0) The core idea (plain English)

A **Control Flow Graph (CFG)** is a picture of "where the program can go next."

- A **Block** is a *chunk of straight-line code* (statements that run one after another with no jump in the middle).
- A **Link (edge)** is a possible jump from one block to another.
    - Example: after an `if`, control can jump to the "then" block or the "else/after" block.
    - Links can have a **condition label** ("exitcase"), like `x > 0` or `not(x > 0)`.

py2cfg works like this:

1. Parse Python source into an **AST** (Abstract Syntax Tree).
2. Walk the AST and create **blocks + links**.
3. Remove "useless" empty blocks.
4. Render the graph with Graphviz.
5. (Optional debug mode) While a debugger is stepping, highlight the blocks/edges that correspond to lines that executed.

---

1) `model.py` — the data model + graph rendering

This file defines the *objects* that represent the CFG and the logic to draw them.

A) `Block`

A `Block` is a "basic block":

- `id`: unique integer ID
- `statements`: list of AST nodes (the code inside the block)
- `func_calls`: list of function names called inside the block (for a call graph overlay)
- `predecessors`: incoming edges (Links)
- `exits`: outgoing edges (Links)
- `func_blocks`: extra nodes used to visualize nested calls/arguments as a call tree
- `highlight` and `outline`: flags used in interactive/debug mode for styling

Important methods:

- `at()` = line number of the **first** statement in the block
- `end()` = line number of the **last** statement
- `is_empty()` = block has no statements
- `get_source()` = convert AST nodes back into source text using `astor`
    - Special-cases:

        - `if/for/while`: only show the header line (like `if x:`) to keep nodes readable
        - function defs: show `def foo(...):...`
- `add_exit(next, exitcase)` creates a `Link(self → next)` and also registers it as a predecessor in `next`.

So a block is basically "a labeled node that knows its code and outgoing/incoming jumps."

B) `Link`

A `Link` is an edge:

- `source`: Block
- `target`: Block
- `exitcase`: the condition that must be true for this jump to happen (an AST expression/compare)
- `highlight`: styling flag

Methods:

- `jumpfrom()` / `jumpto()` = line numbers for the ends/starts
- `get_exitcase()` = pretty-print condition back to source

C) `FuncBlock` and `TryBlock`

These are specialized blocks for visualization / try-handling.

`FuncBlock(Block)`

Used only for the **call graph** display:

- `name`: function name like `foo` or `obj.method`
- `args`: list of FuncBlocks, representing nested calls used as arguments

This creates a tree like:

```
foo( bar(1), baz(qux()) )
```

so `foo` has args `bar` and `baz`, and `baz` has arg `qux`.

`TryBlock(Block)`

Adds:

- `except_blocks`: mapping like `{ "ValueError": blockX, None: blockY }`
  - `None` means "bare except:" (catch-all)

Overrides `get_source()` to show only the try body nicely.

D) `CFG`

This is the main graph object.

It stores:

- `name`, `asynchr`
- `entryblock`
- `finalblocks` (blocks where execution can end)
- `functioncfgs` and `classcfgs` (subgraphs)
- styling flags and line boundaries

Rendering / visualization

py2cfg uses **Graphviz**:

- `node_styles` defines shape/color per AST type (If = diamond, While/For = hexagon, Return = green parallelogram, etc.)
- `_build_key_subgraph()` creates a "legend" in the output (KEY).

How nodes get styled:

- `stylize_node(block)`:
  - looks at the block's first statement type
  - calls a specific handler like `node_ClassDef` if it exists, else `_style_handler`
- `_style_handler` returns `(shape, color, label)` where label is block source with `\l` line-breaks (Graphviz left-justified lines)

How edges get styled:

- `stylize_edge(link)`:
  - tries very specific methods like `If_to_Return`
  - else tries `edge_If`, `edge_While`, etc.
  - else uses `_edge_handler` (default black edge, label = exitcase)

Special edge styling:

- `edge_If`, `edge_While`, `edge_For` color edges:
  - green if the exit label seems to match the "true branch"
  - red otherwise

Graph traversal to draw everything

- `_visit_blocks(graph, block, visited, calls, ...)`
  - DFS traversal
  - creates a node for each block
  - optionally creates call subgraphs for function calls
  - recursively visits targets of exits, then adds edges

String-shortening logic:

- If `short` mode is on, it uses a regex to shorten long string literals (>20 chars) into `"first20..."`.

