PEP 572 -- Assignment Expressions

PEP:	572
Title:	Assignment Expressions
Author:	Chris Angelico <rosuav at gmail.com>, Tim Peters <tim.peters at gmail.com>, Guido van Rossum <guido at python.org>
Status:	Accepted
Type:	Standards Track
Created:	28-Feb-2018
Python-Version:	3.8
Post-History:	28-Feb-2018, 02-Mar-2018, 23-Mar-2018, 04-Apr-2018, 17-Apr-2018, 25-Apr-2018, 09-Jul-2018
Resolution:	https://mail.python.org/pipermail/python-dev/2018-July/154601.html (https://mail.python.org/pipermail/python-dev/2018-July/154601.html)

Contents

- Abstract (#abstract)
- Rationale (#rationale)
  - The importance of real code (#the-importance-of-real-code)
- Syntax and semantics (#syntax-and-semantics)
  - Exceptional cases (#exceptional-cases)
  - Scope of the target (#scope-of-the-target)
  - Relative precedence of `:=` (#relative-precedence-of)
  - Change to evaluation order (#change-to-evaluation-order)
  - Differences between assignment expressions and assignment statements (#differences-between-assignment-expressions-and-assignment-statements)
- Examples (#examples)
  - Examples from the Python standard library (#examples-from-the-python-standard-library)
    - site.py (#site-py)
    - _pydecimal.py (#pydecimal-py)
    - copy.py (#copy-py)
    - datetime.py (#datetime-py)
    - sysconfig.py (#sysconfig-py)
  - Simplifying list comprehensions (#simplifying-list-comprehensions)
  - Capturing condition values (#capturing-condition-values)
  - Fork (#fork)
- Rejected alternative proposals (#rejected-alternative-proposals)
  - Changing the scope rules for comprehensions (#changing-the-scope-rules-for-comprehensions)
  - Alternative spellings (#alternative-spellings)
  - Special-casing conditional statements (#special-casing-conditional-statements)
  - Special-casing comprehensions (#special-casing-comprehensions)
  - Lowering operator precedence (#lowering-operator-precedence)
  - Allowing commas to the right (#allowing-commas-to-the-right)
  - Always requiring parentheses (#always-requiring-parentheses)
- Frequently Raised Objections (#frequently-raised-objections)
  - Why not just turn existing assignment into an expression? (#why-not-just-turn-existing-assignment-into-an-expression)
  - With assignment expressions, why bother with assignment statements? (#with-assignment-expressions-why-bother-with-assignment-statements)
  - Why not use a sublocal scope and prevent namespace pollution? (#why-not-use-a-sublocal-scope-and-prevent-namespace-pollution)
- Style guide recommendations (#style-guide-recommendations)
- Acknowledgements (#acknowledgements)
- Appendix A: Tim Peters's findings (#appendix-a-tim-peters-s-findings)
  - A numeric example (#a-numeric-example)
- Appendix B: Rough code translations for comprehensions (#appendix-b-rough-code-translations-for-comprehensions)
- Appendix C: No Changes to Scope Semantics (#appendix-c-no-changes-to-scope-semantics)
- References (#references)
- Copyright (#copyright)

Abstract (#id4)

This is a proposal for creating a way to assign to variables within an expression using the notation `NAME := expr`.

Rationale (#id5)

Naming the result of an expression is an important part of programming, allowing a descriptive name to be used in place of a longer expression, and permitting reuse. Currently, this feature is available only in statement form, making it unavailable in list comprehensions and other expression contexts.

Additionally, naming sub-parts of a large expression can assist an interactive debugger, providing useful display hooks and partial results. Without a way to capture sub-expressions inline, this would require refactoring of the original code; with assignment expressions, this merely requires the insertion of a few `name :=` markers. Removing the need to refactor reduces the likelihood that the code be inadvertently changed as part of debugging (a common cause of Heisenbugs), and is easier to dictate to another programmer.

The importance of real code (#id6)

During the development of this PEP many people (supporters and critics both) have had a tendency to focus on toy examples on the one hand, and on overly complex examples on the other.

The danger of toy examples is twofold: they are often too abstract to make anyone go "ooh, that's compelling", and they are easily refuted with "I would never write it that way anyway".

The danger of overly complex examples is that they provide a convenient strawman for critics of the proposal to shoot down ("that's obfuscated").

Yet there is some use for both extremely simple and extremely complex examples: they are helpful to clarify the intended semantics. Therefore there will be some of each below.

However, in order to be *compelling*, examples should be rooted in real code, i.e. code that was written without any thought of this PEP, as part of a useful application, however large or small. Tim Peters has been extremely helpful by going over his own personal code repository and picking examples of code he had written that (in his view) would have been *clearer* if rewritten with (sparing) use of assignment expressions. His conclusion: the current proposal would have allowed a modest but clear improvement in quite a few bits of code.

