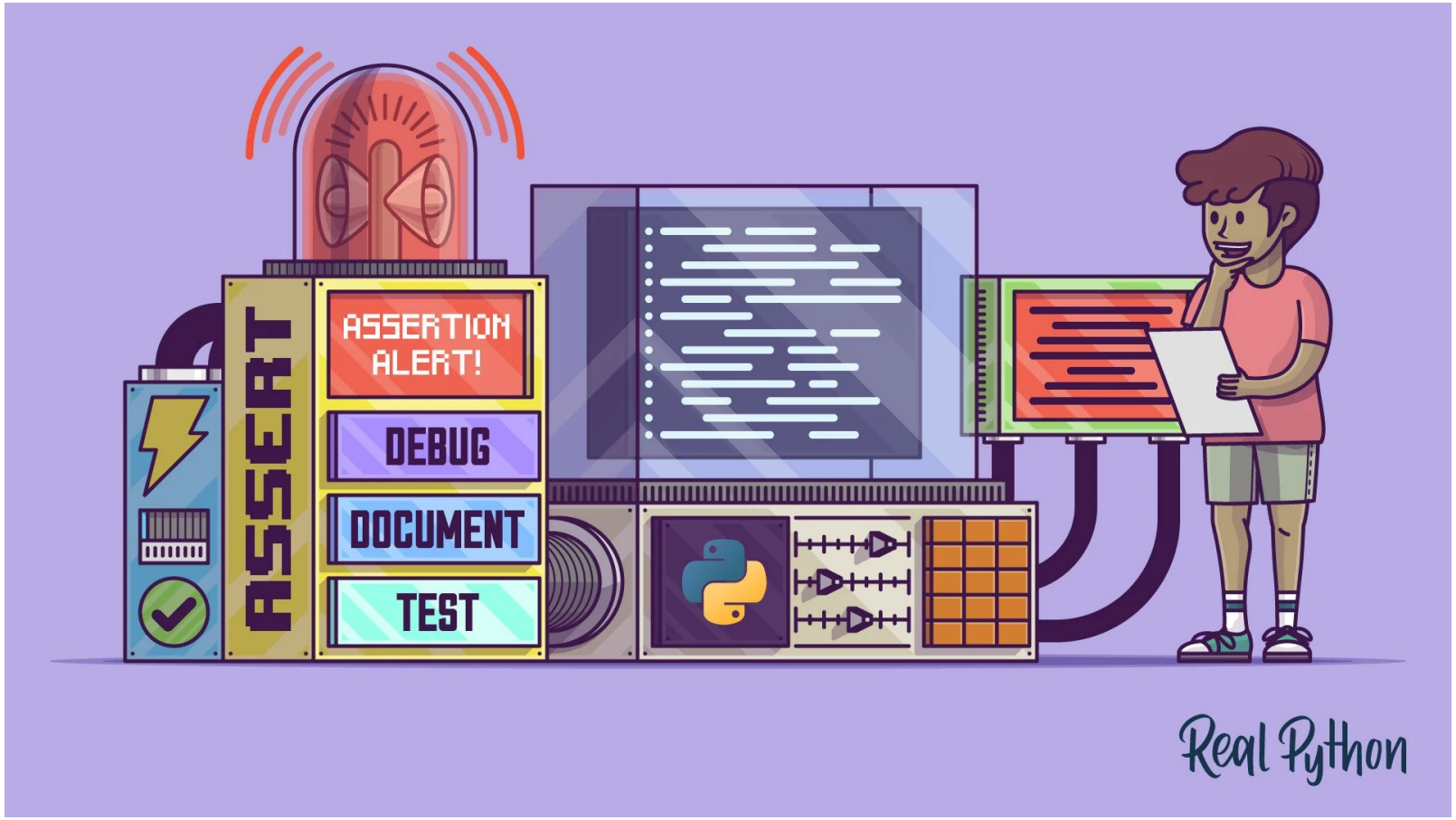
Using Python's `assert` to Debug and Test Your Code



Using Python's `assert` to Debug and Test Your Code

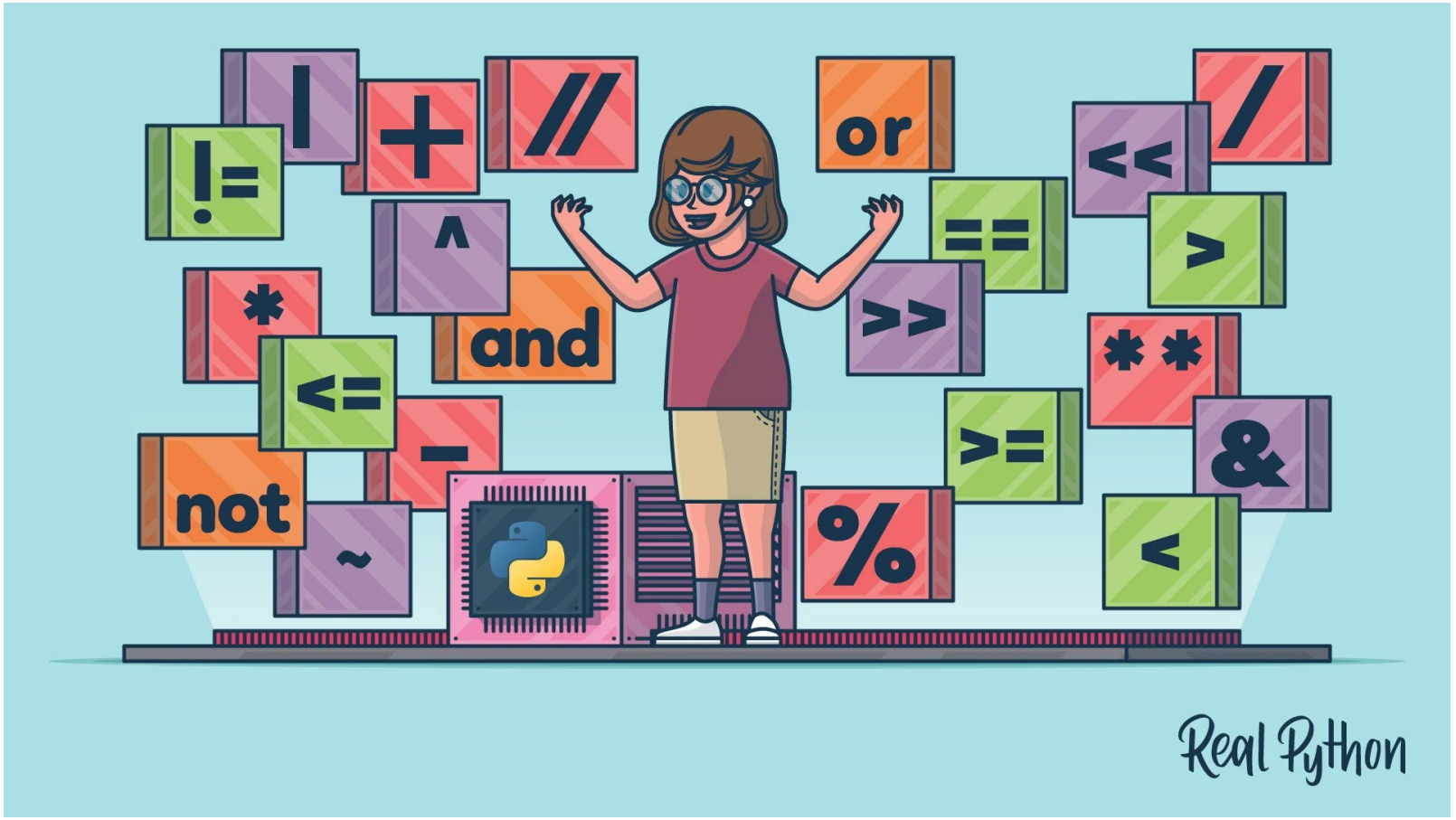
Using Python's `assert` to Debug and Test Your Code

