

Conversion Examples and Boundaries

Python 3.11 -> independently validated deterministic ISO C11 source



WORKS / CONVERTED

DOES NOT WORK / REJECTED

REFERENCE PROFILE

CONVERTER CONTRACT	0.14.3
SOURCE GRAMMAR	Python 3.11
TARGET	deterministic portable ISO C11 source
GUIDE EDITION	expanded examples and readability revision

Document map

Major sections are bookmarked and indexed below. Every conversion unit also carries an individual PDF bookmark.

WORKS / CONVERTED	Accepted source with complete C11 output and no diagnostics.
DOES NOT WORK / REJECTED	Rejected source with its primary PYC diagnostic.

QUICK RULES FOR COMPARISONS AND CONDITIONS	4
PART I - COMPARISONS, IF, ELIF, AND ELSE	5
PART II - POSITIONAL AND KEYWORD PARAMETERS	11
PART III - LOOPS, CONTAINERS, RECORDS, AND INTEGER FLOOR ARITHMETIC . .	16
PART IV - OTHER IMPORTANT BOUNDARIES	22
PART V - APPLICATION PATTERNS AND VERIFIED IMPLEMENTATION LIMITS . . .	24
CURRENT CONVERSION ENVELOPE	31
FINAL REMINDER	32

Purpose

This guide provides standalone Python examples showing what PyCForge converts today and what it deliberately rejects. The examples concentrate on:

- comparisons in if / elif / else;
- chained comparisons and short-circuit Boolean expressions;
- positional, keyword, positional-only, and required keyword-only arguments;
- branch-local variable rules;
- range loops and fixed containers;
- immutable static records;
- practical calculator and numeric-text patterns;
- verified container and loop-target ceilings; and
- common unsupported Python features.

Every example in this guide was exercised directly against the sealed PyCForge 0.15.2 package, whose converter contract is 0.14.3:

- WORKS examples returned Converted with no diagnostics and complete C11.
- DOES NOT WORK examples returned Rejected with the listed primary diagnostic and no generated C.

IMPORTANT

Treat every numbered example as a separate source file. PyCForge rejects an entire conversion unit when any function in that unit is unsupported. It never publishes partial C for the remaining functions.

Reading map

- Part I covers comparisons and structured conditions.
- Part II covers positional and keyword call binding.
- Part III covers loops, fixed containers, records, and floor arithmetic.
- Part IV lists broad unsupported-language boundaries.
- Part V provides compact application patterns and newly verified limits.
- The final sections summarize bundle ceilings and generated-C boundaries.

Each WORKS or DOES NOT WORK heading begins a self-contained conversion unit. The EXPECTED line is separated from the source marker so results and code are easier to scan.

QUICK RULES FOR COMPARISONS AND CONDITIONS

Supported comparison operators

For compatible numeric or Boolean representations:

```
==  !=  <  <=  >  >=
```

Supported condition values

An if, elif, or while condition may use:

- bool;
- int, where zero is false and nonzero is true; or
- float, where +0.0 and -0.0 are false and every other represented value is true.

General string, container, record, object, and dynamic truthiness are not supported.

Important comparison boundaries

- int is compared with int.
- float is compared with float.
- bool is compared with bool.
- int-to-float coercion is not performed.
- General str comparisons are not supported.
- "is", "is not", "in", and "not in" are not supported comparisons.
- Fixed lists, tuples, dictionaries, and records cannot be compared.
- Chained comparisons such as `low <= value < high` are supported.
- Chained operands are evaluated left to right and only once.
- and / or require Boolean-represented operands and preserve Python short-circuit behavior.
- Ordinary new local variables must be declared before entering if / elif / else or loop control flow. Assigning an already-declared compatible local inside those blocks is supported. A range or fixed-container loop target is the documented special loop-local binding.
- Every reachable function path must explicitly return the declared type.

