array computation with Numpy
- Scipy
- Draw common 2D figures with Matplotlib



Python for Oceanography

- Read/Write netCDF files
- Draw data on maps with basemap

Input/Output

Basic I/O: input() & print()

```
In [53]: def sum():
    ...:     ret = 0
    ...:     while True :
    ...:         number=input("input a number to get accumulated summation or q to exit:")
    ...:         if number == "q":
    ...:             print("the summation is: " + str(ret))
    ...:             return
    ...:         ret += int(number)
    ...:

In [54]: sum()

input a number to get accumulated summation or q to exit:45

input a number to get accumulated summation or q to exit:-90

input a number to get accumulated summation or q to exit:q
the summation is: -45
```

File I/O: opening and closing a file

- **Python use the file object to read/write files**
- **function open() returns a file object with different modes**
  - file = open(file_path, mode)
  - file_path: file name to be opened
  - mode:
    - ‘r’: open the file for reading
    - ‘w’: open the file for writing (an existing file with the same name will be erased)
    - ‘a’: open the file for appending
    - ‘b’: open the file in binary mode instead of text mode

    file = open(“work.log”, “w”)

    … …

    file.close()
- **Remember to close a file when we are done: file.close()**
- **Recommend to use with open() as statement when dealing with file objects**

    with open(“work.log”) as file:

        read_data = file.read()

File I/O: reading a file

- **Methods for a file object**

```
In [370]: file = open("work.log", "r+")

In [371]: file.
file.buffer     file.fileno          file.newlines   file.seekable
file.close      file.flush           file.read       file.tell
file.closed     file.isatty          file.readable   file.truncate
file.detach     file.line_buffering  file.readline   file.writable
file.encoding   file.mode            file.readlines  file.write
file.errors     file.name            file.seek       file.writelines
```

- **Reading a file:**
  - **read(size)**: reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode)
  - **readline()**: reads a single line from the file per call
  - **readlines()**: reads all the lines and returned as a list (split at newline)
  - Another fast option to loop all the lines

```
with open('f.txt', 'r') as f:
    for line in f:
        print line
```

File I/O: writing a file

- **file.write(string)** writes the contents of string to the file, returning the number of characters written

```
>>> f.write('This is a test\n')
15
```

- Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them

```
>>> value = ('the answer', 42)
>>> s = str(value)  # convert the tuple to string
>>> f.write(s)
18
```

Exception Handling

Exceptions during code execution

- What if the file doesn't exist when we want to open a file?

```
In [29]: def cat(file_path):
    ...:     with open(file_path) as file:
    ...:         print(file.read())
    ...:
    ...:

In [30]: cat("work.log")
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-30-bd66642317c2> in <module>()
----> 1 cat("work.log")

<ipython-input-29-b6be97e35ec2> in cat(file_path)
      1 def cat(file_path):
----> 2     with open(file_path) as file:
      3         print(file.read())
      4
      5

FileNotFoundError: [Errno 2] No such file or directory: 'work.log'
```

Exceptions handling using **try** – **except** block

```
In [35]: def cat(file_path):
   ...:     try:
   ...:         with open(file_path) as file:
   ...:             print(file.read())
   ...:     except FileNotFoundError:
   ...:         print("[Error] file " + file_path + " doesn't exist!")
   ...:     except:
   ...:         print("[Error] Unexpected error")
   ...:         raise
   ...:     else:
   ...:         print("[Success]")
   ...:     finally:
   ...:         print("This gets executed no matter what")
   ...:

In [36]: cat("work.log")
[Error] file work.log doesn't exist!
This gets executed no matter what
```

Common Python Exceptions

Exceptions	Description
ImportError	Raised when Python cannot find the module/package
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
IOError	Raised when the file cannot be opened
ValueError	Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys
IndexError	Raised if index to sequences is out of bounds
ZeroError	Raised when any division operator used with 0 as divisor

