Introduction to GDB, Python Classes, and Packaging

1. Debugging in GDB

1.1 Should We Demo Debugging Tools?

- Discussion on whether live demonstrations of debugging tools (like GDB) should be included in lectures.
- **Pros of demos:**
  - Shows real-time problem-solving and how functions behave in practice.
- **Cons:**
  - Demos take more time and might cover fewer topics.
  - Risk of being dull if not executed well.

1.2 GDB Essentials

1.2.1 Common GDB Commands

Command	Description
`run` / `r`	Start the program within GDB
`quit` / `q`	Exit GDB
`break <loc>` / `b`	Set breakpoint at function or line
`info break` / `ib`	List all breakpoints
`delete <num>` / `d`	Remove specified breakpoint
`continue` / `c`	Resume execution after break
`step` / `s`	Step into the next line (enters functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns
`print <expr>` / `p`	Evaluate and print a variable or expression
`watch <expr>`	Break when expression value changes

1.2.2 Breakpoints: Types and Management

- **Regular breakpoints:** Pause execution at a specific line or function.
- **Conditional breakpoints:** Only trigger if a condition is true.

```
b sqrt
condition 27 x < 0
```

- **Hardware breakpoints:** Use CPU support for efficient watchpoints (limited number).
- **Pitfalls:**
  - Breakpoints may not trigger if code is optimized/inlined.
  - Use `info break` to check all breakpoints.
  - Remove with `delete <num>`.

1.2.3 Stepping and Execution Control

Command	Action
`step` / `s`	Step into the next line (enters functions)
`next` / `n`	Step over function calls
`stepi`	Step a single machine instruction
`finish`	Run until current function returns

- **Note:** Stepping can be confusing with optimized code; compile with `-O0 -g3` for best results.

1.2.4 Stack-Related Commands

Command	Full Form	Description
`i f`	`info frame`	Shows information about the current function stack frame.
`bt` or `i s`	`backtrace` or `info stack`	Lists call stack trace, showing how program arrived at current function.

- Stack frames record what function called what.
- **Machine-level optimizations** (e.g., tail-call optimization) may cause missing frames in the trace. For example, if a function's last action is to call another, the intermediate frame may be omitted.
- Local variables may be optimized out and hence unavailable in GDB.
- Backtrace only records enough information for the system to continue execution, not every detail of the call history (e.g., loops are not shown).

1.2.5 Expression Evaluation

- `print expr`: Standard way to evaluate an expression.
- `display expr`: Continuously print value after each step.
- **Caution:** Calling functions from GDB can have side effects and may alter program state.
- GDB allows tooling with Python to customize how structures (like graphs) are displayed.
- You can write Python code to help GDB print complex data structures in a readable way.

1.2.6 Register Inspection

```
i r
p $xmm0
```

- `info registers`: Shows machine register values.
- `print $register`: Print a specific register value.
- Useful for performance checks and low-level debugging.
- Registers are fast-access storage; inspecting them can reveal performance-critical code paths.
- **Example:**

  ```
  (gdb) info registers
  (gdb) print $rax
  (gdb) print $xmm0
  ```

1.3 Remote and Cross-Platform Debugging

1.3.1 Debugging Different Architectures

```
target architecture-name
```

- Allows debugging code meant for different architectures (e.g., ARM, x86-64).
- GDB can debug programs running on a different architecture than the host.

1.3.2 Remote Debugging

- GDB can debug programs running on remote machines via serial ports or network (TCP/IP).
- Useful for embedded systems or IoT devices lacking GDB capabilities natively.
- Requires a copy of the executable on both the host and target machines.
- Configuration may involve specifying architecture and connection details (e.g., SSH, serial port).
- **Typical workflow:**
  1. On target: `gdbserver :1234 ./myprog`
  2. On host: `gdb ./myprog`, then `target remote <ip>:1234`
- **Security:** Only use on trusted networks or with SSH tunneling.

```
[Host GDB] --network--> [gdbserver on Target]
      |                         |
   (local exe)            (running exe)
```

1.4 GDB Macros and Extensions

1.4.1 GDB Macros

```
define printlook
  print *(long*)$arg0
end
```

- Custom debugging commands.
- `$arg1`, `$arg2`, etc., can be used for parameter substitution.
- GDB has its own macro language for automating repetitive tasks.
- **When to use:** For simple automation and repetitive command sequences.