Another use of real code is to observe indirectly how much value programmers place on compactness. Guido van Rossum searched through a Dropbox code base and discovered some evidence that programmers value writing fewer lines over shorter lines.

Case in point: Guido found several examples where a programmer repeated a subexpression, slowing down the program, in order to save one line of code, e.g. instead of writing:

```
match = re.match(data)
group = match.group(1) if match else None
```

they would write:

```
group = re.match(data).group(1) if re.match(data) else None
```

Another example illustrates that programmers sometimes do more work to save an extra level of indentation:

```
match1 = pattern1.match(data)
match2 = pattern2.match(data)
if match1:
    result = match1.group(1)
elif match2:
    result = match2.group(2)
else:
    result = None
```

This code tries to match `pattern2` even if `pattern1` has a match (in which case the match on `pattern2` is never used). The more efficient rewrite would have been:

```
match1 = pattern1.match(data)
if match1:
    result = match1.group(1)
else:
    match2 = pattern2.match(data)
    if match2:
        result = match2.group(2)
    else:
        result = None
```

Syntax and semantics (#id7)

In most contexts where arbitrary Python expressions can be used, a **named expression** can appear. This is of the form `NAME := expr` where `expr` is any valid Python expression other than an unparenthesized tuple, and `NAME` is an identifier.

The value of such a named expression is the same as the incorporated expression, with the additional side-effect that the target is assigned that value:

```
# Handle a matched regex
if (match := pattern.search(data)) is not None:
    # Do something with match

# A loop that can't be trivially rewritten using 2-arg iter()
while chunk := file.read(8192):
   process(chunk)

# Reuse a value that's expensive to compute
[y := f(x), y**2, y**3]

# Share a subexpression between a comprehension filter clause and its output
filtered_data = [y for x in data if (y := f(x)) is not None]
```

Exceptional cases (#id8)

There are a few places where assignment expressions are not allowed, in order to avoid ambiguities or user confusion:

- Unparenthesized assignment expressions are prohibited at the top level of an expression statement. Example:

```
y := f(x)  # INVALID
(y := f(x))  # Valid, though not recommended
```

  This rule is included to simplify the choice for the user between an assignment statement and an assignment expression -- there is no syntactic position where both are valid.

- Unparenthesized assignment expressions are prohibited at the top level of the right hand side of an assignment statement. Example:

```
y0 = y1 := f(x)  # INVALID
y0 = (y1 := f(x))  # Valid, though discouraged
```

  Again, this rule is included to avoid two visually similar ways of saying the same thing.

- Unparenthesized assignment expressions are prohibited for the value of a keyword argument in a call. Example:

```
foo(x = y := f(x))  # INVALID
foo(x=(y := f(x)))  # Valid, though probably confusing
```

  This rule is included to disallow excessively confusing code, and because parsing keyword arguments is complex enough already.

- Unparenthesized assignment expressions are prohibited at the top level of a function default value. Example:

```
def foo(answer = p := 42):  # INVALID
    ...
def foo(answer=(p := 42)):  # Valid, though not great style
    ...
```

  This rule is included to discourage side effects in a position whose exact semantics are already confusing to many users (cf. the common style recommendation against mutable default values), and also to echo the similar prohibition in calls (the previous bullet).

- Unparenthesized assignment expressions are prohibited as annotations for arguments, return values and assignments. Example:

```
def foo(answer: p := 42 = 5):  # INVALID
    ...
def foo(answer: (p := 42) = 5):  # Valid, but probably never useful
    ...
```

  The reasoning here is similar to the two previous cases; this ungrouped assortment of symbols and operators composed of `:` and `=` is hard to read correctly.

- Unparenthesized assignment expressions are prohibited in lambda functions. Example:

```
(lambda: x := 1) # INVALID
lambda: (x := 1) # Valid, but unlikely to be useful
(x := lambda: 1) # Valid
lambda line: (m := re.match(pattern, line)) and m.group(1) # Valid
```

  This allows `lambda` to always bind less tightly than `:=`; having a name binding at the top level inside a lambda function is unlikely to be of value, as there is no way to make use of it. In cases where the name will be used more than once, the expression is likely to need parenthesizing anyway, so this prohibition will rarely affect code.

Scope of the target (#id9)

An assignment expression does not introduce a new scope. In most cases the scope in which the target will be bound is self-explanatory: it is the current scope. If this scope contains a `nonlocal` or `global` declaration for the target, the assignment expression honors that. A lambda (being an explicit, if anonymous, function definition) counts as a scope for this purpose.

