The big idea (what this library thinks a CFG is)

A **Control Flow Graph (CFG)** is a picture of "what can run after what" in a program.

This library models a program using:

1) BasicBlock

A **BasicBlock** is a chunk of code that runs in a straight line.

- If execution enters a basic block, it executes its instructions in order.
- It can then "exit" (jump) to one or more other blocks.

2) ControlFlowNode

Inside each BasicBlock, each "instruction" becomes a **ControlFlowNode**.

- Think: one statement/expression of interest → one node.
- A BasicBlock contains a list of these nodes in order.

3) Exits / edges

A BasicBlock has exits to other blocks.

This library splits exits into two types:

- **exits_from_end**: normal flow at the end of the block
- **exits_from_middle**: "interrupting" flow that can happen in the middle, mainly **exceptions**

That's a key design choice:
Any instruction may raise, so they add exception edges from basically everywhere.

4) Frames (the secret sauce)

A **Frame** is like a "scope rule" for control flow.

When you're inside:

- a loop → `continue` and `break` need to know where to jump
- a function → `return` needs to know where to jump; uncaught exceptions go somewhere too
- a try/except → exceptions can be caught
- a try/finally → exits must go *through finally first*

So they keep a **stack of frames** while walking the AST.

File 1: `control_flow_graphviz.py` (turn CFG into a picture)

This file does **not build** the CFG. It only **renders** it using Graphviz via `pygraphviz`.

Key constants

- `LEFT_ALIGN = '\l'`
  - Graphviz labels support `\l` meaning "left-justified line break".
  - They use it so multi-line code displays nicely in a node.

`render(graph, include_src=None, path='/tmp/graph.png')`

- Calls `to_graphviz(...)` to build a Graphviz object
- Then calls `g.draw(path, prog='dot')` to output an image

`trim(line, max_length=30)`

- If a line of source is too long, shorten it and add `...`

`unparse(node)`

- Turns an AST node back into source code using `astunparse.unparse`
- Trims every line
- Cleans whitespace
- Replaces `\n` with `\l` so Graphviz displays it as multiple lines

`get_label_for_instruction(instruction)`

This is important: the label shown in the block depends on whether the instruction has a `.source`.

- If `instruction.source is not None`:
  - it constructs something like:
    `write_names <- original_source_string`
  - Example: assignment-style display
- Else:
  - just unparse the AST node (pretty-print the instruction)

So some nodes show "data flow flavored" text like:

- `x <- y + 2`

`get_label(block)`

Walks through all `block.control_flow_nodes`:

- for each node → get its instruction → produce a line label
- join all lines with `\l`
- always ends with a trailing `\l` (Graphviz formatting convenience)

`to_graphviz(graph, include_src=None)`

Core rendering loop:

For each `block in graph.blocks`:

1. Compute label using `get_label(block)`
2. If the block has a special label like `<start>`, `<exit>`, `<raise>`, `<return>`, `<entry:...>`:
    - make node **bold**
    - prepend that label to the node's label text
3. Node id is `id(block)` (Python object identity)
4. Add edges:
    - for each `next_node in block.next`:
        - if it's in `exits_from_middle`: add dashed edge
        - if it's in `exits_from_end`: add solid edge

So:

- **dashed** edge = "can jump due to exception (or other mid-block interruption)"
- **solid** edge = "normal control flow at end of block"

If `include_src` is provided, they add a big node containing the raw source.

File 2: `control_flow_graphviz_test.py` (tests for graphviz rendering)

Two tests:

`test_to_graphviz_for_all_test_components`

- Uses `inspect.getmembers(tc, predicate=inspect.isfunction)` to iterate all test functions in `control_flow_test_components.py`
- For each function:
    1. Build CFG: `control_flow.get_control_flow_graph(fn)`
    2. Render graphviz: `control_flow_graphviz.to_graphviz(graph)`
- This is basically: "make sure we don't crash for all examples."

`test_get_label_multi_op_expression`

- Builds CFG for `multi_op_expression(): return 1 + 2 * 3`
- Finds the block containing source `'1 + 2 * 3'`
- Ensures the label string matches:
    - `return (1 + (2 * 3))\l`
      This confirms the unparser adds parentheses to reflect AST structure.

File 3: `control_flow_test_components.py` (example programs)

This file is just a zoo of small functions that stress different CFG cases:

- straight-line
- if
- for/while
- nested loops
- try/except/finally
- break/continue in tricky places
- lambda
- generator
- inner function
- class methods

These are used to validate the CFG builder.

File 4: `control_flow_test.py` (tests for CFG correctness)

This file checks the CFG builder behaves how they expect.

It defines helper assertions that use:

- `graph.get_block_by_source('...')`
- `block.exits_to(other_block)`
- `block.raises_to(other_block)` (meaning exception-style edge exists)

Notable: they test both:

- **normal edges** (exits_to)
- **exception edges** (raises_to)

So this library treats exception flow as first-class CFG edges.

File 5: `control_flow_visualizer.py` (generate images for the examples)

This is a little script to generate PNGs under `/tmp/control_flow_graphs/`.