PART I - COMPARISONS, IF, ELIF, AND ELSE

WORKS

W01 Integer comparisons in if / elif / else

This uses <, ==, and <=. The final else covers all remaining values.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W01 -----

```
def classify(value: int) -> int:
    # Integer comparisons can directly control if and elif.
    if value < 0:
        return -1
    elif value == 0:
        return 0
    elif value <= 10:
        return 1
    else:
        return 2
```

----- PYTHON END: W01 -----

WORKS

W02 Floating comparisons and multiple return paths

The operands in every comparison are all float values.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W02 -----

```
def clamp(value: float, low: float, high: float) -> float:
    if value < low:
        return low
    elif value > high:
        return high
    else:
        return value
```

----- PYTHON END: W02 -----

WORKS

W03

Chained comparison

PyCForge preserves Python's meaning:

```
low <= value < high
```

is logically comparable to:

```
low <= value and value < high
```

but the middle operand is evaluated once and reused.

EXPECTED: CONVERTED**NO DIAGNOSTICS**

```
----- PYTHON BEGIN: W03 -----
```

```
def in_window(value: int, low: int, high: int) -> bool:
    return low <= value < high
```

```
----- PYTHON END: W03 -----
```

WORKS

W04

Chained comparison containing a function call

The call to `step(middle)` is evaluated at most once. A later chained operand runs only if the earlier comparison succeeds.

EXPECTED: CONVERTED**NO DIAGNOSTICS**

```
----- PYTHON BEGIN: W04 -----
```

```
def step(value: int) -> int:
    return value + 1
```

```
def ordered(left: int, middle: int, right: int) -> bool:
    return left < step(middle) < right
```

```
----- PYTHON END: W04 -----
```

WORKS

W05

Boolean short-circuit expressions

Both operands of and / or have Boolean representations. Later operands are evaluated only when Python would evaluate them.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W05 -----

```
def choose(first: int, second: int, use_first: bool) -> int:
    if use_first and first >= 0:
        return first
    elif (not use_first) or second != 0:
        return second
    else:
        return 0
```

----- PYTHON END: W05 -----

WORKS

W06

Calls inside and / or retain short-circuiting

positive(second) is skipped when positive(first) already determines the result.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W06 -----

```
def positive(value: int) -> bool:
    return value > 0

def both_positive(first: int, second: int) -> bool:
    return positive(first) and positive(second)

def either_positive(first: int, second: int) -> bool:
    return positive(first) or positive(second)
```

----- PYTHON END: W06 -----

WORKS

W07 Numeric truthiness

An int condition is supported. Zero is false; nonzero is true.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W07 -----

```
def nonzero_or_one(value: int) -> int:
    if value:
        return value
    else:
        return 1
```

----- PYTHON END: W07 -----

WORKS

W08 Declare a local before assigning it in branches

This is the supported pattern. result has a stable int representation before control flow begins.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W08 -----

```
def choose_value(first: int, second: int, use_first: bool) -> int:
    result = 0
    if use_first:
        result = first
    else:
        result = second
    return result
```

----- PYTHON END: W08 -----

DOES NOT WORK

N01 First declaration inside branches

Even though both branches assign result, its first definition occurs inside control flow. PyCForge does not silently turn Python function scope into a narrow C block scope.

EXPECTED: REJECTED

PYC2870

----- PYTHON BEGIN: N01 -----

```
def choose_value(first: int, second: int, use_first: bool) -> int:
    if use_first:
        result = first
    else:
        result = second
    return result
```

----- PYTHON END: N01 -----

DOES NOT WORK

N02 General string comparison

str may cross supported function boundaries as borrowed immutable text, but general Python string comparison is not in the current comparison subset.

EXPECTED: REJECTED

PYC2860

----- PYTHON BEGIN: N02 -----

```
def same_text(left: str, right: str) -> int:
    if left == right:
        return 1
    return 0
```

----- PYTHON END: N02 -----

DOES NOT WORK

N03 Mixed int and float comparison

PyCForge does not insert an implicit int-to-float conversion. Comparison operands must have one exact compatible representation.

