Introducing PythonTA: A Suite of Code Analysis and Visualization Tools

SIGCSE TS 2025

David Liu

February 26, 2025

Welcome!

Introductions (about me)

Associate Professor, Teaching Stream in Computer Science at the University of Toronto

I each courses in CS1/CS2, theory, programming languages, and others

I like to open-source build tools and supervise undergrads contributing to them

This is my first SIGCSE tutorial!

Introductions (about you)

Who are you?

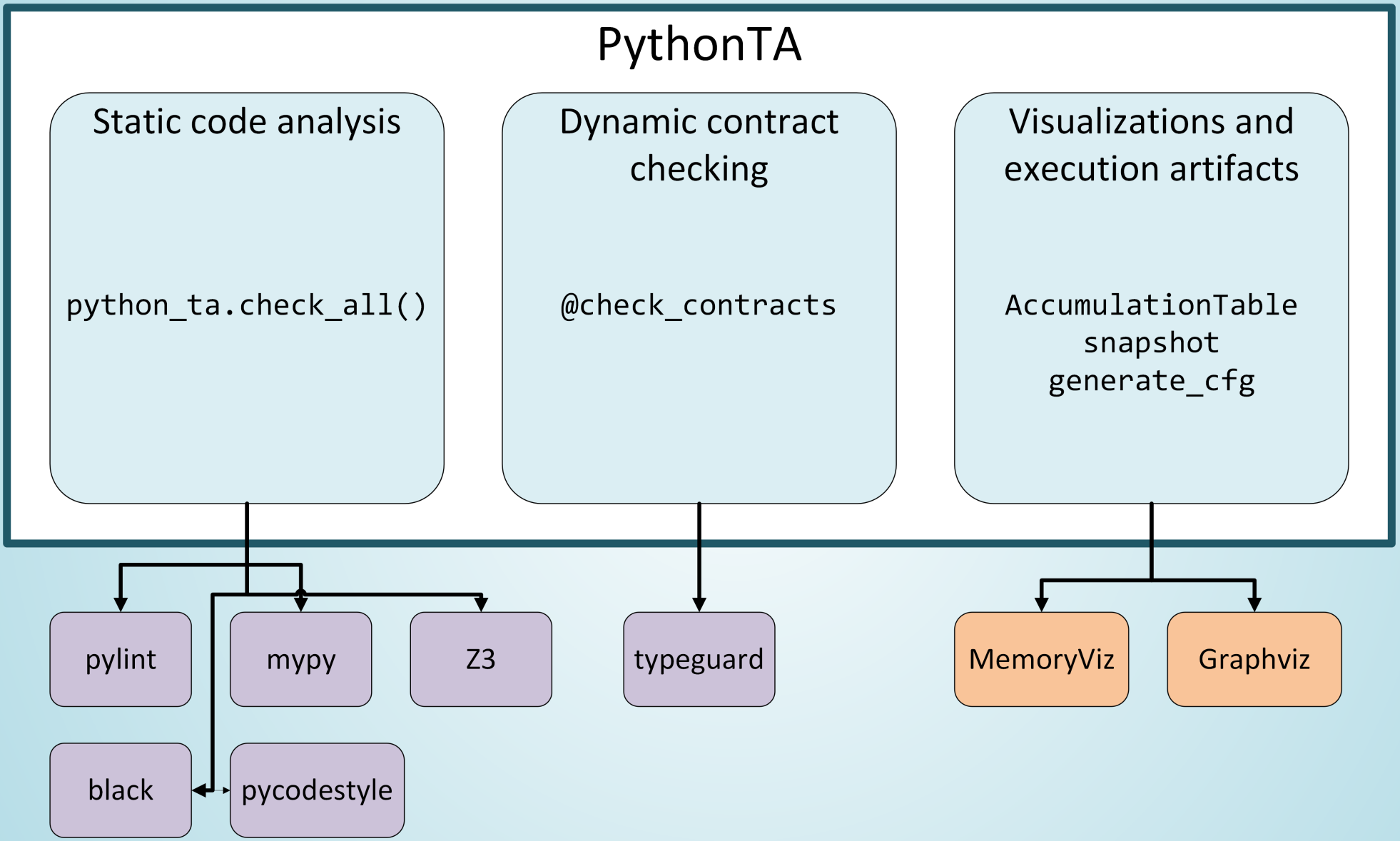
What interested you about this tutorial?

Introductions (about PythonTA)

PythonTA is a open-source suite of code analysis tools designed to help Python learners.

I started the project in 2016.

PythonTA is now used in almost every CS1/CS2 course at the University of Toronto—over 5000 students per year!


PythonTA
Static code analysis
python_ta.check_all()
Dynamic contract checking
@check_contracts
Visualizations and execution artifacts
AccumulationTable
snapshot
generate_cfg
pylint
mypy
Z3
typeguard
MemoryViz
Graphviz
black
pycodestyle

PythonTA's core design principles

1. Beginner-friendly user experience
    - simple Python API
    - accessible messages
2. Customizable to suit instructor needs and course contexts
    - pick and choose from a suite of tools
    - use configuration options to modify behaviour
    - provide machine-readable output formats
3. Integrate easily into existing student/instructor workflows
    - all components can be run locally
    - designed as a library to be called from other frameworks (e.g., autograding scripts)

⚠ Disclaimer: what PythonTA is not

1. PythonTA is not an all-in-one editor or IDE
    - Designed to be called from Python code
2. PythonTA is not an autograding or testing framework
    - But, can be integrated into automated tests
3. PythonTA is not a machine learning or AI tool
    - All functionality uses deterministic algorithms
    - Messages are written by humans

Tutorial objectives

By the end of this tutorial, you will be able to:

- Install and run PythonTA on your computers
- Describe and use the various features of PythonTA
- Design ways to incorporate PythonTA into your courses
- Recognize and mitigate some challenges and limitations of using PythonTA in the classroom
- Identify where to learn more about PythonTA

Agenda

- **Introductions and setup**
- **Part 1: Static code analysis**
  - Technical capabilities
  - Pedagogical ideas and considerations
- **Part 2: Dynamic contract checking**
  - Technical capabilities
  - Pedagogical ideas and considerations
- **Part 3: Generating visualizations and execution artifacts**
  - Technical capabilities
  - Pedagogical ideas and considerations
- **Wrap-up**

Tutorial materials

All materials for this tutorial can be found at https://github.com/pyta-uoft/pyta-sigcse-2025.

You can quickly download a ZIP file containing all materials at https://github.com/pyta-uoft/pyta-sigcse-2025/archive/refs/heads/main.zip.

To install PythonTA, you'll need Python 3.10+.

```
# Windows
$ py -m pip install 'python-ta[z3, cfg]'
# macOS
$ python3 -m pip install 'python-ta[z3, cfg]'
```

For Part 3 of this tutorial, you'll also need to install:

- Graphviz: https://graphviz.org/download/
- Node.js: https://nodejs.org/en/download

Please take a few minutes to download the tutorial materials and install this software on your laptops.

If you're all set up, try running Demos 1-4 under `part1-static-analysis/`.

Part 1: Static code analysis

Static code analysis is the analysis of program code without executing the code.

Some examples common software that perform static analysis:

- compilers
- linters
- integrated development environments (IDEs)

Why static analysis in the Python classroom?

Static analysis can help identify logical errors in student code:

```
if x > 0:
    result = 10
else:
    resutl = 100
return result
```

```
current_sum = None
for number in numbers:
    current_sum += number
return current_sum
```