- Full list of Python built-in exceptions can be found in this document

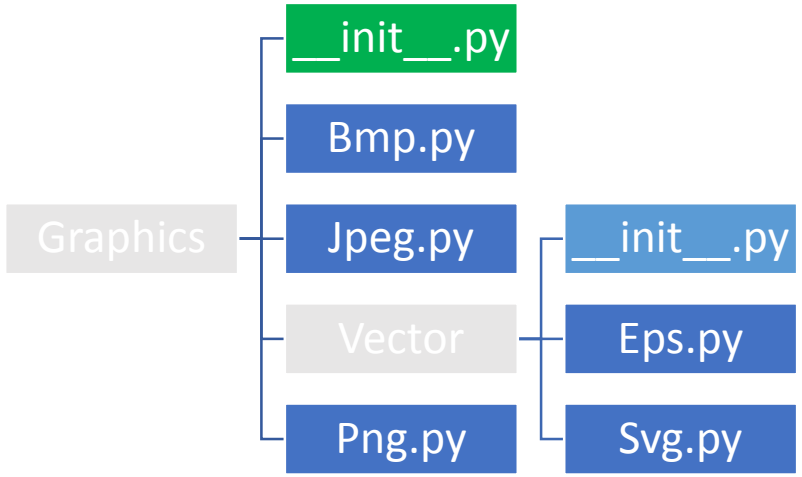
Modules & Packages

Modules & Packages: What & Why ?

- **What**
  - **A Module is a file(.py etc.) which defines functions and variables**
  - **A package is a directory which includes a collection of modules and a __init__.py file**
- **Why**
  - **Code reuse**
    - Routines can be called multiple times within a program
    - Routines can be used from multiple programs
  - **Namespace partitioning**
    - Group data together with functions used for that data
  - **Implementing shared services or data**
    - Can provide global data structure that is accessed by multiple subprograms

Modules & Packages: How ?

- **Package structure sample:**



- ***Graphics/__init__.py:***
- ***__all__ = [“Bmp”, “Jpeg”, “Vector”, “Png”]***

```
import Graphics
image = Graphics.Bmp.load(“image.bmp”)


import Graphics.Vector.Eps as EPS
image = EPS.load(“image.eps”)


from Graphics import *
image = Jpeg.load(“image.jpeg”)
```

- **Items are imported using from and import**
- **Modules/packages are namespaces**

Overview of Python's standard library & 3rd modules

- ***The Python Standard Library***
- ***String services***
  - ***string, re, StringIO, struct***
- ***Data Types***
  - ***datetime, collections, decimal***
- ***Numeric & Mathematical modules***
  - ***math, decimal, random,***
- ***File & Directory Handling***
  - ***os.path, fileinput, csv,*** tempfile
- ***OS Services***
  - ***os, io, logging, time***
- ***Debugging & Profiling***
  - ***bdb, pdb***
- ***Development Tools***
  - ***pydoc, doctest, unittest***
- ***Data Persistence***
  - ***pickle, dbm, sqlite3***
- ***IPC & Networking***
  - ***subprocess, multiprocessing***
- ***Internet Data Handling***
- ***Structured Markup Processing Tools***
- ***Python compiler package***
- ***…***

common used 3rd-party modules

- ***Advanced math, science & engineering***
  - ***NumPy:*** Matlab-ish fundamental package for scientific computing with Python
  - ***Pandas:*** high performance and easy to use data analysis tools
  - ***SciPy:*** a library of algorithms and mathematical tools for python
  - ***SymPy:*** algebraic evaluation, differentiation, expansion, etc.
- ***Data analyzing***
  - ***Matplotlib:*** numerical plotting library
  - ***Altair***: a declarative statistical visualization library based on Vega
- ***Networking***
  - ***requests*** -- if you need to do *any* http requests at all, use this
- ***Web***
  - ujson -- faster than both simplejson and the built-in json modules, handy if you work with lots of/big JSON blobs
  - BeautifulSoup -- for webscraping and/or parsing potentially malformed HTML
  - ***Urlparse – parse url***
  - lxml -- always use this for working with XML data
- ***Development & deploye***
  - nose – a testing framework for python. Must have for testing driven development.
  - docopt -- if you're writing command-line scripts/tools, use this over optparse/argparse
  - ***IPython: interactive python***
  - ***Virtualenv: a tool to create isolated Python environments.***
- ***Image & Graphics***
  - ***Pillow:*** A friendly fork of PIL (Python Imaging Library). It is more user friendly than PIL and is a must have for anyone who works with images.
- ***NLP***
  - ***nltk:*** Natural Language Toolkit

