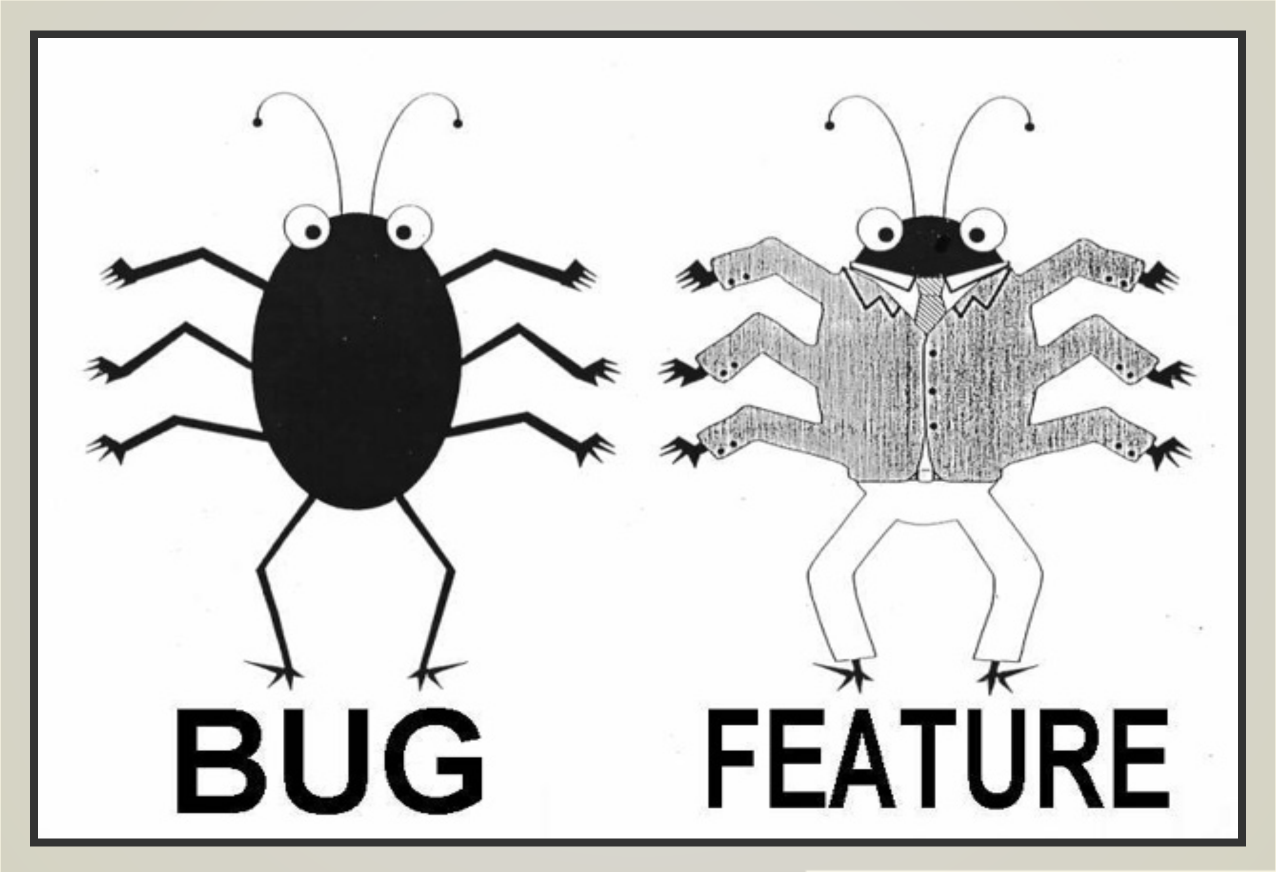
PYTHON II



Revision 04.16.13

NLC INFO

- Rackspace parking sticker works
- Racker breakroom marked 'Laundry' on 1st floor
- Speeding on campus = easy ticket
- Must go off property to smoke
- Please sign in each day. Class starts at 08:30