There is one special case: an assignment expression occurring in a list, set or dict comprehension or in a generator expression (below collectively referred to as "comprehensions") binds the target in the containing scope, honoring a `nonlocal` or `global` declaration for the target in that scope, if one exists. For the purpose of this rule the containing scope of a nested comprehension is the scope that contains the outermost comprehension. A lambda counts as a containing scope.

The motivation for this special case is twofold. First, it allows us to conveniently capture a "witness" for an `any()` expression, or a counterexample for `all()`, for example:

```
if any((comment := line).startswith('#') for line in lines):
    print("First comment:", comment)
else:
    print("There are no comments")

if all((nonblank := line).strip() == '' for line in lines):
    print("All lines are blank")
else:
    print("First non-blank line:", nonblank)
```

Second, it allows a compact way of updating mutable state from a comprehension, for example:

```
# Compute partial sums in a list comprehension
total = 0
partial_sums = [total := total + v for v in values]
print("Total:", total)
```

However, an assignment expression target name cannot be the same as a `for`-target name appearing in any comprehension containing the assignment expression. The latter names are local to the comprehension in which they appear, so it would be contradictory for a contained use of the same name to refer to the scope containing the outermost comprehension instead.

For example, `[i := i+1 for i in range(5)]` is invalid: the `for i` part establishes that `i` is local to the comprehension, but the `i :=` part insists that `i` is not local to the comprehension. The same reason makes these examples invalid too:

```
[[(j := j) for i in range(5)] for j in range(5)]
[i := 0 for i, j in stuff]
[i+1 for i in i := stuff]
```

A further exception applies when an assignment expression occurs in a comprehension whose containing scope is a class scope. If the rules above were to result in the target being assigned in that class's scope, the assignment expression is expressly invalid.

(The reason for the latter exception is the implicit function created for comprehensions -- there is currently no runtime mechanism for a function to refer to a variable in the containing class scope, and we do not want to add such a mechanism. If this issue ever gets resolved this special case may be removed from the specification of assignment expressions. Note that the problem already exists for *using* a variable defined in the class scope from a comprehension.)

See Appendix B for some examples of how the rules for targets in comprehensions translate to equivalent code.

The two invalid cases listed above raise `TargetScopeError`, a new subclass of `SyntaxError` (with the same signature).

Relative precedence of `:=` (#id10)

The `:=` operator groups more tightly than a comma in all syntactic positions where it is legal, but less tightly than all other operators, including `or`, `and`, `not`, and conditional expressions (`A if C else B`). As follows from section "Exceptional cases" above, it is never allowed at the same level as `=`. In case a different grouping is desired, parentheses should be used.

The `:=` operator may be used directly in a positional function call argument; however it is invalid directly in a keyword argument.

Some examples to clarify what's technically valid or invalid:

```
# INVALID
x := 0

# Valid alternative
(x := 0)

# INVALID
x = y := 0

# Valid alternative
x = (y := 0)

# Valid
len(lines := f.readlines())

# Valid
foo(x := 3, cat='vector')

# INVALID
foo(cat=category := 'vector')

# Valid alternative
foo(cat=(category := 'vector'))
```

Most of the "valid" examples above are not recommended, since human readers of Python source code who are quickly glancing at some code may miss the distinction. But simple cases are not objectionable:

```
# Valid
if any(len(longline := line) >= 100 for line in lines):
    print("Extremely long line:", longline)
```

This PEP recommends always putting spaces around `:=`, similar to PEP 8 (/dev/peps/pep-0008)'s recommendation for `=` when used for assignment, whereas the latter disallows spaces around `=` used for keyword arguments.)

Change to evaluation order (#id11)

In order to have precisely defined semantics, the proposal requires evaluation order to be well-defined. This is technically not a new requirement, as function calls may already have side effects. Python already has a rule that subexpressions are generally evaluated from left to right. However, assignment expressions make these side effects more visible, and we propose a single change to the current evaluation order:

- In a dict comprehension `{X: Y for ...}`, Y is currently evaluated before X. We propose to change this so that X is evaluated before Y. (In a dict display like `{X: Y}` this is already the case, and also in `dict((X, Y) for ...)` which should clearly be equivalent to the dict comprehension.)

Differences between assignment expressions and assignment statements (#id12)

Most importantly, since `:=` is an expression, it can be used in contexts where statements are illegal, including lambda functions and comprehensions.