- What Assertions Are and When To Use Them
- How Python's `assert` Statement Works
- Using `assert` To Document, Debug, and Test Your Code
- How Assertions Can Be Disabled To Improve Performance
- Common Pitfalls When Using `assert` Statements

Getting the Most From This Course

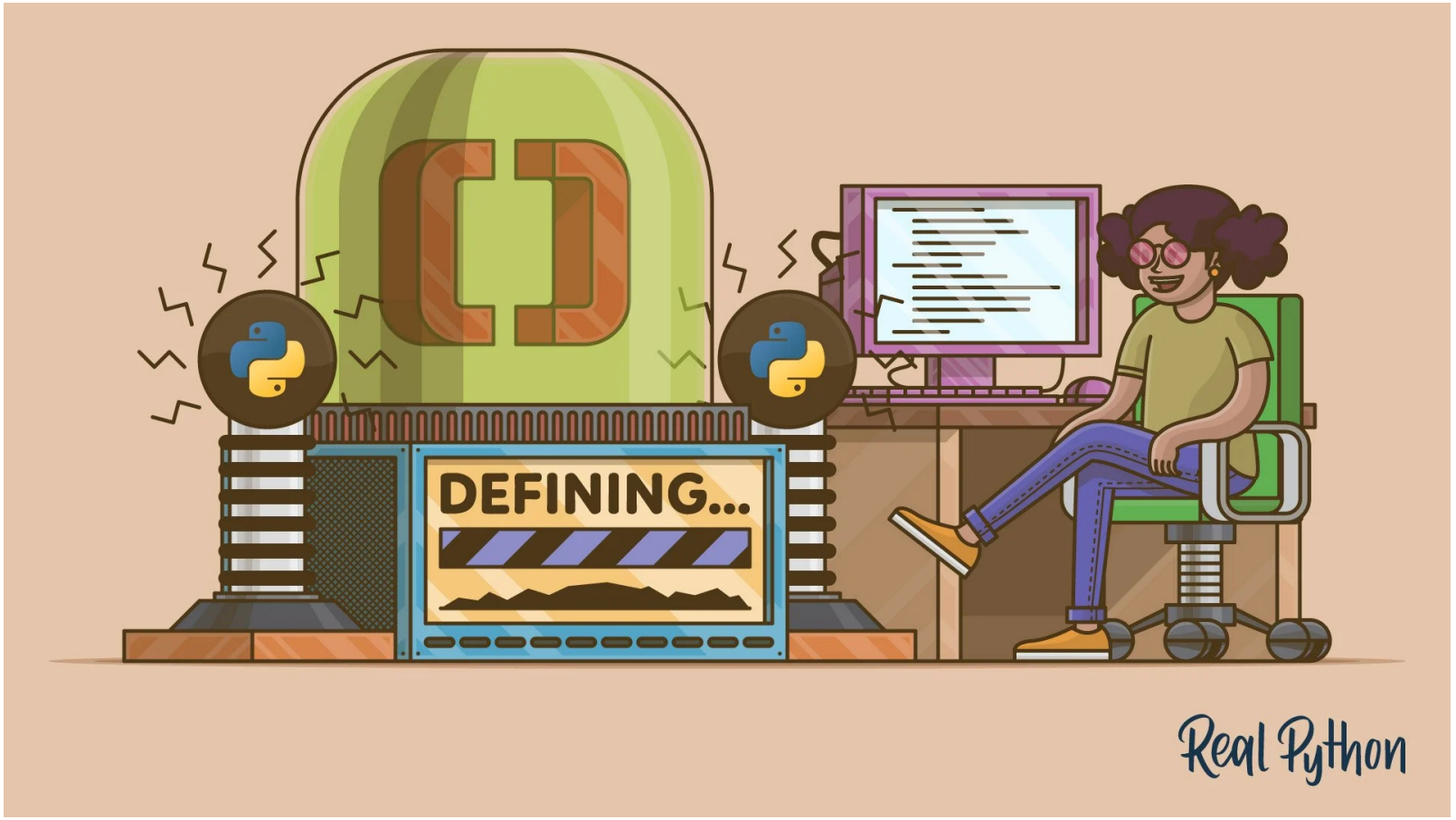
- Expressions and Operators
- Functions
- Conditional Statements
- Exceptions
- Documenting
- Debugging
- Testing

Operators and Expressions in Python



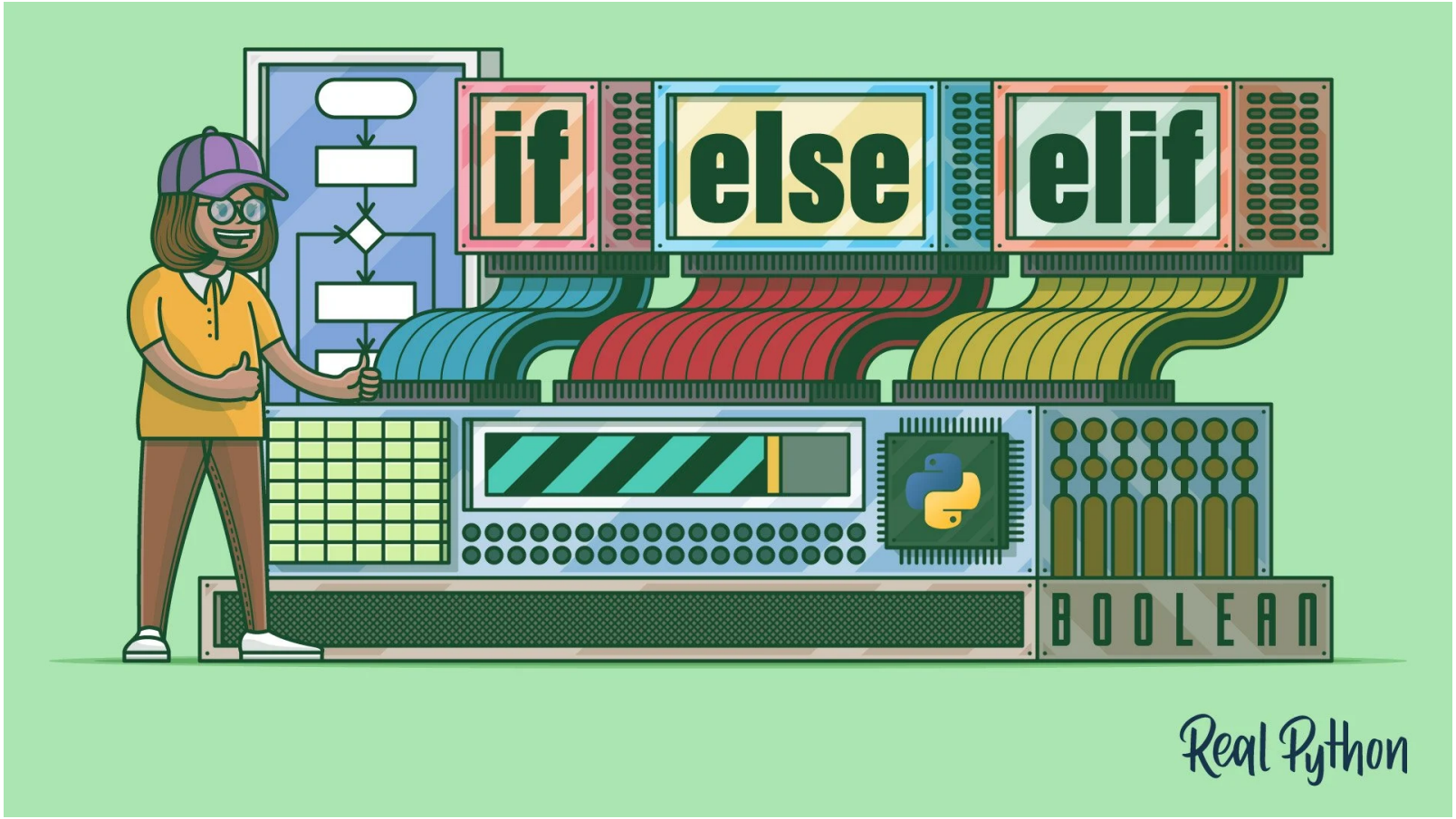
https://realpython.com/python-operators-expressions/