PAPERWORK

- NLC registration
- RU roster
- Email signup sheet
- Core Python Applications, 3rd Edition, Wesley Chun, ISBN 978-0-13-267820-9

INTRODUCTIONS

- Instructor Contact Info
- Introductions
- Any previous programming?
- Why did you sign up?
- Define success for you in this course

LAB INFO

- Labs will use lab machines only as a pass through to access the NLC cloud
- Linux CentOS environment
- ssh -l student -p 401XX nlccloud.edgecloud.com
- but use 192.168.3.10 on campus
- Login creds?
- Available 24x7 during class

THIS COURSE

- Assumes programming concepts contained in Python I or equiv experience

- **What it is:**
- Python (2.7 level)
- (6) Topics: JSON, REST, PEP249 (DB-API), subprocess, threads

- **What it's not:**
- Math or graphics
- Python internals
- Python frameworks

WAYS TO RUN PYTHON

- IPython is is recommended (http://ipython.org) (requires pyreadline)
- Also recommend running in a virtualenv sandbox for custom control of the environment.
- See the cheeseshop for package downloads (http://pypi.python.org)

VIRTUALENV

- Creates an isolated python environment customized to your version and dependency requirements

- Comes with distribute, easy_install, and pip

- Allows control of environments where root authority is lacking

- Install and activate/deactivate:
- python virtualenv.py --distribute <new env>
- cd <new env>
- source bin/activate (or deactivate)

PIP

- pip is the newest python package manager tool
- pip install <package name or package file>
- pip search <package name>
- Use --upgrade option for existing packages

PYTHON REVIEW

- Anatomy of a Module

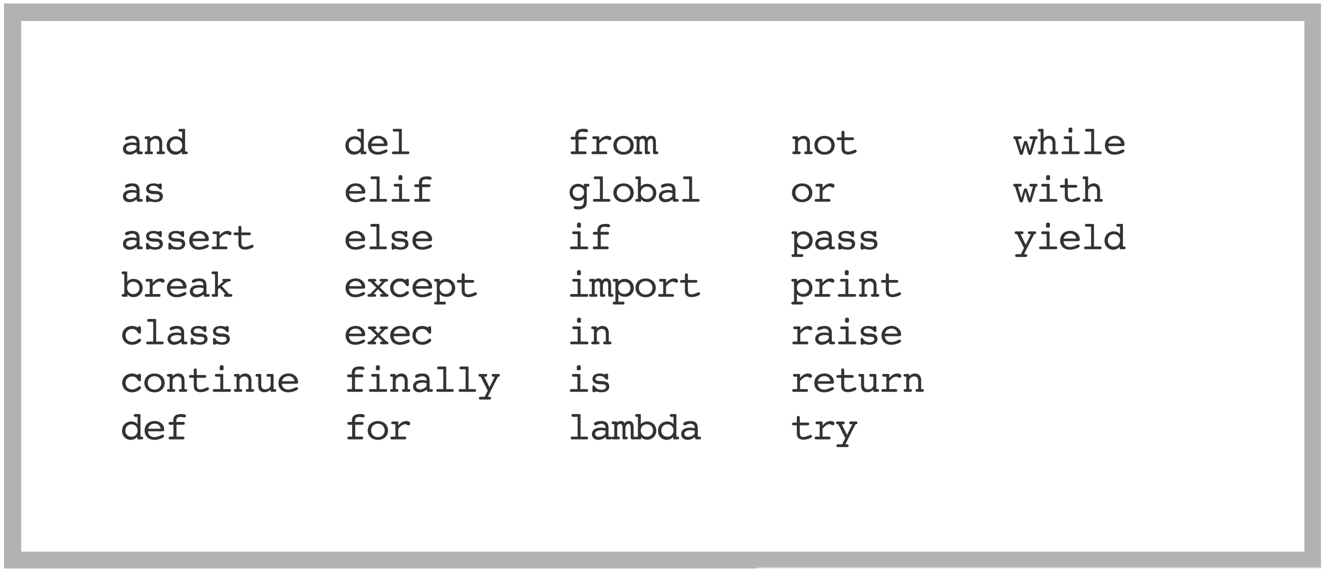
```
# environment and encoding declaration
#!/usr/bin/env python
# -*- coding: utf-8 -*-

Docstrings
Inline documentation

Import(s)
External code sources called modules

Statements
```

