CS 45, Lecture 7
Compilers

Spring 2023

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Learning Goals

In this lecture, we will see:

- How UNIX and C were designed together
- How C compilers work
- How package managers work
- How Python has taken over UNIX scripting

Outline

1. The C Language
2. Compilation
3. Package Management
4. Python

Outline

1. The C Language
2. Compilation
3. Package Management
4. Python

C

- C is a general-purpose programming language from the 1970s.

C

- C is a general-purpose programming language from the 1970s.
- C is used everywhere, and has inspired:
  - Every major operating system
  - Every mainstream programming language

C

- C is a general-purpose programming language from the 1970s.
- C is used everywhere, and has inspired:
  - Every major operating system
  - Every mainstream programming language
- Even if you never write C, you indirectly use it every day.

Before C

- Nowadays, C is considered a weird, old (and sometimes scary!) language.

Before C

- Nowadays, C is considered a weird, old (and sometimes scary!) language.
- The languages before C were *even worse*.

Before C

- Nowadays, C is considered a weird, old (and sometimes scary!) language.
- The languages before C were *even worse*.

This is the “hello, world!” program in B (the language before C):

```
main( ) {
    extern a, b, c;
    putchar(a); putchar(b); putchar(c); putchar('!*n');
}

a 'hell';
b 'o, w';
c 'orld';
```

Early C

- C was designed in the 1970s by Dennis Ritchie, one of the authors of UNIX.

Early C

- C was designed in the 1970s by Dennis Ritchie, one of the authors of UNIX.
- It was officially published in the 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie, commonly called “K&R C”.

Early C

- C was designed in the 1970s by Dennis Ritchie, one of the authors of UNIX.
- It was officially published in the 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie, commonly called “K&R C”.

Here's the “hello, world” program in K&R C:

```
main( ) {
        printf("hello, world");
}
```

Early C

- C was designed in the 1970s by Dennis Ritchie, one of the authors of UNIX.
- It was officially published in the 1978 book *The C Programming Language* by Brian Kernighan and Dennis Ritchie, commonly called “K&R C”.

Here's the “hello, world” program in K&R C:

```
main( ) {
        printf("hello, world");
}
```

The C programming language was used to implement most of UNIX's kernel and userspace.

Problems with K&R C

Early C had some issues:

- The compiler just translated C into assembly language. This assembly then needed to be “assembled” into machine language.

Problems with K&R C

Early C had some issues:

- The compiler just translated C into assembly language. This assembly then needed to be "assembled" into machine language.
- If you didn't tell the compiler what data type a variable was, it just assumed it was an integer.

Problems with K&R C

Early C had some issues:

- The compiler just translated C into assembly language. This assembly then needed to be “assembled” into machine language.
- If you didn't tell the compiler what data type a variable was, it just assumed it was an integer.
- It had no memory protection or error detection, so even minor bugs would cause your program to crash.

Problems with K&R C

Early C had some issues:

- The compiler just translated C into assembly language. This assembly then needed to be "assembled" into machine language.
- If you didn't tell the compiler what data type a variable was, it just assumed it was an integer.
- It had no memory protection or error detection, so even minor bugs would cause your program to crash.
- Some OS-specific functions, like `printf`, had to come from *somewhere*. This meant the machine code had to be "linked" to a C "standard library" which came with the OS.

Modern C

- Now we have *modern* versions of C, like C99, C11, and C23!

Modern C

- Now we have *modern* versions of C, like C99, C11, and C23!
- These versions fix... none of the problems I mentioned.

Modern C

- Now we have *modern* versions of C, like C99, C11, and C23!
- These versions fix... none of the problems I mentioned.
- The C way of doing things is now just considered the “correct” way of doing things, so we're stuck with it.

Modern C

- Now we have *modern* versions of C, like C99, C11, and C23!
- These versions fix... none of the problems I mentioned.
- The C way of doing things is now just considered the "correct" way of doing things, so we're stuck with it.