Defining Your Own Python Function



https://realpython.com/defining-your-own-python-function/

Conditional Statements in Python

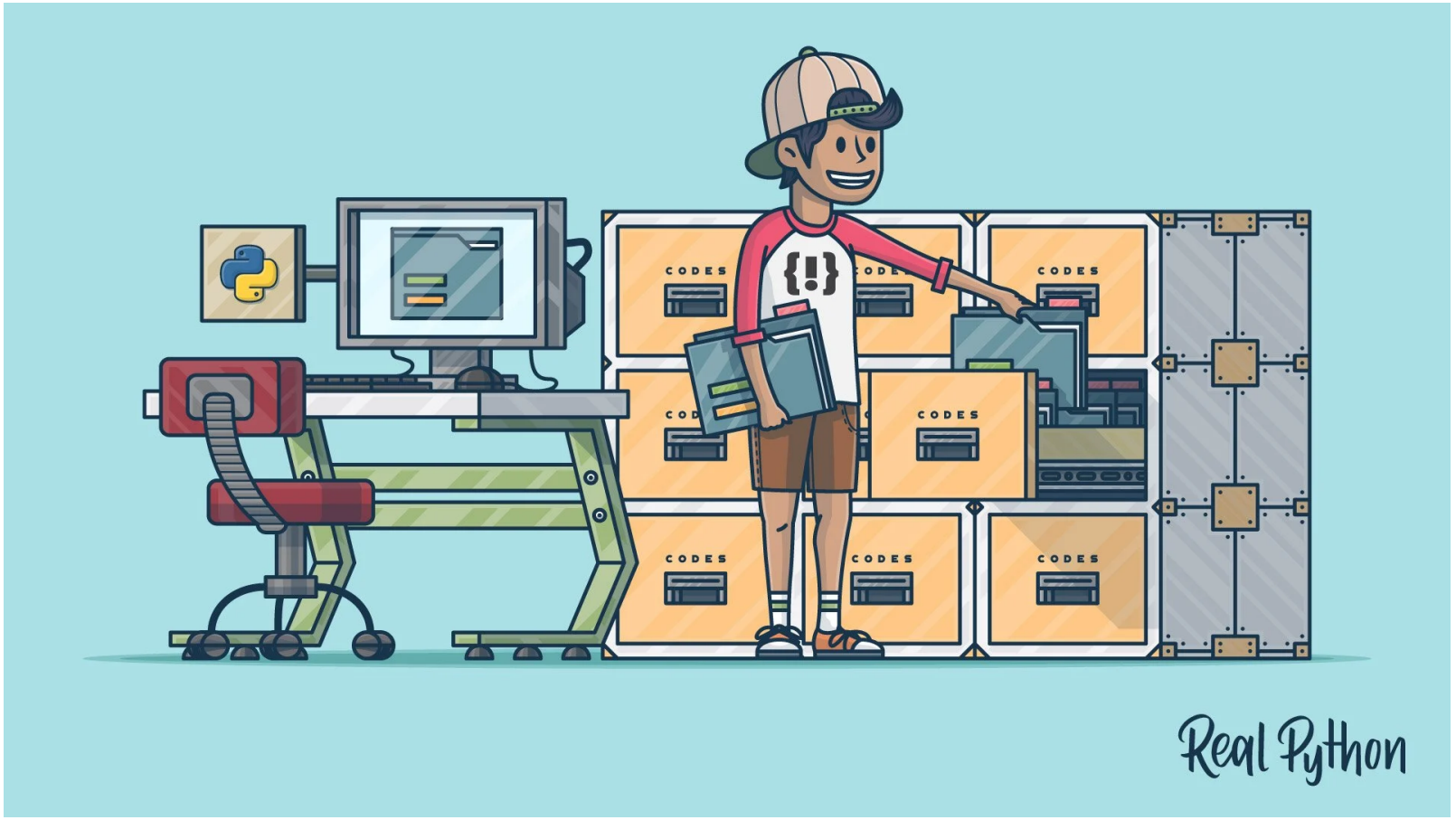


https://realpython.com/python-conditional-statements/

Python Exceptions: An Introduction

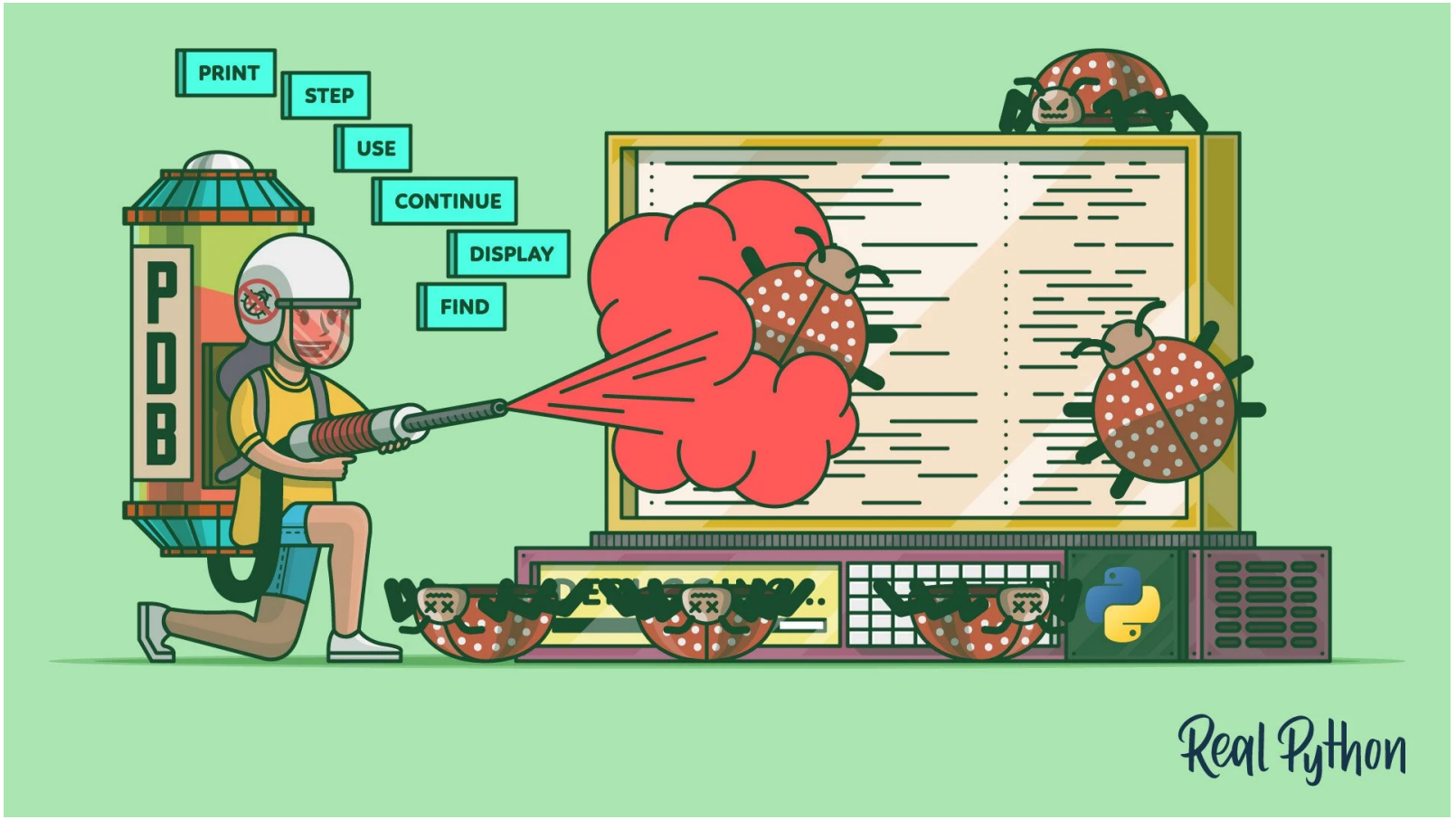
https://realpython.com/python-exceptions/