Conversely, assignment expressions don't support the advanced features found in assignment statements:

- Multiple targets are not directly supported:

```
x = y = z = 0  # Equivalent: (z := (y := (x := 0)))
```

- Single assignment targets other than a single `NAME` are not supported:

```
# No equivalent
a[i] = x
self.rest = []
```

- Priority around commas is different:

```
x = 1, 2  # Sets x to (1, 2)
(x := 1, 2)  # Sets x to 1
```

- Iterable packing and unpacking (both regular or extended forms) are not supported:

```
# Equivalent needs extra parentheses
loc = x, y  # Use (loc := (x, y))
info = name, phone, *rest  # Use (info := (name, phone, *rest))

# No equivalent
px, py, pz = position
name, phone, email, *other_info = contact
```

- Inline type annotations are not supported:

```
# Closest equivalent is "p: Optional[int]" as a separate declaration
p: Optional[int] = None
```

- Augmented assignment is not supported:

```
total += tax  # Equivalent: (total := total + tax)
```

Examples (#id13)

Examples from the Python standard library (#id14)

site.py (#id15)

env_base is only used on these lines, putting its assignment on the if moves it as the "header" of the block.

- Current:

```
env_base = os.environ.get("PYTHONUSERBASE", None)
if env_base:
    return env_base
```

- Improved:

```
if env_base := os.environ.get("PYTHONUSERBASE", None):
    return env_base
```

_pydecimal.py (#id16)

Avoid nested `if` and remove one indentation level.

- Current:

```
if self._is_special:
    ans = self._check_nans(context=context)
    if ans:
        return ans
```

- Improved:

```
if self._is_special and (ans := self._check_nans(context=context)):
    return ans
```

copy.py (#id17)

Code looks more regular and avoid multiple nested if. (See Appendix A for the origin of this example.)

- Current:

```
reductor = dispatch_table.get(cls)
if reductor:
    rv = reductor(x)
else:
    reductor = getattr(x, "__reduce_ex__", None)
    if reductor:
        rv = reductor(4)
    else:
        reductor = getattr(x, "__reduce__", None)
        if reductor:
            rv = reductor()
        else:
            raise Error(
                "un(deep)copyable object of type %s" % cls)
```

- Improved:

```
if reductor := dispatch_table.get(cls):
    rv = reductor(x)
elif reductor := getattr(x, "__reduce_ex__", None):
    rv = reductor(4)
elif reductor := getattr(x, "__reduce__", None):
    rv = reductor()
else:
    raise Error("un(deep)copyable object of type %s" % cls)
```

datetime.py (#id18)

tz is only used for `s += tz`, moving its assignment inside the if helps to show its scope.

- Current:

```
s = _format_time(self._hour, self._minute,
                 self._second, self._microsecond,
                 timespec)
tz = self._tzstr()
if tz:
    s += tz
return s
```

- Improved:

```
s = _format_time(self._hour, self._minute,
                 self._second, self._microsecond,
                 timespec)
if tz := self._tzstr():
    s += tz
return s
```

sysconfig.py (#id19)

Calling `fp.readline()` in the `while` condition and calling `.match()` on the if lines make the code more compact without making it harder to understand.

- Current:

```
while True:
    line = fp.readline()
    if not line:
        break
    m = define_rx.match(line)
    if m:
        n, v = m.group(1, 2)
        try:
            v = int(v)
        except ValueError:
            pass
        vars[n] = v
    else:
        m = undef_rx.match(line)
        if m:
            vars[m.group(1)] = 0
```

- Improved:

```
while line := fp.readline():
    if m := define_rx.match(line):
        n, v = m.group(1, 2)
        try:
            v = int(v)
        except ValueError:
            pass
        vars[n] = v
    elif m := undef_rx.match(line):
        vars[m.group(1)] = 0
```

Simplifying list comprehensions (#id20)

A list comprehension can map and filter efficiently by capturing the condition:

```
results = [(x, y, x/y) for x in input_data if (y := f(x)) > 0]
```

Similarly, a subexpression can be reused within the main expression, by giving it a name on first use:

```
stuff = [[y := f(x), x/y] for x in range(5)]
```

Note that in both cases the variable `y` is bound in the containing scope (i.e. at the same level as `results` or `stuff`).

Capturing condition values (#id21)

Assignment expressions can be used to good effect in the header of an `if` or `while` statement:

```
# Loop-and-a-half
while (command := input("> ")) != "quit":
    print("You entered:", command)

# Capturing regular expression match objects
# See, for instance, Lib/pydoc.py, which uses a multiline spelling
# of this effect
if match := re.search(pat, text):
    print("Found:", match.group(0))
# The same syntax chains nicely into 'elif' statements, unlike the
# equivalent using assignment statements.
elif match := re.search(otherpat, text):
    print("Alternate found:", match.group(0))
elif match := re.search(third, text):
    print("Fallback found:", match.group(0))

# Reading socket data until an empty string is returned
while data := sock.recv(8192):
    print("Received data:", data)
```

Particularly with the `while` loop, this can remove the need to have an infinite loop, an assignment, and a condition. It also creates a smooth parallel between a loop which simply uses a function call as its condition, and one which uses that as its condition but also uses the actual value.

