The big idea (what staticfg thinks a CFG is)

staticfg's CFG model is intentionally simple:

Block

A **Block** is a basic block: an ordered list of AST statements that run straight-line.

- `block.statements`: the code in the block
- `block.exits`: outgoing edges (`Link`s)
- `block.predecessors`: incoming edges (`Link`s)

Link

A **Link** is a directed edge:

- `source -> target`
- plus an optional `exitcase` AST expression representing the condition for that jump

So edges look like:

- `if test` edge labeled `test`
- `else` edge labeled `not test`
- loop edges labeled with loop conditions / iterators, etc.

CFG

A **CFG** has:

- one `entryblock`
- many `finalblocks` (blocks that end execution)
- nested `functioncfgs` (sub-CFGs for inner function definitions)

File 1: `builder.py` — the CFG construction engine

This is the core. It walks the Python AST and creates Blocks + Links.

1) Version compatibility helpers

```
NAMECONSTANT_TYPE = ast.Constant if py>=3.8 else ast.NameConstant
```

This matters because `True/False/None` became `ast.Constant` in newer Python.

2) `invert(node)` — create "not condition"

Used to build else-edges and false-branches.

Cases:

- If it's `Compare`: invert operator (`==` → `!=`, `<` → `>=`, etc.)
- If it's a boolean constant: flip it
- Otherwise: wrap in `not (...)`

So `invert(x > y)` becomes something like `x <= y`.

---

3) `merge_exitcases(exit1, exit2)`

Used during CFG cleanup when removing empty blocks.

If you remove an empty middle block, you may have:

- pred → empty with condition A
- empty → succ with condition B

New merged edge should represent: **A AND B**.

So merge rules:

- both exist → `(A and B)`
- only one exists → keep it
- none → none

---

4) `CFGBuilder` state (the important stacks)

```
self.after_loop_block_stack = []
self.curr_loop_guard_stack = []
self.current_block = None
self.separate_node_blocks = separate
```

These stacks are how `break`/`continue` work:

- `after_loop_block_stack[-1]` = where `break` jumps

- `curr_loop_guard_stack[-1]` = where `continue` jumps (loop guard)

So this library handles loops using explicit stacks, not "frames" like python-graphs.

---

5) `build(name, tree, asynchr=False, entry_id=0)`

High-level build steps:

1. Create a `CFG`
2. Set `current_id = entry_id`
3. Create the entry block via `new_block()`
4. `visit(tree)` — AST walk builds blocks/edges
5. `clean_cfg(entryblock)` — remove empty blocks
6. return CFG

Important behavior

- There's **no dedicated `<exit>` or `<raise>` block** like python-graphs.
- CFG "ends" at blocks with no exits; those are collected in `cfg.finalblocks`.

---

6) Block/edge creation primitives

`new_block()`

Increments `current_id` and returns `Block(current_id)`.

`add_statement(block, statement)`

Just appends the AST node into `block.statements`.

`add_exit(block, nextblock, exitcase=None)`

Creates a `Link` and updates:

- `block.exits.append(link)`
- `nextblock.predecessors.append(link)`

So edges are stored in both directions.

---

7) `new_loopguard()`

This is a neat “optimization”:

If the current block is empty and has no exits, reuse it as the loop guard.
Otherwise create a new guard block and jump to it.

This prevents creating useless empty blocks around loops.

---

8) Nested function CFGs: `new_functionCFG(node, asynchr=False)`

When you see a function definition inside the current CFG, staticfg:

- creates a sub-CFG for the function body
- stores it in `self.cfg.functioncfgs[node.name]`

But it **does not inline** that function’s flow into the parent CFG.
It just records it separately.

---

9) `clean_cfg(block, visited=[])` — pruning empty blocks

This is similar in spirit to python-graphs’ pruning, but simpler.

If a block is empty:

- For each predecessor link `pred` (pred.source → block, case P)
- For each exit link `exit` (block → exit.target, case E)
- Add a new edge `pred.source → exit.target` with merged condition `P AND E`

Then remove:

- the old predecessor link from pred.source.exits
- the old exit link from exit.target.predecessors

Then recurse forward.

Important gotcha

The function signature uses `visited=[]` as a default, which is a Python footgun (shared across calls).
In practice it often “works” but can lead to surprising behavior across multiple builds.

How staticfg builds control structures (visitor methods)

staticfg is an `ast.NodeVisitor`. It overrides visitors for key statements.

Straight-line statements

`visit_Assign`, `visit_AugAssign`, `visit_Expr`, etc.

They do:

1. `add_statement(current_block, node)`
2. `goto_new_block(node)` — depending on configuration

`goto_new_block(node)`

If `separate_node_blocks` is enabled:

- make a new block
- connect current → new
- move current_block to new

Then `generic_visit(node)` so calls inside expressions get found.

So staticfg can work in two styles:

- **packed blocks** (default): many statements per block
- **one-statement blocks** (if `separate=True`)

Function calls tracking: `visit_Call`

This does not affect control-flow edges.
It only records `block.func_calls.append(func_name)` for visualization.

