**Go and Python**

**Using Python to illustrate Dynamic linking with Golang**

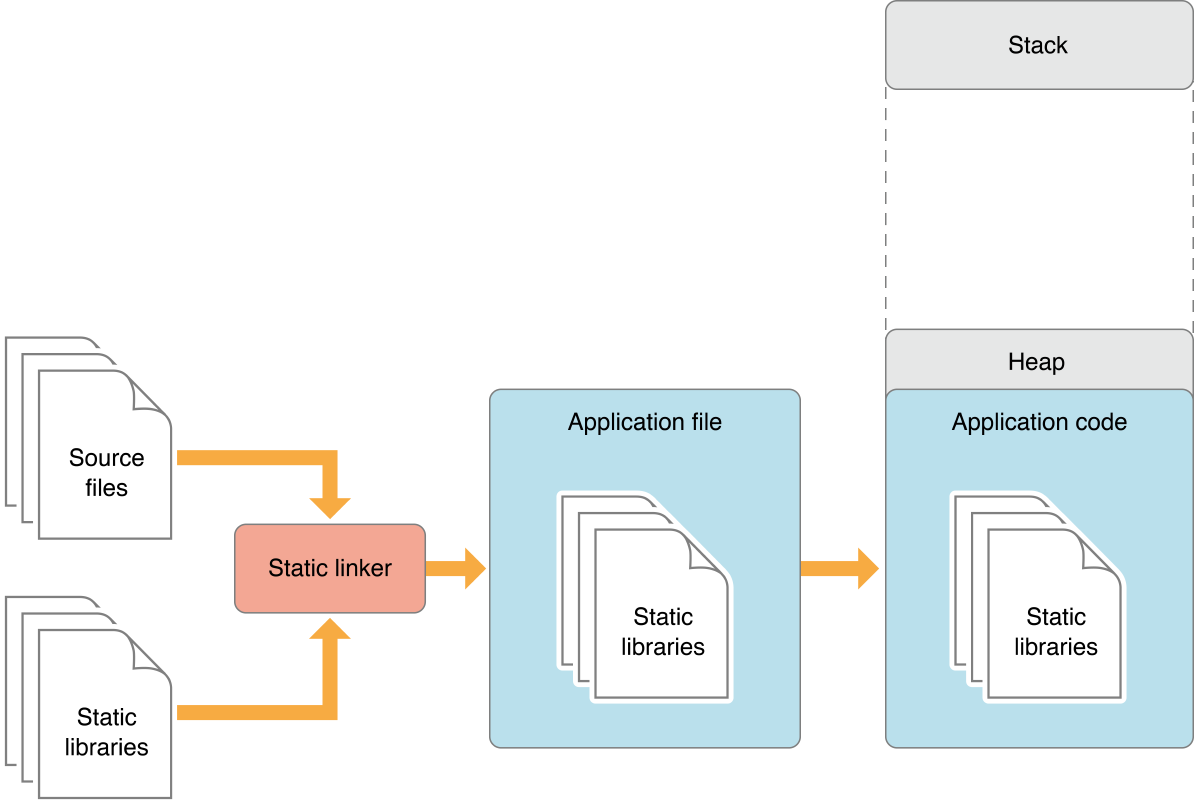
**System programming summary to this point**

In order to be used, object code must either be placed in an executable file, a library file, or an [object file](https://en.wikipedia.org/wiki/Object_file).

Object files can in turn be [linked](https://en.wikipedia.org/wiki/Linker_(computing)) to form an executable file or library file.

A linker links several object (and library) files to generate an executable.