Documenting Python Code: A Complete Guide



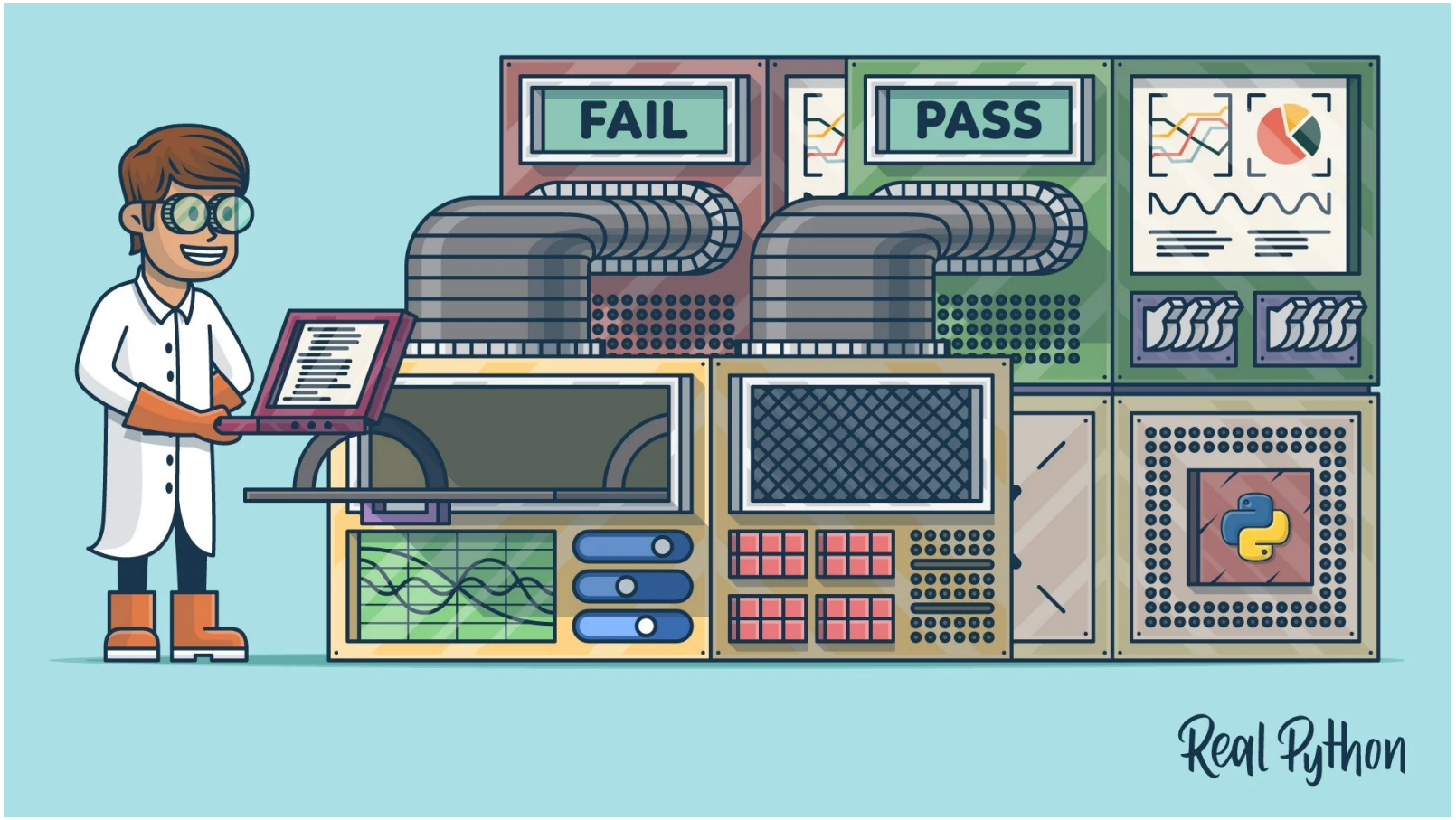
https://realpython.com/documenting-python-code/

Python Debugging With Pdb



https://realpython.com/python-debugging-pdb/

Getting Started With Testing in Python



https://realpython.com/python-testing/

Bpython Interpreter

https://bpython-interpreter.org/

Let’s Get Started!

Using Python's `assert` to Debug and Test Your Code

1. **Getting to Know Assertions**
2. Understanding Python's `assert` Statement
3. Exploring Common Assertion Formats
4. Documenting Your Code With Assertions
5. Debugging Your Code With Assertions
6. Disabling Assertions in Production
7. Testing Your Code With Assertions
8. Understanding Common Pitfalls of `assert`

Getting to Know Assertions

- Useful During Development
- Found in Other Languages
- Used for:
    - Documenting
    - Debugging
    - Testing

What Are Assertions?

- Statements
- Sanity Checks During Development
- Test Correctness of Code
- Check if Specific Conditions Are True
- Assertion Failure:
    - Indicates a Bug
    - Terminates Execution

What Do Assertions Check?

- Invariants *are* Invariant
- Assumptions:
  - Preconditions
  - Postconditions
- This argument `is not None`
- Return Value is `str`

What Are Assertions Good For?

- Debugging
  - Ensure New Bugs Are Not Added
- Documenting
- Testing

The Primary Role of Assertions

- Trigger Alarms When a Bug Appears
- Make Sure a Condition Remains True
- Otherwise Throw an Error

Checking Conditions

- Preconditions
- Postconditions

Checking Conditions

- Preconditions - Check Input is Valid
- Postconditions

Checking Conditions

- Preconditions - Check Input is Valid
- Postconditions - Check Output is Valid
- Condition Remains True
- Can Include Optional Message

Use in Documentation

- Action Taken
- Active Unlike:
  - Comments
  - Docstrings

Testing

- Test Cases
- Concise
- Check if Condition is Met

When Not To Use Assertions

- Data Processing
- Data Validation
- Error Handling

Next: Understanding Python's `assert` Statements

Using Python's `assert` to Debug and Test Your Code

1. Getting to Know Assertions
2. **Understanding Python's `assert` Statement**
3. Exploring Common Assertion Formats
4. Documenting Your Code With Assertions
5. Debugging Your Code With Assertions
6. Disabling Assertions in Production
7. Testing Your Code With Assertions
8. Understanding Common Pitfalls of `assert`

Understanding Python's `assert` Statement

- `assert` Is a Statement, Not a Function

The Syntax of the `assert` Statement

- `assert` Keyword
- Expression or Condition to Test
- Optional Message
- Condition Should be `True`
- `False` Condition Raises an `AssertionError`

The Syntax of the `assert` Statement

```
assert expression[, assertion_message]
```

- `expression` Is Tested For Truthiness
- If `expression` Is `False`, `AssertionError` is Raised
- `assertion_message` Is Optional, But Encouraged

The Syntax of the `assert` Statement

- Parentheses Not Needed to Group `expression` and `assertion_message`

The Syntax of the `assert` Statement

```
number = 42

assert (
    number > 0 and isinstance(number, int),
    f"number greater than 0 expected, got: {number}"
)
```

- Using `(` & `)` To Split Long Lines Is Common Practice
- This Turns the Expression and Message Into a Two-Item Tuple

