Advanced Python

Here, there be dragons

Topics to cover

- Ctypes - all the power of C, all the confusion of combining two languages into one program
- Regular Expressions - Using a robust engine to search for strings.
- Multithreading - Python’s jilted lover
- Fuzzing - Using bad input to do bad things

CTYPES

C + Python = Fun (and mayhem)

Ctypes

- Ctypes are a means of using C code within a Python program.
- They allow you to do things which may not have been included in Python, or which require special modules/libraries.
- They are tricksy beasts, and require a bit of setup.

But first! A discussion of C

- C is a lower-level language than Python (still high-level compared to Assembly, but we’re going to avoid that rabbit hole)
- Most operating systems are written some variation of C, which is compiled into machine-code so that the processor can read it.
- It’s faster than Python, and capable of more granular work, but it’s also harder to use, and takes more work.

Calling Conventions in C

- C primarily uses two calling conventions on x86 platforms (the kind of chip you have in your computer): cdecl and stdcall.
- In Windows C, stdcall is called WINAPI and cdecl is called WINAPIV.

Calling Conventions in C

- Stdcall is used for functions which always take the same number of arguments.

- def foo(a,b,c)

- cdecl is used for functions which take a variable number of arguments.

- def bar (a,...)

DLL’s

- A DLL (Dynamically Linked Library) is used to gain access to functions from other code. They’re similar to Python modules, but require a bit more legwork
- You can use DLL’s to access the various ctype functions

Major DLL’s

- kernel32.dll - exports memory management, I/O, and process/thread functions.
- user32.dll - Makes GUI’s possible. This is what is implemented to make the OS user friendly.
- msvcrt.dll - allows programmers to use Linux-C style code without making major source code changes

Data Types

CTYPES	C TYPE	PYTHON TYPE
c_bool	_Bool	bool(1)
c_char	char	1-character string (ANSI)
c_wchar	wchar_t	1-character string (Unicode)
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long

Data Types (cont)

CTYPES	C TYPE	PYTHON TYPE
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	_int64/long long	int/long
c_ulonglong	unsigned _int 64/unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float

Data Types (cont)

CTYPES	C TYPE	PYTHON TYPE
c_char_p	char * (NULL termination)	string or None
c_wchar_p	wchar_t * (NULL termination)	unicode or None
c_void_p	void *	int/long or None

Pointers in CTYPES

- Long story short, a Pointer is a memory address. Python doesn't generally use these, for various reasons.
- Pseudo-code Example:

```
int a = 0;
        In C, this is how you create an integer variable named "a", and assign
        it the value of 0
int *b = &a;
        This is how you create a pointer ("b") to the value "a".
```

Pointers in CTYPES (cont)

- After creating the variables, you can access the value by using either the original, or by dereferencing the pointer.
  a = 1;
  *b = 1; ⇐ The star accesses the value pointed to by b

Why bother with Pointers?

- The general response to pointers is rarely a positive one. They seem complicated and difficult to handle. So why does anyone bother? The answer lies in the difference between passing by reference or passing by value.

Reference VS Value

- C passes by **reference** - meaning that you pass the address of the data, rather than copying it all over.
- Python passes by **value** - meaning that any time you pass a variable to a function, the called function copies the entire contents of that variable.

Reference VS Value (cont)

- In passing by reference, any changes you make to the addressed data will show for any functions which access that data.
- In passing by value, you can do whatever you like to the data you receive, the original won't be altered.

Assigning data types

- Like so much else in Python, CTYPES data types are just classes with some dressing up. To create a CTYPES int, you simply use:

  c_int(value)

Calling CTYPES Functions

- We’ll start out with something simple: printf.
- printf is similar to Python’s “print” statement, with a few differences we don’t need to discuss here.
- You call it as follows:

  cdll.msvcrt.printf(“STRING”)

Calling CTYPES Functions (cont)

- Now let’s try something with a bit more of a punch: MessageBoxA
- All those friendly Error boxes Windows throws up are created using a MessageBox function, and it’s a good demo of using User32 to do GUI work.