1.4.2 Python Extensions in GDB

- GDB supports scripting in Python for advanced automation and pretty-printing.
- **Example:**

  ```
  # .gdbinit or loaded via 'source'
  python
  class MyPrinter:
      def __init__(self, val):
          self.val = val
      def to_string(self):
          return f"MyStruct: {self.val['field']}"
  def register_printers(objfile):
      gdb.pretty_printers.append(MyPrinter)
  end
  ```

- **When to use:** For complex data visualization, integration, or automation.
- **Comparison:** Use macros for simple tasks, Python for complex logic.

1.5 Alternative Debugging Strategies

1.5.1 Comparison Table: Assertions, Exceptions, Logging, Tracing

Strategy	Purpose	Example	Pros	Cons
Assertion	Catch bugs early	`assert(x >= 0);`	Fast, simple, runtime check	Can be compiled out, no recovery
Exception	Handle errors	`try: ... except ...`	Structured, recoverable	Can obscure flow, overhead
Logging	Record events	`print()`, logs	Persistent, post-mortem	Can be noisy, performance
Tracing	External monitoring	`strace`, `ltrace`	No code change, system-wide	Limited detail, setup needed

1.5.2 Debugging Process Diagram

```
[Observe Failure]
      |
      v
[Reproduce & Stabilize]
      |
      v
[Minimize Test Case]
      |
      v
[Form Hypothesis]
      |
      v
[Test Hypothesis]
      |
      v
[Fix & Retest]
```

1.5.3 Assertions

C-style:

```
assert(x >= 0);
```

- If `x < 0`, the program prints error and aborts.
- Can be compiled away using `-DNDEBUG`.
- **Restriction:** Assertions should be free of side effects; otherwise, program behavior may differ when assertions are compiled out.
- Difference from `unreachable`: Assertions are runtime checks; `unreachable` is a message to the compiler about code paths.

1.5.4 Exception Handling

Python-style:

```
try:
    do_something()
except ZeroDivisionError:
    handle_issue()
```

Alternative C-style:

```
if (do_something() < 0) {
    handle_issue();
}
```

Feature	Try-Catch	Manual Checking
Clearer mainline code	Yes	No
Easier to track flow	No	Yes

- **Try-catch**: Mainline code is clearer, but error handling can be less explicit.
- **Manual checking**: Error handling is explicit, but mainline code is cluttered.
- Both have pros and cons; know when to use each.

1.5.5 Logging and Tracing

- Print/log statements to track execution.
- **Logs:** Explicit in code (e.g., print statements, server logs). Levels: DEBUG, INFO, WARNING, ERROR, CRITICAL.
- **Traces:** Generated by external tools (e.g., `strace` for Linux, which logs system calls).
- Useful for debugging production systems where attaching a debugger is impractical.
- **Best practices:** Use log levels, avoid logging sensitive data, rotate logs.

1.5.6 Checkpoint-Restart

- Save application state at intervals.
- On crash, reload checkpoint to reproduce or recover state.
- Useful for reproducing bugs in long-running programs.
- **Tools:** `checkpoint`/`restart` in GDB, application-level checkpointing libraries.
- **Limitations:** May not capture all state (e.g., open sockets, external resources).

1.5.7 Barricades

- Architecture for legacy modernization.
- Build a clean subsystem separated by a "barricade".
- Only allow access via validated interfaces.
- Barricade checks all data crossing from messy to clean code, ensuring reliability.

Versioned cleanup pattern:

1. Start with this structure:

```
[Messy Code] --barricade--> [Clean Code]
```

2. Gradually clean messy code and push barricade outward.

```
+-----------------+    Barricade    +-----------------+
|  Messy Legacy  | ---------------> |     Clean New    |
|      Code       | <--------------- |      Code        |
+-----------------+   (interface)   +-----------------+
```

- Only validated data/operations cross the barricade.
- Over time, the barricade moves as more code is cleaned up.