Fork (#id22)

An example from the low-level UNIX world:

```
if pid := os.fork():
    # Parent code
else:
    # Child code
```

Rejected alternative proposals (#id23)

Proposals broadly similar to this one have come up frequently on python-ideas. Below are a number of alternative syntaxes, some of them specific to comprehensions, which have been rejected in favour of the one given above.

Changing the scope rules for comprehensions (#id24)

A previous version of this PEP proposed subtle changes to the scope rules for comprehensions, to make them more usable in class scope and to unify the scope of the "outermost iterable" and the rest of the comprehension. However, this part of the proposal would have caused backwards incompatibilities, and has been withdrawn so the PEP can focus on assignment expressions.

Alternative spellings (#id25)

Broadly the same semantics as the current proposal, but spelled differently.

1. `EXPR as NAME`:

   ```
   stuff = [[f(x) as y, x/y] for x in range(5)]
   ```

   Since `EXPR as NAME` already has meaning in `import`, `except` and `with` statements (with different semantics), this would create unnecessary confusion or require special-casing (e.g. to forbid assignment within the headers of these statements).

   (Note that `with EXPR as VAR` does *not* simply assign the value of `EXPR` to `VAR` -- it calls `EXPR.__enter__()` and assigns the result of *that* to `VAR`.)

   Additional reasons to prefer `:=` over this spelling include:

   - In `if f(x) as y` the assignment target doesn't jump out at you -- it just reads like `if f x blah blah` and it is too similar visually to `if f(x) and y`.
   - In all other situations where an `as` clause is allowed, even readers with intermediary skills are led to anticipate that clause (however optional) by the keyword that starts the line, and the grammar ties that keyword closely to the as clause:
     - `import foo as bar`
     - `except Exc as var`
     - `with ctxmgr() as var`

     To the contrary, the assignment expression does not belong to the `if` or `while` that starts the line, and we intentionally allow assignment expressions in other contexts as well.
   - The parallel cadence between
     - `NAME = EXPR`
     - `if NAME := EXPR`

     reinforces the visual recognition of assignment expressions.

2. `EXPR -> NAME`:

   ```
   stuff = [[f(x) -> y, x/y] for x in range(5)]
   ```

   This syntax is inspired by languages such as R and Haskell, and some programmable calculators. (Note that a left-facing arrow `y <- f(x)` is not possible in Python, as it would be interpreted as less-than and unary minus.) This syntax has a slight advantage over 'as' in that it does not conflict with `with`, `except` and `import`, but otherwise is equivalent. But it is entirely unrelated to Python's other use of `->` (function return type annotations), and compared to `:=` (which dates back to Algol-58) it has a much weaker tradition.

3. Adorning statement-local names with a leading dot:

   ```
   stuff = [[(f(x) as .y), x/.y] for x in range(5)] # with "as"
   stuff = [[(.y := f(x)), x/.y] for x in range(5)] # with ":="
   ```

   This has the advantage that leaked usage can be readily detected, removing some forms of syntactic ambiguity. However, this would be the only place in Python where a variable's scope is encoded into its name, making refactoring harder.

4. Adding a `where:` to any statement to create local name bindings:

   ```
   value = x**2 + 2*x where:
       x = spam(1, 4, 7, q)
   ```

   Execution order is inverted (the indented body is performed first, followed by the "header"). This requires a new keyword, unless an existing keyword is repurposed (most likely `with:`). See PEP 3150 (/dev/peps/pep-3150) for prior discussion on this subject (with the proposed keyword being `given:`).

5. `TARGET from EXPR`:

   ```
   stuff = [[y from f(x), x/y] for x in range(5)]
   ```

   This syntax has fewer conflicts than `as` does (conflicting only with the `raise Exc from Exc` notation), but is otherwise comparable to it. Instead of paralleling `with expr as target:` (which can be useful but can also be confusing), this has no parallels, but is evocative.

Special-casing conditional statements (#id26)

One of the most popular use-cases is `if` and `while` statements. Instead of a more general solution, this proposal enhances the syntax of these two statements to add a means of capturing the compared value:

```
if re.search(pat, text) as match:
    print("Found:", match.group(0))
```

This works beautifully if and ONLY if the desired condition is based on the truthiness of the captured value. It is thus effective for specific use-cases (regex matches, socket reads that return '' when done), and completely useless in more complicated cases (e.g. where the condition is `f(x) < 0` and you want to capture the value of `f(x)`). It also has no benefit to list comprehensions.

Advantages: No syntactic ambiguities. Disadvantages: Answers only a fraction of possible use-cases, even in `if`/`while` statements.