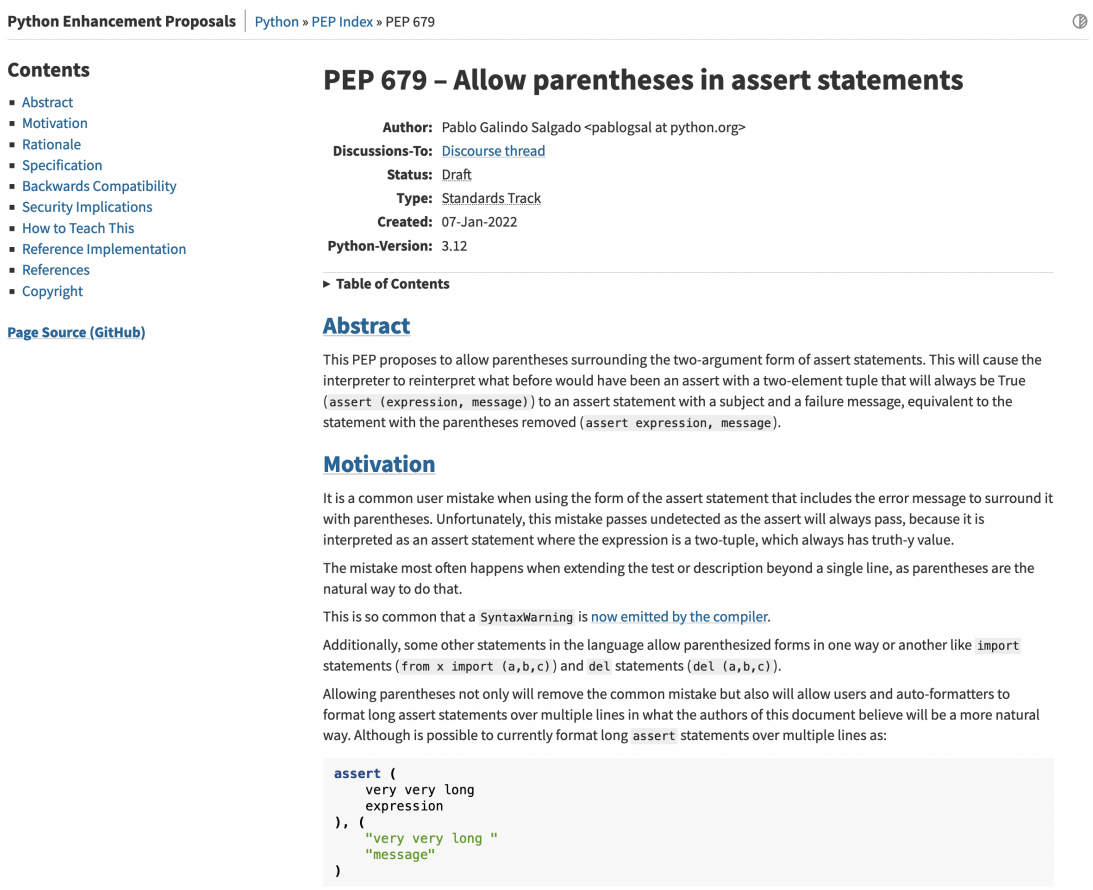
The Syntax of the `assert` Statement

```
number = 42

assert number > 0 and isinstance(number, int), \
    f"number greater than 0 expected, got: {number}"
```

- Use `\` To Join Two Physical Lines Into One Logical Line
- Appropriate Line Length Retained Without Logical Errors

PEP 679 – Allow Parentheses in Assert Statements



Python Enhancement Proposals | Python » PEP Index » PEP 679

Contents

- Abstract
- Motivation
- Rationale
- Specification
- Backwards Compatibility
- Security Implications
- How to Teach This
- Reference Implementation
- References
- Copyright

Page Source (GitHub)

PEP 679 – Allow parentheses in assert statements

Author: Pablo Galindo Salgado <pablogsal at python.org>
Discussions-To: Discourse thread
Status: Draft
Type: Standards Track
Created: 07-Jan-2022
Python-Version: 3.12

▸ Table of Contents

Abstract

This PEP proposes to allow parentheses surrounding the two-argument form of assert statements. This will cause the interpreter to reinterpret what before would have been an assert with a two-element tuple that will always be True (`assert (expression, message)`) to an assert statement with a subject and a failure message, equivalent to the statement with the parentheses removed (`assert expression, message`).

Motivation

It is a common user mistake when using the form of the assert statement that includes the error message to surround it with parentheses. Unfortunately, this mistake passes undetected as the assert will always pass, because it is interpreted as an assert statement where the expression is a two-tuple, which always has truth-y value.

The mistake most often happens when extending the test or description beyond a single line, as parentheses are the natural way to do that.

This is so common that a `SyntaxWarning` is now emitted by the compiler.

Additionally, some other statements in the language allow parenthesized forms in one way or another like `import` statements (`from x import (a,b,c)`) and `del` statements (`del (a,b,c)`).

Allowing parentheses not only will remove the common mistake but also will allow users and auto-formatters to format long assert statements over multiple lines in what the authors of this document believe will be a more natural way. Although is possible to currently format long `assert` statements over multiple lines as:

```
assert (
    very very long
    expression
), (
    "very very long "
    "message"
)
```

https://peps.python.org/pep-0679/

The `AssertionError` Exception

- Raised When Assertion Expression Equates to `False`
- Message Used as Argument to `AssertionError`
- Exception Breaks Program Execution
- Generally Not Raised Explicitly
- Should Not Be Caught in `try` … `except` Block
- Concrete Exception That Should be Raised, Not Subclassed

Next: Exploring Common Assertion Formats

Using Python's `assert` to Debug and Test Your Code

Exploring Common Assertion Formats

Other Assertion Formats

- Boolean-Valued Functions
- Python Objects
- Comparison Expressions
- Boolean Expressions
- Python Expressions

Next: Documenting Your Code With Assertions

Using Python's `assert` to Debug and Test Your Code

1. Getting to Know Assertions
2. Understanding Python's `assert` Statement
3. Exploring Common Assertion Formats
4. **Documenting Your Code With Assertions**
5. Debugging Your Code With Assertions
6. Disabling Assertions in Production
7. Testing Your Code With Assertions
8. Understanding Common Pitfalls of `assert`

Documenting Your Code With Assertions

- Effective
- Specific Condition Clearly Indicated

Documenting Your Code With Assertions

- Effective
- Powerful
- Documents Intentions
- Avoids Hard-To-Find Bugs

Next: Debugging Your Code With Assertions

Using Python's `assert` to Debug and Test Your Code

Debugging Your Code With Assertions

- Debugging Aid
- Testing Conditions That Should Remain True
- Failure Should Indicate Bug in Code

An Example Of Debugging With Assertions

- Debug Code During Development
- Ensure Conditions Are and Remain True
- If the Assertion Becomes False, a Bug Is Present

Debugging `Circle`

- `assert` Works as a Watchdog for Invalid `.radius` Values
- Points to Specific Problem:
  - `.radius` Is Now Negative

Considerations When Debugging With Assertions

- Used to State Preconditions:
  - Checking Value Before Computation
- Used To State Postconditions:
  - Checking for Valid Return Value
- Conditions Should Remain `True` At All Times:
  - Becoming `False` Should Indicate a Bug

`AssertionError` Should Not Be Caught

- Do Not Use `try` ... `except` Blocks With `AssertionError`
- `AssertionError` Should Lead to Program Termination

The Goal Of Assertions in Debugging

- Informing Developers of Unrecoverable Errors
- Not For Expected Errors
- Uncovering Programmers' Errors
- Useful During Development, Not Production

Next: Disabling Assertions in Production

Using Python's `assert` to Debug and Test Your Code

Disabling Assertions in Production

- Development Cycle Complete
- Code:
  - Reviewed
  - Tested
- All Assertions Pass
- Code Ready For Release

Why Disable Assertions?