Here's the "hello, world" program, rewritten in modern C:

```
#include <stdio.h>
int main(int argc, char **argv) {
  printf("hello, world\n");
}
```

Outline

1. The C Language
2. Compilation
3. Package Management
4. Python

How source code becomes machine code

The modern C-style process of compilation can be broken into three steps:

1. A COMPILER turns C code into assembly code (`cc`).
2. An ASSEMBLER turns assembly code into machine code (`as`).
3. A LINKER takes many different pieces of machine code (often from different source code files) and weaves them into a single program (`ld`).

How source code becomes machine code

The modern C-style process of compilation can be broken into three steps:

1. A COMPILER turns C code into assembly code (`cc`).
2. An ASSEMBLER turns assembly code into machine code (`as`).
3. A LINKER takes many different pieces of machine code (often from different source code files) and weaves them into a single program (`ld`).

Modern C compilers let you do all of these steps with a single command, but they still do each step separately behind the scenes.

Compiler Input

A COMPILER reads source code, like this:

```
#include <stdio.h>
int main(int argc, char **argv) {
  printf("hello, world\n");
}
```

This is meant to be human-readable and portable. The same code would work on Linux on an Intel CPU or macOS on an ARM CPU.

We can compile this by running `cc -S hello.c -o hello.s` to get an assembly file called `hello.s`.

Compiler Output

A COMPILER writes assembly code, like this:

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
```

```
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.2.1 20230111"
        .section        .note.GNU-stack,"",@progbits
```

This is technically still considered human-readable!

Assembler Input

An ASSEMBLER reads assembly code, like this:

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
```

```
        subq    $16, %rsp
        movl    %edi, -4(%rbp)
        movq    %rsi, -16(%rbp)
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        call    puts@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (GNU) 12.2.1 20230111"
        .section        .note.GNU-stack,"",@progbits
```

We can assemble this by running `as -c hello.s -o hello.o` to get an object file called “hello.o”.

Assembler Output

An ASSEMBLER writes machine code, which is not human-readable, but the disassembly looks like this:

```
Disassembly of section .text:

0000000000000000 <main>:
 0:     55                      push   %rbp
 1:     48 89 e5                mov    %rsp,%rbp
 4:     48 83 ec 10             sub    $0x10,%rsp
 8:     89 7d fc                mov    %edi,-0x4(%rbp)
 b:     48 89 75 f0             mov    %rsi,-0x10(%rbp)
 f:     48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 16 <main+0x16>
16:     48 89 c7                mov    %rax,%rdi
19:     e8 00 00 00 00          call   1e <main+0x1e>
1e:     b8 00 00 00 00          mov    $0x0,%eax
23:     c9                      leave
24:     c3                      ret
```

The call to `printf` is missing because the assembler doesn't know where it is! This incomplete machine code is called an OBJECT FILE.

Linker Input

An LINKER reads incomplete machine code in object files:

```
Disassembly of section .text:

0000000000000000 <main>:
 0:     55                      push   %rbp
 1:     48 89 e5                mov    %rsp,%rbp
 4:     48 83 ec 10             sub    $0x10,%rsp
 8:     89 7d fc                mov    %edi,-0x4(%rbp)
 b:     48 89 75 f0             mov    %rsi,-0x10(%rbp)
 f:     48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # 16 <main+0x16>
16:     48 89 c7                mov    %rax,%rdi
19:     e8 00 00 00 00          call   1e <main+0x1e>
1e:     b8 00 00 00 00          mov    $0x0,%eax
23:     c9                      leave
24:     c3                      ret
```

We could theoretically link this using `ld` to get an executable, but the exact command is complicated and system-dependent. Instead, we'll ask `cc` to do it:

```
cc hello.o -o hello
```

Linker Output

A LINKER writes complete machine code as an executable binary; once again, let's look at the disassembly:

```
0000000000001139 <main>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 83 ec 10             sub    $0x10,%rsp
    1141:       89 7d fc                mov    %edi,-0x4(%rbp)
    1144:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
    1148:       48 8d 05 b5 0e 00 00    lea    0xeb5(%rip),%rax        # 2004 <_IO_stdin_used+0x4>
    114f:       48 89 c7                mov    %rax,%rdi
    1152:       e8 d9 fe ff ff          call   1030 <puts@plt>
    1157:       b8 00 00 00 00          mov    $0x0,%eax
    115c:       c9                      leave
    115d:       c3                      ret
```