EXPECTED: REJECTED

PYC2850

----- PYTHON BEGIN: N03 -----

```
def mixed_less(left: int, right: float) -> int:
    if left < right:
        return 1
    return 0
```

----- PYTHON END: N03 -----

DOES NOT WORK

N04 Identity comparison

"is" and "is not" are outside the supported comparison operators.

EXPECTED: REJECTED

PYC2828

----- PYTHON BEGIN: N04 -----

```
def same_identity(left: int, right: int) -> int:
    if left is right:
        return 1
    return 0
```

----- PYTHON END: N04 -----

DOES NOT WORK

N05 Membership comparison

"in" and "not in" are not supported. A fixed container may be iterated, but it is not a general comparison operand.

EXPECTED: REJECTED

PYC3403

----- PYTHON BEGIN: N05 -----

```
def contains(value: int) -> int:
    values = (1, 2, 3)
    if value in values:
        return 1
    return 0
```

----- PYTHON END: N05 -----

DOES NOT WORK

N06 String truthiness

General dynamic string truthiness is outside the supported condition profile.

EXPECTED: REJECTED

PYC2850

----- PYTHON BEGIN: N06 -----

```
def has_text(value: str) -> int:
    if value:
        return 1
    return 0
```

----- PYTHON END: N06 -----

PART II - POSITIONAL AND KEYWORD PARAMETERS

Function signature rules

- Every admitted ordinary source-function parameter and return has an exact int, float, bool, or str annotation. The special static-record `__init__` form returns None.
- Positional-only parameters are supported with Python's "/" marker.
- Ordinary positional-or-keyword parameters are supported.
- Required keyword-only parameters are supported after Python's "*" marker.
- Required keyword-only means there is no default.
- Positional and explicit keyword actual values are evaluated once, from left to right in source order.
- After validation, PyCForge arranges temporary values into formal parameter order for the C call.
- Every parameter must be supplied exactly once with the exact type.
- Only a directly resolved same-module or explicitly imported source function can be called.
- Defaults, omitted parameters, *args, **kwargs, and call unpacking are not supported.

WORKS

W09

Positional, reordered keyword, and mixed calls

The keyword call may spell arguments in an order different from the function declaration. The mixed call uses one leading positional value and explicit keywords for the remaining parameters.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W09 -----

```
def clamp(value: int, low: int, high: int) -> int:
    if value < low:
        return low
    elif value > high:
        return high
    else:
        return value
```

```
def positional_example(value: int) -> int:
    return clamp(value, 0, 100)
```

```
def keyword_example(value: int) -> int:
    return clamp(high=100, value=value, low=0)
```

```
def mixed_example(value: int) -> int:
    return clamp(value, high=100, low=0)
```

----- PYTHON END: W09 -----

WORKS

W10

Required keyword-only parameters

use_limit and limit are required keyword-only parameters because they appear after "*". They have annotations but no defaults.

The call intentionally supplies limit before use_limit. Actual expressions retain source evaluation order, while the generated C temporary values are arranged into the function's formal order.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W10 -----

```
def choose(value: int, *, use_limit: bool, limit: int) -> int:
    if use_limit and value > limit:
        return limit
    else:
        return value
```

```
def apply_limit(value: int, enabled: bool) -> int:
    return choose(value, limit=10, use_limit=enabled)
```

----- PYTHON END: W10 -----

WORKS

W11

Positional-only, ordinary keyword, and keyword-only

value is positional-only because it appears before "/". offset is positional-or-keyword. enabled is required keyword-only because it appears after "*".