Object-Oriented Programming(OOP) in Python

OOP (Object-Oriented Programing) in Python

- **A software item that contains variables and methods**
- **OOP focuses on**
  - **Encapsulation**
    - Dividing the code into a public interface, and a private implementation of the interface
  - **Inheritance**
    - The ability to create subclasses that contain specializations of their parents
  - **Polymorphism**
    - The ability to overload method of parents so that they have appropriate behavior based on their context

OOP in Python(1): class definition & object instantiation

- **Class definition syntax:**

  class subclass[(superclass)]**:**

  [attributes and methods]

- **Object instantiation syntax:**

  object = class()

- **Attributes and methods invoke:**

  object.attribute

  object.method()

OOP in Python(2): a class sample

```
class Person:
    ''' description about class Person
    >>> jim = Person("jim")
    '''
    # define an constructor
    def __init__(self, name, age=18):
        # public instance attribute
        self.name = name
         # private instance attribute
        self.__age = age
    def setAge(self, value):
        self.__age = value
        tempAge = value # stack variable
    def getAge(self):
        return self.__age
    def info(self):
        return self.name, self.__age
```

```
>>> tom = Person("tom", 20)
>>> tom.name
'tom'
>>> tom.__age
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute '__age'
>>> tom.getAge()
20
>>> tom.setAge(30)
>>> tom.getAge()
30
>>> tom.tempAge
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Person' object has no attribute 'tempAge'
>>> isinstance(tom, Person)
True
```

- **No need to declare the attributes in class**
- **The first parameter is similar to "this" pointer in C++**
- **Some concepts in python OOP: class, attribute, function, method**

OOP in Python(3): class attribute VS. instance attribute

```
class Person:
    count = 0 # define a public class attribute
    # define the constructor
    def __init__(self, name, age=18):
        # public instance attribute
        self.name = name
        # private instance attribute
        self.__age = age
        Person.count += 1
    def setAge(self, value):
        self.__age = value
    def getAge(self):
        return self.__age
    def info(self):
        return self.name, self.__age
```

```
>>> Person
<class '__main__.Person'>
>>> Person.count
0
>>> jim = Person("jim")
>>> jim.info()
('jim', 18)
>>> Person.count
1
>>> jim
<__main__.Person object at 0x2a992729b0>
>>> jim.count
1
>>> tom = Person("tom", 30)
>>> Person.count
2
```

- **Class attributes:** Variables defined in the class definition are class attributes; they are shared by instances
- **Instance attributes:** can be set in a method with *self.name = value*

OOP in Python(4): class attribute VS. instance attribute

```
class Person:
    count = 0 # define a public class attribute
    # define the constructor
    def __init__(self, name, age=18):
        # public instance attribute
        self.name = name
        self.__age = age
        Person.count += 1
    def setAge(self, value):
        self.__age = value
    def getAge(self):
        return self.__age
    def info(self):
        return self.name, self.__age
```

```
>>> jim = Person("jim")
>>> Person.count
1
>>> tom = Person("tom", 30)
>>> Person.count
2
>>> Person.country = "China"
>>> jim.country
'China'
>>> tom.country
'China'
>>> jim.country = "USA"
>>> jim.country
'USA'
>>> Person.country
'China'
>>> tom.country
'China'
```

- Both class attribute & instance attribute can be added during runtime
- Instance attribute will hides a class attribute with the same name

OOP in Python(5): special attributes in Python

Except for user-defined class attributes in Python, class has some special attributes. They are provided by object module.