Now the call to `printf` is fixed (actually called `puts` because of a compiler optimization). This is because the program is now “linked” to the system C library, *libc*, which has a definition of `printf`/`puts`.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.
- On Linux, these files end with `.so`; on macOS, they sometimes end with `.dylib` instead. The Windows equivalent ends with `.dll`.
- `cc -shared test.c -o test.so` creates a shared object file containing all the functions from `test.c`.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.
- On Linux, these files end with `.so`; on macOS, they sometimes end with `.dylib` instead. The Windows equivalent ends with `.dll`.
- `cc -shared test.c -o test.so` creates a shared object file containing all the functions from `test.c`.
- These "shared objects" are linked at runtime by a DYNAMIC LINKER, which is part of the OS.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.
- On Linux, these files end with `.so`; on macOS, they sometimes end with `.dylib` instead. The Windows equivalent ends with `.dll`.
- `cc -shared test.c -o test.so` creates a shared object file containing all the functions from `test.c`.
- These "shared objects" are linked at runtime by a DYNAMIC LINKER, which is part of the OS.
  - These `so` files are usually in `/usr/lib/`
  - The environment variable `$LD_LIBRARY_PATH` can add more locations.

Shared Object Files

- We can compile C into a SHARED OBJECT, which is a library any program can use.
- On Linux, these files end with `.so`; on macOS, they sometimes end with `.dylib` instead. The Windows equivalent ends with `.dll`.
- `cc -shared test.c -o test.so` creates a shared object file containing all the functions from `test.c`.
- These "shared objects" are linked at runtime by a DYNAMIC LINKER, which is part of the OS.
  - These `so` files are usually in `/usr/lib/`
  - The environment variable `$LD_LIBRARY_PATH` can add more locations.
- The `ldd` command tells you which shared libraries a program requires to run; if they're not installed, the program will give an error and exit.

So why does anyone use C?

C code is buggy, hacky, and hard to understand. So why is it so popular?

So why does anyone use C?

C code is buggy, hacky, and hard to understand. So why is it so popular?

- It's fast: C translates really well into assembly, so C programs are orders of magnitude faster than programs written in some other languages.

So why does anyone use C?

C code is buggy, hacky, and hard to understand. So why is it so popular?

- It's fast: C translates really well into assembly, so C programs are orders of magnitude faster than programs written in some other languages.
- It's omnipresent: everyone else uses C for everything, so the only way to interact with their code is using C.

So why does anyone use C?

C code is buggy, hacky, and hard to understand. So why is it so popular?

- It's fast: C translates really well into assembly, so C programs are orders of magnitude faster than programs written in some other languages.
- It's omnipresent: everyone else uses C for everything, so the only way to interact with their code is using C.
- It's required: UNIX is *defined* in terms of C. A UNIX-based OS **must** have a C compiler. All the interfaces to ask the OS for anything are designed for C programs.

Outline

1. The C Language
2. Compilation
3. Package Management
4. Python

Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)

Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)
- C is so tightly integrated into the OS that we install C libraries the same way we install programs: a package manager.

Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)
- C is so tightly integrated into the OS that we install C libraries the same way we install programs: a package manager.
- On Linux, this comes with the OS: `apt`, `dnf`, `rpm`, `pacman`, etc.

Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)
- C is so tightly integrated into the OS that we install C libraries the same way we install programs: a package manager.
- On Linux, this comes with the OS: `apt`, `dnf`, `rpm`, `pacman`, etc.
  - If you've ever heard of a Linux "distribution", like Ubuntu, Debian, or Fedora, that basically means what package manager it uses.

Package Managers

- We get basic functions like `printf` and file I/O from *libc*, which comes with the OS.
- How do we get more complicated functions? (besides writing them ourselves)
- C is so tightly integrated into the OS that we install C libraries the same way we install programs: a package manager.
- On Linux, this comes with the OS: `apt`, `dnf`, `rpm`, `pacman`, etc.
  - If you've ever heard of a Linux “distribution”, like Ubuntu, Debian, or Fedora, that basically means what package manager it uses.
- On macOS, this is something we have to install, like `brew`.