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W11 -----

```
def select(value: int, /, offset: int, *, enabled: bool) -> int:
    if enabled:
        return value + offset
    else:
        return value
```

```
def run(value: int, enabled: bool) -> int:
    return select(value, offset=2, enabled=enabled)
```

----- PYTHON END: W11 -----

DOES NOT WORK

N07 Default parameter

Positional defaults and keyword-only defaults are not yet supported.

EXPECTED: REJECTED

PYC2911

----- PYTHON BEGIN: N07 -----

```
def adjust(value: int, amount: int = 1) -> int:
    return value + amount
```

```
def run(value: int) -> int:
    return adjust(value)
```

----- PYTHON END: N07 -----

DOES NOT WORK

N08 Required keyword-only parameter passed positionally

enabled is declared after "*", so it must be supplied by its exact name.

EXPECTED: REJECTED

PYC2904

----- PYTHON BEGIN: N08 -----

```
def select(value: int, *, enabled: bool) -> int:
    if enabled:
        return value
    return 0
```

```
def run(value: int, enabled: bool) -> int:
    return select(value, enabled)
```

----- PYTHON END: N08 -----

DOES NOT WORK

N09 Positional-only parameter passed by keyword

value and factor appear before "/", so neither may be bound by name.

EXPECTED: REJECTED

PYC2912

----- PYTHON BEGIN: N09 -----

```
def scale(value: int, factor: int, /) -> int:
    return value * factor
```

```
def run(value: int) -> int:
    return scale(value=value, factor=2)
```

----- PYTHON END: N09 -----

DOES NOT WORK

N10**Starred argument unpacking**

The source is valid Python when value contains exactly two characters, but PyCForge does not model runtime argument unpacking.

EXPECTED: REJECTED**PYC2910****----- PYTHON BEGIN: N10 -----**

```
def choose(first: str, second: str) -> str:
    return first
```

```
def run(value: str) -> str:
    return choose(*value)
```

----- PYTHON END: N10 -----

DOES NOT WORK

N11**Unknown keyword name**

Every keyword must name one exact unbound parameter.

EXPECTED: REJECTED**PYC2912****----- PYTHON BEGIN: N11 -----**

```
def combine(first: int, second: int) -> int:
    return first + second
```

```
def run(first: int, second: int) -> int:
    return combine(first=first, missing=second)
```

----- PYTHON END: N11 -----

DOES NOT WORK

N12 Keyword argument to range

The recognized range loop form accepts one to three positional integer arguments. range keywords are not part of the direct source-function keyword call feature.

Python's built-in range also rejects keyword arguments at runtime. PyCForge reports this call shape statically and emits no C.

EXPECTED: REJECTED

PYC2842

----- PYTHON BEGIN: N12 -----

```
def count() -> int:
    total = 0
    for value in range(stop=3):
        total = total + value
    return total
```

----- PYTHON END: N12 -----

PART III - LOOPS, CONTAINERS, RECORDS, AND INTEGER FLOOR ARITHMETIC

WORKS

W12 range loop, continue, break, comparison, and modulo

range is positional. Modulo uses the statically safe direct literal divisor 2.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W12 -----

```
def sum_even(limit: int) -> int:
    total = 0
    for value in range(limit):
        if value % 2 != 0:
            continue
        total = total + value
        if total > 100:
            break
    return total
```

----- PYTHON END: W12 -----

WORKS

W13 while condition with meaningful break and continue

while may use int truthiness. continue skips accumulation when the decremented value is even, so only odd values are added. while ... else is not supported.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W13 -----

```
def sum_odd_below(value: int) -> int:
    total = 0

    while value:
        if value < 0:
            break
        value = value - 1
        if value % 2 == 0:
            continue
        total = total + value

    return total
```

----- PYTHON END: W13 -----

DOES NOT WORK

N13 **while ... else**

Loop else clauses are outside the current structured loop profile.