Attributes Name	Description
__dict__	Dict variable of class name space
__doc__	Document reference string of class
__name__	Class name
__module__	Module name consisting of class
__base__	The tuple including all the superclasses

OOP in Python(6): instance method, class method, and static method

```
class Person:
    count = 0 # define a public class attribute
    # define the constructor
    def __init__(self, name, age=18):
        self.name = name
        self.__age = age
        Person.count += 1
    # define an instance method
    def getAge(self):
        return self.__age
    @classmethod
    def getCount(cls):
        return cls.count
    @staticmethod
    def isAnimal():
        return True
```

```
>>> Person.count
0
>>> jim = Person("jim")
>>> Person.count
1
>>> tom = Person("tom", 30)
>>> Person.getCount()
2
>>> jim.getCount()
2
>>> Person.isAnimal()
True
```

- **Instance method:** the first parameter is an instance object. **The most used method in classes**
- **Class method:** the first parameter is the class object
- **Static method:** no pre-defined parameters by Python

OOP in Python(7): special methods in Python

If the class implement some special methods, then its instances can used in various scenarios.

Methods Name	Usage	Description
__init__(self[,args…])	obj = className(args)	Constructor (with any optional arguments)
__del__(self)	*del obj*	Destructor, deletes an object
__bool__(self)	bool(obj)	To support if x: …
__hash__(self)	hash(obj)	Can be used as key of dict, or used as element of set
__str__(self)	str(x)	Printable string representation
… …		

OOP in Python(8): special methods in Python

Methods Name	Usage	Methods Name	Usage
__abs__(self)	abs(x)	__float__(self)	Float(x)
__add__(self, other)	*x + y*	__pos__(self)	+x
__iadd__(self, other)	x += y	__neg__(self)	-x
__radd__(self, other)	y + x	__mod__(self, other)	X%y
__sub__(self, other)	x – y	__truediv(self, other)	x/y
__pow__(self, other)	X ** y	__and__(self, other)	X&y
__xor__(self, other)	X ^ Y	__rshift__(self, other)	X>>y
__lshift__(self, other)	X << y	__invert__(self)	~x
… …			

OOP in Python(9): Inheritance & Polymorphism

```
class Person:
    count = 0
    def __init__(self, name, age=18):
        self.name = name
        self.__age = age
        Person.count += 1
    def info(self):
        return self.name, self.__age
class Employee(Person):
    def __init__(self, name, age, employer="DJEL"):
        super().__init__(name, age)
        self.employer = employer
    def info(self):
        return super().info(), self.employer
```

```
>>> issubclass(Employee, Person)
True
>>> issubclass(Person, object)
True
>>> issubclass(Person, Person)
True
>>> persons = [Person("jim"), Employee("tom",
25), Employee("kate", 30, "IBM")]
>>> for person in persons:
...     print(person.info())
...
('jim', 18)
(('tom', 25), 'DJEL')
(('kate', 30), 'IBM')
```

- subclass inherits all the public attributes and methods from super class
- If __init__() not defined in subclass, a default __init__() will be provided by the system and the __init__() of the first super class will be called

OOP in Python(10): Multiple Inheritance

- **A class definition with multiple base classes looks as follows:**

```
class DerivedClass(Base1, Base2, Base3 …)
        <statement-1>
        <statement-2>
        … …
```

- **The only rule necessary to explain the semantics is the *resolution rule* used for class attribute references. This is *depth-first, left-to-right*.**
  - **if an attribute is not found in DerivedClass, it is searched in Base1, then recursively in the super classes of Base1**
  - **and only if it is not found there, it is searched in Base2, and so on.**