Dependency Management

- When you install a program, it may depend on other programs or libraries.

Dependency Management

- When you install a program, it may depend on other programs or libraries.
  - For example, `firefox` depends on `ffmpeg`.

Dependency Management

- When you install a program, it may depend on other programs or libraries.
    - For example, `firefox` depends on `ffmpeg`.
- Those programs/libraries may themselves depend on other programs/libraries.

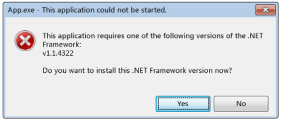
Dependency Management

- When you install a program, it may depend on other programs or libraries.
  - For example, `firefox` depends on `ffmpeg`.
- Those programs/libraries may themselves depend on other programs/libraries.
- Trying to figure out the entire dependency tree is somewhere between annoying and impossible, depending on the piece of software.

Dependency Management

- When you install a program, it may depend on other programs or libraries.
  - For example, `firefox` depends on `ffmpeg`.
- Those programs/libraries may themselves depend on other programs/libraries.
- Trying to figure out the entire dependency tree is somewhere between annoying and impossible, depending on the piece of software.

Without a dependency manager of some sort, you get annoying errors like this:



Solutions

One solution is STATIC LINKING.

- This means every dependency of a program must be *compiled into* that program's binary file.
- This is common nowadays; the iOS and Android app stores (more or less) use this.

Another solution is PACKAGING programs.

- This means every program comes with a *list* of dependencies which are automatically downloaded/installed alongside it.
- This is more widespread; UNIX uses it by default, as do programming language-specific tools like `pip`.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.
- Package managers may be *global*, like `apt` or `brew`, where any program on the same computer can later use a dependency.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.
- Package managers may be *global*, like `apt` or `brew`, where any program on the same computer can later use a dependency.
- Global package managers have the downside that they may have version conflicts; different programs may need different versions of shared libraries.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.
- Package managers may be *global*, like `apt` or `brew`, where any program on the same computer can later use a dependency.
- Global package managers have the downside that they may have version conflicts; different programs may need different versions of shared libraries.
- Package managers can also be *local* like `npm`, where they install dependencies into a specific “project”.

Package Managers

- A PACKAGE MANAGER is a tool which installs programs or libraries, and automatically takes care of resolving dependencies.
- Package managers may be *global*, like `apt` or `brew`, where any program on the same computer can later use a dependency.
- Global package managers have the downside that they may have version conflicts; different programs may need different versions of shared libraries.
- Package managers can also be *local* like `npm`, where they install dependencies into a specific "project".
- Local package managers have the downside that they may do redundant work; different programs may get duplicate copies of the same library.

Outline

1. The C Language

2. Compilation

3. Package Management

4. Python

Python

- Python is a "new"[1] programming language which has taken over a lot of the traditional roles of C within UNIX.

[1] The 90s is "new" as far as UNIX goes.

Python

- Python is a "new"[1] programming language which has taken over a lot of the traditional roles of C within UNIX.
- Python is a SCRIPTING LANGUAGE like `bash`; typically a programmer writes short scripts which combine tools written by someone else.

[1] The 90s is "new" as far as UNIX goes.

Python

- Python is a “new”[1] programming language which has taken over a lot of the traditional roles of C within UNIX.
- Python is a SCRIPTING LANGUAGE like `bash`; typically a programmer writes short scripts which combine tools written by someone else.
- Python is INTERPRETED; there's no compilation step needed, but you can't run a `.py` file on a computer that doesn't have `python` installed.

[1] The 90s is “new” as far as UNIX goes.

Python

- Python is a "new"[1] programming language which has taken over a lot of the traditional roles of C within UNIX.
- Python is a SCRIPTING LANGUAGE like `bash`; typically a programmer writes short scripts which combine tools written by someone else.
- Python is INTERPRETED; there's no compilation step needed, but you can't run a `.py` file on a computer that doesn't have `python` installed.
- Python is **compatible with C**.