EXPECTED: REJECTED

PYC2830

----- PYTHON BEGIN: N13 -----

```
def count_down(value: int) -> int:
    while value:
        value = value - 1
    else:
        return 1
    return 0
```

----- PYTHON END: N13 -----

WORKS

W14 **Fixed lists, tuples, dictionaries, and iteration**

Containers are nonempty, homogeneous, function-local, and assigned once. Indices and dictionary keys are compile-time literals. Negative tuple indices are allowed when their normalized index is in bounds.

Each fixed container may contain at most 64 elements. A loop target is a special local binding, but every later loop in the same function must use a new target name.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W14 -----

```
def container_examples() -> int:
    values = [10, 20, 30]
    offsets = (1, 2, 3)
    weights = {1: 4, 2: 5, 3: 6}

    total = values[0] + offsets[-1] + weights[2]

    for value in values:
        total = total + value

    for key in weights:
        total = total + key

    return total
```

----- PYTHON END: W14 -----

DOES NOT WORK

N14 Dynamic list index

List and tuple indices must be compile-time signed integer literals.

EXPECTED: REJECTED

PYC3404

----- PYTHON BEGIN: N14 -----

```
def lookup(index: int) -> int:
    values = [10, 20, 30]
    return values[index]
```

----- PYTHON END: N14 -----

DOES NOT WORK

N15 Container mutation

append, resizing, element assignment, and other mutation are unsupported. The append call reaches the broader container aliasing, escape, rebinding, or scalar-use boundary first.

EXPECTED: REJECTED

PYC3403

----- PYTHON BEGIN: N15 -----

```
def change() -> int:
    values = [1, 2, 3]
    values.append(4)
    return values[0]
```

----- PYTHON END: N15 -----

DOES NOT WORK

N16 List comprehension

Comprehensions create a dynamic container construction path outside the fixed literal profile.

EXPECTED: REJECTED

PYC3406

----- PYTHON BEGIN: N16 -----

```
def make_values() -> int:
    values = [item for item in range(4)]
    return values[0]
```

----- PYTHON END: N16 -----

WORKS

W15 Immutable static record

This narrow class-shaped form maps to a block-scope const C struct value. The record is bound once to a fresh local in the same module and observed through direct read-only field access. Static-record class declarations must precede every top-level function in the same conversion unit.

EXPECTED: CONVERTED**NO DIAGNOSTICS**

----- PYTHON BEGIN: W15 -----

```
class Point:
    x: int
    y: int
    visible: bool

    def __init__(self, x: int, y: int, visible: bool) -> None:
        self.x = x
        self.y = y
        self.visible = visible

def point_score(x: int, y: int, visible: bool) -> int:
    point = Point(x, y, visible)
    if point.visible:
        return point.x + point.y
    else:
        return 0
```

----- PYTHON END: W15 -----

DOES NOT WORK

N17 Record constructor keywords

Static-record construction currently requires exact positional arguments. The source-function keyword-call feature does not widen record construction.

EXPECTED: REJECTED**PYC3605**

----- PYTHON BEGIN: N17 -----

```
class Point:
    x: int
    y: int

    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

def run() -> int:
    point = Point(x=1, y=2)
    return point.x
```

----- PYTHON END: N17 -----

DOES NOT WORK

N18 General record method

The accepted class form is a static record declaration, not the general Python object and method model.

EXPECTED: REJECTED

PYC3604

----- PYTHON BEGIN: N18 -----

```
class Counter:
    value: int

    def __init__(self, value: int) -> None:
        self.value = value

    def increment(self) -> int:
        return self.value + 1

def run(value: int) -> int:
    counter = Counter(value)
    return counter.increment()
```

----- PYTHON END: N18 -----

WORKS

W16 Bounded integer floor division and modulo

The right operand is a direct safe signed integer literal. PyCForge uses frozen helpers to preserve Python floor-division and remainder semantics.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W16 -----

```
def bucket(value: int) -> int:
    quotient = value // 10
    remainder = value % 10
    return quotient + remainder
```

----- PYTHON END: W16 -----

DOES NOT WORK

N19 Dynamic floor-division divisor

The divisor for integer `//` or `%` must be a direct statically safe literal. Zero, -1, `INT64_MIN`, dynamic, calculated, floating, and out-of-range divisors are rejected. The snippet below is the representative dynamic-divisor case.