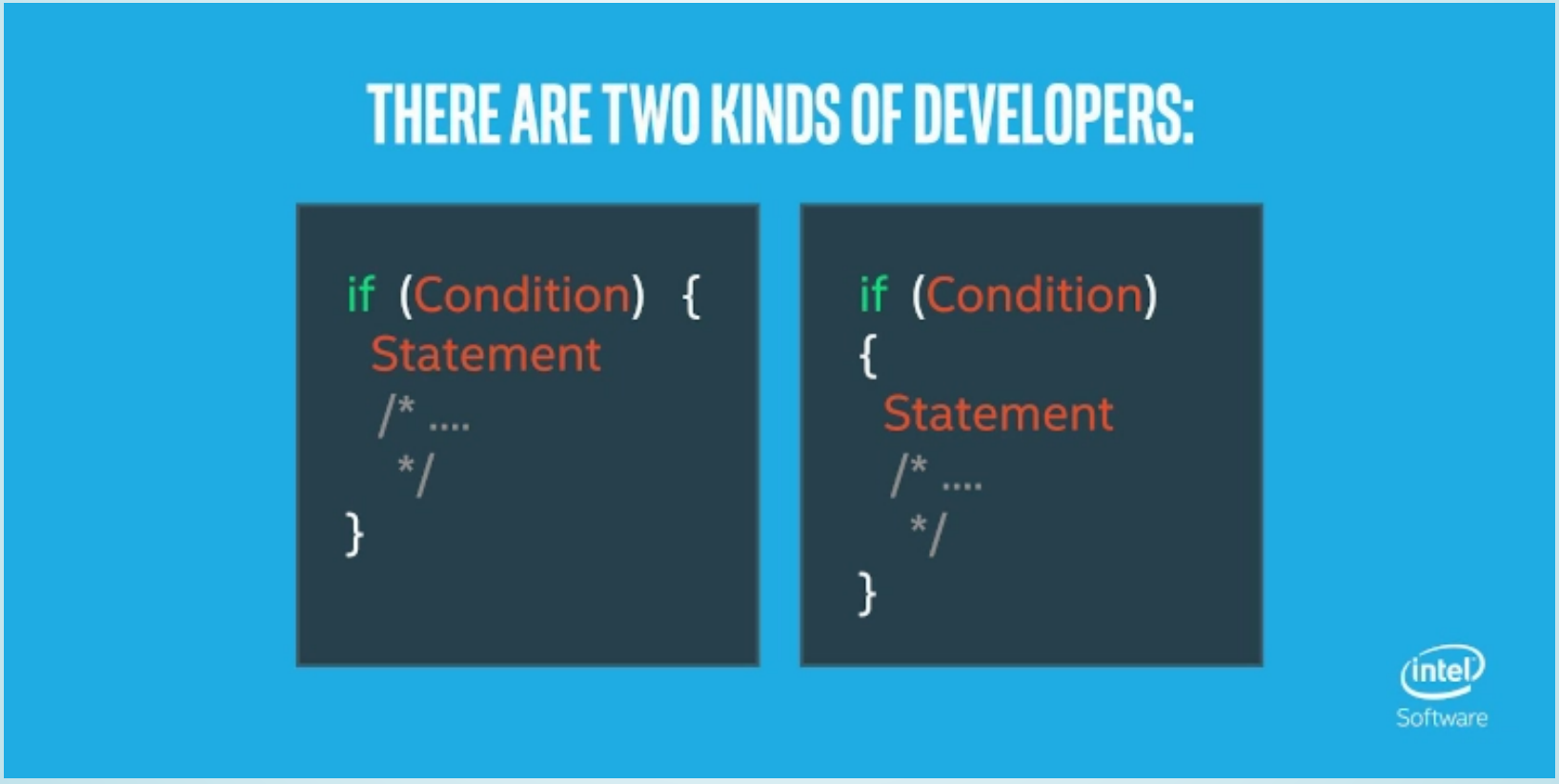
Why static analysis in the Python classroom?

Static analysis can provide feedback on aspects of code quality beyond correctness:

```
def add_one(x: int) -> int:
    """Return x + 1"""
    return x + 1
    return x + 1
```

Why static analysis in the Python classroom?

Automated static analysis tools can provide on-demand, consistent feedback at scale.



Why PythonTA?

Existing Python linters are powerful but are aimed at professionals. IDEs perform static analysis but don't expose accessible APIs.

PythonTA began as an extension of Pylint, with:

- a beginner-friendly default configuration
- custom static checks
- custom error messages
- a simple Python-based interface

```
import python_ta
python_ta.check_all()
```

```
import doctest
doctest.testmod()
```

PythonTA code analysis checks

A **check** in PythonTA (and pylint) is an algorithm that:

- processes Python source code
- looks for code patterns that indicate an issue
- reports issues via an error code, message, and source code context

We'll look at four types of checks made by PythonTA:

- Formatting checks: lexical properties of code
- Style checks: code issues that don't affect correctness
- Logical checks: code issues that affect correctness
- Forbidden usage checks: code disallowed by the instructor

Formatting checks

```
def my_function()->int:


    x=1
    y =     x * 1;
    return y
```

Demo: `demo1_formatting_checks.py`

Style checks

```
def calculate_sum(numbers: list[int]) -> int:
    current_sum = 0
    for i in range(len(numbers)):
        current_sum += numbers[i]
    return current_sum


def greet(age: int) -> str:
    if 10 < age and age < 13:
        return 'Hi!'
    else:
        return "Hello"
```

Demo: `demo2_style_checks.py`

Logical checks

```
def calculate_sum(numbers: list[int]) -> int:
    current_sum = None
    for number in numbers:
        current_sum += number
    return current_sum


def search(numbers: list[int], item: int) -> str:
    for number in numbers:
        if number == item:
            return 'Found'
        else:
            return 'Not found'
```

Demo: `demo3_logical_checks_demo.py`

Forbidden usage checks

```
def calculate_average(a: int, b: int) -> int:
    """Return the average of a and b, rounded down to the nearest integer.
    You may NOT use any functions from the "math" module.
    """

def calculate_sum(numbers1: list[int], numbers2: list[int]) -> int:
    """Return the sum of the numbers in both given lists.
    You may NOT use any loops (instead use an appropriate built-in function).
    """

def add_one(x: int) -> int:
    """Return x + 1.
    Remember to remove all print statements before submitting your work.
    """
```

Demo: `demo4_forbidden_usage_checks.py`

Finding out more

- PythonTA documentation:
  - https://www.cs.toronto.edu/~david/pyta/checkers/
- Pylint documentation:
  - https://pylint.readthedocs.io/en/stable/user_guide/checkers/featu
- Code examples:
  - https://github.com/pyta-uoft/pyta/tree/master/examples/pylint
  - https://github.com/pyta-uoft/pyta/tree/master/examples/custom_checkers

Configuring the static code analysis

Passing configuration options

Specify configuration options using the `config` argument to `python_ta.check_all` in one of two ways:

- passing a dictionary of option name-value pairs
- passing a path to a `.pylintrc` file

Disabling checks

Use the `disable` option to disable checks by error code or name.

```
python_ta.check_all(config={
    'disable': [
        'E9970',  # or, 'missing-param-type'
        'missing-return-type'  # or, 'E9971'
    ]
})
```

Demo: `demo5_disabling_checks.py`

Running specific checks

Use `disable=['all']` and `enable=[...]` together to run only specific checks.

```
python_ta.check_all(config={
    'disable': ['all'],
    'enable': ['unnecessary-indexing']
})
```

Demo: `demo5_disabling_checks.py`

Output format

Use the `output-format` option to change the report format:

- `python_ta.reporters.PlainReporter`
  - plaintext report
- `python_ta.reporters.ColorReporter`
  - like plaintext, but uses ANSI terminal colour codes
- `python_ta.reporters.JSONReporter`
  - reports errors in machine-readable JSON format
- `python_ta.reporters.HTMLReporter`
  - opens report in a web browser (default)

Demo: any of the previous files!

Customizing error messages

```
def greet(age: int) -> str:
    if 10 < age and 13 >= age:  # Slightly different condition
        return 'Hi!'
    else:
        return "Hello"
```

```
R1716 (chained-comparison)  Number of occurrences: 1.
  [Line 18] You can simplify this boolean operation into a chained comparison.
            Instead of 'a < b and b < c', use 'a < b < c'.
   16  def greet(age: int) -> str:
   17      """Return a greeting based on the given age."""
   18      if 10 < age and 13 >= age:
              ----------------------
   19          return 'Hi!'
   20      else:
```

Customizing error messages

You can customize all error messages by creating a TOML file with the following structure:

```
["<module>".<CheckerClass>]

<error code> = "<new message>"
```

Customizing error messages

Goal: replace message for `R1716` (`chained-comparison`).

1. First, we need to find the module and class where it's defined.
   - Pylint documentation
2. Then we need to create a `.toml` file containing the following:

```
# demo-config/custom_messages.toml
["pylint.checkers.refactoring".RefactoringChecker]
R1716 = "<my new message for R1716>"
```

3. Set the `messages-config-path` configuration option:

```
'messages-config-path': 'demo-config/custom_messages.toml'
```

Demo: `demo6_customizing_messages.py`

Customizing error messages: limitations

```
"Index-based for loop (with index `%s`) can be simplified
by looping over the elements of %s directly."
```

Some checks pass context-sensitive information (e.g., variable names) into messages using conversion specifiers (e.g., `%s`).

The main **limitation** of error message customization is that it must use the same string conversion specifiers as the original message.

```
# Original
"Loop/comprehension index `%s` can be simplified by \
accessing the elements directly in the for loop or \
comprehension, for example, `for my_variable in %s`."
```

PythonTA's default message configuration

https://github.com/pyta-uoft/pyta/blob/master/python_ta/config/messages_config.toml

More configuration options

For more information on PythonTA-specific options, see the project documentation.

- `extra-imports: list[str]`
  - allowed module imports
- `allowed-io: list[str]`
  - functions where I/O operations are allowed
- `pyta-number-of-messages: int`
  - maximum number of messages per error type to emit

PythonTA also inherits all pylint configuration options.

Using configuration files

Configuration options can be specified in an INI file.

```
# demo-config/.pylintrc
[CUSTOM PYTA OPTIONS]
output-format = python_ta.reporters.PlainReporter

[FORMAT]
max-line-length = 120

[FORBIDDEN IMPORT]
extra-imports = datetime, statistics
```

```
python_ta.check_all(config='demo-config/.pylintrc')
```

Demo: `demo7_configuration_file.py`

Additional options

Autoformatting

❌ Formatting issues are common and annoying.