[1] The 90s is "new" as far as UNIX goes.

Python Hello World

Python looks a lot more friendly than C:

```
#!/usr/bin/env python3
print("Hello, World!")
```

- Note the shebang line, like a shell script, but calling `python3` instead of `bash`.

Python Hello World

Python looks a lot more friendly than C:

```
#!/usr/bin/env python3
print("Hello, World!")
```

- Note the shebang line, like a shell script, but calling `python3` instead of `bash`.
- There are two incompatible versions of Python; `python` is usually Python 2, but `python3` is Python 3. **Use `python3` for anything new you write.**

Creating a new Python Project

- If you're writing a little standalone script, you can just create a Python file like you'd create a shell script. Remember the shebang line!

Creating a new Python Project

- If you're writing a little standalone script, you can just create a Python file like you'd create a shell script. Remember the shebang line!
- If you're writing a more complicated program, you probably want to create a new project.

Creating a new Python Project

- If you're writing a little standalone script, you can just create a Python file like you'd create a shell script. Remember the shebang line!
- If you're writing a more complicated program, you probably want to create a new project.
- As far as Python knows, a project is just a directory, so we can create a new Python project with `mkdir`.

Dependencies

- Much like C programs, we sometimes want our Python program to depend on code someone else wrote.

Dependencies

- Much like C programs, we sometimes want our Python program to depend on code someone else wrote.
- We can install programs with the Python package manager, `pip` (or more accurately, `pip3`).

Dependencies

- Much like C programs, we sometimes want our Python program to depend on code someone else wrote.
- We can install programs with the Python package manager, `pip` (or more accurately, `pip3`).
- However, we generally don't want to copy the C approach of installing libraries globally, since then we could only have one version of a library installed for all our Python scripts.

Dependencies

- Much like C programs, we sometimes want our Python program to depend on code someone else wrote.
- We can install programs with the Python package manager, `pip` (or more accurately, `pip3`).
- However, we generally don't want to copy the C approach of installing libraries globally, since then we could only have one version of a library installed for all our Python scripts.
- Instead, we create a VIRTUAL ENVIRONMENT and install our required dependencies in there.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.
- We can create a virtual environment named “myenv” in the current directory by running the command `python -m venv myenv`.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.
- We can create a virtual environment named “myenv” in the current directory by running the command `python -m venv myenv`.
- We can ACTIVATE the virtual environment by running the command `source ./myenv/bin/activate`.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.
- We can create a virtual environment named "myenv" in the current directory by running the command `python -m venv myenv`.
- We can ACTIVATE the virtual environment by running the command `source ./myenv/bin/activate`.
- We could deactivate the environment by running `deactivate`.

Virtual Environments

- A VIRTUAL ENVIRONMENT is like a little bubble isolated from other Python programs on your computer.
- Anything you install within a virtual environment will stay inside it.
- We can create a virtual environment named “myenv” in the current directory by running the command `python -m venv myenv`.
- We can ACTIVATE the virtual environment by running the command `source ./myenv/bin/activate`.
- We could deactivate the environment by running `deactivate`.

Let's practice this! First, create a directory named `python-test`; then create and activate an environment named `myenv` inside it!

Requirements

Let's install some packages inside your new virtual environment.

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.
- Now, let's install the `numpy` package, which lets us do vector math: `pip3 install numpy`.

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.
- Now, let's install the `numpy` package, which lets us do vector math: `pip3 install numpy`.
- We can now launch a Python Read-Eval-Print-Loop (REPL) by running `python3`. This lets us type Python commands and have them immediately executed, just like the shell!

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.
- Now, let's install the `numpy` package, which lets us do vector math: `pip3 install numpy`.
- We can now launch a Python Read-Eval-Print-Loop (REPL) by running `python3`. This lets us type Python commands and have them immediately executed, just like the shell!
- Inside the REPL, run `import numpy` to import the `numpy` library we just installed.

Requirements

Let's install some packages inside your new virtual environment.