1.5.8 Stronger Isolation Mechanisms

- **Interpreters:** Execute untrusted code through a carefully written interpreter. Slow but safe. Used in Chrome, Emacs, GDB, etc.
- **Virtual Machines and Containers:**
  - Virtual machines emulate entire systems, providing strong isolation.
  - Containers share host OS but isolate user space. Lighter, but less isolated than VMs.
  - Trade-off: Security vs. performance.

---

2. Python Programming Constructs

2.1 Functions and Lambdas

```
f = lambda x, y: x + y + 1
g = f
```

- Functions are first-class callables.
- Can be assigned to other variables, passed as arguments, or returned from functions.
- Lambda expressions create anonymous functions.
- **Example:**

  ```
  def apply_func(f, x):
      return f(x)
  print(apply_func(lambda y: y * 2, 5))  # 10
  ```

2.2 Python Classes and Inheritance

2.2.1 Classes as Objects

- In Python, classes are regular objects.
- Can be dynamically created and assigned.
- Each class has a `__dict__` that stores its attributes (a dictionary mapping names to values).
- `__dict__` is a reserved name; don't modify unless you know what you're doing.
- **Dynamic class creation example:**

  ```
  MyClass = type('MyClass', (object,), {'x': 42})
  obj = MyClass()
  print(obj.x)  # 42
  ```

2.2.2 Inheritance Model and Method Resolution Order (MRO)

```
class C(A, B):
    ...
```

- Multiple inheritance allowed.
- Python uses **depth-first left-to-right** search for method resolution, avoiding re-visits.
- **MRO Example:**

  ```
  class A: pass
  class B(A): pass
  class C(A): pass
  class D(B, C): pass
  ```

```
print(D.__mro__)
# (<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>)
```

- **Comparison Table:**
Language	MRO Strategy
Python	Depth-first, left-to-right
C++	Depth-first, ambiguous
Java	Single inheritance only
JavaScript	Prototype chain

2.3 Namespaces and Scoping

- Namespace: Collection mapping names to objects (essentially a `dict`).
- Each class, module, and function defines its own namespace.
- Namespaces prevent name collisions and organize code.
- **Diagram:**

```
[Global Namespace]
      |
      +-- [Module Namespace]
              |
              +-- [Class Namespace]
                      |
                      +-- [Instance Namespace]
```

---

3. Python Modules and Import System

3.1 Basic Module Mechanics

```
# foo.py
a = 19
def f(x): return x + 1
class C: ...
```

- Modules are Python files (`.py`) that define variables, functions, and classes.
- `import foo` creates a new namespace, executes `foo.py` in that namespace, and binds `foo` in the importer's namespace.
- Use `foo.a` to access `a` from the module.
- **Pitfalls:**
  - Circular imports (A imports B, B imports A) can cause issues; Python tries to avoid re-importing modules.
  - Name collisions: If you already have a variable named `foo`, importing a module with the same name will overwrite it.

3.1.1 Selective Import

```
from foo import f
```

- Imports only `f` from `foo` into the current namespace.
- `from foo import *` imports all names, but can cause name collisions and is discouraged unless necessary.

3.1.2 Import Styles Table

Style	Example	Pros	Cons
`import foo`	`import math`	Namespace isolation	Verbose
`from foo import f`	`from math import sin`	Direct access, less typing	Name collisions possible
`from foo import *`	`from math import *`	Quick, all names imported	Collisions, unclear origin

3.1.3 Circular Imports: Detection and Solutions

- **Detection:** Import errors, partially initialized modules, or `AttributeError` at import time.
- **Solutions:**

- Refactor shared code into a third module.
- Use local imports inside functions.
- Avoid top-level code that triggers imports.

3.1.4 Inspecting Modules

- `dir(foo)` lists all names in the module.
- The `builtins` module contains all built-in functions and names.

3.1.5 Running Modules as Scripts

- Running `python foo.py` creates a new namespace and sets `__name__` to `__main__`.
- Common idiom:

```
if __name__ == "__main__":
    # test code or main application
```

- Allows modules to be used both as importable libraries and as standalone scripts.