Two modes:

`render_functions(functions)`

For each function:

- build CFG
- get source via `program_utils.getsource(function)`
- render CFG image including the source node

`render_filepaths(filepaths)`

For raw source files:

- read file
- build CFG from source string
- render image

Now the main course: `control_flow.py` (the actual CFG builder)

This is where the CFG is built.

A) Entry point: `get_control_flow_graph(program)`

`program` can be:

- a function object
- a source string
- an AST node

Steps:

1. Convert "program" to AST using `program_utils.program_to_ast(program)`
2. Create a `ControlFlowVisitor`
3. Run it on the AST

4. Return its graph

B) `ControlFlowGraph` (the graph container)

`__init__`

- `self.blocks = []`
- `self.nodes = []`
- Creates a special `start_block`:
  - `prunable=False` (don't delete it later)
  - label `<start>`

`new_block(node=None, label=None, prunable=True)`

Creates a new `BasicBlock`, sets its `.graph`, appends it.

Lookup helpers

There are lots:

- find blocks by source snippet
- find blocks by AST node identity
- find entry blocks for functions
- etc.

These exist mostly for testing and analysis.

`prune()` and `compact()`

This library builds a CFG that may contain many empty "connector" blocks, then cleans it:

- **prune**: remove empty blocks that are allowed to be removed
- **merge**: merge block A into block B if:
  - A has exactly one normal successor
  - successor is prunable
  - successor has no other predecessors
  - exception-exits compatibility matches
- **compact**: after structure stabilizes, compute indexes so ControlFlowNodes can find "next within block" quickly

This "build messy → prune → merge → finalize" pipeline is very common in CFG builders.

C) `BasicBlock` (edges, branches, and pruning)

A `BasicBlock` stores:

Graph structure

- `next`: set of successor blocks
- `prev`: set of predecessor blocks

Contents

- `control_flow_nodes`: list of ControlFlowNode

Edges classified

- `exits_from_middle`: successors reachable mid-block (exceptions)
- `exits_from_end`: successors reachable only at end-block (normal flow)

Branch maps

- `branches`: e.g. `{True: true_block, False: false_block}`
- `except_branches`: used for exception-header matching logic
- `reraise_branches`: used for "exception continues after finally" logic

`add_exit(block, interrupting=False, branch=None, except_branch=None, reraise_branch=None)`

- Always adds to `next` and updates `prev`
- Puts the successor in `exits_from_middle` if interrupting, else `exits_from_end`
- Optionally records the branch metadata

`prune()`

If the block is empty and prunable:

- reconnect all its predecessors directly to all its successors
- preserve whether predecessor edges were "from middle" or "from end"
- preserve branches that used to point to this block
- remove this block's edges

This is how they delete empty glue blocks without breaking meaning.

`merge()`

If block has one normal successor and successor is "safe" to merge:

- absorb successor's edges and nodes
- redirect successor's outgoing edges to current block
- keep identities

This is how straight-line code becomes a single block.

D) `ControlFlowNode` (instruction-level navigation)

Each ControlFlowNode wraps one Instruction and knows which block it's in.

`next`

- If there's another instruction in the same block → that's next
- Else, next is the first instruction of each successor block
- If successor block is empty, pruning guarantees it has no successors

`next_from_end`

Same idea, but only follow `exits_from_end` (ignore exception exits).

`prev`

Similar logic but backward.

`branches` / `get_branches(...)`

Only meaningful if this node is the last instruction in its block.
It translates block-level branch edges into node-level branch edges.

If the next block has no instructions, it returns the block label
like `'<exit>'` or `'<raise>'` instead of a node.

E) Frames (how return/break/continue/raise behave correctly)

A **Frame** stores "where exits go" depending on what you're inside.

Kinds:

- MODULE
- LOOP

- FUNCTION
- TRY_EXCEPT
- TRY_FINALLY

Each frame stores some important blocks, like:

- loop: `continue_block`, `break_block`
- function: `return_block`, `raise_block`
- try/finally: `final_block`, `final_block_end`
- try/except: `handler_block`

The visitor keeps a stack `self.frames`.
Innermost is last.

F) `ControlFlowVisitor` (the actual builder)

This is the "walk the AST and create blocks and edges" engine.

1) `run(node)`

- uses graph.start_block as initial
- visits module
- calls `self.graph.compact()` at end
    - (note: their `ControlFlowGraph.compact()` prunes + merges + indexes)

2) `visit(node, current_block)`

- If node is an "instruction AST node":
    - add it as a new instruction to `current_block`
- Then dispatch to a method `visit_<NodeType>` if it exists

So:

- *some nodes* are instructions (like Assign, Return, Compare, Call, etc.)
- *some nodes* are control-structure nodes (If, For, While, Try, FunctionDef, etc.)

3) `add_instruction(...)` and exception edges

After adding an instruction to a block:

- if the block has no exception exits yet:

- call `raise_through_frames(block, interrupting=True)`

That means: **the moment the block gets its first real instruction, they add exception edges "out of it" according to frames.**

This is the reason their CFGs include detailed exception flow.

---

4) `raise_through_frames(...)` (exception flow logic)

This is one of the most important functions.