Interactive styling:

- `border_style` and `fillcolor` use `highlight`/`outline` flags in debug mode.

Utility operations

- `__iter__()` yields all blocks in this CFG and sub-CFGs (functions/classes)
- `own_blocks()` yields only blocks in this CFG (BFS from entry)
- `bsearch(lineno)` finds which block contains a line number (binary search on sorted blocks)
- `outline_block(lineno)` sets `outline=True` for the block at that line
- `highlight_blocks(lines)` sets `highlight=True` for blocks covering executed lines
- `find_path(finalblock)` tries to find one path from entry to a final block (DFS)

---

2) `builder.py` — turns an AST into blocks + links

This is the heart: it *constructs* the CFG.

A) `invert(node)`

Given a condition AST node, return its logical negation:

- `a == b` becomes `a != b`
- `x < y` becomes `x >= y`
- `True` becomes `False`
- otherwise wraps with `not (...)`

This is used for “false branch” edges.

B) `merge_exitcases(exit1, exit2)`

When you remove an empty block, you might connect a predecessor directly to a successor. If both edges had conditions, the real condition should be: `exit1 AND exit2`.

So it creates `ast.BoolOp(And, [exit1, exit2])`.

C) Try-handling helpers: `TryEnum` + `TryStackObject`

This builder tracks what “phase” it’s in for a try:

- building except handlers
- building body
- building else
- building finally

`TryStackObject` holds:

- `try_block`: the TryBlock node for this try
- `after_block`: where flow continues after try/except/else
- `has_final`: whether there’s a finally
- `iter_state`: which part is being visited right now

Why a stack? Because tries can be nested.

D) `CFGBuilder(ast.NodeVisitor)`

This uses Python’s visitor pattern:

- `visit_If` handles `if`
- `visit_While` handles `while`
- etc.

It maintains:

- `self.cfg`: the CFG being built
- `self.current_block`: where new statements go “right now”
- `self.current_id`: to give unique block ids
- `loop_stack`: used for break/continue targets (stack of `(after_loop, loop_guard)`)
- `try_stack`: used for raise/return interactions with try/finally

Build entry points

- `build(name, tree, ...)`
  - creates CFG
  - creates entry block
  - visits the AST
  - calls `clean_cfg` to remove empty blocks
- `build_from_src` parses source with `ast.parse`
- `build_from_file` loads file then calls build_from_src

E) Block creation helpers

- `new_block(statement=None)` creates a Block with a new ID and optional first statement.
- `new_try_block` same but TryBlock
- `new_func_block` creates FuncBlock for call tree rendering
- `add_statement(block, statement)` just appends statement to block
- `add_exit(block, nextblock, exitcase)` creates a Link and wires it up

F) The key structural trick: "loop guard blocks"

`new_loopguard()` ensures loops have a dedicated "test" node:

- If current block is empty and has no exits, reuse it as the guard.
- Otherwise create a new block, link current → guard.

This makes the CFG shape for loops consistent: a guard node that branches to body or after-loop.

G) Sub-CFGs for classes and functions

When the builder sees:

- `class X: ...`
- `def f(): ...`

It:

1. Adds the ClassDef/FunctionDef statement into the current block (so CFG shows that definition line)
2. Builds a new CFG for the body:

    - creates a new `CFGBuilder` (sharing the same stacks buffer)
    - builds CFG on `ast.Module(body=node.body)`
3. Stores it in `cfg.classcfgs[name]` or `cfg.functioncfgs[name]`
4. Updates line range info (`lineno`, `end_lineno`)
5. Updates `current_id` so block IDs remain unique across subgraphs

So the main graph keeps "definition nodes", and the function/class body becomes a nested subgraph.