**Static Linking**

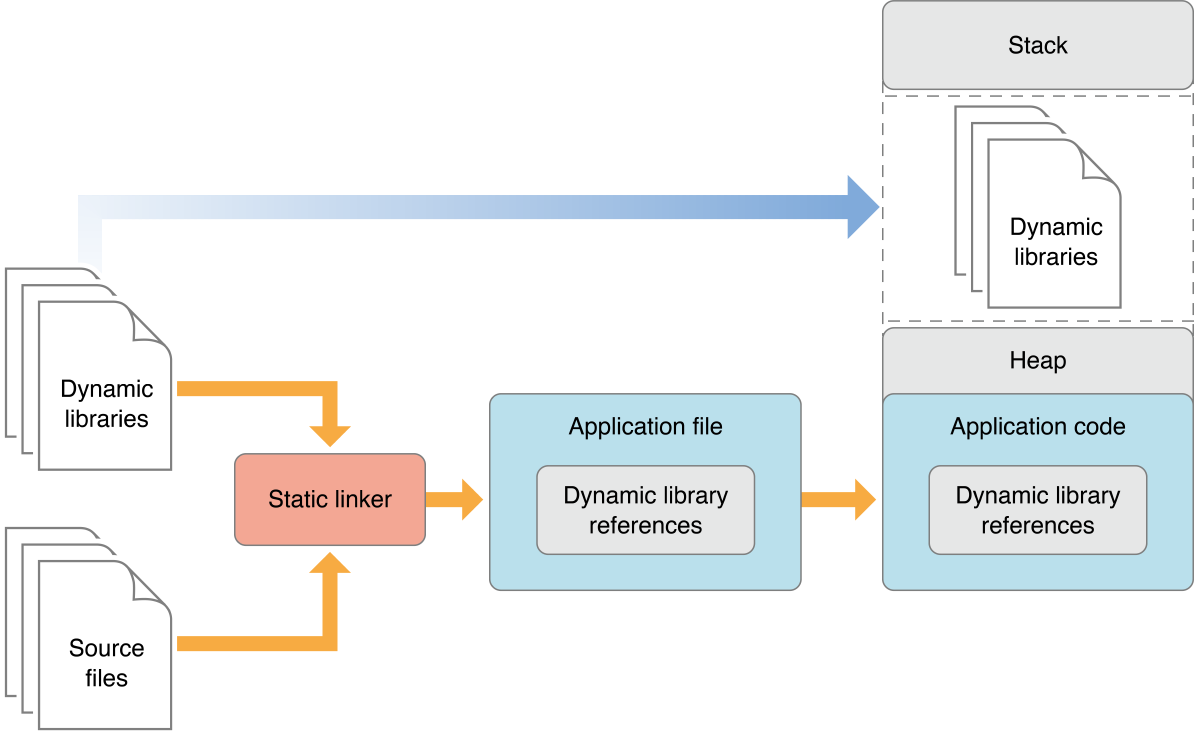


When a binary is launched, the binary code—which includes the code of the

static libraries it was linked with—is loaded into the binary address space.

Linking many static libraries into a binary produces a large file.

**Dynamic linking**



A better approach is for a binary to load code into its address space when it’s actually needed, either at launch time or at runtime. The type of library that provides this flexibility is called dynamic library. Dynamic libraries are not statically linked into client apps; they don't become part of the executable file. Instead, dynamic libraries can be loaded (and linked) into a binary either when the binary is launched or as it runs.

**Statically linking is conceptually simpler and practically simpler.**

**Dynamic linking can lead to a maintenance nightmare.**

**Python and Dynamic linking**

**Python supports dynamically loadable modules via its import statement which goes out to the filesystem looking for a shared library’s as well as a “.py” files**

**We can make Golang system libraries that will be dynamically loaded at runtime by python**

**Dealing with the maintenance nightmare**

**Application binary interface (ABI)**

As application programmers we are already familiar with the concept of an API. The API allows organised access to the types you use in your code.

An ABI is very similar at the machine-language level you access the library though the ABI the ABI defines the types you use in your code only on a lower level.

ABIs help you deal with the maintenance nightmare that arise as a result of dynamic linking. If you code to use a particular library which is later updated, you will need to rebuild the code to be compatible with the updates. If however the updated library uses the same ABI, then this step is avoided. Two versions of a library are "binary-compatible" when they have the same ABI.

**Python stands out as It posses a very stable ABI**

System programmers may have to deal with ABIs directly when writing programs in a mix of programming languages, using [foreign function call](https://en.wikipedia.org/wiki/Foreign_function_call) interfaces between them.

<https://en.wikipedia.org/wiki/Application_binary_interface>

**Foreign function interface (FFI)**

A foreign function interface (FFI) is a mechanism by which a program written in one [programming language](https://en.wikipedia.org/wiki/Programming_language) can call routines or make use of services written in another

<https://en.wikipedia.org/wiki/Foreign_function_interface>

[Java](https://en.wikipedia.org/wiki/Java_(programming_language)) refers to its FFI as the JNI ([Java Native Interface](https://en.wikipedia.org/wiki/Java_Native_Interface)) or JNA ([Java Native Access](https://en.wikipedia.org/wiki/Java_Native_Access)).

