LECTURE 7

The Standard Library Part 1:
Built-ins, time, sys, and os

THE PYTHON LANGUAGE

Believe it or not, you now have all the Python syntax and structures you need already. At this point, we can turn our attention to writing applications in Python.

There will still be some points to be made about the Python language as we continue the course, but they will be brought to your attention when they come up.

For now, let's start by learning some of the libraries that every Python programmer must know.

THE PYTHON STANDARD LIBRARY

The Python Standard Library is a collection of modules that are distributed with every Python installation. It is a vast assortment of useful tools and interfaces, which covers a very wide range of domains.

Besides the standard library, there is also the Python Package Index (PyPI), the official third-party repository for everything from simple modules to elaborate frameworks written by other Python programmers. As of right now, there are 60,660 packages in PyPI.

We will start by spending the next couple of lectures covering the most commonly used modules in the standard library. Then, we will spend the rest of the semester covering widely-used third party packages.

STANDARD LIBRARY: BUILT-INS

We've already learned about a lot of data types – such as numbers and lists – which are part of the "core" of Python. That is, you don't need to import anything to use them.

However, it's the standard library that actually defines these types, as well as many other built-in components.

We've already learned all about the built-in data types so we won't re-cover that material but we'll start by looking at what other "built-ins" are defined by the standard library.

STANDARD LIBRARY: BUILT-IN CONSTANTS

There are a few built-in constants defined by the standard library:

- `True`: true value of a bool type.
- `False`: false value of a bool type.
- `__debug__`: true if Python was started with the –O option.

```
a = True
b = False
if a is True:
    print "a is true."
else:
    print "a is false"
if b is True:
    print "b is true."
else:
    print "b is false."
```

STANDARD LIBRARY: BUILT-IN CONSTANTS

- `None`: used to represent the absence of a value. Similar to the null keyword in many other languages.

```
conn = None
try:
    database = MyDatabase(db_host, db_user, db_password, db_database)
    conn = database.connect()
except DatabaseException:
    pass

if conn is None:
    print('The database could not connect')
else:
    print('The database could connect')
```

STANDARD LIBRARY: BUILT-IN CONSTANTS

- `NotImplemented`: returned when a comparison operation is not defined between two types.

  This constant is meant to be used in conjunction with "rich comparison" methods, `__lt__()`, `__eq__()`, etc. Behind the scenes, when we execute the following statement:

  ```
  >>> a < b
  ```

  Python is really executing this statement:

  ```
  >>> a.__lt__(b)
  ```

  The `NotImplemented` constant allows us to indicate that `a` does not have `__lt__()` defined for `b`'s type, so perhaps we should try calling `b`'s `__ge__()` method with `a` as an argument.

STANDARD LIBRARY: BUILT-IN CONSTANTS

```python
class A:
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        if isinstance(other, B):
            print('Comparing an A with a B')
            return other.value == self.value
        print('Could not compare A with other')
        return NotImplemented

class B:
    def __init__(self, value):
        self.value = value
    def __eq__(self, other):
        print('Could not compare B with other')
        return NotImplemented
```

```python
>>> a = A(2)
>>> b = B(2)
>>> a == b # a.__eq__(b)
Comparing an A with a B
True
>>> b == a # b.__eq__(a)
Could not compare B with other
Comparing an A with a B
True
```

STANDARD LIBRARY: BUILT-IN CONSTANTS

- `Ellipsis`: for custom use in extended slicing syntax (not used by any built-in function).

```
>>> class TwoDimList:
        def __init__(self, data):
            self.data = data
        def __getitem__(self, item):
            if item is Ellipsis:
                return [x[i] for x in self.data for i in range(len(self.data))]
            else:
                return self.data[item]
>>> x = TwoDimList([[1,2,3],[4,5,6],[7,8,9]])
>>> x[0] # x.__getitem__(0)
[1, 2, 3]
>>> x[...]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

STANDARD LIBRARY: BUILT-IN FUNCTIONS

There are a huge number of built-in functions which are always available.

We've seen a good number of these already and most of them are "manual" calls for actions typically done another way.

		Built-in Functions		
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	
delattr()	help()	next()	setattr()	
dict()	hex()	object()	slice()	
dir()	id()	oct()	sorted()	

STANDARD LIBRARY: TIME

The time module is responsible for providing time-related functions and conversion methods. You can obtain access to time's methods and attributes with the `import time` statement.

The most commonly used methods are:

- `time.time()` – returns the time in seconds since the epoch (typically 1/1/1970).
- `time.sleep(s)` – suspends execution for *s* seconds.
- `time.clock()` – returns the current processor time in seconds.

STANDARD LIBRARY: TIME

Here, we create two small timing functions which measure time passed over a call to `time.sleep()`.

The `time.time()` method simply measures elapsed wall clock time.

The `time.clock()` method, however, only measures time during which the CPU is actively working on behalf of the program. When we sleep, we are suspending the program for some time so the CPU is not active during the sleeping time.