- First, make sure the environment is activated.
- Now, let's install the `numpy` package, which lets us do vector math: `pip3 install numpy`.
- We can now launch a Python Read-Eval-Print-Loop (REPL) by running `python3`. This lets us type Python commands and have them immediately executed, just like the shell!
- Inside the REPL, run `import numpy` to import the `numpy` library we just installed.
- Try running `numpy.array([1, 2, 3]) + numpy.array([4, 5, 6])`.

Python Scripting

Let's write a Python script.

In fact, let's translate a shell script we wrote in Lecture 4—`my_folder.sh`—into Python!

Create a new file called `my_folder.py` and open it in your favorite editor.

Python Scripting: Example

```
akshay@akshays-thinkpad ~ % python my_folder.py akshay akshay.txt
```

```python
#!/usr/bin/env python3
import os
import sys

def make_my_folder(folder_name, file_name):
    os.mkdir(folder_name)
    os.chdir(folder_name)
    with open(file_name, 'a'):
        pass

make_my_folder(sys.argv[1], sys.argv[2])
```

Python Scripting: Example 2

Let's write a Python script that uses a dependency, `vector_norm.py`.

```
akshay@akshays-thinkpad ~ % python vector_norm.py 1 2 3
```

```
#!/usr/bin/env python3
import numpy as np
import sys

vector = np.array(sys.argv[1:])
print(np.linalg.norm(vector))
```

Packaging a Virtual Environment

The virtual environment you created is customized to your computer, so what do we do if we want to send a Python project to someone else?

Packaging a Virtual Environment

The virtual environment you created is customized to your computer, so what do we do if we want to send a Python project to someone else?

1. Create a `requirements.txt` file by running `pip3 freeze > requirements.txt`.

Packaging a Virtual Environment

The virtual environment you created is customized to your computer, so what do we do if we want to send a Python project to someone else?

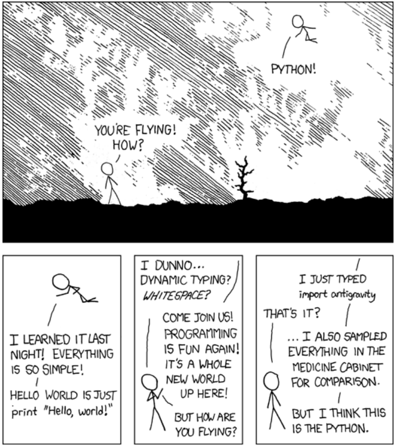
1. Create a `requirements.txt` file by running `pip3 freeze > requirements.txt`.
2. Send your Python files (`*.py`) and `requirements.txt` to someone else.

Packaging a Virtual Environment

The virtual environment you created is customized to your computer, so what do we do if we want to send a Python project to someone else?

1. Create a `requirements.txt` file by running `pip3 freeze > requirements.txt`.
2. Send your Python files (`*.py`) and `requirements.txt` to someone else.
3. Have the other person create a virtual environment and run `pip3 install -r requirements.txt` inside it to install all the requirements.

What's so great about Python?



Source: xkcd 353

What's so great about Python?

Python is fundamentally based on C, but hides it under really good abstractions.

What's so great about Python?

Python is fundamentally based on C, but hides it under really good abstractions.

If you really need to use C from inside Python, you can! But most of the time you never need to.

What's so great about Python?

Python is fundamentally based on C, but hides it under really good abstractions.

If you really need to use C from inside Python, you can! But most of the time you never need to.

Python scripts can link against C shared object files. The entire C software ecosystem is usable from inside Python.

Python-C Foreign Function Interface

Say we have a C file `add.c`:

```c
int add(int a, int b) { return a + b; }
```

Python-C Foreign Function Interface

Say we have a C file `add.c`:

```c
int add(int a, int b) { return a + b; }
```

We can compile it into `add.so`:

```
cc -shared add.c -o add.so
```

Python-C Foreign Function Interface

Say we have a C file `add.c`:

```c
int add(int a, int b) { return a + b; }
```

We can compile it into `add.so`:

```
cc -shared add.c -o add.so
```

We can use it from Python:

```python
import ctypes
lib = ctypes.CDLL("./add.so")
print(lib.add(1, 2))
```