EXPECTED: REJECTED**PYC3702**

----- PYTHON BEGIN: N19 -----

```
def bucket(value: int, divisor: int) -> int:
    return value // divisor
```

----- PYTHON END: N19 -----

PART IV - OTHER IMPORTANT BOUNDARIES

DOES NOT WORK

N20 Recursion

Direct and mutual recursion are outside the current closed call graph.

EXPECTED: REJECTED

PYC2920

----- PYTHON BEGIN: N20 -----

```
def factorial(value: int) -> int:
    if value <= 1:
        return 1
    return value * factorial(value - 1)
```

----- PYTHON END: N20 -----

DOES NOT WORK

N21 Exception handling

try / except and Python exception behavior are not modeled.

EXPECTED: REJECTED

PYC2812

----- PYTHON BEGIN: N21 -----

```
def safe_divide(value: float, divisor: float) -> float:
    result = 0.0
    try:
        result = value / divisor
    except:
        result = 0.0
    return result
```

----- PYTHON END: N21 -----

Unsupported by default

Anything not explicitly admitted by the converter contract is rejected. Important examples include:

- async and await;
- generators and yield;
- lambdas, nested functions, and closures;
- decorators;
- recursion;
- exceptions;
- pattern matching;
- arbitrary-precision behavior outside the signed-64 contract;
- power and bitwise operations;

- dynamic or floating floor division and modulo;
- runtime string formation, concatenation, indexing, slicing, formatting, and general str methods;
- mutable, nested, aliased, escaping, or over-capacity containers;
- dynamic list, tuple, or dictionary lookup;
- reused loop-target names inside one function;
- general classes, inheritance, properties, and methods;
- static records passed as parameters or returned from functions;
- eval, exec, and reflection;
- plain, relative, star, local, conditional, and dynamic imports;
- third-party package or host-environment discovery; and
- executable module-level statements and general globals.

Outside the converter's scope

Compilation, linking, loading, and execution of generated C are host-toolchain operations rather than rejected Python source features. PyCForge emits and validates C11 source, then stops.

PART V - APPLICATION PATTERNS AND VERIFIED IMPLEMENTATION LIMITS

This part turns the earlier rules into practical conversion patterns. It also records implementation ceilings verified directly against PyCForge 0.15.2.

WORKS

W17

Four-operation calculator with an explicit status channel

The operation is an int code because general string comparison is unsupported:

```
1 = addition
2 = subtraction
3 = multiplication
4 = division
```

The status result distinguishes success, an unsupported operation, and division by zero. The value function is independently safe and returns 0.0 when it cannot produce a result.

EXPECTED: CONVERTED

NO DIAGNOSTICS

```
----- PYTHON BEGIN: W17 -----
```

```
def calculator_status(operation: int, right: float) -> int:
    status = 0

    if operation < 1 or operation > 4:
        status = 1
    elif operation == 4 and right == 0.0:
        status = 2

    return status

def calculator_value(operation: int, left: float, right: float) -> float:
    result = 0.0

    if operation == 1:
        result = left + right
    elif operation == 2:
        result = left - right
    elif operation == 3:
        result = left * right
    elif operation == 4:
        if right != 0.0:
            result = left / right

    return result
```

```
----- PYTHON END: W17 -----
```

WORKS

W18

Split fixed tables with distinct loop targets

A fixed container may contain at most 64 elements. Larger lookup tables can be split into several admitted fixed containers.

Every loop uses a distinct target name. The example is deliberately small so the binding pattern remains easy to read.