☑ They are also the easiest to fix automatically.

Pass `autoformat=True` to run Black before checking code:

```
# Autoformat with Black.
# ⚠ This modifies the file!
python_ta.check_all(autoformat=True, ...)
```

Demo: `demo8_autoformat.py`

Saving reports

By default, PythonTA reports are transient (printed to standard output or rendered in a web browser).

Reports can be saved using the `output` argument:

```
# Write report to file pyta_report.txt
python_ta.check_all(output='pyta_report.txt', ...)
```

Demo: `demo9_output_file.py`

Invoking PythonTA

So far, we've only called `python_ta.check_all` in one way, directly from the file being checked.

Now, let's look at other ways of invoking PythonTA.

Using the Python API

We can pass a first argument to `python_ta.check_all`, which is a path or list of paths to check.

```
python_ta.check_all(
    [
        'demo1_formatting_checks.py',
        'demo2_style_checks.py',
        'demo3_logical_checks.py',
        'demo4_forbidden_usage_checks.py'
    ],
    config={'output-format':
                'python_ta.reporters.PlainReporter'}
)
```

Demo: `demo10_multiple_files.py`

Command-line interface

It is also possible to invoke PythonTA in the command-line:

```
python_ta [OPTIONS] [FILENAMES]...
```

For example:

```
python_ta \
  --output-format=python_ta.reporters.PlainReporter \
  demo1_formatting_checks.py \
  demo4_forbidden_usage_checks.py
```

Or, with a configuration file:

```
python_ta --config config/.pylintrc \
  demo7_configuration_file.py
```

Programmatically analysing errors

`python_ta.check_all` returns a reporter object that records the issues found.

```
reporter = python_ta.check_all('my_file.py', ...)
messages = reporter.messages['my_file.py']  # Message list
messages[0].to_dict()  # Inspect a single message
```

Demo: `demo11_unittest.py`

Incorporating PythonTA static analysis into your courses

As a tool to support students while programming

- Provide starter code that includes a call to `python_ta.check_all`.

```
if __name__ == '__main__':
    import python_ta
    python_ta.check_all(config=...)
```

- If you are using a lot of configuration options, a separate configuration file hides complexity from students
- Two kinds of advice:
  - "Run PythonTA and fix issues regularly while working"
  - "Run PythonTA and fix issues when reviewing/refactoring"
- Incorporate PythonTA feedback into public test suites

As a tool for evaluation/grading

- Incorporate PythonTA feedback into public/private test suites
  - Recommended: ensure the PythonTA configuration used for grad is identical to the one provided to students
- Sample grading scheme:

> *PythonTA checks are worth 15% of the assignment grade. Each issue reported by PythonTA results in a 1.5% deduction, up to a maximum of 15% for 10 issues reported.*

As opportunities for targeted practice

Sample exercise. When running PythonTA on this code:

```
def search(numbers: list[int], item: int) -> str:
    """Return 'Found' if <item> appears in <numbers>, and
    'Not found' otherwise.
    """
    for number in numbers:
        if number == item:
            return 'Found'
        else:
            return 'Not found'
```

we see the message

```
"This loop will only ever run for one iteration"
```

Explain the bug in the code, and then fix it.

Challenges and limitations

1. Not all students will run PythonTA.
    - 💡 Demo PythonTA during lecture/tutorial time
    - 💡 Integrate it into existing ways students run code
2. Students are annoyed by PythonTA.
    - 💡 Discuss the importance of code quality (beyond correctness)
    - 💡 Discuss how automated tools can improve code consistency
    - 💡 Enable autoformatting
3. Configuring PythonTA is time-consuming.
    - 💡 Reuse configurations across assessments
    - 💡 Only enable specific checks you care about

Discussion!

What questions do you have about the technical aspects of the tool?

What questions do you have about using the tool in your courses?

What opportunities and challenges do you see in using this tool?

What improvements would you suggest to make the tool more suitable for your courses?

Part 2: Dynamic contract checking

Function contracts

For our CS1/CS2 courses, we commonly use the concept of contracts to specify function behaviour:

- Preconditions are properties that must be satisfied by the function caller, and can be assumed to be true by the implementation
  - includes parameter type annotations
  - e.g., "`n > 0`", "the input list is non-empty"
- Postconditions are properties that must be satisfied when the function ends
  - includes the return type annotation
  - e.g., "the return value is `>= 0`", "the input list is now sorted"

We say a function implementation is correct according a contract when for all inputs that satisfy the preconditions, the postconditions are also satisfied.

Challenge: incorrect code may inadvertently violate these contracts, introducing bugs that might be revealed much later.

```
def get_max(lst: list[int]) -> int:
    """Precondition: lst != []"""
    m = None
    for number in lst:
        if m is None or number > m:
            m = number
    return m
```

```
def main(lst: list[int]) -> int:
    max_item = get_max(lst)

    # lots of code...

    return max_item + 1   # Uh oh!
```

Checking contracts dynamically

PythonTA supports a mechanism for checking contracts dynamically, i.e., while code is executing.

Why dynamic?

1. Limited ability to statically verify logical conditions in general
2. Impacting code execution has greater visibility to students

python_ta.contracts

The submodule `python_ta.contracts` defines the `check_contracts` decorator:

```
from python_ta.contracts import check_contracts

@check_contracts
def my_function(x: int) -> int:
    ...
```

This decorator causes functions to **automatically verify contracts**, raising `AssertionError`s when they are violated.

check_contracts: function type annotations

```
@check_contracts
def add_to_list(numbers: list[int],
                new_num: int, n: int) -> list[int]:
    if n <= 0:
        return numbers.copy()
    elif n == 1:
        return numbers.append(new_num)
    else:
        return numbers + ([new_num] * n)
```

Demo: `demo1_function_type_annotations.py`

This functionality is built on the typeguard library.

Checking preconditions

`check_contracts` extracts preconditions from function docstrings.

```
@check_contracts
def my_function(...) -> ...:
    """...

    Preconditions:
    - <precondition 1>
    - <precondition 2>
    - ...
    """
```

Preconditions written in Python syntax are verified; ones written in natural language are ignored.

Demo: `demo2_function_contracts.py`

Checking postconditions

`check_contracts` also parses postconditions, with `$return_value` representing the function return value:

```
@check_contracts
def my_function(...) -> ...:
    """...

    Postconditions:
    - <postcondition 1>
    - <postcondition 2>
    - ...
    """
```

Demo: `demo2_function_contracts.py`

Note about verification order

When a function is decorated by `check_contracts`, it performs these steps when called:

1. Before the function is called:
    1. Verify parameter type annotations
    2. Verify preconditions parsed from the function docstring (top-down order)
2. Call the function on the provided arguments
3. After the function returns:
    1. Verify return type annotation
    2. Verify postconditions parsed from the function docstring (top-down order)

This order allows you to write pre- and postconditions that assume the arguments and return value have the correct type!

Classes and contract checking

`check_contracts` can also be used to decorate classes!

When a class is decorated with `check_contracts`, all of its methods are decorated automatically.

Additionally, class invariants are checked.

```
@check_contracts
class MyClass:
    ...
```

Class invariants

A class (representation) invariant is properties that every class instance must satisfy. These invariants are a special form of precondition **and** postcondition of all (public) instance methods.

```
class Person:
    """A class representing a person.

    Representation Invariants:
    - self.name != ""
    - self.age >= 0
    """
    name: str
    age: int
```

Instance attribute type annotations are a form of class invariant.

Invariant checking

When a class is decorated with `check_contracts`:

- invariants are extracted from attribute type annotations and the class docstring
- invariants are verified:
  - after each call to an instance method
  - after an instance attribute is reassigned outside of an instance method

Demo: `demo3_class_contracts.py`

Note about inheritance

Given a class `A` that is decorated with `check_contracts`, a subclass `B` of `A`:

- “Inherits” the verification of all instance attribute type annotations declared in `A`
- “Inherits” the verification of all invariants parsed from the docstring of `A`

Demo: `demo4_inheritance.py`

⚠ Technical limitations

- Does not distinguish between public and “private” methods

```
def my_method(self):
    self._update_attr1()  # Violate invariant
    self._update_attr2()  # Restore invariant
```

- Mutation of an instance attribute does not trigger verification

```
class A:
    x: list[int]

a = A()
a.x.append('David')
```

Tips on writing conditions as Python expressions

1. Translate standard boolean operations:

a and b	`a and b`
a or b	`a or b`
if a then b	`not a or b`
a if and only if b	`a == b`

2. Use comprehensions and `all`/`any` for collections:

every number is positive	`all(n > 0 for n in numbers)`
there is a positive number	`any(n > 0 for n in numbers)`

3. You can always define a separate function!

Incorporating dynamic contract checking into your courses

Integrating into assessments

In your starter code, apply `check_contracts` to selected functions and/or classes.