Special-casing comprehensions (#id27)

Another common use-case is comprehensions (list/set/dict, and genexps). As above, proposals have been made for comprehension-specific solutions.

1. `where`, `let`, or `given`:

   ```
   stuff = [(y, x/y) where y = f(x) for x in range(5)]
   stuff = [(y, x/y) let y = f(x) for x in range(5)]
   stuff = [(y, x/y) given y = f(x) for x in range(5)]
   ```

   This brings the subexpression to a location in between the 'for' loop and the expression. It introduces an additional language keyword, which creates conflicts. Of the three, `where` reads the most cleanly, but also has the greatest potential for conflict (e.g. SQLAlchemy and numpy have `where` methods, as does `tkinter.dnd.Icon` in the standard library).

2. `with NAME = EXPR`:

   ```
   stuff = [(y, x/y) with y = f(x) for x in range(5)]
   ```

   As above, but reusing the `with` keyword. Doesn't read too badly, and needs no additional language keyword. Is restricted to comprehensions, though, and cannot as easily be transformed into "longhand" for-loop syntax. Has the C problem that an equals sign in an expression can now create a name binding, rather than performing a comparison. Would raise the question of why "with NAME = EXPR:" cannot be used as a statement on its own.

3. `with EXPR as NAME`:

   ```
   stuff = [(y, x/y) with f(x) as y for x in range(5)]
   ```

   As per option 2, but using `as` rather than an equals sign. Aligns syntactically with other uses of `as` for name binding, but a simple transformation to for-loop longhand would create drastically different semantics; the meaning of `with` inside a comprehension would be completely different from the meaning as a stand-alone statement, while retaining identical syntax.

Regardless of the spelling chosen, this introduces a stark difference between comprehensions and the equivalent unrolled long-hand form of the loop. It is no longer possible to unwrap the loop into statement form without reworking any name bindings. The only keyword that can be repurposed to this task is `with`, thus giving it sneakily different semantics in a comprehension than in a statement; alternatively, a new keyword is needed, with all the costs therein.

Lowering operator precedence (#id28)

There are two logical precedences for the `:=` operator. Either it should bind as loosely as possible, as does statement-assignment; or it should bind more tightly than comparison operators. Placing its precedence between the comparison and arithmetic operators (to be precise: just lower than bitwise OR) allows most uses inside `while` and `if` conditions to be spelled without parentheses, as it is most likely that you wish to capture the value of something, then perform a comparison on it:

```
pos = -1
while pos := buffer.find(search_term, pos + 1) >= 0:
    ...
```

Once find() returns -1, the loop terminates. If `:=` binds as loosely as `=` does, this would capture the result of the comparison (generally either `True` or `False`), which is less useful.

While this behaviour would be convenient in many situations, it is also harder to explain than "the := operator behaves just like the assignment statement", and as such, the precedence for `:=` has been made as close as possible to that of `=` (with the exception that it binds tighter than comma).

Allowing commas to the right (#id29)

Some critics have claimed that the assignment expressions should allow unparenthesized tuples on the right, so that these two would be equivalent:

```
(point := (x, y))
(point := x, y)
```

(With the current version of the proposal, the latter would be equivalent to `((point := x), y)`.)

However, adopting this stance would logically lead to the conclusion that when used in a function call, assignment expressions also bind less tight than comma, so we'd have the following confusing equivalence:

```
foo(x := 1, y)
foo(x := (1, y))
```

The less confusing option is to make `:=` bind more tightly than comma.

Always requiring parentheses (#id30)

It's been proposed to just always require parenthesize around an assignment expression. This would resolve many ambiguities, and indeed parentheses will frequently be needed to extract the desired subexpression. But in the following cases the extra parentheses feel redundant:

```
# Top level in if
if match := pattern.match(line):
    return match.group(1)

# Short call
len(lines := f.readlines())
```

Frequently Raised Objections (#id31)

Why not just turn existing assignment into an expression? (#id32)

C and its derivatives define the **=** operator as an expression, rather than a statement as is Python's way. This allows assignments in more contexts, including contexts where comparisons are more common. The syntactic similarity between `if (x == y)` and `if (x = y)` belies their drastically different semantics. Thus this proposal uses `:=` to clarify the distinction.

With assignment expressions, why bother with assignment statements? (#id33)

The two forms have different flexibilities. The `:=` operator can be used inside a larger expression; the **=** statement can be augmented to **+=** and its friends, can be chained, and can assign to attributes and subscripts.