OOP in Python (11): reference & exercise

- References:
  - https://docs.python.org/3/reference/compound_stmts.html#class-definitions
  - http://anandology.com/python-practice-book/object_oriented_programming.html#classes-and-objects
  - http://lgiordani.com/blog/2014/08/20/python-3-oop-part-1-objects-and-types/#.VepnnPmqqko
  - http://www.tutorialspoint.com/python/python_classes_objects.htm
- Exercise 2
  - 2.1 Extend your sort function in Exercise 1 to be a package so that can be imported in other Python files.
  - 2.2 Implement a **stack** class which supports general methods such as push(item), pop(), etc. (Hint: you could use the basic data type in Python)

Interacting between Python & C

- Python call C
- C call Python (C API)

Python Programming/Extending with C

- **Wrapping:** making your code (primarliy C/C++) available from Python
  - **Python C API**
  - **Ctypes**
  - **SWIG**
  - **Cython:** probably the most advanced one and the one you should consider using first
  - **… …**

- Overall, extending the Python is not a easy job.
  - All of these techniques may crash (segmentation fault) the Python interpreter, which is (usually) due to bugs in the C code.
  - You will need a C compiler for most of the examples.

Python C API (CPython): Overview

- **Advantages**
  - Requires no additional libraries
  - Lots of low-level control
  - Entirely usable from C++

- **Disadvantages**
  - May requires a substantial amount of effort
  - Much overhead in the code
  - Must be compiled
  - High maintenance cost
  - No forward compatibility across Python versions as C-Api changes

Python C API (CPython): wrap c codes in .c file

```
#include <Python.h>
#include <math.h>

/*  wrapped cosine function */
static PyObject* cos_func(PyObject* self, PyObject* args)
{
   double value;
   double answer;

   /*  parse the input, from python float to c double */
   if (!PyArg_ParseTuple(args, "d", &value))
      return NULL;

   /* call cos from libm */
   answer = cos(value);

   /*  construct the output from cos, from c double to python float */
   return Py_BuildValue("f", answer);
}

/*  define functions in module */
static PyMethodDef CosMethods[] =
{
    {"cos_func",          /* char* , name of the method */
     cos_func,            /* PyCFunction, pointer to the C implemenation */
     METH_VARARGS,  /* flat bit indicating how the call should be constructed */
     "evaluate the cosine“                        /* pointer to the contents of the docstring */
    },
    {NULL, NULL, 0, NULL}
};

/* Create the module */
static struct PyModuleDef cosModule =
{
   PyModuleDef_HEAD_INIT,
   "cosModule",       /* name of module */
   "no description",   /* module documentation, may be NULL*/
   -1,              /* size of per-interpreter state of the module, or -1 if the module keeps stat in global variables. */
   CosMethods    /* A pointer to a table of module-level functions. Can be NULL if no functions are present. */
};

PyMODINIT_FUNC PyInit_cos_module(void)
{
   return PyModule_Create(&cosModule);
}
```

- **Include <Python.h>**
- **Wrap the C function**
  - Parse the input params from python to C
  - Call the C function
  - Parse the output value from C to python
- **Declare functions in this module**
  - **PyMethodDef**
- **Initializing C modules**
  - **PyModuleDef**
  - **PyModule_Create()**

Python C API: build the C-extentions from a setup.py

- The standard python build system **distutils** supports compiling C-extensions from asetup.py

```
from distutils.core import setup, Extension

# define the extension module
cos_module = Extension('cos_module',
sources=['cos_module.c'])

# run the setup
setup(ext_modules=[cos_module])
```