Alternatively, apply the decorater inside your test cases:

```
from student_file import my_function
from python_ta.contracts import check_contracts

class TestMyFunction(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.wrapped = check_contracts(my_function)

    def test_simple(self):
        self.assertEqual(
            type(self).wrapped(...), ...
        )
```

Demo: `demo5_unittest.py`

Exercises in logical translation

Sample exercise. Given the following function, translate the English preconditions into valid Python syntax.

```
@check_contracts
def replace_n_times(numbers: list[int], new_item: int, old_item: int, n: int) -> None:
    """Replace the first <n> occurrences of <old_item> with <new_item> in <numbers>.

    Preconditions:
    - n is positive
    - new_item and old_item are distinct numbers
    - old_item appears at least n times in numbers
    """
```

Check your work using PythonTA by calling your function on inputs that violate the preconditions.

Demo: `demo6_sample_exercises.py`

Exercises in class representation design

Sample exercise. Given the following class, write two representation invariants in Python syntax that would be appropriate to enforce.

```
@check_contracts
class Person:
    """A class representing a person.

    Representation Invariants:
    - ...
    - ...
    """
    given_name: str         # The person's given name
    family_name: str        # The person's family name
    age: int                # The person's age (in years)
    friends: list[Person]   # A list of the person's friends
```

Check your work using PythonTA by creating instances of this class that violate your representation invariants; you should see an `AssertionError`.

Challenges and limitations

1. Some conditions are hard/impossible to represent as Python expressions
   - 💡 Keep writing conditions in English
     - `check_contracts` can still check type annotations
   - 💡 Define helper functions to verify conditions
2. Contract checking slows down function calls
   - 💡 Set `python_ta.contracts.ENABLE_CONTRACT_CHECKING` `False` to disable contract checking
   - 💡 Use `check_contracts` as a debugging tool only

Discussion!

What questions do you have about the technical aspects of the tool?

What questions do you have about using the tool in your courses?

What opportunities and challenges do you see in using this tool?

What improvements would you suggest to make the tool more suitable for your courses?

Bonus: logical specification in static analysis

The Z3 Theorem Prover

```
# part2-dynamic-contract-checking/z3/z3_intro.py
import z3
x = z3.Int('x')
y = z3.Int('y')
z = z3.Int('z')

z3.solve(
    x * x + y * y == z * z,
    x > 0,
    y > 0,
    z == 0,
)
# Output: no solution
```

Enabling Z3 usage

PythonTA integrates the Z3 Theorem Prover to (optionally) enhance its static analysis.

Install `z3-solver` (or `python-ta[z3]`) and use the `z3` configuration option to enable:

```
python_ta.check_all(config={
    'z3': True
})
```

Condition analysis (1)

PythonTA can detect some impossible (i.e., always `False`) conditions.

```
def greet_v1(age: int) -> str:
    """Return a greeting.

    Preconditions:
    - age >= 0
    """
    if age >= 18:
        return 'Hello, you adult, you'
    elif age >= 100:
        return 'Wow, good for you!'
    else:
        return 'Hi!'
```

```
def greet_v2(age: int) -> str:
    """Return a greeting.

    Preconditions:
    - age >= 0
    """
    if age < 0:
        return 'You are not born yet'
    elif age >= 18:
        return 'Hello, you adult, you'
    else:
        return 'Hi!'
```

Demo: `z3/z3_condition_checking.py`

Condition analysis (2)

PythonTA can detect some redundant (i.e., always `True`) conditions.

```
def calculate(x: int, numbers: list[int]) -> int:
    """Return the sum of <numbers> multiplied by <x>."""
    if x == 0:
        return 0

    current_sum = 0
    for number in numbers:
        if x != 0:
            current_sum += number * x
    return current_sum
```

Demo: `z3/z3_condition_checking.py`

Control flow analysis

```
def greet(age: int) -> str:
    """Return an age-appropriate greeting.

    Preconditions:
    - age >= 0
    """
    if age >= 100:
        return 'Wow, good for you!'
    elif age >= 18:
        return 'Hello, you adult, you'
    elif age <= 17:
        return 'Hi!'
```

Demo: `z3_control_flow_analysis.py`

⚠ Technical limitations

PythonTA's Z3 logical analysis is (currently!) very limited.

- No support for local variables other than function parameters
- No support for parameters being reassigned
- No support for function calls (e.g., `len`) or comprehensions

Part 3: Generating visualizations and execution artifacts

Story so far

Part 1: Static code analysis

- “here’s my code, where can I improve it?”

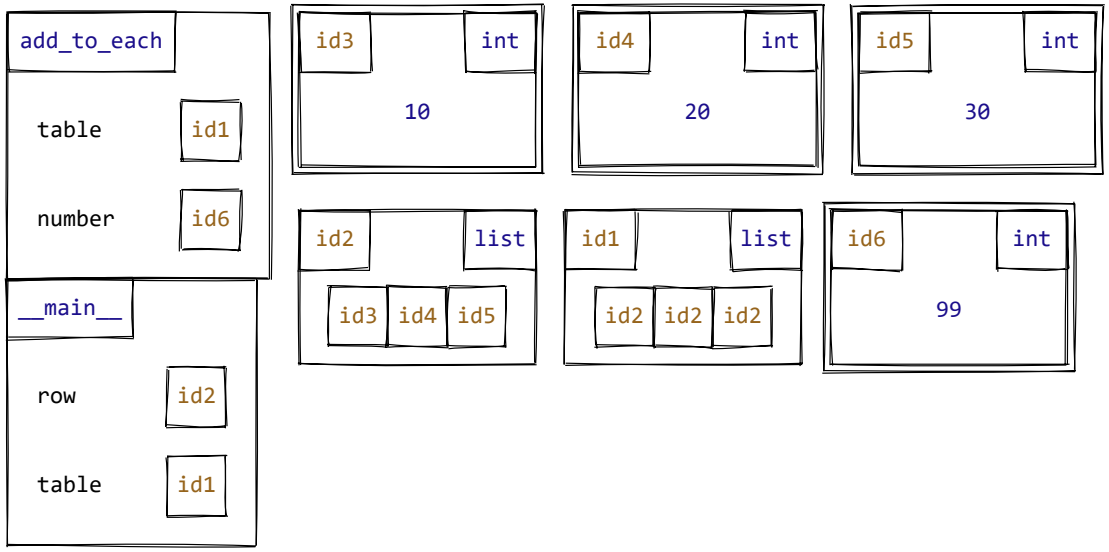
Part 2: Dynamic contract checking

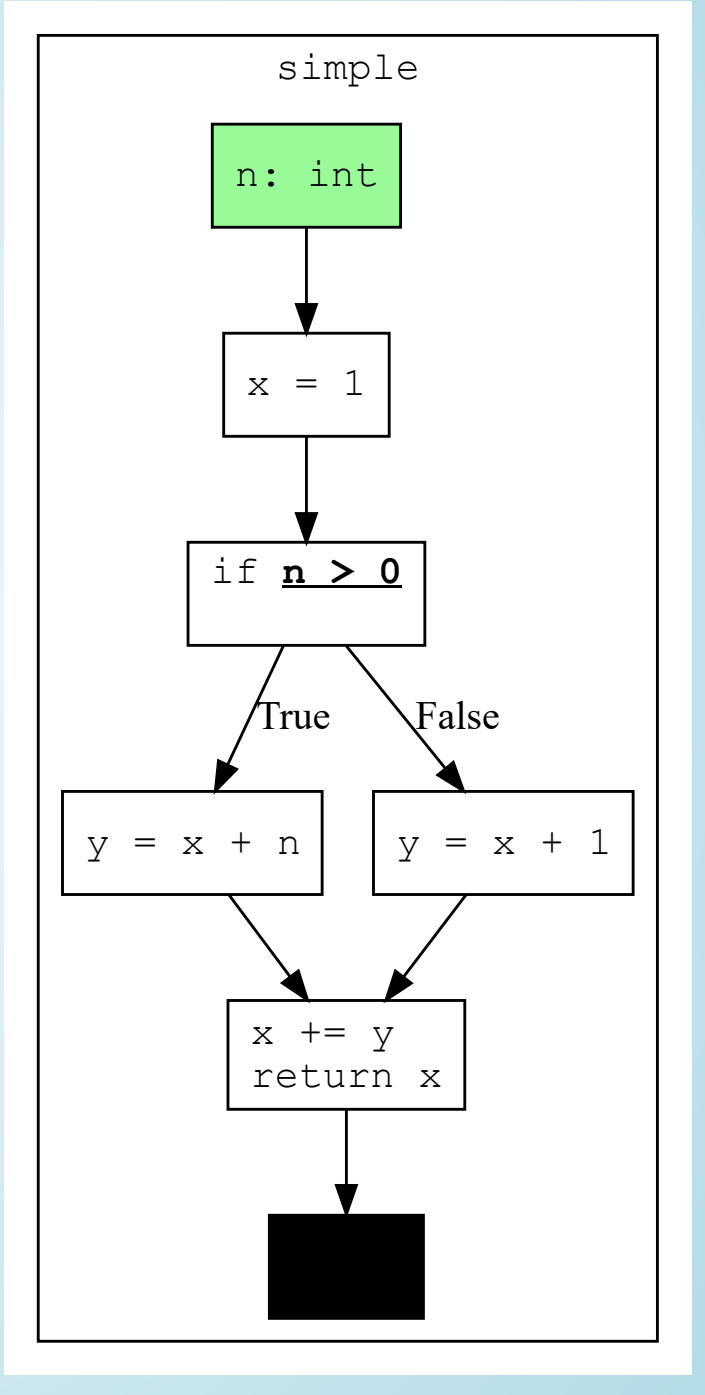
- “help me identify bugs by flagging contract violations”