H) `clean_cfg` — removing empty blocks

This is important.

If a block is empty, it "splices it out":

For each predecessor `pred` and each exit `exit` of the empty block:

- create a new link: `pred.source → exit.target`
- condition becomes `pred.exitcase AND exit.exitcase`
- remove old links from the structures to avoid dangling references

Then it recurses.

Result: the final CFG is cleaner (fewer meaningless "connector" nodes).

---

3) The actual control-flow construction (the visitor methods)

This is the most important part to "understand how it builds CFGs."

`visit_Expr`, `visit_Assign`, `visit_AnnAssign`, `visit_AugAssign`, `visit_Import`, `visit_ImportFrom`

These are simple:

- put the statement in the current block
- keep going

No new blocks or edges created.

`visit_Call`

This is special because py2cfg draws a *call subgraph*.

It extracts a function name (`visit_func`) from patterns like:

- `foo(...)`
- `obj.method(...)`
- `a[b](...)` (subscript call)
- nested calls like `foo(bar())`

Then it builds `FuncBlock`s to represent calls and call arguments as a tree.

Key idea:

- If you're currently visiting a call that is an **argument** of another call, it attaches as a child `args` under the parent FuncBlock.
- If you're not inside another call, it attaches to the current CFG block as a "call made here".

That's what `_callbuf` is: it's a stack of "currently-being-built calls".

So:

- start visiting a call → push its FuncBlock to `_callbuf`
- visit args → nested calls become children
- finish call → pop

`visit_If`

The CFG shape created is:

```
(cond block) --[test]--> (if body)
(cond block) --[not test]--> (else or after)
(if body end) --> (after)
(else body end) --> (after)
```

Step-by-step:

1. Ensure condition is in its own block if needed:
    - If current block already has statements, make a new `cond_block` and link into it.
    - Otherwise just add `If` statement into the current block.
2. Build `if_block` (true branch target)
3. Link `current(cond) → if_block` labeled with `node.test`
4. Build `afterif_block` (join point)
5. If there is an else:
    - make `else_block`

    - link cond → else_block with `invert(test)`
    - visit else statements in else_block
    - link else end → afterif_block
      Otherwise:
    - link cond → afterif_block with `invert(test)` (false falls through)
6. Visit if body in `if_block`
7. Link if end → afterif_block
8. Continue from afterif_block

That's classic CFG formation for if/else.

`visit_While`

It creates:

- a guard block that contains the `While` node
- true edge to body
- false edge to after-while
- body links back to guard
- uses `loop_stack` so `break`/`continue` know where to go

Steps:

1. `loop_guard = new_loopguard()`
2. current = loop_guard, add `While` statement there
3. if test contains calls, it visits them (for call graph)
4. create `while_block` (body)
5. guard → body labeled `test`
6. create `afterwhile_block`
7. guard → after labeled `not test`
8. push `(afterwhile_block, loop_guard)` onto loop_stack
9. visit body statements in while_block
10. pop loop_stack
11. body_end → guard (back edge)
12. continue building in afterwhile_block

`visit_For`

Very similar pattern but the "condition" is the iterator:

- guard block holds the `For` node
- one edge to body labeled `node.iter`

- one edge to afterfor (unlabeled)
- body links back to guard
- loop_stack used for break/continue

Note: it does not model StopIteration explicitly; it's a simplified CFG representation.

`visit_Break` and `visit_Continue`

These use `loop_stack[0]` (top loop context):

- `break` jumps to `after_block`
- `continue` jumps to `loop_guard`

After adding the jump, it sets `current_block = new_block()` so code after break/continue goes into a fresh block (since control doesn't flow linearly anymore).