Why not use a sublocal scope and prevent namespace pollution? (#id34)

Previous revisions of this proposal involved sublocal scope (restricted to a single statement), preventing name leakage and namespace pollution. While a definite advantage in a number of situations, this increases complexity in many others, and the costs are not justified by the benefits. In the interests of language simplicity, the name bindings created here are exactly equivalent to any other name bindings, including that usage at class or module scope will create externally-visible names. This is no different from `for` loops or other constructs, and can be solved the same way: `del` the name once it is no longer needed, or prefix it with an underscore.

(The author wishes to thank Guido van Rossum and Christoph Groth for their suggestions to move the proposal in this direction. [2] (#id3))

Style guide recommendations (#id35)

As expression assignments can sometimes be used equivalently to statement assignments, the question of which should be preferred will arise. For the benefit of style guides such as PEP 8 (/dev/peps/pep-0008), two recommendations are suggested.

1. If either assignment statements or assignment expressions can be used, prefer statements; they are a clear declaration of intent.
2. If using assignment expressions would lead to ambiguity about execution order, restructure it to use statements instead.

Acknowledgements (#id36)

The authors wish to thank Nick Coghlan and Steven D'Aprano for their considerable contributions to this proposal, and members of the core-mentorship mailing list for assistance with implementation.

Appendix A: Tim Peters's findings (#id37)

Here's a brief essay Tim Peters wrote on the topic.

I dislike "busy" lines of code, and also dislike putting conceptually unrelated logic on a single line. So, for example, instead of:

```
i = j = count = nerrors = 0
```

I prefer:

```
i = j = 0
count = 0
nerrors = 0
```

instead. So I suspected I'd find few places I'd want to use assignment expressions. I didn't even consider them for lines already stretching halfway across the screen. In other cases, "unrelated" ruled:

```
mylast = mylast[1]
yield mylast[0]
```

is a vast improvement over the briefer:

```
yield (mylast := mylast[1])[0]
```

The original two statements are doing entirely different conceptual things, and slamming them together is conceptually insane.

In other cases, combining related logic made it harder to understand, such as rewriting:

```
while True:
    old = total
    total += term
    if old == total:
        return total
    term *= mx2 / (i*(i+1))
    i += 2
```

as the briefer:

```
while total != (total := total + term):
    term *= mx2 / (i*(i+1))
    i += 2
return total
```

The `while` test there is too subtle, crucially relying on strict left-to-right evaluation in a non-short-circuiting or method-chaining context. My brain isn't wired that way.

But cases like that were rare. Name binding is very frequent, and "sparse is better than dense" does not mean "almost empty is better than sparse". For example, I have many functions that return `None` or `0` to communicate "I have nothing useful to return in this case, but since that's expected often I'm not going to annoy you with an exception". This is essentially the same as regular expression search functions returning `None` when there is no match. So there was lots of code of the form:

```
result = solution(xs, n)
if result:
    # use result
```

I find that clearer, and certainly a bit less typing and pattern-matching reading, as:

```
if result := solution(xs, n):
    # use result
```

It's also nice to trade away a small amount of horizontal whitespace to get another _line_ of surrounding code on screen. I didn't give much weight to this at first, but it was so very frequent it added up, and I soon enough became annoyed that I couldn't actually run the briefer code. That surprised me!

There are other cases where assignment expressions really shine. Rather than pick another from my code, Kirill Balunov gave a lovely example from the standard library's `copy()` function in `copy.py`:

```
reductor = dispatch_table.get(cls)
if reductor:
    rv = reductor(x)
else:
    reductor = getattr(x, "__reduce_ex__", None)
    if reductor:
        rv = reductor(4)
    else:
        reductor = getattr(x, "__reduce__", None)
        if reductor:
            rv = reductor()
        else:
            raise Error("un(shallow)copyable object of type %s" % cls)
```

The ever-increasing indentation is semantically misleading: the logic is conceptually flat, "the first test that succeeds wins":

```
if reductor := dispatch_table.get(cls):
    rv = reductor(x)
elif reductor := getattr(x, "__reduce_ex__", None):
    rv = reductor(4)
elif reductor := getattr(x, "__reduce__", None):
    rv = reductor()
else:
    raise Error("un(shallow)copyable object of type %s" % cls)
```

Using easy assignment expressions allows the visual structure of the code to emphasize the conceptual flatness of the logic; ever-increasing indentation obscured it.

A smaller example from my code delighted me, both allowing to put inherently related logic in a single line, and allowing to remove an annoying "artificial" indentation level:

```
diff = x - x_base
if diff:
    g = gcd(diff, n)
    if g > 1:
        return g
```

became:

```
if (diff := x - x_base) and (g := gcd(diff, n)) > 1:
    return g
```

That `if` is about as long as I want my lines to get, but remains easy to follow.