Part 3: Generating visualizations and execution artifacts

- “help me understand my code in different modalities”

iteration	number	current_sum	numbers
0	N/A	0	[10, 20, 30, 4
1	10	10	[10, 20, 30, 4
2	20	30	[10, 20, 30, 4
3	30	60	[10, 20, 30, 4
4	40	100	[10, 20, 30, 4





Loop tracing tables

```python
def calculate_sum(numbers: list[int]) -> int:
  current_sum = 0
  for number in numbers:
    current_sum += number
    print(f'number={number}, current_sum={current_sum}')
  return current_sum
```

Print-based debugging is quick and easy, but is often imprecise.

Debuggers are excellent tools, but (1) have a higher barrier to entry, and (2) typically show a snapshot at a single point in time.

`python_ta.debug.AccumulationTable`

PythonTA exports a context manager, `AccumulationTable`, that aims to:

- Standardize and simplify this form of print-based debugging in loops
- Output data in a structured, easy-to-parse format
- Support students in both understanding correct algorithms and identifying bugs in incorrect code

python_ta.debug.AccumulationTable

```
from python_ta.debug import AccumulationTable


def calculate_sum(numbers: list[int]) -> int:
    current_sum = 0

    with AccumulationTable(['current_sum']):
        for number in numbers:
            current_sum += number

    return current_sum
```

```
# Output

iteration   number    current_sum
----------  --------  -----------
0           N/A       0
1           10        10
2           20        30
3           30        60
4           40        100
```

Demo: `demo1_loop_tracing.py`

`AccumulationTable` technical details

- Initialization: `AccumulationTable(names: list[str])`
  - `names` contains the expressions to evaluate and list
- Usage: `with AccumulationTable(...):` wraps a loop, and does the following:

  1. Record a single row capturing the state immediately before entering the loop
  2. At the end of each completed loop iteration, record the state
  3. After the block completes, print the loop tracing table

⚠ Technical limitations

- Cannot record state in the middle of a loop iteration, only at the end
- Does not log iterations of nested inner loops
- Does not log iterations of multiple loops in the same context (just the first loop)

`AccumulationTable` sample exercises

The following function has a bug!

```
def sum_positives(numbers: list[int]) -> int:
    """Return the sum of the given numbers
    that are positive."""
    current_sum = 0

    with AccumulationTable(['numbers']):
        for number in numbers:
            if number > 0:
                current_sum = number
    return current_sum
```

Example function output:

```
>>> numbers = [10, -3, -4, 20, 5]
>>> r = sum_positives(numbers)

iteration  number    current_sum
---------  --------  -----------
0          N/A       0
1          10        10
2          -3        10
3          -4        10
4          20        20
5          5         5

>>> r
5
```

At what iteration did the error first occur? What value(s) in the table are incorrect, and what should their correct values be? What is the likely source of this error?

`AccumulationTable` sample exercises

Here is the Euclidean algorithm for calculating the greatest common divisor of two numbers.

```
def euclidean_gcd(a: int, b: int) -> int:
    """Return the gcd of a and b.

    Preconditions:
    - a >= 0
    - b >= 0
    """
    x, y = a, b
    with AccumulationTable(['x', 'y']):
        while y > 0:
            x, y = y, x % y

    return x
```

Suppose we call `euclidean_gcd(20, 14)`. Complete the loop tracing table below.

iteration	x	y
0	20	14
1		
2		
3		

Illustrating loop invariants

`AccumulationTable` can be used to demonstrate loop invariants.

Example: the extended Euclidean algorithm.

```
def extended_euclidean_gcd(a: int, b: int) -> tuple[int, int, int]:
    """Return the greatest common divisor of a and b,
    and integers p and q such that:

        p * a + q * b = gcd(a, b)

    Preconditions:
    - a >= 0
    - b >= 0
    """
```

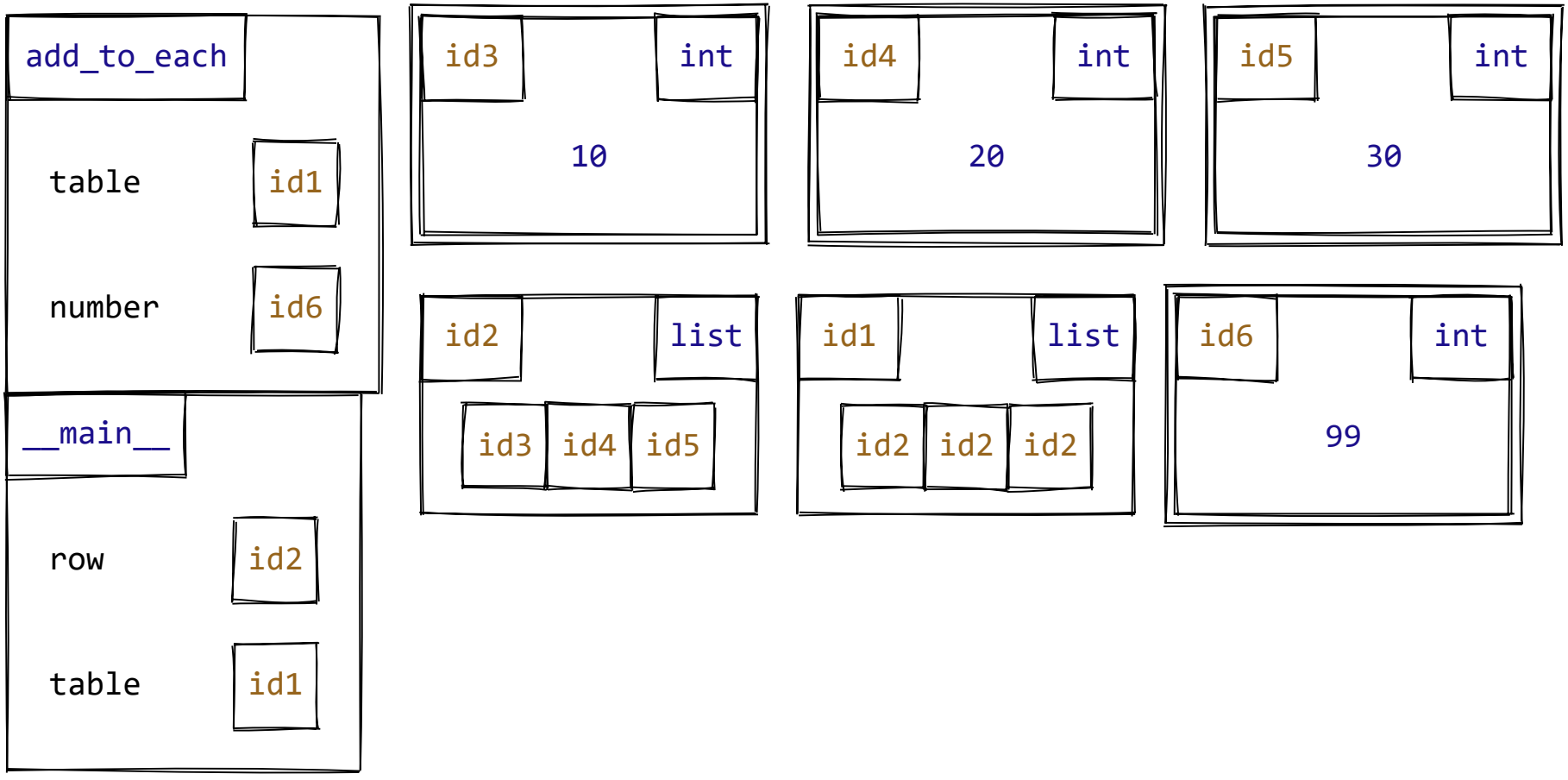
Demo: `demo1_loop_tracing.py`

Memory model visualizations

```
def add_to_each(table: list[list[int]], number: int) -> No
    """Append <number> to each row in the table.
    """
    for row in table:
        row.append(number)
```

```
row = [10, 20, 30]
table = [row, row, row]
add_to_each(table, 99)
print(table)
```

```
# Output:
[[1, 2, 3, 99, 99, 99],
 [1, 2, 3, 99, 99, 99],
 [1, 2, 3, 99, 99, 99]]
```


add_to_each
table
id1
number
id6
__main__
row
id2
table
id1
id3
int
10
id4
int
20
id5
int
30
id2
list
id3
id4
id5
id1
list
id2
id2
id2
id6
int
99