PYTHON KEYWORDS



```
and       del       from      not       while
as        elif      global    or        with
assert    else      if        pass      yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

- Reserved for use by Python

TUPLE TYPE

- Tuples are sequences of other objects that cannot be changed (they are “immutable”)

- Tuples can contain any type of object, and can be sliced (remember, they’re ordered)

- A one element tuple is formed by (element,) where the comma makes it a tuple

- Reference: Section 5.6

LIST TYPE

- Lists are sequences that can be changed (“mutable”)
- Lists can contain other object types and be sliced
- A list is a set of objects separated by commas and closed in square brackets
- Reference: Section 5.6

DICTIONARY TYPE

- A dictionary (“dict”) type is a set of expressions that are associated 1 to 1 with other expressions
- A dict provides a handy “mapping” between a “key” expression and its associated “value” expression
- A dict is not an ordered sequence, it’s a mapping.
- Reference: Sect. 5.8 Mapping Types

SET TYPE

- A set is a mutable group of unordered immutable objects with unique values i.e. no duplicates (so the set is mutable, but not the objects within it)

- Uses the ‘{}’ symbols just like a dictionary, but doesn’t have ‘:’ i.e. no key/value pairs

- Slicing is not allowed (why?), but iterators work

- Set operations are quite powerful. See Sect 5.7

YIELD STATEMENT

- yield is similar to return, but suspends execution of the called function instead of ending the function
- On the next call to the function, yield picks up where it left off, with all identifier values still holding the same values i.e. loses its stack entry
- Appearance of yield signals a generator

WITH STATEMENT

- The with statement is used to run a suite of other statements under a “context manager”
- Special methods __entry__() and __exit() are called to setup and takedown a “context”
- Common for doing i/o (which auto-closes the file handle) and threading (which auto-acquires and releases lock)

IMPORT & FROM STATEMENTS

- To make a set of identifiers in another module available for use by your module, you must first use import or from

- import pulls in identifiers from module(s) but the module name must be used as a prefix (only module name is added to local namespace)

- from pulls in identifiers from modules but avoids need to prefix with the module name (identifier is added to local namespace)

THE CALCULATOR PACKAGE

```
calculator
   __init__.py
operations
   __init__.py
   arithmetic.py
display
   __init__.py
   scientific.py
   algebraic.py
```

FILE OBJECTS

Use the with statement if possible.
Why?

```
with open(“file_path”,<mode>) as fh:
# <mode> = “r” for read only; ”rb”
#          “w” for write only; “wb”
#          “a” for append
for line in fh:
    <suite>
all_contents = fh.read() # read all chars
one_line = fh.readline()
as_list = fh.readlines()
```

LAB01

JSON

- Javascript Object Notation
- Part of Javascript language def
- Easy for parse for humans and machines
- Language independent, a “lightweight data interchange format”
- Built on:

```
Name/value pair collection (dict in Python)
Ordered list of values (list in Python)
```

JSON

- An “object” is an unordered set of name:value pairs (called “members”) separated by commas and surrounded by curly braces
- { Name1:Value2, Name2:Value,… }
- An “array” is a set of “elements” surrounded by square brackets
- [element1, element2,…]
- Elements can be strings, numbers, true, false, null, object, or array

JSON

- Strings are made of:
- Unicode chars (\uhhhh)
- \,\\,/,\b,\f,\n,\r,\t
- Numbers are signed integer, decimal, or exponent (“e”) flavors only
- Whitespace is fine

JSON MODULE

- import json
- Python object to serialized JSON object:

  - json.dump(python_obj, fo, **kwargs)
  - json.dumps(python_obj, **kwargs)

- JSON serialized object back to Python object:

  - python_obj = json.load(fo, **kwargs)
  - python_obj = json.loads(string, **kwargs)
  - “fo” is a “file-like” object supporting write.
    **kwargs have additional options
  - Note: JSON is not a ”framed” protocol
    i.e.can’t append multiple JSON objects to same file

JSON MODULE

- JSON keys in key/value pairs are always strings. Unlike Python.
- Default json module encoder only encodes “ASCII-compatible” strings. Use u for other encodings
- Same name in name:value pairs uses the last one
- Out of range floats are handled correctly (nan, inf, -inf)

PYTHON TO JSON TRANSLATIONS

```
dict -> object
list, tuple -> array
str, unicode -> string
int, float -> number
True, False -> true, false
None -> null
```

JSON TO PYTHON TRANSLATIONS

```
object -> dict
array -> list
string -> unicode
int -> int
real -> float
true, false -> True, False
null -> None
```

COMPLEX SERIALIZATIONS

- Accessing a deep object serialization can be a challenge
- Remember: only simple objects, maps, and arrays compose a JSON serial object
- Use subscriptions and keys to get around

PICKLE AND SHELVE

- The pickle module does a similar job to JSON, but is Python specific
- Not good for machine data interchange
- Allows multiple pickled objects to be dumped to the same file (and must be loaded in same order on way back) but shelve is better solution
- The shelve module essentially provided a persistent dict for pickled objects in a database

PERFORMANCE

- The builtin json module can be slow
- Other pypi packages have C extensions to speed them up e.g. simplejson, yajl, python-cjson, and UltraJSON

LAB02

REST

- A simple, client/server web services API currently in favor
- Way of locating and manipulating “resources” (usually XML or JSON documents) on a network
- Commonly based HTTP protocol (GET, POST, PUT, DELETE)
- Stateless (all state on client or server)
- Simple, predictable resource pathing scheme based on URL

REST

- RESTful services generally map a CRUD interface (Create, Read, Update, Delete) by URL mappings that embed data e.g. GET v2/{tenant_id}/servers

- HTTP POST -> Create
- GET -> Read
- PUT -> Update
- DELETE -> Delete

- HTTP GET calls to a RESTful service should not change state i.e. read only

OPENSTACK REST API

- Openstack uses REST to implement it’s user-controlled cloud provisioning service

- Requires a set of “endpoint” URL’s which have service request data appended e.g.

- Adding “v2.0/tokens” to the Identity service “endpoint” URL, inserting login credentials, and POST’ing will return a “token” that allows use of the v2.0 API

- e.g. Adding “v2/{tenant_id}/servers/ips for the Compute service will return server IP addresses

OPENSTACK REST API

- See the Openstack API Reference at http://api.openstack.org/ for complete API

- Workflow to use Openstack API:

  - Obtain tenant id and API key and authenticate (24 hour timeout)
  - Extract token id and appropriate endpoint URL for the desired service from the response
  - Send API request(s) to the appropriate service endpoint(s) including the X-Auth-Token HTTP header for each request
  - If a 401 HTTP response occurs, re-authenticate

URLLIB2 MODULE

- Client functions to access URL’s
- import urllib2
- urllib2.urlopen(url[,data]) is a common call signature, where url is the URL of the target resource, and data is to be sent to the server
- Only HTTP uses data currently. If data exists, it must be URL encoded
- If data exists, HTTP GET becomes a POST.

URLLIB2 MODULE

- A “file-like object” is returned from urlopen() which can be accessed with file semantics (read, readlines, etc.)
- Raises:

```
URLError (subclass of IOError)
HTTPError (subclass of URLError)
```

- Also takes urllib2.Request objects, useful for including HTTP headers in a dict (or use Request add_header method)
- Can handle redirections, HTTP error responses, cookies, proxies, HTTP authentication

LAB03

DAY 1 SUMMARY/Q&A

BASIC SQL

- Structured Query Language (SQL) is covered in the Intro to MySQL class and many other places

```
CREATE DATABASE / DROP DATABASE
CREATE TABLE
INSERT
SELECT
```

- DB-API defines a standard interface with which to access a relational database from Python

- Version 2 is current – defined in PEP249 http://www.python.org/dev/peps/pep-0249/

- Allows choice of thread support and parameter formatting

DB-API

- Has a defined Exception hierarchy:

```
StandardError
Warning
Error
Interface Error
Database Error
```

- Supprts “connection objects” to access database

```
close()
commit()
rollback()
cursor()
```

DB-API

- Supports cursor objects (essentially a result set iterator for databases) e.g.

```
execute()
fetchone()
fetchmany()
fetchall()
```

- Supports binding of database specific constructors for time and date formats to match the target database, and convert Python types to database types

- Many implementations: we will use a MySQL database and package mysql-python

MYSQLDB DB-API

- A PEP-249 implementation for MySQL database
- Multiple threads can share the module (but need their own Connection objects)
- Connection parameters (most common):

```
host (default localhost)
user
passwd
db (default no db)
port (default 3306: MySQL standard)
use_unicode=True
paramstyle (defaults to % format chars; column values only!)
```

MYSQLDB DB-API

- Connection.cursor() emulates a cursor (MySQL does not support cursors directly)
- Connection.commit() and Connection.rollback() work for transactions
- User Guide at http://mysql-python.sourceforge.net/MySQLdb.html
- API at http://mysql-python.sourceforge.net/MySQLdb-1.2.2/

MYSQLDB DB-API

- Examples:

```
import MySQLdb
lab_conn = MysqlDB.connect(host=“x”,
                           user=“me”,
                           passwd=“secret”,
                           db=“lab”)
lab_cursor = lab_conn.cursor()
lab_cursor.execute(“select * from bugs
                           where bug_type = %s
                           and genus = %i”,
                           (“butterfly”,3))
results = lab_cursor.fetchall()
```

LAB04

PYTHON SPAWNED PROCESSES

- Python can spawn and control entire processes using the subprocess module
- Generally means redirecting the basic file descriptors (stdin, stdout, stderr) to gain programmatic access
- Forks a new process and uses pipes for redirection
- As usual, beware of invoking a process based on direct or indirect user input

SUBPROCESS.CALL()

- Use subprocess.Popen class for most use cases, unless a convenience method fits
- Run command specified in args and *, wait for completion, and return exit code:
- subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False)
- Don’t use pipes for stdout or stderr
- Only use shell=True if need shell features

SUBPROCESS.CHECK_CALL()

- check_call() raises CalledProcessError exception if exit code not 0:

  subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False)

- Don’t use pipes for stdout or stderr

- Only use shell=True if need shell features

SUBPROCESS.CHECK_OUTPUT()

- check_output() returns process output as a byte string:

  my_out = subprocess.check_output(args, *, stdin=None, stdout=None, stderr=None, shell=False, universal_newlines=False)

- Don’t use pipes for stdout or stderr

- universal_newlines is to convert all line endings to \n

SUBPROCESS MODULE

- Can use subprocess.PIPE in stdin, stdout, stderr
- Can subprocess.STDOUT in stderr
- subprocess.Popen(args) constructor has many keyword args too.
- See http://docs.python.org/library/subprocess.html#module-subprocess
- Popen args should be a string or sequence (sequence preferred)

SUBPROCESS MODULE

- Note: if args is a string, must use shell=True to specify args
- Once again, don’t use shell=True with user entered data.
- shell=False does not call a shell either directly or indirectly, allowing all characters (even metacharacters) to be passed safely

SUBPROCESS MODULE

- Popen methods:
  - poll() sees if child is terminated
  - wait() waits for child to terminate
  - communicate(input=None) sends input to child stdin. Use instead of stdin.write()
  - send_signal(signal) sends given signal to child
  - terminate() and kill() send respective signals to child

SUBPROCESS MODULE

- Popen.std* are file objects if std*=PIPE was used in Popen constructor
- Popen.pid is child process ID
- Popen.returncode is child exit code. Set by poll(), wait(). None if child not terminated. If negative, indicates child was terminated by that signal number.

LAB05

DAY 2 SUMMARY/Q&A

THREADS

- Threads are independently scheduled streams of instructions that run within a single OS process
- Threads have their own stack, registers, scheduling attributes, priorities, signal mask, and thread local memory
- Threads allow logical concurrency of execution (and possibly parallel execution if configured)
- Threads introduce the need for synchronization!

THREADS

- Threads share the same instructions (bytecodes) same identifier bindings, same open files, and other resources
- Threads cannot exist outside of an OS process
- Threads are "lightweight" – the overhead of creating them is much less than creating a real process
- Threads are used for capturing a higher % of available cycles on a single CPU, running parallel work if multiple CPU's, realtime systems, an asynchronous event handling.

THREADS

- Thread synchronization
  - coarse (involving an entire call)
  - fine (a section of code)
  - very fine (a single identifier)
- Thread synchronization failure can cause “deadly embrace” and/or “race conditions”
- Use thread-safe libraries

THREADS

- User threads vs. kernel threads
- A programming abstraction
- Kernel thread designs differ substantially by OS