- Slower Performance
- Greater Memory Usage

Disabling Assertions

- `python -O` or `python -OO`
- Setting The `PYTHONOPTIMIZE` Environment Variable

Understanding The `__debug__` Constant

- Related to `assert`
- Boolean
- Defaults to `True`

The Value of `__debug__`

Mode	`__debug__`
Normal	`True`
Optimized	`False`

Code Equivalent to `assert`

```
if __debug__:
    if not expression:
        raise AssertionError(assertion_message)
```

Development Workflow

- Normal : `__debug__` = `True`
  - Assertions Functional
  - Development and Testing
- Optimized : `__debug__` = `False`
  - Assertions Disabled
  - Production

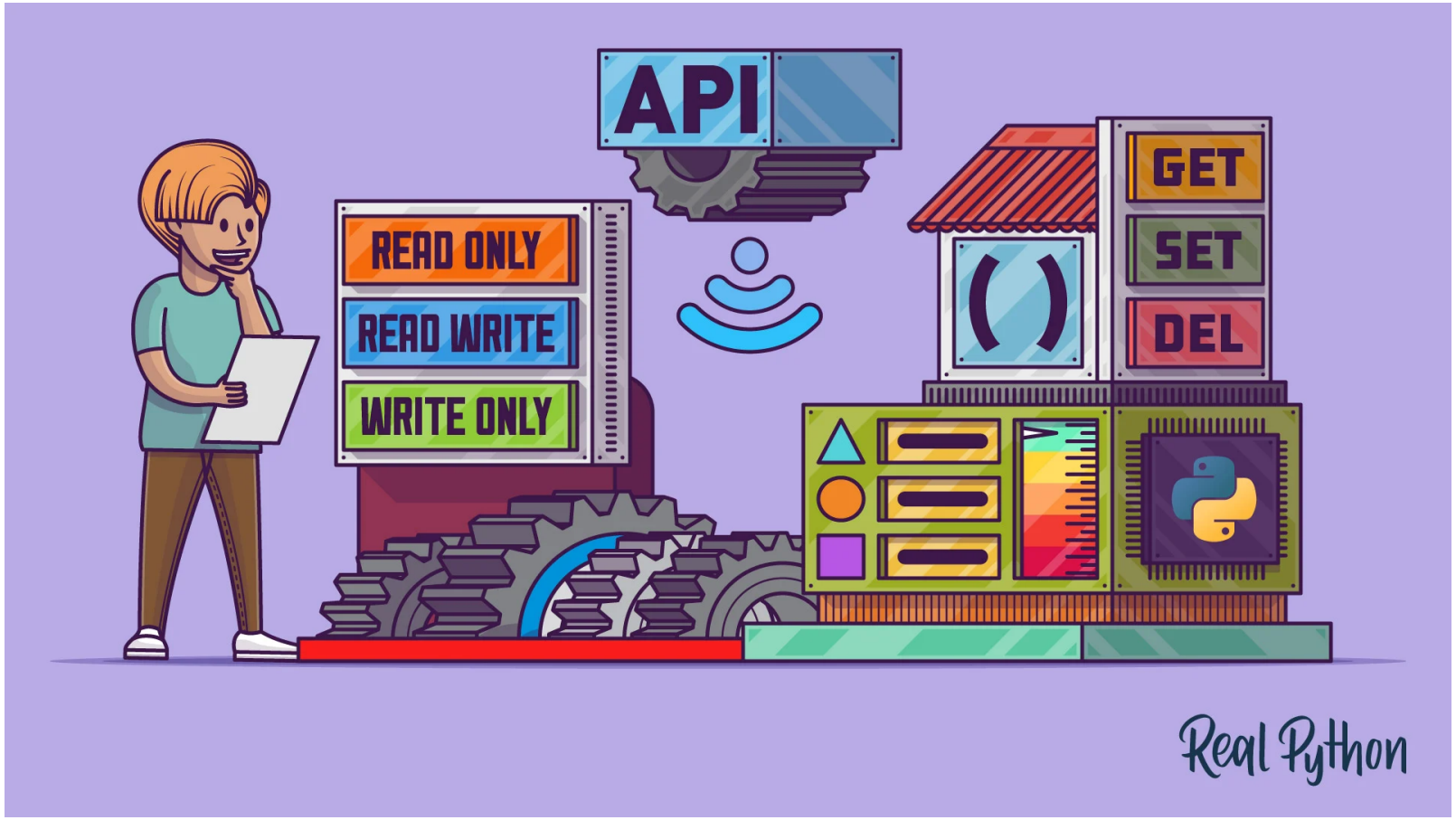
Running Python With the `-O` or `-OO` Options

- `python -O`
  - Sets `__debug__` to `False`
  - Disables `assertions`
  - Disables conditionals checking `__debug__`
- `python -OO`
  - Also Disables Docstrings

Disabled Assertions

- The Main Reason To Not Use `assert` For Validation
- Production Code Typically Runs In Optimized Mode

Managing Attributes With Python's property()



https://realpython.com/courses/property-python/

Code Using `__debug__` Conditionals

- Also Disabled in Optimized Mode

Running Python in Optimized Mode

- `python -O`
- `python -OO`
- Repetitive
- Error-Prone

Next: Setting the `PYTHONOPTIMIZE` Environment Variable

Using Python's `assert` to Debug and Test Your Code

1. Getting to Know Assertions
2. Understanding Python's `assert` Statement
3. Exploring Common Assertion Formats
4. Documenting Your Code With Assertions
5. Debugging Your Code With Assertions
6. Disabling Assertions in Production
   - 6.1 **Setting the `PYTHONOPTIMIZE` Environment Variable**
7. Testing Your Code With Assertions
8. Understanding Common Pitfalls of `assert`

Setting the `PYTHONOPTIMIZE` Environment Variable

- `PYTHONOPTIMIZE` = `"anything"` Is Equivalent to `python -O`

Launch Python In The Normal Way

Setting `PYTHONOPTIMIZE` To An Integer

Value	Equivalent
`1`	`python -O`
`2`	`python -OO`

Other `PYTHONOPTIMIZE` Values

- Any Integer Value is Accepted
- Python Only Implements Two Levels
- `0` Disables Optimization