Regular Expressions

Like Where's Waldo, but harder

re module

- In Python, Regular Expressions are accomplished by way of the “re” module. The Syntax takes a bit of getting used to, but when you get the hang of things, you’re able to do some pretty cool magic.

RE Syntax

- Regular Expressions consist of two types of characters, *ordinary* (‘A’,‘a’,‘B’,‘b’, and so on) and *special* (‘.’,‘*’,‘?’, and so on).
- We’re going to use a fairly simple Regular Expression to get the hang of how the Syntax looks.

‘\d{3}-\d{3}-\d{4}’

- At a glance, this seems pretty arcane. A bunch of \’s, {}’s, and numbers. However, this is actually a pattern to find a phone number (American, it doesn’t deal with international numbers).

‘\d{3}-\d{3}-\d{4}’

- \d ⇐ This is a special character which looks for any integer (it’s the same as writing [0-9])
- {3} ⇐ This specifies how many of a character you’re looking for. In this case, we want three (or four, at the end) numbers.
- \- ⇐ This is exactly what it looks like. It’s nothing more or less than the “-” character.

Special Characters in RE

- “.” ⇐ Matches any character (except newlines)
- “[]” ⇐ Matches anything in a given set
  [a-zA-z] Matches any letter
  [a-z] Matches any lowercase letter
- “^” ⇐ Matches anything *not* in a given set.
  ex: [^a1] matches any character except a or 1
- “*” ⇐ Matches 0 or more of a given character or set

Special Characters in RE (cont)

- “+” ⇐ Matches 1 or more of a character or set
- “?” ⇐ Matches 0 or 1 of a character or set
- “a|b” ⇐ Matches either “a” or “b”

re Methods

- re.compile ⇐ turns the pattern into something the regex engine can understand.
- re.search ⇐ scans a string for the pattern and returns a MatchObject for the first match
- re.match ⇐ only checks the beginning of the string for the pattern.

re Methods (cont)

- re.findall ⇐ returns *every* match, rather than only the first.
- re.finditer ⇐ returns an iterator over all matches
- re.sub ⇐ replaces the first instance of a match with a given string

Match Objects

- When a pattern finds a match in a string, it returns what is known as a “match object”. Like any other object, this has several Methods associated.

Match Object Methods

- reobject.group ⇐ returns subgroups of a Match Object.
- reobject.start/reobject.end ⇐ return the string indices at which the match starts and ends
- reobject.re ⇐ contains the regex pattern matched

Multi-threading

PEBCAK

“Python’s Jilted Lover”

- During the introduction, I referred to Multi-threading as “Python’s Jilted Lover”. This is for a reason which is actually somewhat complicated, and requires a bit of understanding as to how multi-threading works.
- The first thing we need to do is define what threads actually *are*, and how they’re different from processes.

Processes and Threads

- Processes are full programs. They have their own PID (process ID) and their own PEB (Process Environment Block).