- Threads may or may not be implemented in the kernel
- “Green” threads are only implemented in the user process, and not mapped to kernel threads

POSIX THREADS

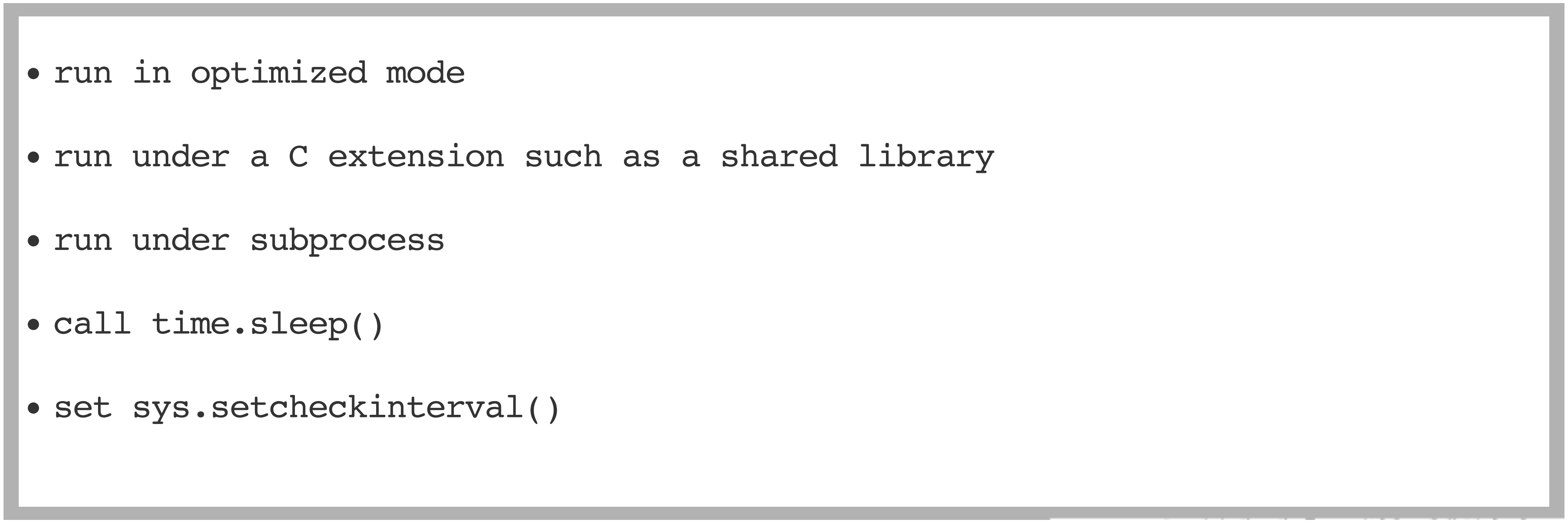
- POSIX (Portable Operating System Interface for Unix) established the POSIX threads (pthreads) standard
- Pthreads provides a standard interface to maximize available machine resources and minimize complexity e.g. in parallel programming, load balancing, problem partitioning, communications handlers
- Most OS vendors and modern languages have ported the pthreads model, including Python

CPYTHON THREADS

- CPython does map user threads to actual kernel threads, BUT…
- CPython is not “thread safe”
- CPython forces a thread to obtain the Global Interpreter Lock (GIL) before running, effectively serializing threads to protect Python interpreter memory
- CPython forces GIL release every 100 bytecode instructions, allowing other threads to run

CPYTHON THREADS

- Thread-safe C extensions bypass the GIL
- Any blocking I/O also releases the GIL
- There exists ways to minimize or eliminate the effect of the CPython GIL



- run in optimized mode
- run under a C extension such as a shared library
- run under subprocess
- call time.sleep()
- set sys.setcheckinterval()

THREADING MODELS

- Some common threading models exist:

- Boss/worker
  - Boss thread creates worker threads, then loops receiving work requests from a queue or socket

  - Worker threads loop on work requests from Boss

THREADING MODELS

- Peer
  - All threads work on the backlog without a “manager” thread
  - All threads generate their own work requests
- Pipeline
  - Each thread does a stage of a work pipeline
  - Each thread accepts work from “previous” thread, passes to “next” thread

THREADING MODULE

- Main objects are:
    - wait for a condition using a lock:
      threading.Condition()

    - wait for an event:
      threading.Event()

    - lock a critical section or variable:
      threading.Lock() or RLock()

    - classic semaphore acquire and release based on count:
      threading.Semaphore()

THREADING MODULE

- Semaphore with a constained upper bound: threading.BoundedSemaphore()
- Create a thread: threading.Thread()
- Run a thread after timer expires: threading.Timer()
- To create a thread, call Thread constructor with a callable object (or override run() method), then the thread start()

THREADING MODULE

- Calling another threads join() causes the caller to wait for thread to end
- Pass arguments to new thread using the args and kwargs keywords
- Locks have acquire() and release() methods that can be automatically invoked on the with statement

THREADING MODULE

- Condition variables have a wait() to wait for the condition to be reached
- Condition variables also have notify() and notify_all() that can be used to indicate to other threads that the condition variable has changed
- Semaphore objects also have acquire() and release() for easy use with with
- Event objects have set(), clear(), and wait() which indicate if event has happened or not

LAB06

PYTHON DECORATORS

- Decorators are syntactical sugar for function wraps
- Start with '@' followed by the decorator name
- The decorator immediately precedes the decorated function and is called instead when the decorated function is called

PYTHON DECORATORS

- The decorated function reference is passed to the decorator, which can call it or replace it
- Decorated function arguments are also available to the decorator
- The decorator returns a new function which is executed
- See the built-in *@property* for an example

LAB07

WSGI

- Web Server Gateway Interface (PEP333)
- Standardized interface to web app server to allow portability of apps
- WSGI app is a callable with parameters for environment dictionary and a callback function that sets HTTP response code and headers
- WSGI server simply calls the WSGI app and waits for completion, then returns the response to the client
- Almost all python app servers and apps are WSGI compliant

LAB08

DAY 3 SUMMARY/Q&A

THE END