It tries to derive a readable call name:

- `foo()` → `"foo"`
- `obj.m()` → `"obj.m"`
- some other node types → fallback to class name

`visit_If`

CFG shape:

- Current block contains the If node (as a statement)
- Create `if_block` for the True branch
- Create `afterif_block` for code after if/else
- If else exists, create `else_block`

Edges:

- current → if_block with `node.test`
- if else exists:
  - current → else_block with `invert(node.test)`
  - else_end → afterif_block (if not already terminated)
- if no else:
  - current → afterif_block with `invert(node.test)`
- if_end → afterif_block (if not already terminated)

Subtle detail

They check `if not self.current_block.exits:` before adding the fallthrough edge.
That's how they avoid adding a "normal fallthrough" when the branch ended with `break`, etc.

`visit_While`

CFG shape:

1. Determine / create `loop_guard`
2. loop_guard has the `While` node as a statement
3. Push loop stacks:
   - guard stack (for continue)
   - after-loop stack (for break)
4. Create:
   - `while_block` for body
   - `afterwhile_block` for loop exit

Edges:

- guard → body with `node.test`
- guard → after with `invert(node.test)`
  except when `invert(test)` is literally `False` (i.e., `while True:`), then skip that edge
- body_end → guard (if no break happened)

Then pop stacks.

So the loop back-edge is explicit, and continue uses the guard stack.

`visit_For`

Their for-loop encoding is slightly less semantic than python-graphs.

1. Create / reuse loop_guard
2. Put `For` node as a statement in guard
3. Create `for_block` (body), `afterfor_block` (after loop)
4. Edges:

- guard → for_block with `node.iter` (iterator expression used as "condition label")
- guard → afterfor_block with no condition (unconditional)
- body_end → guard (if not broken)

So unlike python-graphs, it doesn't encode the "iteration finished" condition precisely; it's more "there is an iterator edge" + "there is an exit edge".

`visit_Break` / `visit_Continue`

Very direct:

- break: add edge to `after_loop_block_stack[-1]`
- continue: add edge to `curr_loop_guard_stack[-1]`

No special unreachable block is created here; instead, the branch's current block now has exits, so later fallthrough edges won't be added.

`visit_Return`

- add return statement to current block
- mark current block as a final block: `cfg.finalblocks.append(current_block)`
- then set `current_block = new_block()` **without connecting to it**

That last step is how they ensure “code after return” is unreachable and not represented in CFG.

`visit_Yield`

- sets `cfg.asynchr = True`
- creates afteryield_block and connects current → afteryield_block
  So yield is treated like it “continues” flow.

`visit_FunctionDef` / **`visit_AsyncFunctionDef`**

- add the function definition statement to current block
- build a sub-CFG for the function body and store it under `cfg.functioncfgs`

Again: this is separate from main control flow.

Not implemented: `visit_Raise`

It’s literally `# TODO pass`
So staticfg **does not model exceptions** in the CFG right now.

That’s a major difference vs python-graphs, which models exception flow aggressively.

File 2: `model.py` — data structures + Graphviz rendering

`Block`

Holds:

- `id`
- `statements`
- `func_calls`
- `predecessors`
- `exits`

`get_source()`

Turns block statements into source text for graph nodes using `astor.to_source`.

Special casing:

- For `If/For/While`: only show the first header line (`if x:`) not the whole body
- For function defs: show `def f(...):...`

So nodes stay compact.

`get_calls()`

Pretty prints recorded calls.

`Link`

Stores:

- `source`
- `target`
- `exitcase` (AST expr)

`get_exitcase()`

Uses `astor.to_source(exitcase)` to render edge labels.

`CFG`

Stores:

- `name`
- `asynchr`
- `entryblock`
- `finalblocks`
- `functioncfgs`

Visualization

`build_visual(...)` uses `graphviz` Python package:

- Each block is a node labeled with `block.get_source()`
- Each exit is an edge labeled with exitcase
- Each "calls node" is a dashed edge to a special box node if `block.func_calls` exists
- Nested function CFGs are included as Graphviz subgraphs (`cluster...`)

File 3: `__init__.py`

Just exports:

- `CFGBuilder`, `Block`, `Link`, `CFG`

What to take away (how staticfg differs from python-graphs)

staticfg is "classic CFG builder":

- blocks + conditional edges
- stacks for loops
- cleanup removes empty connector blocks
- optional "one statement per block" mode
- nested function CFGs stored separately
- **no exception edges** (Raise not implemented)

python-graphs is "analysis-grade CFG":

- instruction-level nodes inside blocks
- explicit modeling of:
  - raise paths
  - try/except/finally semantics
  - exception edges from "any instruction may raise"
- special blocks: `<start>`, `<exit>`, `<raise>`, `<return>`, `<entry:...>`
- pruning + merging is more semantics-aware