```
>>> import time
>>> def timer():
...     s = time.time()
...     time.sleep(5)
...     e = time.time()
...     print (e-s)
>>> def cpu_timer():
...     s = time.clock()
...     time.sleep(5)
...     e = time.clock()
...     print (e-s)
>>> timer()
5.00614309311
>>> cpu_timer()
0.000121
```

STANDARD LIBRARY: TIME

There are some additional useful time methods but they all depend on the `struct_time` class so we'll cover that first. The `struct_time` class is also defined in the time module. It is a class which simply has 9 attributes for describing a particular time.

Index	Attribute	Values
0	`tm_year`	(for example, 1993)
1	`tm_mon`	range [1, 12]
2	`tm_mday`	range [1, 31]
3	`tm_hour`	range [0, 23]
4	`tm_min`	range [0, 59]
5	`tm_sec`	range [0, 61]; see **(2)** in `strftime()` description
6	`tm_wday`	range [0, 6], Monday is 0
7	`tm_yday`	range [1, 366]
8	`tm_isdst`	0, 1 or -1; see below

STANDARD LIBRARY: TIME

The `struct_time` class is unique in that it uses a *named tuple* interface. You can access the attributes of the class using either the attribute name (e.g. `t.tm_year`) or an index (e.g. `t[0]`).

The `time.strftime(`*`format`*`[, `*`t`*`])` method can be used to convert a `struct_time` object *t* into a readable format. A table of the possible format string arguments is found here.

STANDARD LIBRARY: TIME

- `time.asctime([t])` – converts a `struct_time` object *t* into a specific formatted output string. If *t* is not provided, the current time is used.
- `time.gmtime([s])` – converts a time expressed in seconds *s* since the epoch to a `struct_time` object in UTC. If *s* is not provided, `time.time()` is used.
- `time.localtime([s])` – like `time.gmtime([s])`, but converts to a local time.
- `time.mktime(t)` – inverse of `time.localtime([s])`. Converts a `struct_time` object *t* in local time to seconds since the epoch.

STANDARD LIBRARY: TIME

```
>>> time.time()
1433264623.282071
>>> time.gmtime()
time.struct_time(tm_year=2015, tm_mon=6, tm_mday=2,
tm_hour=17, tm_min=3, tm_sec=58, tm_wday=1, tm_yday=153,
tm_isdst=0)
>>> time.localtime()
time.struct_time(tm_year=2015, tm_mon=6, tm_mday=2,
tm_hour=13, tm_min=4, tm_sec=8, tm_wday=1, tm_yday=153,
tm_isdst=1)
>>> time.asctime(time.localtime())
'Tue Jun 2 13:05:00 2015'
>>> time.strftime("%A, %B %d, %Y",time.localtime())
'Tuesday, June 02, 2015'
```

STANDARD LIBRARY: SYS

The sys module provides access to some variables used or maintained by the interpreter as well as some methods for interacting with the interpreter. It allows you to receive information about the runtime environment as well as make modifications to it.

To use the sys module, just execute the `import sys` statement.

STANDARD LIBRARY: SYS

As we've already seen, one of the most common ways to use the sys module is to access arguments passed to the program. This is done with the `sys.argv` list.

The first element of the `sys.argv` list is always the module name, followed by the whitespace-separated arguments.

```
import sys
for i in range(len(sys.argv)):
    print "sys.argv[" + str(i) + "] is " + sys.argv[i]

$ python testargs.py here are some arguments
sys.argv[0] is testargs.py
sys.argv[1] is here
sys.argv[2] is are
sys.argv[3] is some
sys.argv[4] is arguments
```

STANDARD LIBRARY: SYS

The sys.path variable specifies the locations where Python will look for imported modules. The sys.path variable is also a list and may be freely manipulated by the running program. The first element is always the "current" directory where the top-level module resides.