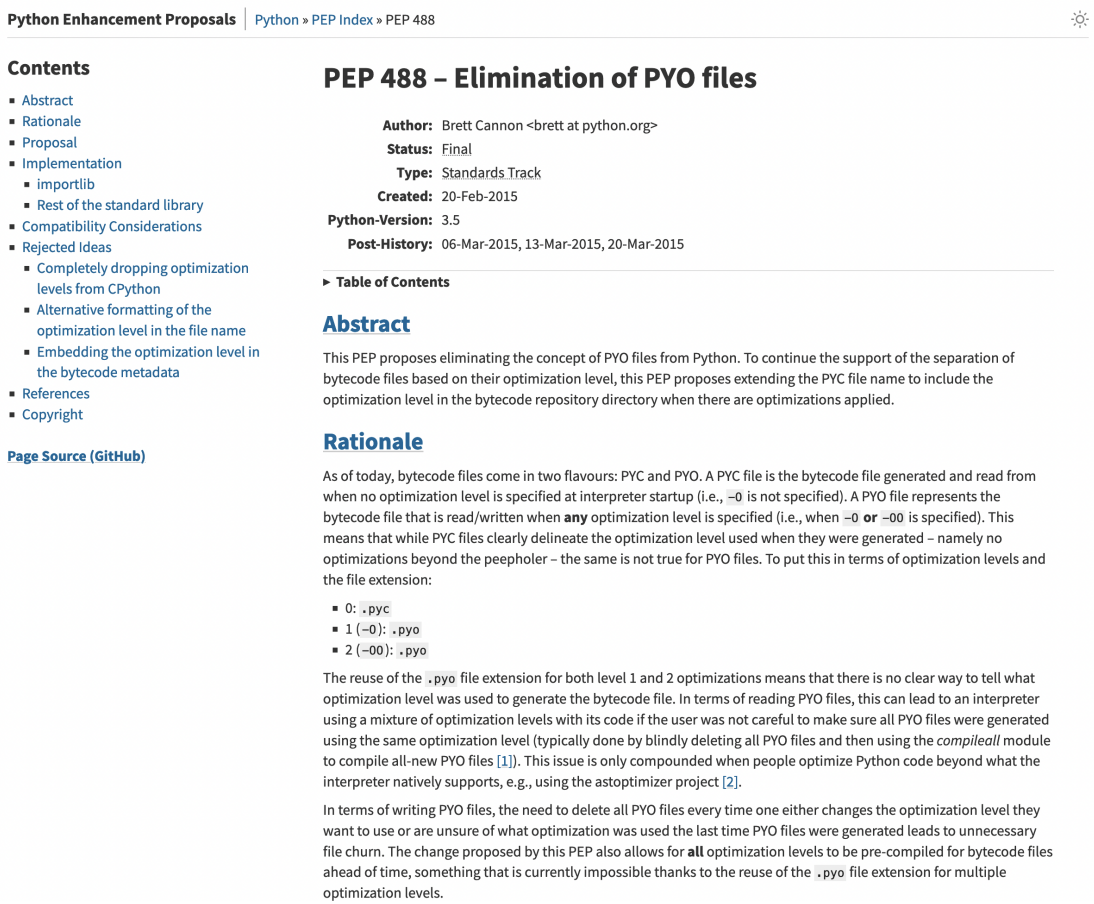
Running Python in Optimized Mode

- Bytecode Compiled On-The-Fly
- Bytecode Stored in `__pycache__/`
- `.pyc` File Names:
  - Original Module
  - Interpreter Name and Version
  - Optimization Level

`.pyc` Files

Command	`PYTHONOPTIMIZE`	Filename
`python circle.py`	`0`	`circle.cpython-311.pyc`
`python -O circle.py`	`1`	`circle.cpython-311.opt-1.pyc`
`python -OO circle.py`	`2`	`circle.cpython-311.opt-2.pyc`

PEP 488 - Elimination of PYO Files



Python Enhancement Proposals | Python » PEP Index » PEP 488

Contents

- Abstract
- Rationale
- Proposal
- Implementation
  - importlib
  - Rest of the standard library
- Compatibility Considerations
- Rejected Ideas
  - Completely dropping optimization levels from CPython
  - Alternative formatting of the optimization level in the file name
  - Embedding the optimization level in the bytecode metadata
- References
- Copyright

Page Source (GitHub)

PEP 488 – Elimination of PYO files

Author: Brett Cannon <brett at python.org>
Status: Final
Type: Standards Track
Created: 20-Feb-2015
Python-Version: 3.5
Post-History: 06-Mar-2015, 13-Mar-2015, 20-Mar-2015

▸ Table of Contents

Abstract

This PEP proposes eliminating the concept of PYO files from Python. To continue the support of the separation of bytecode files based on their optimization level, this PEP proposes extending the PYC file name to include the optimization level in the bytecode repository directory when there are optimizations applied.

Rationale

As of today, bytecode files come in two flavours: PYC and PYO. A PYC file is the bytecode file generated and read from when no optimization level is specified at interpreter startup (i.e., `-O` is not specified). A PYO file represents the bytecode file that is read/written when **any** optimization level is specified (i.e., when `-O` **or** `-OO` is specified). This means that while PYC files clearly delineate the optimization level used when they were generated – namely no optimizations beyond the peepholer – the same is not true for PYO files. To put this in terms of optimization levels and the file extension:

- 0: `.pyc`
- 1 (`-O`): `.pyo`
- 2 (`-OO`): `.pyo`

The reuse of the `.pyo` file extension for both level 1 and 2 optimizations means that there is no clear way to tell what optimization level was used to generate the bytecode file. In terms of reading PYO files, this can lead to an interpreter using a mixture of optimization levels with its code if the user was not careful to make sure all PYO files were generated using the same optimization level (typically done by blindly deleting all PYO files and then using the *compileall* module to compile all-new PYO files [1]). This issue is only compounded when people optimize Python code beyond what the interpreter natively supports, e.g., using the astoptimizer project [2].

In terms of writing PYO files, the need to delete all PYO files every time one either changes the optimization level they want to use or are unsure of what optimization was used the last time PYO files were generated leads to unnecessary file churn. The change proposed by this PEP also allows for **all** optimization levels to be pre-compiled for bytecode files ahead of time, something that is currently impossible thanks to the reuse of the `.pyo` file extension for multiple optimization levels.

https://peps.python.org/pep-0488/

Next: Testing Your Code With Assertions

Using Python's `assert` to Debug and Test Your Code

Testing Your Code With Assertions

- Comparing Observed and Expected Values
- Assertions Check for True Conditions
- `pytest` Third-Party Library:
  - Uses `assert` Statements
  - Most Test Cases Are Suited to `assert`

`pytest` Use of `assert` Statements

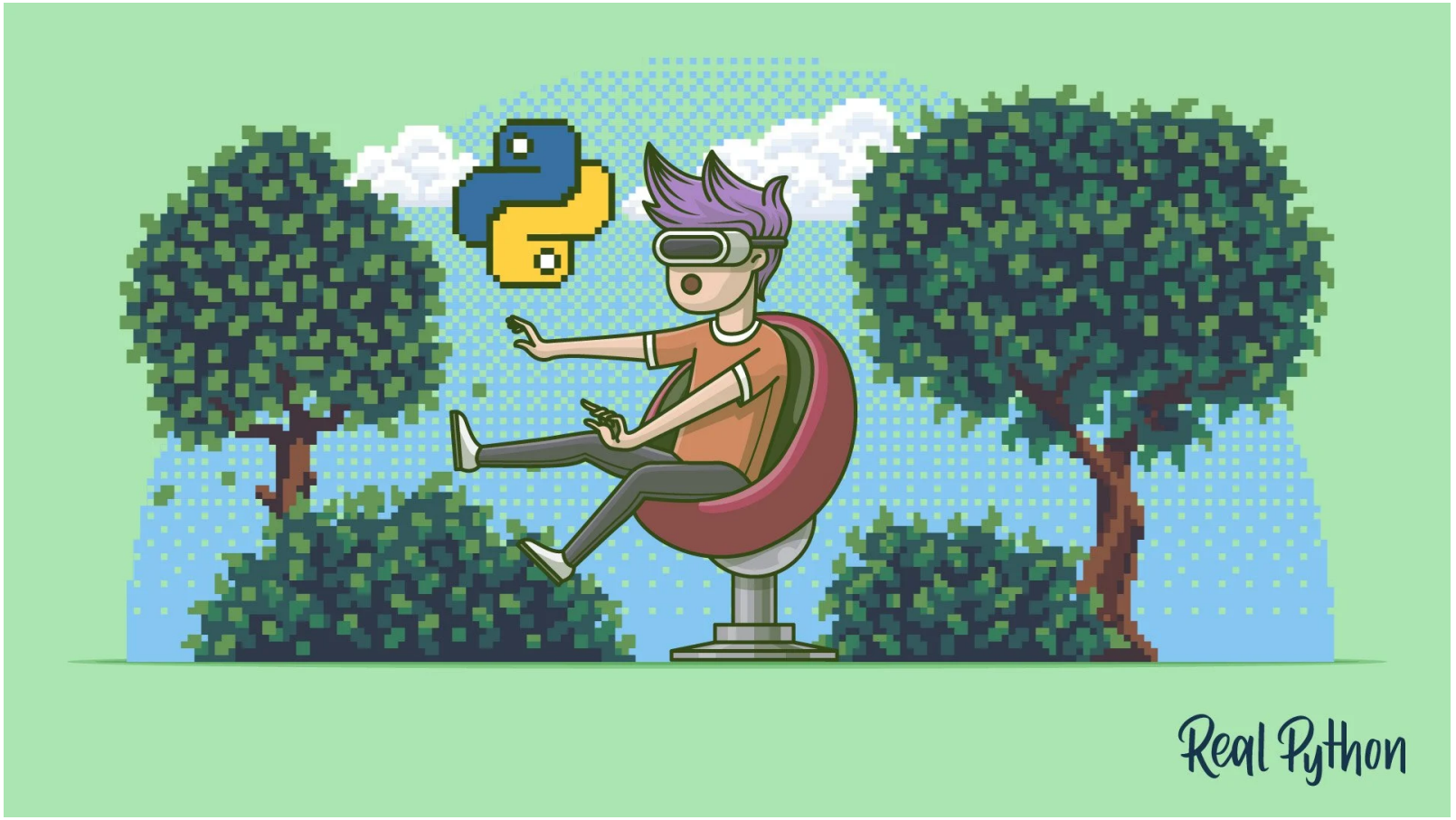
- Low Entry Barrier
- Flatter Learning Curve
- No Imports Needed For Many Test Cases