`visit_Return`

Return ends control flow (in that function CFG).

How it's handled:

1. If current block already has statements, it creates a fresh `return_block` and jumps to it (so return is its own terminal statement block).
2. Add the Return statement.
3. If inside a try with a finally (and we're not currently building the finally), it **inserts the finally execution** before treating it as terminal.
    - It makes `after_return` block
    - links return_block → after_return
    - then after_return → return_block (this back-edge is weird-looking, but it's used as a trick to model "finally executes on leaving" and preserve graph structure for interactive rendering)
    - then visits the `finalbody`
4. Adds this block to `cfg.finalblocks`.
5. Sets `current_block = new_block()` without linking to it, so anything after return is unreachable and won't appear connected.

`visit_Try` and `visit_Raise`

These are the hairiest parts.

`visit_Try`

It builds:

- a `TryBlock` holding the `Try` statement
- separate blocks for each except handler
- body block contents
- optional else block
- optional finally blocks
- a join block `after_tryblock`

It also pushes a `TryStackObject` so that `raise` and `return` can consult "am I inside a try right now, and where do I go?"

Core behavior:

- It first creates try_block and links current → try_block
- It registers except handlers into `try_block.except_blocks`
- It visits handler bodies, each ending in `after_tryblock`
- Then it visits the try body inside `try_block`
- If there's an `else`, it creates and visits else_block
- It links the end to after_tryblock
- If there's a `finally`, it visits finalbody statements and then links onward

This code also has special handling if a `Raise` appears in finally.

`visit_Raise`

This decides where control jumps when an exception is raised.

Cases:

- If not in a try: treat as terminal-ish (creates a new block and returns)
- If in try:
    - identify the exception type name (`StopIteration`, etc.)
    - walk the try stack:
        - if we're in the try BODY and there is a matching handler: link raise → that except block
        - else if there is a bare except: link raise → bare except block
        - if we're in EXCEPT or ELSE and there's a finally: simulate that the finally executes before propagating
        - if we're not in FINAL and there is a finally: run finally before leaving

At the end it moves to a new block, because flow after a raise is not linear.

This is doing an approximation of Python's exception propagation rules, mainly for visualization/debug stepping.

---

4) `_source_db.py` — indexing CFGs by file + function + line

This creates a database so the debugger UI can quickly find:

- "What CFG corresponds to this function?"
- "What CFG starts at this line?"

`CFG_Database.index(filename, cfg)`:

- stores the module CFG under:
  - `at_lineno[(filename, module_start_line)]`
  - `at_function[(filename, "<module>")]`
- for each function CFG inside:
  - store by `funcgraph.lineno` and by function name

It also sorts blocks by their starting line (`block.at()`), which makes `bsearch()` fast.

So this file exists mainly to support **interactive mode**: "highlight the right block when I'm stopped at line X."

---

5) `_serve.py` — static file server for the generated graphs

This does two things:

1. Creates a temp directory `~/.py2cfg` (or under a chosen directory) by copying a prototype web UI from `_proto_`.
2. Starts `http.server.SimpleHTTPRequestHandler` to serve that directory.

It also disables cache headers, so when you regenerate an SVG with the same name, the browser doesn't keep showing an old copy.

---

6) `_runner.py` — the CLI wrapper (`py2cfg` command)

This is the command-line entry point.

Normal mode:

- `CFGBuilder(short=args.short).build_from_file(...)`
- `cfg.build_visual(...)` outputs svg/png/pdf etc.

Debug mode (`--debug`):

- runs **pudb** (a console debugger)
- forks a child process to run `_serve.serve(port)` (the web server)
- hooks into pudb's debugger callbacks using `_debug_hooks.debug_init(...)`
- while you step in the debugger, it keeps generating + sending updated CFG SVGs to the browser UI

This is why the project feels "interactive": it combines a debugger + live CFG rendering.