The verified NumericText8 translator uses one 64-code tuple for ASCII 32 through 95, then maps ASCII 96 through 126 arithmetically.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W18 -----

```
def letter_code_at(index: int) -> int:
    upper_codes = (65, 66, 67)
    lower_codes = (97, 98, 99)
    position = 0
    result = 0

    for upper_code in upper_codes:
        if position == index:
            result = upper_code
            break
        position = position + 1

    if result == 0:
        for lower_code in lower_codes:
            if position == index:
                result = lower_code
                break
            position = position + 1

    return result
```

----- PYTHON END: W18 -----

WORKS

W19

Class-backed numeric text formation

General runtime string construction is unsupported, but a narrow immutable record can hold fixed numeric character codes. Standalone functions may construct that record and pack its read-only fields into a signed-64 result.

This compact teaching model mirrors the verified NumericText8 engine: printable ASCII uses direct codes 32 through 126, zero means unused storage, and base 128 keeps one character in each numeric digit. The fourth base-128 digit stores this example's length.

A production implementation should validate length, character range, and zero-padding before packing.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W19 -----

```
class NumericText3:
    length: int
    code0: int
    code1: int
    code2: int

    def __init__(
        self,
        length: int,
        code0: int,
        code1: int,
        code2: int,
    ) -> None:
        self.length = length
        self.code0 = code0
        self.code1 = code1
        self.code2 = code2

def form_numeric_text3(
    length: int,
    code0: int,
    code1: int,
    code2: int,
) -> int:
    text = NumericText3(length, code0, code1, code2)
    return (
        text.length * 2097152
        + text.code0
        + text.code1 * 128
        + text.code2 * 16384
    )
```

----- PYTHON END: W19 -----

WORKS

W20

Packed numeric character selection and fixed slicing

Dynamic container indexing is unsupported. A bounded selector can instead use if / elif and direct literal divisors.

For a canonical three-character W19 input, slice_last_two selects positions one and two and forms a new two-character packed value whose length digit is 2.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W20 -----

```
def numeric_code_at(packed: int, index: int) -> int:
    result = 0

    if index == 0:
        result = packed % 128
    elif index == 1:
        result = (packed // 128) % 128
    elif index == 2:
        result = (packed // 16384) % 128

    return result

def slice_last_two(packed: int) -> int:
    first = numeric_code_at(packed, 1)
    second = numeric_code_at(packed, 2)
    return 2 * 2097152 + first + second * 128
```

----- PYTHON END: W20 -----

WORKS

W21

Borrowed string passthrough without string operations

str is admitted as borrowed immutable text at a supported function boundary. The value may pass through unchanged, but the source cannot compare, combine, index, slice, format, or otherwise interpret its contents.

EXPECTED: CONVERTED

NO DIAGNOSTICS

----- PYTHON BEGIN: W21 -----

```
def echo_text(value: str) -> str:
    return value
```

----- PYTHON END: W21 -----

DOES NOT WORK

N22 Fixed container with 65 elements

The current fixed-container capacity is 64. A 65-element literal exceeds that ceiling even when the only access uses a compile-time literal index.

EXPECTED: REJECTED

PYC3401

----- PYTHON BEGIN: N22 -----

```
def first_value() -> int:
    values = (
        0, 1, 2, 3, 4, 5, 6, 7,
        8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23,
        24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39,
        40, 41, 42, 43, 44, 45, 46, 47,
        48, 49, 50, 51, 52, 53, 54, 55,
        56, 57, 58, 59, 60, 61, 62, 63,
        64,
    )
    return values[0]
```

----- PYTHON END: N22 -----

DOES NOT WORK

N23 Reusing a loop target

A range or fixed-container loop target is a special binding. It cannot reuse a parameter, ordinary local, or target from an earlier loop in the same function.