- Build & Usage

```
[wenwu@p01bc interfacing_with_c]$ pwd
/home/wenwu/python/exercise/interfacing_with_c
[wenwu@p01bc interfacing_with_c]$ python3 setup.py build_ext --inplace
running build_ext
building 'cos_module' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
prototypes -fPIC -I/home/wenwu/include/python3.4m -c cos_module.c -o
build/temp.linux-x86_64-3.4/cos_module.o
gcc -pthread -shared build/temp.linux-x86_64-3.4/cos_module.o -o
/home/wenwu/python/exercise/interfacing_with_c/cos_module.cpython-
34m.so
[wenwu@p01bc interfacing_with_c]$ python3
Python 3.4.3 (default, Aug 12 2015, 21:24:10)
[GCC 3.4.5 20051201 (Red Hat 3.4.5-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cos_module
>>> cos_module
<module 'cos_module' from
'/home/wenwu/python/exercise/interfacing_with_c/cos_module.cpython-
34m.so'>
>>> dir(cos_module)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'cos_func']
>>> cos_module.cos_func(0.0)
1.0
>>> cos_module.cos_func(3.1415926)
-0.999999999999998
```

Extention with Ctypes: Overview

Ctypes is a *foreign function library* for Python. It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python.

- **Advantages**
  - Part of the Python standard library
  - Does not need to be compiled
  - Wrapping code entirely in Python
- **Disadvantages**
  - Requires code to be wrapped to be available as a shared library (roughly speaking *.dll in Windows *.so in Linux and *.dylib in Mac OSX.)
  - No good support for C++

Extention with Ctypes: Example

- Wrap the functions in dynamic library in a module file (.py)

- ctypesWrap.py

```
#!/usr/bin/env python3
""" Example of wrapping cos function from math.h using ctypes. """

import ctypes
from ctypes.util import find_library

# find and load the library
libm = ctypes.cdll.LoadLibrary("libm.so")
# set the argument type
libm.cos.argtypes = [ctypes.c_double]
# set the return type
libm.cos.restype = ctypes.c_double

# wrap the functions defined in the library
def cos_func(arg):
    ''' Wrapper for cos from math.h '''
    return libm.cos(arg)
```

- Usage

```
[wenwu@p01bc interfacing_with_c]$ python3
Python 3.4.3 (default, Aug 12 2015, 21:24:10)
[GCC 3.4.5 20051201 (Red Hat 3.4.5-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import ctypesWrap
>>> ctypesWrap
<module 'ctypesWrap' from
'/home/wenwu/python/exercise/interfacing_with_c/ctypesWrap.py'>
>>> dir(ctypesWrap)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'cos_func', 'ctypes', 'find_library', 'libm']
>>> ctypesWrap.cos_func(0)
1.0
>>> ctypesWrap.cos_func(1.0)
0.5403023058681398
>>> ctypesWrap.cos_func("string")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/wenwu/python/exercise/interfacing_with_c/ctypesWrap.py",
line 16, in cos_func
    return libm.cos(arg)
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
```

Extention with Cython: Overview

Cython is both a ***programming language*** based on Python, that is ***translated*** into C/C++ codes, and finally ***compiled*** into binary extension modules that can be loaded into a regular Python session.

- **Advantages**
  - Python like language for writing C-extensions
  - Autogenerated code
  - Supports incremental optimization
  - Includes a GNU debugger extension
  - Support for C++ (Since version 0.13)
- **Disadvantages**
  - Must be compiled
  - Requires an additional library (but only at build time, at this problem can be overcome by shipping the generated C files)

Extention with Cython: Example

- Cos_module.pyx

```
""" Example of wrapping cos function from math.h using Cython. """

# declare a function from externel library
cdef extern from "math.h":
    double cos(double arg)

# define a pure Python function to wrap the C function
def cos_func(arg):
    return cos(arg)
```

- ctypesWrap.py

```
#!/usr/bin/env python3
""" setup  for cython wraping """

from distutils.core import setup, Extension
from Cython.Distutils import build_ext

setup(
    cmdclass={'build_ext': build_ext},
    ext_modules=[Extension("cos_module", ["cos_module.pyx"])]
)
```