```
import sys

print "path has", len(sys.path), "members"

sys.path.insert(0, "samples")
import sample

sys.path = []
import math
```

```
$ python systest.py
path has 8 members
Hello from the sample module!
Traceback (most recent call last):
  File "systest.py", line 9, in ?
    import math
ImportError: No module named math
```

STANDARD LIBRARY: SYS

Note that there are some modules that are always available to the interpreter because they are built-in. The sys module is one of them. Use `sys.builtin_module_names` to see which modules are built-in.

```python
import sys

print "path has", len(sys.path), "members"

sys.path.insert(0, "samples")
import sample

sys.path = []
import math
```

```
$ python systest.py
path has 8 members
Hello from the sample module!
Traceback (most recent call last):
  File "systest.py", line 9, in ?
    import math
ImportError: No module named math
```

STANDARD LIBRARY: SYS

The `sys.modules` dictionary contains all of the modules currently imported.

```
>>> import sys
>>> sys.modules.keys()
['copy_reg', 'sre_compile', '_sre', 'encodings', 'site', '__builtin__',
'sysconfig', '__main__', 'encodings.encodings', 'math', 'abc',
'posixpath', '_weakrefset', 'errno', 'encodings.codecs', 'sre_constants',
're', '_abcoll', 'types', '_codecs', 'encodings.__builtin__',
'_warnings', 'encodings.latin_1', 'genericpath', 'stat', 'zipimport',
'_sysconfigdata', 'warnings', 'UserDict', 'sys', 'codecs', 'readline',
'os.path', 'signal', 'traceback', 'linecache', 'posix',
'encodings.aliases', 'exceptions', 'sre_parse', 'os', '_weakref']
```

The sys.platform attribute gives information about the operating system.

```
>>> sys.platform
'linux2'
```

STANDARD LIBRARY: SYS

The sys.version attribute provides information about the interpreter including version, build number, and compiler used. This string is also displayed when the interpreter is started.

```
$ python2.7
Python 2.7.5 (default, Oct 5 2013, 01:47:54)
[GCC 3.4.3 20041212 (Red Hat 3.4.3-9.EL4)] on linux2
```

STANDARD LIBRARY: SYS

The `sys.stdin`, `sys.stdout`, and `sys.stderr` attributes hold the file object corresponding to standard input, standard output, and standard error, respectively. Just like every other attribute in the sys module, these may also be changed at any time!

If you want to restore the standard file objects to their original values, use the `sys.__stdin__`, `sys.__stdout__`, and `sys.__stderr__` attributes.

STANDARD LIBRARY: SYS

The `sys.exit([status])` function can be used to exit a program gracefully. It raises a `SystemExit` exception which, if not caught, will end the program.

The optional argument *status* can be used to indicate a termination status. The value 0 indicates a successful termination while an error message will print to stderr and return 1.

The sys module also defines a `sys.exitfunc` attribute. The function object specified by this attribute is used to perform “cleanup actions” before the program terminates.

STANDARD LIBRARY: OS

The os module provides a common interface for operating system dependent functionality.

Most of the functions are actually implemented by platform-specific modules, but there is no need to explicitly call them as such.

STANDARD LIBRARY: OS

We've already seen how the os module can be used to work with files. We know that there are built-in functions to open and close files but os extends file operations.

- `os.rename(current_name, new_name)` renames the file *current_name* to *new_name*.
- `os.remove(filename)` deletes an existing file named *filename*.

STANDARD LIBRARY: OS

There are also a number of directory services provided by the os module.

- `os.listdir(dirname)` lists all of the files in directory *dirname*.
- `os.getcwd()` returns the current directory.
- `os.chdir(dirname)` will change the current directory.

```
>>> os.listdir("demos")
['frac.py', 'dogs.py', 'csv_parser.py']
>>> os.listdir(".")
['lect5.py', 'demos', 'lect3.py']
>>> os.getcwd()
'/home/faculty/carnahan/CIS4930'
>>> os.chdir(os.getcwd() + "/demos")
>>> os.getcwd()
'/home/faculty/carnahan/CIS4930/demos'
>>> os.rename("dogs.py", "cats.py")
>>> os.listdir(".")
['frac.py', 'cats.py', 'csv_parser.py']
>>> os.remove("cats.py")
>>> os.listdir(".")
['frac.py', 'csv_parser.py']
```

STANDARD LIBRARY: OS

- Use `os.mkdir(`*`dirname`*`)` and `os.rmdir(`*`dirname`*`)` to make and remove a *single* directory.
- Use `os.makedirs(`*`path/of/dirs`*`)` and `os.removedirs(`*`path/of/dirs`*`)` to make and remove a hierarchy of directories.
- Make sure directories are empty before removal!

STANDARD LIBRARY: OS