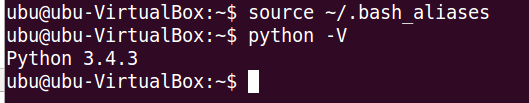
**Example Using Golang to illustrate Dynamic linking**

need python 3.4

python 3.4 is installed on the stable release of Ubuntu 14.0

~/.bash\_aliases

alias python=python3



**Step 1 Build a C object for Python**

**CPython**

C API to interact with it and for dealing with Python types.

**Stable Application Binary Interface**

Since Python 3.2, a subset of the API has been declared to guarantee a stable ABI.

<https://docs.python.org/3.4/c-api/stable.html>

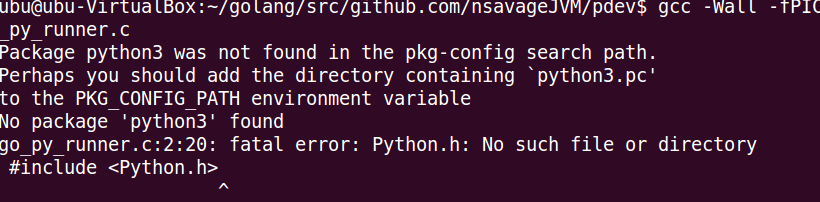
**Good Object linking for CPython exists**

<https://docs.python.org/3.4/extending/extending.html>

We can first illustrate the simple concepts involved by building a Python package using the C api with a simple object/function

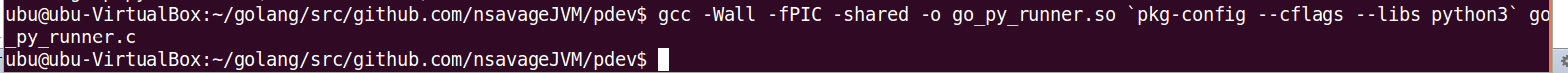
gcc -Wall -fPIC -shared -o go\_py\_runner.so `pkg-config --cflags --libs python3` go\_py\_runner.c

**solution for**



solution **sudo apt-get install python3-dev**

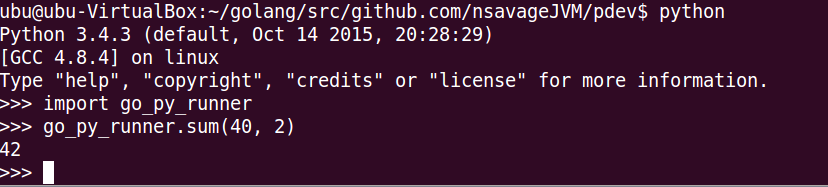
now errors are gone



now we have the object, we can import it as a python package



**run python import object file and call function**



**Simple recipe**

We exposed a PyInit\_go\_py\_runner(void) function, which calls PyModule\_Create passing in a pointer to a PyModuleDef metadata object.

This produces a PyMethodDef object with a pointer to our PyObject \* sum(PyObject \*, PyObject \*) function, which is the object code we want go\_py\_runner.sum() to run.

**Building a Go Python module**

now we have the basics we can look at the equivalent go code that is building a Go object file and import as a Python module.

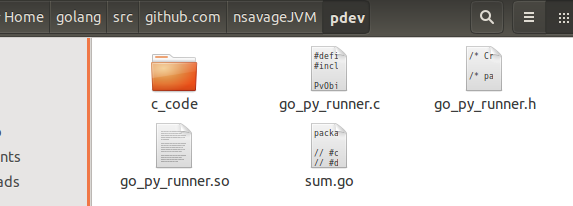
**Command cgo** <https://golang.org/cmd/cgo/>

// more complicated c toolchain

**gcc -Wall -fPIC -shared -o go\_py\_runner.so `pkg-config --cflags --libs python3` go\_py\_runner.c**

// simplified one language toolchain

**go build -buildmode=c-shared -o go\_py\_runner.so**



**python3 -c 'import go\_py\_runner; print(go\_py\_runner.sum(2, 40))'**