- Build & Usage

```
[wenwu@p01bc interfacing_with_c]$ python3 cythonWrap.py build_ext --inplace
running build_ext
cythoning cos_module.pyx to cos_module.c
building 'cos_module' extension
gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -
fPIC -I/home/wenwu/include/python3.4m -c cos_module.c -o build/temp.linux-
x86_64-3.4/cos_module.o
gcc -pthread -shared build/temp.linux-x86_64-3.4/cos_module.o -o
/home/wenwu/python/exercise/interfacing_with_c/cos_module.cpython-34m.so
[wenwu@p01bc interfacing_with_c]$ ls
build  cos_module.c  cos_module.cpython-34m.so  cos_module.pyx  ctypesWrap.py
cythonWrap.py  __pycache__  setup.py

[wenwu@p01bc interfacing_with_c]$ python3
Python 3.4.3 (default, Aug 12 2015, 21:24:10)
[GCC 3.4.5 20051201 (Red Hat 3.4.5-2)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import cos_module
>>> cos_module
<module 'cos_module' from
'/home/wenwu/python/exercise/interfacing_with_c/cos_module.cpython-34m.so'>
>>> cos_module.cos_func(1.0)
0.5403023058681398
```

Embedding Python in C/C++: Overview

- The problem with most compiled languages is that the resulting application is fixed and inflexible.
  - The input/output format of our applications changes may need to changes in our codes
- Embedding Python in C/C++
  - Python to handle the input and reprocess the data for desired output format
  - The C/C++ application would rarely change, even no need to re-build the applications
- The API library provides a bunch of C routines to initialize the Python Interpreter, call into your Python modules and finish up the embedding.

Embedding Python in C/C++: call Python scripts

- Initialize the Python Interpreter
  - Py_Initialize()
- Evaluate the Python scripts
  - PyRun_SimpleString()
  - PyRun_SimpleFile()
- Shut down the Python Interpreter
  - Py_Finalize()

```
#include <Python.h>

int main(int argc, char *argv[])
{
 Py_SetProgramName(argv[0]);  /* optional but recommended */

 Py_Initialize();

 PyRun_SimpleString("from time import time,ctime\n"
                    "print('Today is', ctime(time()))\n");

Py_Finalize();
 return 0;
}
```

- Build & Usage

```
[wenwu@p01bc c_call_python]$ pwd
/home/wenwu/python/exercise/c_call_python
[wenwu@p01bc c_call_python]$ python3.4-config --cflags
-I/home/wenwu/include/python3.4m -I/home/wenwu/include/python3.4m  -fno-strict-
aliasing  -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
[wenwu@p01bc c_call_python]$ gcc -I/home/wenwu/include/python3.4m -
I/home/wenwu/include/python3.4m -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -
Wstrict-prototypes -c c_call_string.c
c_call_string.c:5:34: warning: "/*" within comment
[wenwu@p01bc c_call_python]$ ls
c_call_func.c  c_call_string.c  c_call_string.o
[wenwu@p01bc c_call_python]$ python3.4-config --ldflags
-L/home/wenwu/lib/python3.4/config-3.4m -L/home/wenwu/lib -lpython3.4m -lpthread -ldl  -
lutil -lm  -Xlinker -export-dynamic
[wenwu@p01bc c_call_python]$ gcc -L/home/wenwu/lib/python3.4/config-3.4m -
L/home/wenwu/lib -lpython3.4m -lpthread -ldl -lutil -lm -Xlinker -export-dynamic -o
c_call_string c_call_string.o
c_call_string.o(.text+0x5): In function `main':
/home/wenwu/python/exercise/c_call_python/c_call_string.c:6: undefined reference to
`Py_Initialize'
... ...
collect2: ld returned 1 exit status
[wenwu@p01bc c_call_python]$ gcc -L/home/wenwu/lib/python3.4/config-3.4m -
L/home/wenwu/lib -o c_call_string c_call_string.o -lpython3.4m -lpthread -ldl -lutil -lm -Xlinker
-export-dynamic
[wenwu@p01bc c_call_python]$ ls
c_call_func.c  c_call_string  c_call_string.c  c_call_string.o
[wenwu@p01bc c_call_python]$ ./c_call_string
Today is Sun Sep  6 10:07:24 2015
```