It computes: "If an exception happens here, where can it go?"

It uses `get_current_exception_handling_frames()` which returns the chain of frames that matter for exceptions:

- try/finally frames (must run finally)
- then either try/except, or function/module raise block as the final catcher

Then it walks those frames from inner to outer:

If it hits TRY_FINALLY:

- add edge to the finally block (interrupting or end depending)
- then switch "current location" to `final_block_end`
- after finally, exception is now re-raised (so it's not "interrupting" anymore; it's at the end)
- it also marks a `reraise_branch=True` to label that path

If it hits TRY_EXCEPT:

- add edge to handler header block

If it hits FUNCTION or MODULE:

- add edge to that frame's `raise_block`

Net effect:

- exceptions flow *through* finally blocks, then into except or raise blocks

---

G) How each control structure builds blocks/edges

`visit_Module`

Creates:

- `<exit>` block (unprunable)
- `<raise>` block (unprunable)
  Push MODULE frame with those exits.
  Visit module body statements.
  Then add normal edge from end to `<exit>`.
  Pop frame.
  Move `<exit>` and `<raise>` to end (purely for nicer ordering).

`visit_FunctionDef` / **`visit_Lambda`**

Two things happen:

(1) In the *outer* block (where the function is defined):

They add instructions for:

- default args
- decorators
- and then a "write" to the function name (like assigning the function object)

(2) They separately build the function body CFG-like region:

`handle_function_definition(...)` creates:

- `<return>` block
- `<raise>` block
- `<entry:name>` block
- `fn_block` (first real body block)

Edges:

- entry → fn_block
- end of body → return block

It also pushes a FUNCTION frame while visiting the body, so `return` and uncaught exceptions know where to go.

`visit_If`

Inside current_block:

- add instruction for the test expression (node.test)

Create:

- `after_block`
- `true_block`
- optional `false_block`

Edges:

- current → true_block (branch=True)
- current → false_block (branch=False) OR current → after_block (branch=False if no else)
- end(true) → after_block
- end(false) → after_block

Then returns `after_block` as the "place control continues".

`visit_While` **and** **`visit_For`**

Both call `handle_Loop(...)`.

Loop layout:

- current_block → test_block
- test_block contains the test or iterator-target instruction
- test_block branches:
  - True → body_block
  - False → else_block or after_block

They push a LOOP frame where:

- continue goes to `test_block`
- break goes to `after_block`

Then:

- body_block ends by exiting back to test_block

This is classic CFG structure.

`visit_Try`

This one is the hardest, but here's the plain-English behavior:

They create:

- `after_block` (where try statement finishes)
- handler header blocks + handler body blocks
- optionally a `final_block` for finally, and compute its end block

If there's a finally:

They push a TRY_FINALLY frame with:

- `final_block` and `final_block_end`
  and connect:
- final_block_end → after_block, marked with `reraise_branch=False`
  - meaning "normal completion path after finally"

If there are except handlers:

They push TRY_EXCEPT frame pointing to the first handler header block.

Then they create `try_block` and connect:

- current_block → try_block
- visit try body into try_block
- on normal completion:
  - try_end → else_block (if present) else → final_block/after_block

Then they pop TRY_EXCEPT frame (because handlers are about to be built explicitly).

Handler building

They chain handler headers like:

- handler0_header (match?) → handler0_body
- if doesn't match → handler1_header
- ...
- last header if doesn't match and there's no bare except:
  - "reraise through frames" from end-of-last-header

Each handler body exits to final_block/after_block.

Finally:

- else block (if present) exits to final_block/after
- if there was finally, pop TRY_FINALLY frame

`visit_Return`, `visit_Break`, `visit_Continue`

All use `handle_ExitStatement(...)`.

What that helper does:

- If you're inside one or more try/finally frames:
  - route control through each finally block first
- then finally exit to the target (return_block / break_block / continue_block)
- then create an unreachable `after_block` (no edge to it)

That unreachable `after_block` is how they represent:
"code after return/break/continue doesn't run".

`visit_Raise`

- Calls `raise_through_frames(current_block, interrupting=False)`
  - meaning the exception happens at the end of this statement, not "mid-block"
- Creates an unreachable after_block

What to take away (the "mental model")

If you remember only 4 things, remember these:

1. **Blocks contain instructions.** Straight-line code merges into one block.
2. **Edges represent possible next blocks.** Solid = normal end-of-block, dashed = exception-style mid-block.
3. **Frames are how they get correctness** for:
   - return/break/continue (must pass through finally)
   - exceptions (must pass through finally, may go to except handlers, else to raise block)
4. **They build first, then prune+merge**, so the final graph looks clean.

Quick contrast vs py2cfg (since you're comparing libraries)

- **py2cfg** attaches edges mostly from explicit syntax constructs and uses its own Block/Link types with exit conditions.
- **python-graphs** is more "analysis-style":
  - has *instruction-level nodes*
  - models exceptions aggressively ("any instruction may raise")

- uses frames to route exception/return/break/continue correctly through try/finally

That's why python-graphs CFGs tend to look more "semantics aware" around exceptions.