The `snapshot` function

We can generate these diagrams dynamically using `snapshot` and the MemoryViz library.

```
from python_ta.debug.snapshot import snapshot

snapshot(save=True, memory_viz_args=[...])
```

`memory_viz_args` is a list of command-line arguments to MemoryViz. We'll use:

```
memory_viz_args=['--output=snapshot.svg', '--width=1000']
```

Demo: `demo2_snapshot.py`

Customizing the diagrams

If you want more control over the diagram output, you can instead capture a JSON representation of the program state and modify it before passing it to MemoryViz. **Demo**: `demo3_snapshot_json.py`

1. Modify the `snapshot` call to output and save JSON data.

```
open('snapshot.json', 'w').write(
    json.dumps(snapshot_to_json(snapshot()), indent=2))
```

2. Customize the data in `snapshot.json` (more later).

3. Invoke `memory-viz`:

```
$ npx memory-viz@latest --output=snapshot2.svg \
    --width=1000 snapshot.json
```

MemoryViz JSON format

The MemoryViz JSON format is (roughly) the following:

- The top-level element is a list of objects
- Each object has a `type`
- `type: ".frame"` objects represent stack frames. Each has:
  - `name` (string)
  - `value` (object) – a mapping from variables to ids
- Other objects represent Python objects. Each has:
  - `id` (int) – a unique identifier
  - `value` – the value of the object

Customization examples

1. Highlighting objects:

```
{..., "style": ["highlight"]}
```

2. Removing parts:

```
{"type": "list", "id": 1, "value": []}
{"type": "int", "id": null, "value": null}
```

3. Reordering objects

Ideas for incorporating `snapshot` diagrams

1. Generate diagrams to illustrate concepts (e.g., aliasing)
2. Generate diagrams live during class
3. Generating partial diagrams for exercises or tests

Bonus: (experimental) line-by-line visualizations

```
from python_ta.debug import SnapshotTracer

def add_to_each(table: list[list[int]], number: int) -> No
    with SnapshotTracer(output_directory='_out',
                        webstepper=True,
                        memory_viz_args=['--width=1000']):
        for row in table:
            row.append(number)

if __name__ == '__main__':
    row = [10, 20, 30]
    table = [row, row, row]
    add_to_each(table, 99)
    print(table)
```

Demo: `demo4_snapshot_tracer.py`

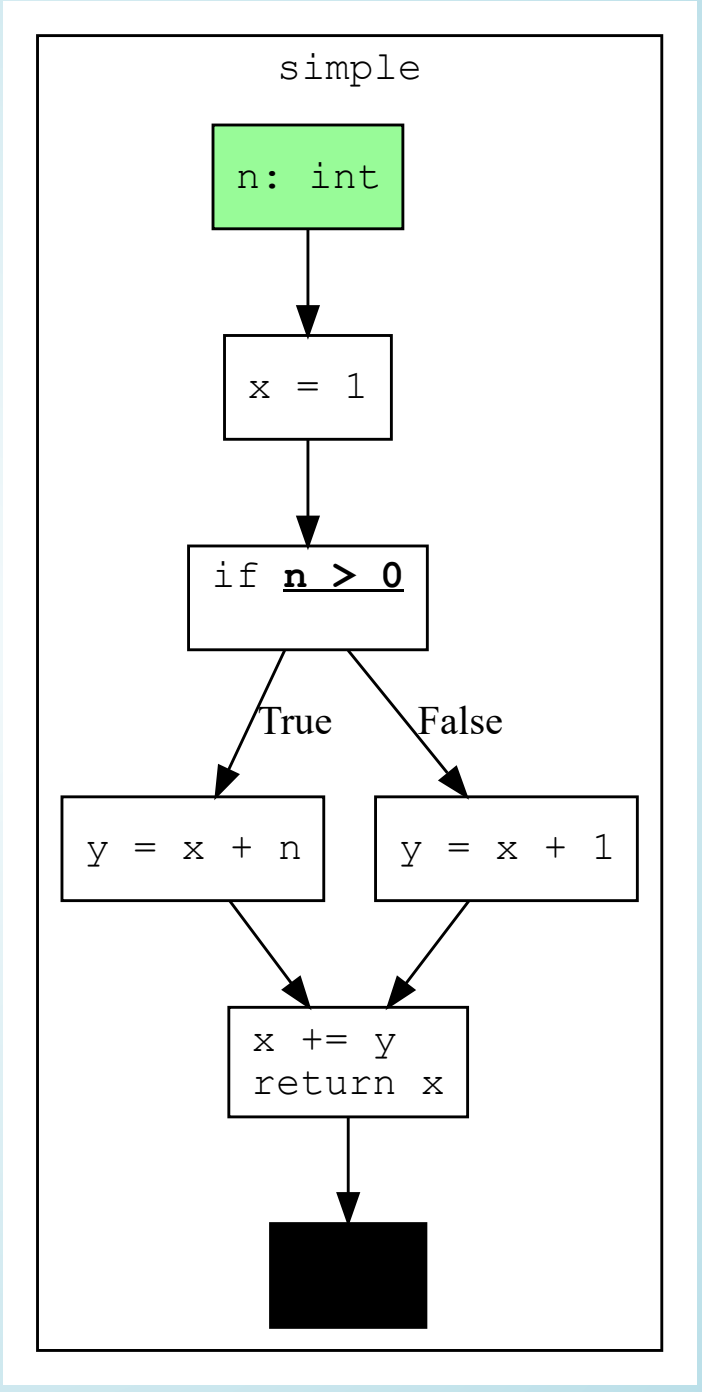
⚠ Technical limitations

`SnapshotTracer` is a new feature, and currently has some pretty significant limitations.

1. `SnapshotTracer` does not step into (or out of) function calls.
2. `SnapshotTracer` is quite slow!

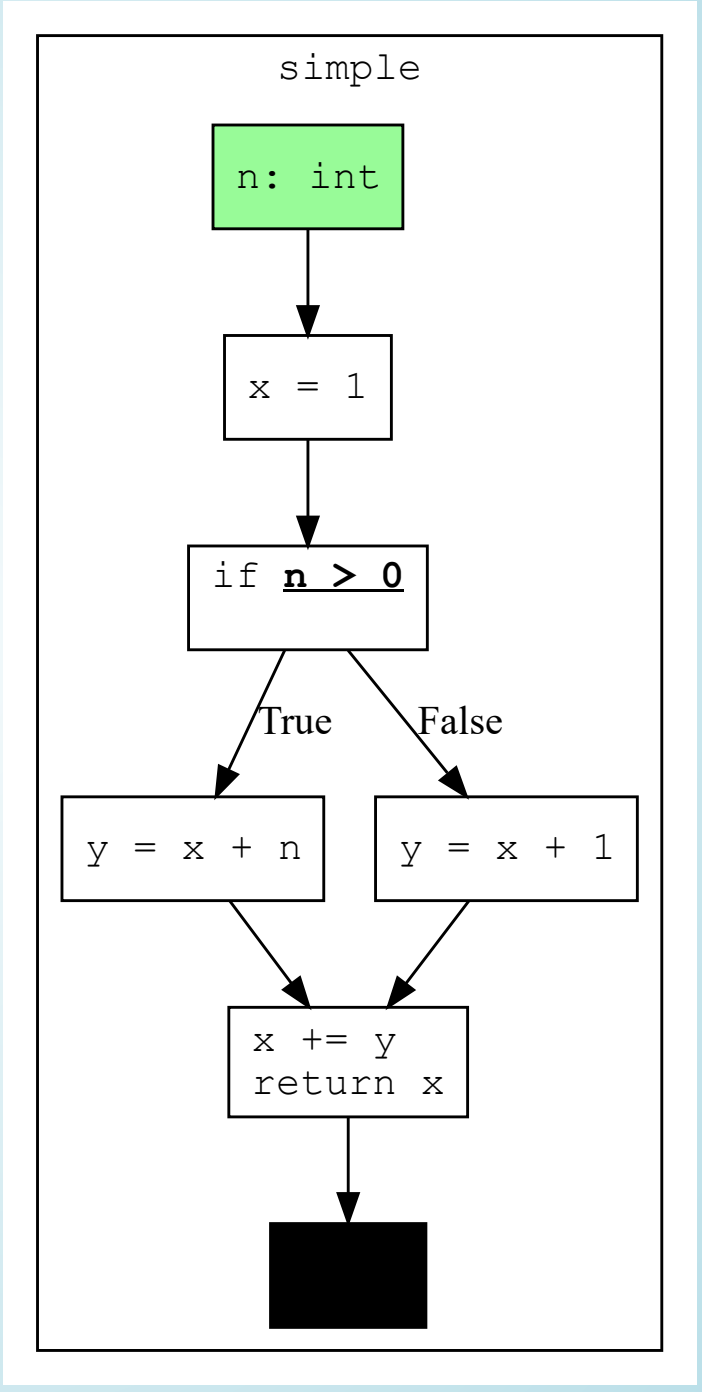
Control flow graphs

```
def simple(n: int) -> int:
    x = 1
    if n > 0:
        y = x + n
    else:
        y = x + 1
    x += y
    return x
```



A **control flow graph** is a directed graph representing the possible execution paths in code.