3.1.6 Import System Search Path Diagram

```
[Current Directory] -> [PYTHONPATH dirs] -> [Standard Library] -> [Site-packages]
```

3.2 Packages and Hierarchical Organization

- Packages are directories containing an `__init__.py` file.
- Packages allow hierarchical organization of modules (tree structure).
- Dots in import names correspond to directory structure (e.g., `import a.b.c` looks for `a/b/c.py`).
- `__init__.py` can perform package-level initialization.
- Relative imports (e.g., `from . import foo`) allow navigation within the package hierarchy.
- **init.py roles:**
    - Marks a directory as a package.
    - Can execute initialization code or expose selected submodules.
    - Pitfall: Missing `__init__.py` in older Python (❤️.3) breaks imports.

ASCII Diagram: Python Package/Module Hierarchy

```
my_app/
├── __init__.py
├── utils/
│   ├── __init__.py
│   └── math_helpers.py
└── models/
    ├── __init__.py
    └── user.py
```

- Each directory with `__init__.py` is a package.
- Modules are `.py` files within packages.

3.2.1 Comparison Table: Packages in Python vs. Other Languages

Language	Package Structure	Initialization File	Import Syntax
Python	Directory tree	`__init__.py`	`import a.b.c`
Java	Directory tree	none	`import a.b.C`
Node.js	Directory tree + package.json	none	`require('a/b/c')`

3.2.2 Multiple Libraries and PYTHONPATH

- Python searches for modules using the `PYTHONPATH` environment variable (colon-separated list of directories).
- The `import` statement searches these directories in order.

3.2.3 Package Managers and Virtual Environments

- **pip**: Python's package manager for installing, upgrading, and removing packages.
  - `pip install numpy` installs the NumPy library.
  - `pip uninstall numpy` removes it.
  - `pip list` shows installed packages.
  - `pip list --outdated` shows outdated packages.
  - `pip list --format=json` outputs in JSON format.
- **Dependencies:** pip installs dependencies automatically, but this can sometimes pull in unwanted packages.
- **Extending Python:** Most extensions are done via libraries, but you can also modify the Python interpreter itself (written in C) for deeper changes. Such changes are proposed via Python Enhancement Proposals (PEPs).
- **Virtual environments:**
  - `venv` (standard library, Python 3.3+): `python -m venv venv`
  - `virtualenv` (older, more features): `virtualenv venv`
  - `pipenv`, `conda` (alternative tools): manage dependencies and environments.
  - **Best practice:** Always use a virtual environment for projects to avoid dependency conflicts.

3.2.4 Comparison to Other Ecosystems

- Node.js uses `npm` for package management; similar issues and workflows as Python's `pip`.

---

4. Build Tools and Packaging for Compiled Languages (Preview)

- Building and packaging for compiled languages (C, C++, Rust) is more complex than for interpreted languages.
- Executables are platform-specific; even machines with the same architecture may have subtle differences.
- The build/distribution/install process involves multiple roles:
  - **Developers**: Write the code.
  - **Builders**: Compile the code.
  - **Distributors**: Ship executables to users.
  - **Installers**: Install executables on end-user machines.
- Simple scripts can automate builds, but more robust tools are needed for complex projects.
- **Build automation tools:**
  - `make`, `CMake`, `Bazel`, `Ninja` for C/C++/Rust.
  - Handle dependencies, platform differences, and incremental builds.
- **Comparison Table: Interpreted vs. Compiled Build/Distribution**

Aspect	Interpreted (Python)	Compiled (C/C++/Rust)
Portability	High (source code)	Low (binary format)
Build Step	None (run directly)	Required (compile/link)
Distribution	Source or bytecode	Platform-specific binaries
Dependency Mgmt	pip, venv, etc.	System pkg mgr, static/dyn link

ASCII Diagram: Build Process Stakeholder Flow

```
[Developer]
     |
     v
[Builder]
     |
     v
[Distributor]
     |
     v
[Installer]
     |
     v
[End User]
```

- Each role may be automated or manual.
- Artifacts flow from source code to executable to installed application.
- **Build pipeline diagram:**

```
[Source Code] -> [Build Tool] -> [Binary/Package] -> [Distribution] -> [Installation] -> [User Execution]
```