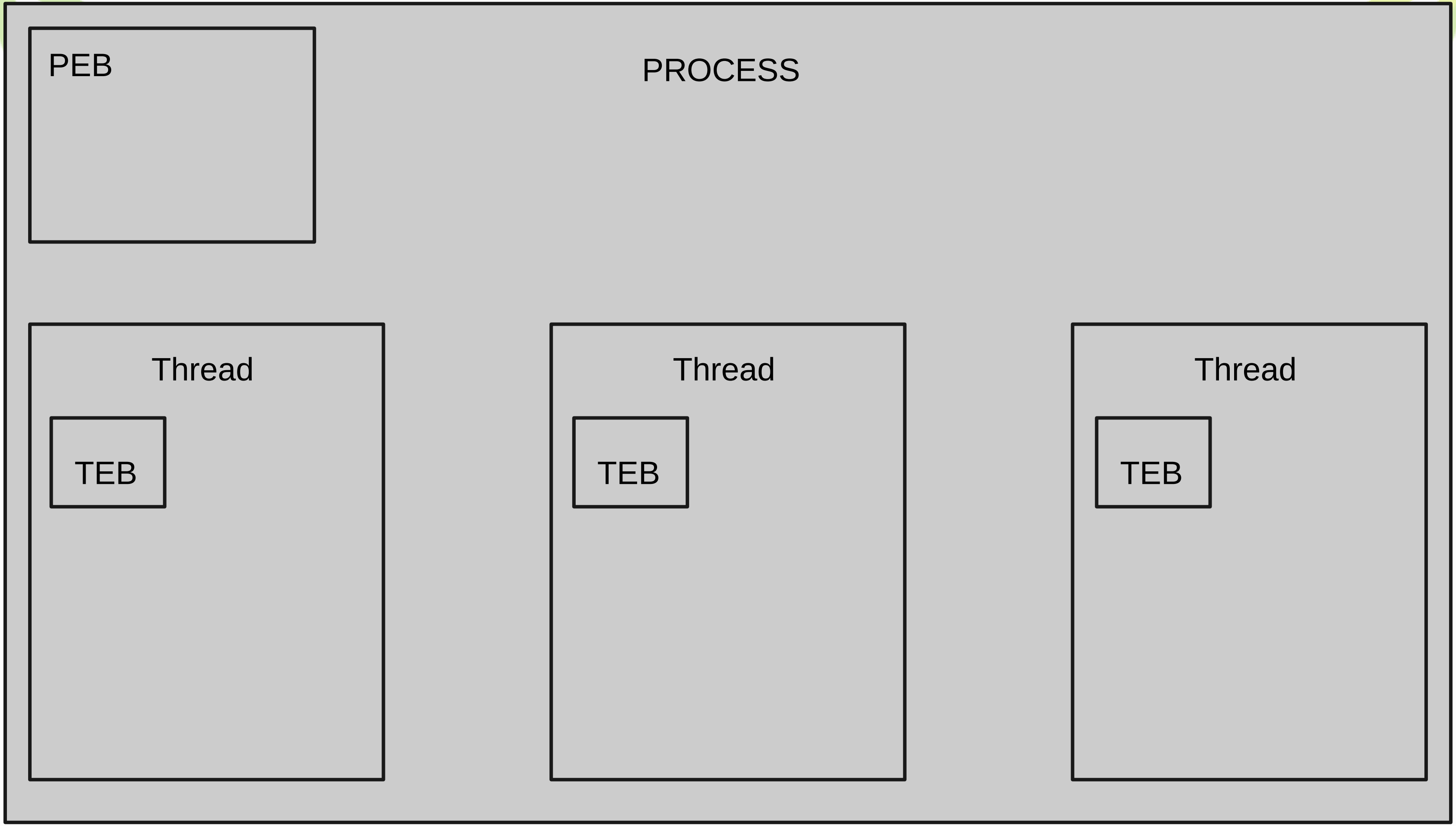
- Processes can contain multiple threads.

- If a process terminates, the associated threads do so as well.

- Threads are separate execution paths within a process. They don’t have PID’s, but they do each have a TEB (Thread Environment Block).

- Threads can only be associated with one Process.

- Processes can continue after threads terminate (so long as there is at least one thread left)


PEB
PROCESS
Thread
TEB
Thread
TEB
Thread
TEB

Advantages of Multi-threading

- Speed ⇐ by having more threads work on the same problem, you can increase speed dramatically.
- Granularity ⇐ especially in network applications, having a thread for each client allows for more fine control over the connections.
- Robustness ⇐ if one thread crashes and burns, you can continue the program and avoid a potential catastrophic termination

Disadvantages of Multi-threading

- Debugging ⇐ It’s harder to track down errors over multiple threads.