- **Nodes** in the graph represent basic blocks: a sequence of statements that has only one entry and exit point.
- **Edges** between basic blocks represent possible jumps in control flow.



Control flow graphs and PythonTA

PythonTA's static analysis builds a control flow graph to perform some of its checks.

It also allows users to visualize these control flow graphs using GraphViz.

```
from python_ta.cfg import generate_cfg

generate_cfg(
  module,          # Optional (default: current module)
  auto_open,                 # Optional (default: False)
  visitor_options: ...,   # Configuration options
)
```

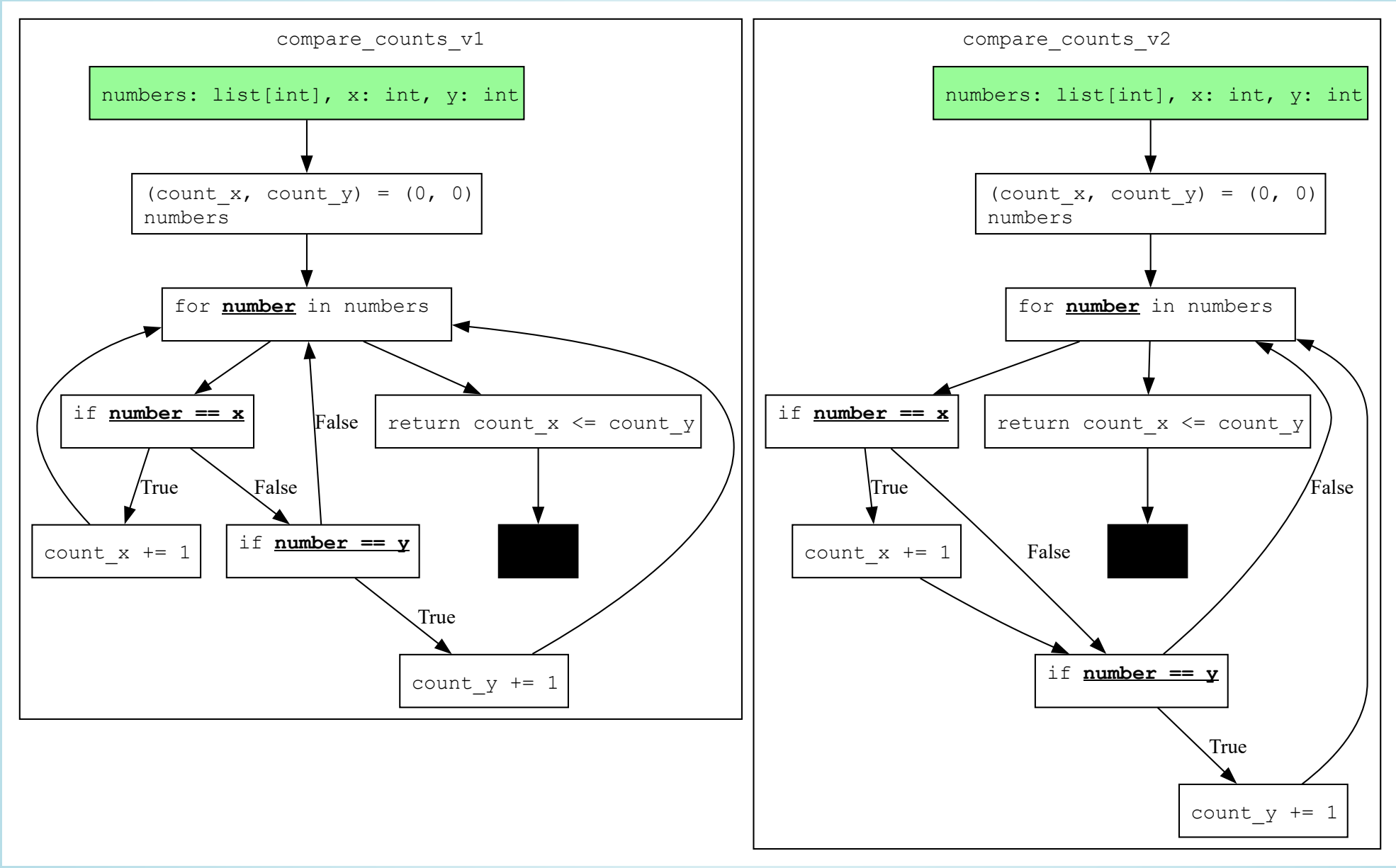
Demo: `demo5_control_flow_graphs.py`

Ideas for incorporating control flow graphs

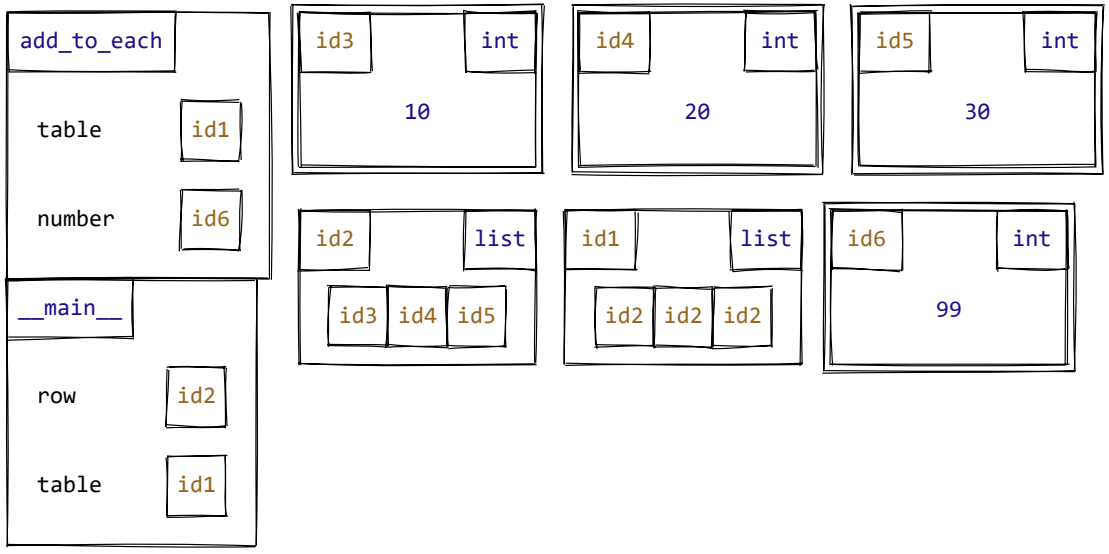
Use to illustrate differences in control flow structures.

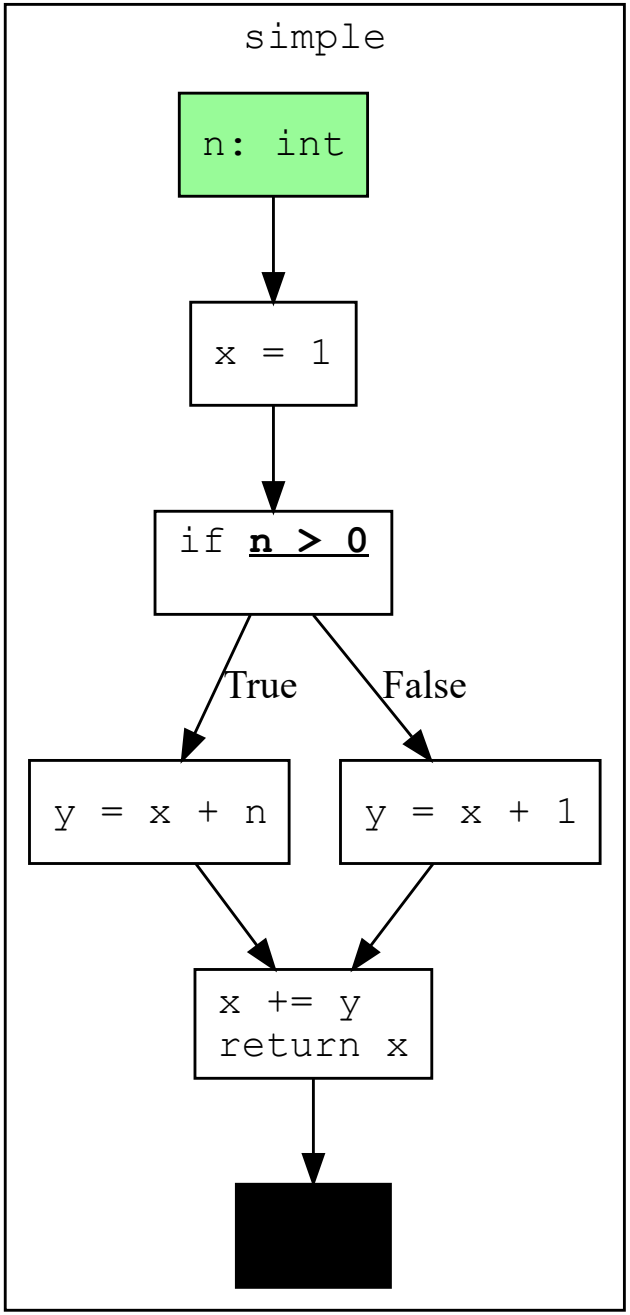
```
def compare_counts_v1(
    numbers: list[int],
    x: int, y: int
    ) -> bool:
  count_x, count_y = 0, 0
  for number in numbers:
    if number == x:
      count_x += 1
    elif number == y:
      count_y += 1

  return count_x <= count_y
```

```
def compare_counts_v2(
    numbers: list[int],
    x: int, y: int
    ) -> bool:
  count_x, count_y = 0, 0
  for number in numbers:
    if number == x:
      count_x += 1
    if number == y:
      count_y += 1

  return count_x <= count_y
```