```
>>> os.makedirs("dir1/dir2/dir3")
>>> os.listdir(".")
['frac.py', 'dir1', 'csv_parser.py']
>>> f = open("dir1/dir2/dir3/test", "w")
>>> f.write("hi!")
>>> f.close()
>>> for line in open("dir1/dir2/dir3/test", "r"):
...     print line
...
hi!
>>> os.remove("dir1/dir2/dir3/test")
>>> os.removedirs("dir1/dir2/dir3")
>>> os.listdir(".")
['frac.py', 'csv_parser.py']
```

STANDARD LIBRARY: OS

The `os.walk(`*`path`*`)` method will generate a tuple (*dirpath*, *dirnames*, *filenames*) for each directory found by traversing the directory tree rooted at *path*.

```
>>> os.makedirs("dir1/dir2/dir3")
>>> os.listdir(".")
['frac.py', 'dir1', 'football.csv', 'csv_parser.py']
>>> os.mkdir("dir1/dir2/dir4")
>>> f = open("dir1/dir2/d2file", "w")
>>> f = open("dir1/dir2/dir3/d3file", "w")
>>> f = open("dir1/dir2/dir4/d4file", "w")
>>> path = os.getcwd()
>>> for (path, dirs, files) in os.walk(path):
...     print "Path: ", path
...     print "Directories: ", dirs
...     print "Files: ", files
...     print "---"
```

STANDARD LIBRARY: OS

The `os.walk(`*`path`*`)` method will generate a tuple *(dirpath, dirnames, filenames)* for each directory found by traversing the directory tree rooted at *path*.

```
Path:  /home/faculty/carnahan/CIS4930/demos
Directories:  ['dir1']
Files:  ['frac.py','football.csv','csv_parser.py']
---
Path:  /home/faculty/carnahan/CIS4930/demos/dir1
Directories:  ['dir2']
Files:  []
---
Path:
/home/faculty/carnahan/CIS4930/demos/dir1/dir2
Directories:  ['dir4', 'dir3']
Files:  ['d2file']
---
Path:
/home/faculty/carnahan/CIS4930/demos/dir1/dir2/dir4
Directories:  []
Files:  ['d4file']
---
Path:
/home/faculty/carnahan/CIS4930/demos/dir1/dir2/dir3
Directories:  []
Files:  ['d3file']
---
```

STANDARD LIBRARY: OS

The os module includes an `os.stat(path)` method which will return file attributes related to the path provided (equivalent to stat() system call).

Result is a stat structure which includes

- `st_size`: size of file in bytes.
- `st_atime`: time of most recent access.
- `st_uid`: user id of owner.
- `st_nlink`: number of hard links.

```
>>> import os
>>> stat_info = os.stat("football.csv")
>>> stat_info
posix.stat_result(st_mode=33216,
st_ino=83788199L, st_dev=20L,
st_nlink=1, st_uid=87871, st_gid=300,
st_size=648L, st_atime=1422387494,
st_mtime=1421257389,
st_ctime=1421257413)
>>> stat_info.st_mtime
1421257389.0
```

OS SERVICES

The `os.system(`*cmd*`)` function executes the argument *cmd* in a subshell. The return value is the exit status of the command.

```
>>> os.system("ls")
csv_parser.py dir1 football.csv frac.py
0
>>> os.system("touch newfile.txt")
0
>>> os.system("ls")
csv_parser.py dir1 football.csv frac.py newfile.txt
0
```

STANDARD LIBRARY: OS

The `os.exec(path, args)` function will start a new process from *path* using the *args* as arguments, replacing the current one. Alternatives include `os.execve()`, `os.execvp()`, etc as usual. Arguments depend on version used.

```
$ python2.7
Python 2.7.5 (default, Oct 5 2013, 01:47:54)
[GCC 3.4.3 20041212 (Red Hat 3.4.3-9.EL4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> os.execvp("python2.7", ("python2.7", "csv_parser.py"))
Aston_Villa has a minimum goal difference of 1
$
```

STANDARD LIBRARY: OS

Combine the `os.exec*()` functions with `os.fork()` and `os.wait()` to spawn processes from the current process. The former makes a copy of the current process, the latter waits for a child process to finish. Use `os.spawn()` on Windows.

```
import os
import sys

pid = os.fork()

if not pid:
    os.execvp("python2.7", ("python2.7", "csv_parser.py"))
os.wait()
```