- Resource fights ⇐ if multiple threads are trying to access the same data, explosions will happen

Python’s little problem

- Multithreading is problematic in Python because of a fundamental part of the standard Python Interpreter; the Global Interpreter Lock.
- The GIL only allows a single Python instruction to be executed at a time, meaning parallelization is essentially neutralized in Python Multi-threading. The speedup normally gained is nowhere to be found, because no matter how many threads you have, they’re bottlenecked by the GIL.

But it’s not so bad

- This bottlenecking doesn’t remove the benefit of multithreading for the sake of multiple network connections, or for the sake of program robustness. Only the speed is affected.
- The damage to the Multi-threading speedup can also be neutralized by utilizing an interpreter which doesn’t have the GIL. Jpython is typically a favorite, though Stackless uses a remarkable implementation which gains all the benefits (or at least most) without many of the drawbacks of multi-threading.

Thread Creation Methods

- import threading
- threading.Thread() ⇐ instantiate a thread
- threadobj.start() ⇐ begin execution
- threadobj.isAlive() ⇐ see if thread is extant

Resources

- As mentioned, if multiple threads attempt to access the same resource, there will be a Michael Bay-esque explosion.
- The solution to sharing resources without creating race conditions to access data is by use of things like semaphores and mutexes

Semaphores

- A semaphore is used to guard limited resources. For example, if you only have 5 slots available for a task, a semaphore can be used to ensure that only 5 slots are allowed in use.
- This is accomplished by means of an internal counter. Every time someone acquires the semaphore, this counter decreases. Every time someone releases the semaphore, this counter increases.

Mutexes

- A mutex (mutually exclusive) lock is a means of controlling access to a resource. For example, if one thread is using a file handle, a mutex prevents another thread from accessing the file handle.

Fuzzing

The Art of Hitting Things with Sticks

What is Fuzzing?

- While every topic in this class has been necessary for a security professional using Python, Fuzzing is unique in that it is pretty much *only* useful to security professionals.

How does Fuzzing work?

- You find an application which takes input (pretty much any of them), and start sending data. You start fairly simply; misspelled commands, bad arguments, etc. As you progress, you get into the bigger stuff; sending gigs of data, or sending files with bits flipped to change the control paths.

Types of Fuzzing

- String Mutation (the most common type) - taking a string or letter, and changing it to find a break condition.

  “a” * 1000000000 #this will break pretty much anything if it’s insecure.

- Metadata/File Format Fuzzing - A bit (actually, a lot) more complicated, but useful when trying to break applications which take in files.

- Malformed Arguments - If you know what kinds of arguments a file takes, you can malform them and do some pretty interesting damage.

What makes Fuzzing possible?

- Like most any exploitable condition, user error. Fuzzing is only useful against systems which improperly sanitize input, or which take more data than they can handle.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
        char password = “SlithyToves”;
        char input[11] = {0};
        int bpassaccept = 0;

        strcpy(input, argv[1]);
        if (strcmp(input , password)==0)
        {
                bpassaccept = 1;
    }
        if (bpassaccept != 0)
        {
                grantaccess();
    }
        return 0;
}
```

Password cracking

- This is included in the Fuzzing section, since it's a very related task. Fuzzing involves sending random data to crash a target system. Password cracking (at least, the brute force kind) involves simply trying lots of things quickly. It will also allow us to get a sense of what fuzzer-style code looks like.

```
from hashlib import md5
import sys


def passcrack(pass_hash):
    for i in range(1001):                    #try 1-1000
        m = md5()#reset m
        m.update(str(i))                     #calculate the hash
        test_hash = m.hexdigest()
        if (test_hash != pass_hash):         #check the hash
            print "Failed: %s\t%s" % (test_hash, pass_hash)
        else:
            print "Success: %d" % i
            return

m=md5()
m.update(str(sys.argv[1]))
passcrack(m.hexdigest())
```