compare_counts_v1
numbers: list[int], x: int, y: int
(count_x, count_y) = (0, 0)
numbers
for number in numbers
if number == x
False
return count_x <= count_y
True
False
count_x += 1
if number == y
True
count_y += 1
compare_counts_v2
numbers: list[int], x: int, y: int
(count_x, count_y) = (0, 0)
numbers
for number in numbers
if number == x
return count_x <= count_y
True
False
count_x += 1
False
if number == y
True
count_y += 1

iteration	number	current_sum	numbers
0	N/A	0	[10, 20, 30, 40]
1	10	10	[10, 20, 30, 40]
2	20	30	[10, 20, 30, 40]
3	30	60	[10, 20, 30, 40]
4	40	100	[10, 20, 30, 40]





Discussion!

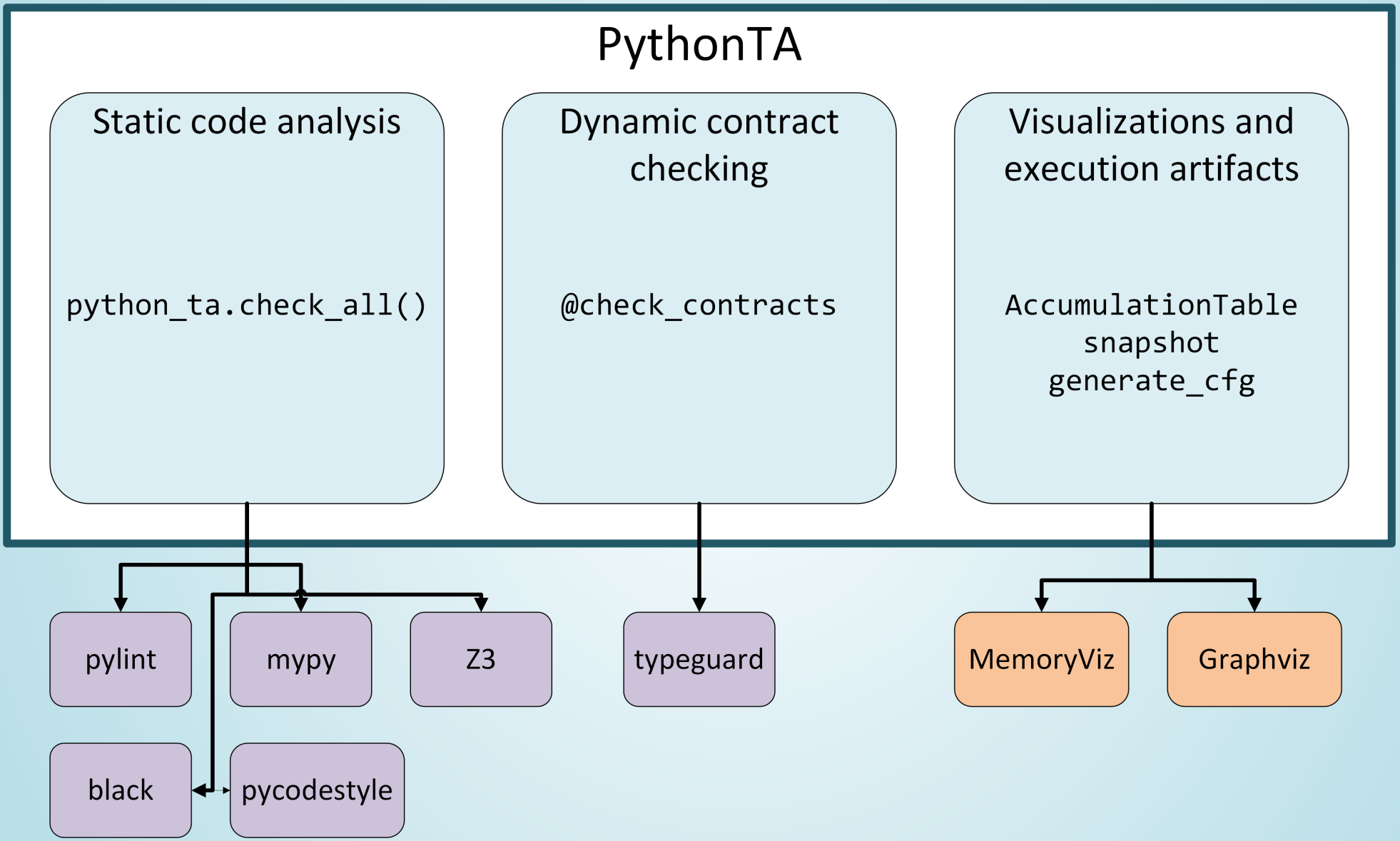
What questions do you have about the technical aspects of the tools?

What questions do you have about using the tools in your courses?

What opportunities and challenges do you see in using this tools?

What improvements would you suggest to make the tools more suitable for your courses?

Wrap up


PythonTA
Static code analysis
python_ta.check_all()
Dynamic contract checking
@check_contracts
Visualizations and execution artifacts
AccumulationTable
snapshot
generate_cfg
pylint
mypy
Z3
typeguard
MemoryViz
Graphviz
black
pycodestyle

Summary: Part 1 – Static Code Analysis

You learned:

- How to run PythonTA's static analysis
  - `python_ta.check_all`
  - CLI
- How to configure PythonTA's static analysis
  - `config` argument; configuration file
  - disable checks
  - override error messages
- Ways to incorporate PythonTA static analysis into your classroom

Summary: Part 2 – Dynamic Contract Checking

You learned:

- How to enable contract checking (`python_ta.contracts.check_contracts`)
- How to write conditions that are understood by PythonTA
- Ways to incorporate contract checking into your classroom

Summary: Part 3 – Generating Visualizations and Execution Artifacts

You learned:

- How to create loop tracing tables using `python_ta.debug.AccumulationTable`
- How to generate single memory model diagrams using `python_ta.debug.snapshot`
- How to generate an interactive code stepping visualizations using `python_ta.debug.SnapshotTracer`
- How to generate control flow diagrams using `python_ta.cfg.generate_cfg`
- Ways to incorporate these visualizations/artifacts in your classroom

Acknowledgements

Student contributors 🎉

Lorena Buciu
Simon Chen
Freeman Cheng
Ivan Chepelev
Yianni Culmone
Daniel Dervishi
Nigel Fong
Adam Gleizer
Ibrahim Hasan
Niayesh Ilkhani
Craig Katsube
Rebecca Kay
Christopher Koehler
David Kim
Simeon Krastnikov
Ryan Lee
Christopher Li
Hoi Ching (Herena) Li
Hayley Lin
Bruce Liu
Merrick Liu
Wendy Liu
Yibing (Amy) Lu
Maria Shurui Ma
Aina Fatema Asim Merchant
Shweta Mogalapalli
Ignas Panero Armoska
Justin Park
Harshkumar Patel
Varun Sahni
Eleonora Scognamiglio
Stephen Scott
Amr Sharaf
Richard Shi
Kavin Singh
Alexey Strokach
Sophy Sun
Utku Egemen Umut
Sarah Wang
Lana Wehbeh
Jasmine Wu
Raine Yang
Philippe Yu
Yi Cheng (Michael) Zhao

PythonTA has received funding from teaching and learning grants offered by the University of Toronto.

Learning more & Staying in touch

Check out the **GitHub** repository: https://github.com/pyta-uoft/pyta

Visit the project documentation:
https://www.cs.toronto.edu/~david/pyta/

Join the PythonTA users mailing list: david@cs.toronto.edu

Questions, feedback, or anything else: david@cs.toronto.edu

THANK YOU for coming!