Embedding Python in C/C++: call Python functions (object-based)


```c
/*  just a sample code, not complete c file */
#include <Python.h>
int main(int argc, char *argv[])
{
   PyObject *pName, *pModule, *pFunc, *pValue;
   if (argc < 3) {
      fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
      return 1;
   }
   Py_Initialize();

   pName = PyUnicode_FromString(argv[1]);
   pModule = PyImport_Import(pName);
   Py_DECREF(pName);
   if (pModule != NULL) {
      pFunc = PyObject_GetAttrString(pModule, argv[2]);
      /* pFunc is a new reference */
      if (pFunc && PyCallable_Check(pFunc)) {
         pValue = PyObject_CallObject(pFunc, NULL);
         Py_DECREF(pValue);
      }
      Py_XDECREF(pFunc);
      Py_DECREF(pModule);
   }

   Py_Finalize();
   return 0;
}
```

```python
def multiply(a, b):
      print("will compute", a, "*", b)
      return a*b
```

- Initialize the Python Interpreter
  - Py_Initialize()
- Call the Python functions
  - Convert data values from C to Python
  - Perform a function call to a Python interface routine
  - Convert the data values from the call from Python to C
- Shut down the Python Interpreter
  - Py_Finalize()

```
[wenwu@p01bc c_call_python]$ ./c_call_func func_module.py multiply 3 2
Traceback (most recent call last):
  File "<frozen importlib._bootstrap>", line 2218, in _find_and_load_unlocked
AttributeError: 'module' object has no attribute '__path__'

During handling of the above exception, another exception occurred:

ImportError: No module named 'func_module.py'; 'func_module' is not a package
Failed to load "func_module.py"
[wenwu@p01bc c_call_python]$ ./c_call_func func_module multiply 3 2
will compute 3 * 2
Result of call: 6
```

The Python Embedding API

C API	Python equivalent	Description
PyImport_ImportModule	import module	Imports a module into the Python
PyImport_ReloadModule	reload(module)	Reloads the specified module
PyImport_GetModuleDict	sys.modules	Returns a dictionary object containing the list of loaded modules
PyDict_GetItemString	dict[key]	Gets the value for a corresponding dictionary key
PyDict_SetItemString	dict[key] = value	Sets a dictionary key's value
PyDict_New	dict = {}	Creates a new dictionary object
PyObject_GetAttrString	getattr(obj, attr)	Gets the attribute for a given object
PyEval_CallObject	apply(function, args)	Calls a function with arguments in args
PyRunString	eval(expr), exec expr	Executes expr as a Python statement
PyRun_File	execfile(filename)	Executes the file filename
PySetProgramName	sys.argv[0] = name	Changes the name of the Python program typically set on the command line
… …		

Summary on interacting between Python and C/C+ +

- C/C++ to provide new modules to extend the Python interpreter
  - Benefit the efficiency of C/C++ codes
  - Reuse the C codes, libraries already have
- Python provide the interface to interact with Python interpreter
  - Benefit the “learning curve” of Python
  - Make use of the advantages of Python modules (Regular expression, text-handling, etc.)
  - Adding scripting ability(flexibility) to your applications (similar with Lua)

Reference

- https://scipy-lectures.github.io/advanced/interfacing_with_c/interfacing_with_c.html
- https://docs.python.org/3.4/c-api/index.html
- https://en.wikibooks.org/wiki/Python_Programming/Extending_with_C
- https://docs.python.org/3.4/library/ctypes.html#loading-dynamic-link-libraries
- http://cython.org/
- https://docs.python.org/3.4/extending/index.html
- http://www.codeproject.com/Articles/11805/Embedding-Python-in-C-C-Part-I
- https://www6.software.ibm.com/developerworks/education/l-pythonscript/l-pythonscript-ltr.pdf