Standard Library `unittest` Module

- API of `.assert*()` Methods:
  - `assertEqual(a, b)`
  - `assertNotEqual(a, b)`
  - `assertTrue(x)`
  - `assertFalse(x)`
  - `assertIs(a, b)`
  - … and so on

Installing `pytest`

Python Virtual Environments Primer



https://realpython.com/python-virtual-environments-a-primer/

Checking The Value Of `PYTHONOPTIMIZE`

`pytest` And `assert`

- Close Integration
- Detailed Information About Failures

`pytest` Failing Assertion Detail

```
>       assert pow(10, 2) == 42
E       assert 100 == 42
E        +  where 100 = pow(10, 2)
```

- Root Cause of Failure Revealed

Next: Understanding Common Pitfalls of `assert`

Using Python's `assert` to Debug and Test Your Code

Understanding Common Pitfalls of `assert`

- Assertions Can Be Misused
- Should Be Used for Debugging and Testing During Development
- Should Not Be Used for Production Functionality:
  - Processing And Validating Data
  - Handling Errors
  - Running Operations with Side Effects
- Production Use Can Impact Performance
- Assertions Are Enabled by Default

Using `assert` for Data Processing and Validation

Handling Errors with `assert`

- Misused as a Quick Form of Error Handling
- Production Code Removes Assertions
- Error Checks No Longer Present

Never Catch `AssertionError`

- Silences Failing Assertions
- Catch Concrete Exceptions Instead
- Only Use Assertions To Catch Bug-Related Errors
- Remember Assertions Can Be Disabled

Next: More Common Pitfalls of `assert`

Using Python's `assert` to Debug and Test Your Code

More Common Pitfalls of `assert`

Running `assert` on Expressions With Side Effects

- Operations With Side Effects:
  - Modify State of Out-of-Scope Objects
- Occurs with Each Assertion
- May Alter Program State and Behaviour

Operations Without Side Effects

- Pure Functions
  - Input
  - Return Without Modification of State

Impacting Performance with `assert`

- Impact on Code Performance
- Assertions with:
  - Long Compound Conditions
  - Long Running Functions
  - Classes with Costly Instantiation

Impacting Performance with `assert`

- Execution Time
- Memory Usage

Avoiding Performance Issues in Production

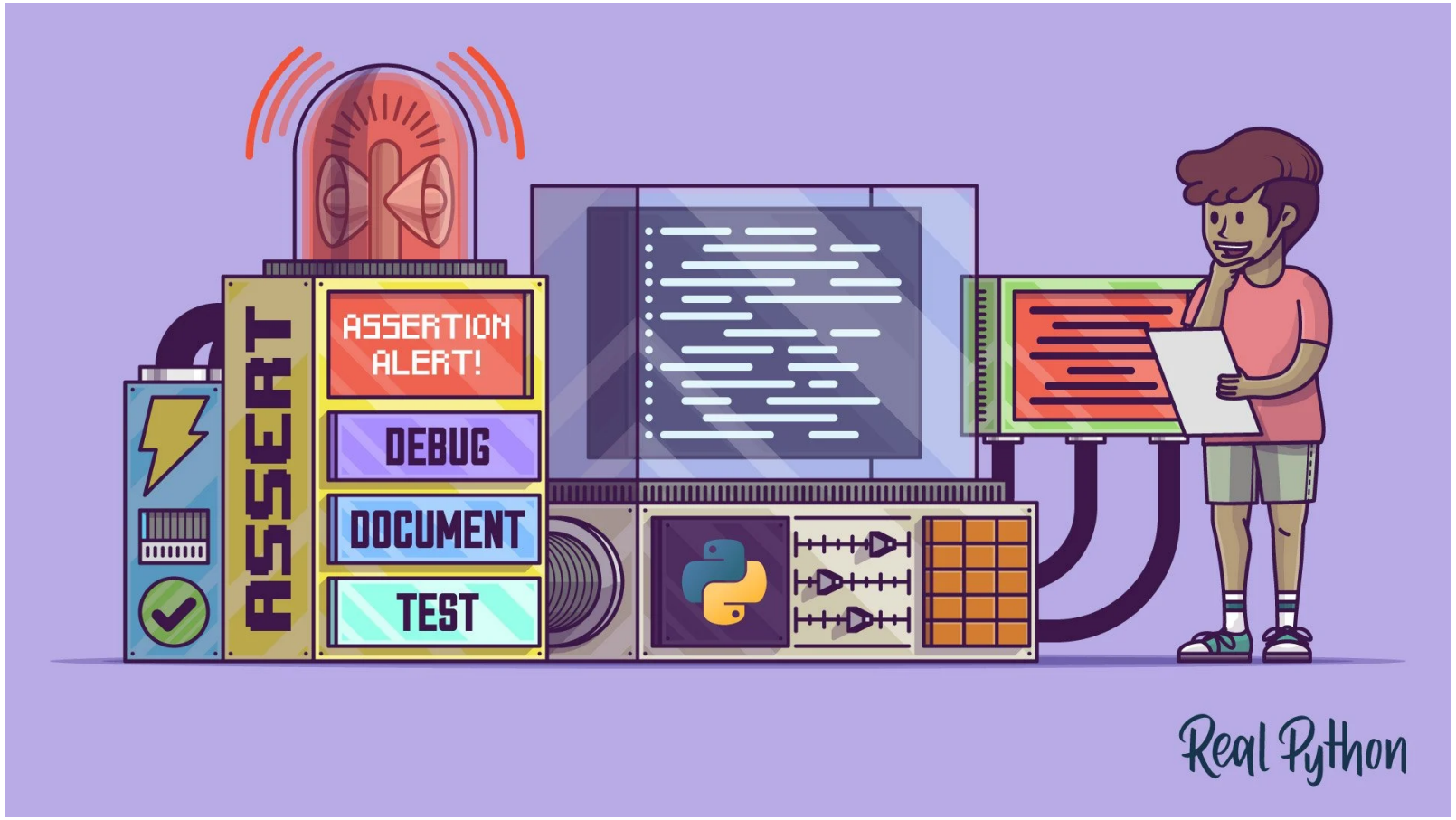
- `python -O` or `python -OO`
- Set `PYTHONOPTIMIZE`

Having `assert` Statements Enabled by Default

- Python Enables Assertions By Default
- Other Languages Disable Assertions By Default
- Assuming This Behavior Is a Source of Confusion

Next: Summary

Using Python's `assert` to Debug and Test Your Code : Summary



Summary

`assert` Used To:

- Document
- Debug
- Test

Summary

- What Assertions Are and When To Use Them
- How Python's `assert` Statement Works
- `assert` for Documenting, Debugging and Testing Code
- Disabling Assertions To Improve Performance
- Common Pitfalls When Using `assert` Statements

Summary