EXPECTED: REJECTED

PYC2944

----- PYTHON BEGIN: N23 -----

```
def sum_two_groups() -> int:
    first = (1, 2)
    second = (3, 4)
    total = 0

    for value in first:
        total = total + value

    for value in second:
        total = total + value

    return total
```

----- PYTHON END: N23 -----

DOES NOT WORK

N24 Static record returned from a function

Static records are local implementation details. They cannot be parameters or return values.

EXPECTED: REJECTED

PYC3606

----- PYTHON BEGIN: N24 -----

```
class Pair:
    left: int
    right: int

    def __init__(self, left: int, right: int) -> None:
        self.left = left
        self.right = right

def make_pair(left: int, right: int) -> Pair:
    pair = Pair(left, right)
    return pair
```

----- PYTHON END: N24 -----

DOES NOT WORK

N25 General string slicing

str may cross a supported scalar boundary, but direct string subscripting and slicing are not part of the admitted operation set.

EXPECTED: REJECTED

PYC2930

----- PYTHON BEGIN: N25 -----

```
def first_character(value: str) -> str:
    return value[0:1]
```

----- PYTHON END: N25 -----

DOES NOT WORK

N26 General string concatenation

String concatenation remains outside the operation-free borrowed-string boundary. Numeric formation such as W19 is the current deterministic replacement pattern.

EXPECTED: REJECTED

PYC2823

----- PYTHON BEGIN: N26 -----

```
def join_text(left: str, right: str) -> str:
    return left + right
```

----- PYTHON END: N26 -----

DOES NOT WORK

N27 Dynamic dictionary lookup

Fixed dictionaries support compile-time literal keys. A runtime key parameter cannot select a value from the dictionary.

For numerical translation, use fixed-container iteration with a separate position counter, explicit comparisons, or arithmetic mapping.

EXPECTED: REJECTED

PYC3404

----- PYTHON BEGIN: N27 -----

```
def translate(code: int) -> int:
    table = {65: 1, 66: 2}
    return table[code]
```

----- PYTHON END: N27 -----

DOES NOT WORK

N28 Static-record field mutation

Static-record fields become immutable after the exact constructor finishes. A later field assignment is not admitted.

EXPECTED: REJECTED

PYC3607

----- PYTHON BEGIN: N28 -----

```
class NumericText2:
    length: int
    code0: int
    code1: int

    def __init__(
        self,
        length: int,
        code0: int,
        code1: int,
    ) -> None:
        self.length = length
        self.code0 = code0
        self.code1 = code1

def change(length: int, code0: int, code1: int) -> int:
    text = NumericText2(length, code0, code1)
    text.code0 = 65
    return text.code0
```

----- PYTHON END: N28 -----

Generated C note for floor arithmetic

Examples using integer `//` or `%` cause PyCForge to emit frozen helpers that preserve Python floor semantics. The helpers are valid ISO C11 and the conversion still completes with no diagnostics.

With GCC `-Wall -Wextra`, the current helper spelling produces a `-Wparentheses` warning around its sign-comparison expression. A build that promotes every warning to an error can temporarily add `-Wno-parentheses`. A future renderer revision can remove the warning by adding explicit parentheses without changing semantics.

CURRENT CONVERSION ENVELOPE

The default aggregate SourceBundle ceilings are:

- 1,000,000 UTF-8 source bytes;
- 100,000 source lines;
- 250,000 tokens;
- 100,000 AST nodes;
- nesting depth 128 per document;
- 64 explicitly supplied source documents; and
- 4,096 normalized import edges.

PyCForge supports only absolute function imports resolved inside that explicit bundle:

```
from exact.module.id import direct_function
```

It does not search the filesystem, environment, installed packages, network, or Python import system. All accepted documents become one deterministic C11 translation unit.

FINAL REMINDER

PyCForge is a Python-to-C source transpiler. A successful conversion means it produced complete, independently validated C11 source under the current contract. It stops there: it does not compile, link, load, or execute the generated C.

PYTHON SOURCE

VALIDATED C11

NO COMPILE / LINK / EXECUTE