---

7) `_debug_hooks.py` — the live "highlight executed path" system

This file is basically:
"while debugging, track executed lines and repaint the CFG showing where you are."

Pieces involved:

A) WebSocket server

Uses `websocket_server.WebsocketServer`.

- When a browser connects (`new_client`):
  - build the KEY graph SVG
  - send message telling client where to find it

Then later it sends messages like:

```
{ "type": "cfg", "filepath": "cfg/foo.svg", "layer": 0 }
```

so the browser can reload/display updated CFGs.

B) Stack level tracking

They invent a "stack level marker" using a local variable `__py2cfg_stacklevel` injected into frames.

- `_set_entry_frame` remembers the entry frame (the main program being debugged)
- `_get_stack_level` walks `f_back` frames to compute nesting depth in a stable way

This is used so they can know "did we step into a function?" and which CFG to show.

C) CFG database (`cfg_db`)

On first stepping, it builds the CFG of the entire file, indexes it (`cfg_db.index(entry, cfg)`), so later it can quickly fetch the right CFG by function name.

D) ProcessFactory (the clever/chaotic part)

This is the most non-obvious part.

They want to know: **which lines got executed since the last stop** (to highlight blocks).

They do it by:

- forking a "factory" process that can repeatedly fork a child that continues execution under `sys.settrace`
- trace function pushes `(filename, funcname, lineno, event, stacklevel)` into a multiprocessing queue
- a thread (`QueueMGR`) reads that queue and appends executed line numbers into a cache

Then `draw_graph()`:

- calls `cfg.highlight_blocks(executed_lines)`
- calls `cfg.outline_block(current_line)`
- builds a new SVG (interactive styling)
- dispatches it to the browser via websocket

So the workflow is:

1. debugger stops on a line
2. highlight current block
3. fork/trace to learn the line execution path as you step
4. keep repainting

E) Hooking into pudb

`debug_init()` replaces pudb's handlers:

- `Debugger.user_call`
- `Debugger.user_line`
- `Debugger.user_return`
- `Debugger.user_exception`

with wrappers that:

- manage stack of ProcessFactory instances
- generate graphs for callers and callees

- dispatch updated graphs

So pudb runs as normal, but these wrappers continuously produce CFG visuals.

---

8) `__init__.py` — client helper to request CFG from a server

This defines `request_cfg` using `functools.singledispatch`.

Two modes:

Mode 1: `request_cfg(relpath: str, lineno: int, hostname, port) -> CFG`

- sends a JSON-RPC-like request to `http://host:port/build`
- server returns pickled CFG bytes base64-encoded
- decodes + unpickles into a CFG object

Mode 2: `@request_cfg(cfgptr)`

If you pass a list as `cfgptr`, it returns a decorator that:

- on function definition, requests the CFG for that function from the server
- stores it into `cfgptr[0]`

So you can do something like:

```
cfg_holder = [None]

@request_cfg(cfg_holder)
def myfunc(): ...
```

and later inspect `cfg_holder[0]`.

---

9) Putting it together: "How the CFG is built" in one storyline

When you run `CFGBuilder.build_from_src`:

1. `ast.parse` turns text into AST nodes.
2. Builder starts with an empty entry block.
3. It visits statements in order:
    - normal statements get appended to current block

    - control statements (`if/for/while/try/return/break/continue/raise`) **split blocks** and **add edges**
4. Loops:
    - create a guard block
    - guard branches to body and after-loop
    - body links back to guard
5. If:
    - condition branches to if-body and else/after
    - both branches converge at after-if
6. Try:
    - creates handler blocks, body block(s), optional else, optional finally
    - uses a stack so nested tries and raises can find correct handler/finally behavior
7. Cleanup pass removes empty blocks by "splicing" edges and AND-ing conditions.
8. CFG object now contains:
    - nodes (blocks) with code
    - edges with conditions
    - sub-CFGs for functions and classes
9. Rendering phase traverses blocks, styles nodes/edges, and writes Graphviz output.