So, in all, in most lines binding a name, I wouldn't use assignment expressions, but because that construct is so very frequent, that leaves many places I would. In most of the latter, I found a small win that adds up due to how often it occurs, and in the rest I found a moderate to major win. I'd certainly use it more often than ternary `if`, but significantly less often than augmented assignment.

A numeric example (#id38)

I have another example that quite impressed me at the time.

Where all variables are positive integers, and a is at least as large as the n'th root of x, this algorithm returns the floor of the n'th root of x (and roughly doubling the number of accurate bits per iteration):

```
while a > (d := x // a**(n-1)):
    a = ((n-1)*a + d) // n
return a
```

It's not obvious why that works, but is no more obvious in the "loop and a half" form. It's hard to prove correctness without building on the right insight (the "arithmetic mean - geometric mean inequality"), and knowing some non-trivial things about how nested floor functions behave. That is, the challenges are in the math, not really in the coding.

If you do know all that, then the assignment-expression form is easily read as "while the current guess is too large, get a smaller guess", where the "too large?" test and the new guess share an expensive sub-expression.

To my eyes, the original form is harder to understand:

```
while True:
    d = x // a**(n-1)
    if a <= d:
        break
    a = ((n-1)*a + d) // n
return a
```

Appendix B: Rough code translations for comprehensions (#id39)

This appendix attempts to clarify (though not specify) the rules when a target occurs in a comprehension or in a generator expression. For a number of illustrative examples we show the original code, containing a comprehension, and the translation, where the comprehension has been replaced by an equivalent generator function plus some scaffolding.

Since `[x for ...]` is equivalent to `list(x for ...)` these examples all use list comprehensions without loss of generality. And since these examples are meant to clarify edge cases of the rules, they aren't trying to look like real code.

Note: comprehensions are already implemented via synthesizing nested generator functions like those in this appendix. The new part is adding appropriate declarations to establish the intended scope of assignment expression targets (the same scope they resolve to as if the assignment were performed in the block containing the outermost comprehension). For type inference purposes, these illustrative expansions do not imply that assignment expression targets are always Optional (but they do indicate the target binding scope).

Let's start with a reminder of what code is generated for a generator expression without assignment expression.

- Original code (EXPR usually references VAR):

```
def f():
    a = [EXPR for VAR in ITERABLE]
```

- Translation (let's not worry about name conflicts):

```
def f():
    def genexpr(iterator):
        for VAR in iterator:
            yield EXPR
    a = list(genexpr(iter(ITERABLE)))
```

Let's add a simple assignment expression.

- Original code:

```
def f():
    a = [TARGET := EXPR for VAR in ITERABLE]
```

- Translation:

```
def f():
    if False:
        TARGET = None  # Dead code to ensure TARGET is a local variable
    def genexpr(iterator):
        nonlocal TARGET
        for VAR in iterator:
            TARGET = EXPR
            yield TARGET
    a = list(genexpr(iter(ITERABLE)))
```

Let's add a `global TARGET` declaration in `f()`.

- Original code:

```
def f():
    global TARGET
    a = [TARGET := EXPR for VAR in ITERABLE]
```

- Translation:

```
def f():
    global TARGET
    def genexpr(iterator):
        global TARGET
        for VAR in iterator:
            TARGET = EXPR
            yield TARGET
    a = list(genexpr(iter(ITERABLE)))
```

Or instead let's add a `nonlocal TARGET` declaration in `f()`.

- Original code:

```
def g():
    TARGET = ...
    def f():
        nonlocal TARGET
        a = [TARGET := EXPR for VAR in ITERABLE]
```

- Translation:

```
def g():
    TARGET = ...
    def f():
        nonlocal TARGET
        def genexpr(iterator):
            nonlocal TARGET
            for VAR in iterator:
                TARGET = EXPR
                yield TARGET
        a = list(genexpr(iter(ITERABLE)))
```

Finally, let's nest two comprehensions.

- Original code:

```
def f():
    a = [[TARGET := i for i in range(3)] for j in range(2)]
    # I.e., a = [[0, 1, 2], [0, 1, 2]]
    print(TARGET)  # prints 2
```

- Translation:

```
def f():
    if False:
        TARGET = None
    def outer_genexpr(outer_iterator):
        nonlocal TARGET
        def inner_generator(inner_iterator):
            nonlocal TARGET
            for i in inner_iterator:
                TARGET = i
                yield i
        for j in outer_iterator:
            yield list(inner_generator(range(3)))
    a = list(outer_genexpr(range(2)))
    print(TARGET)
```

Appendix C: No Changes to Scope Semantics (#id40)

Because it has been a point of confusion, note that nothing about Python's scoping semantics is changed. Function-local scopes continue to be resolved at compile time, and to have indefinite temporal extent at run time ("full closures"). Example:

```
a = 42
def f():
    # `a` is local to `f`, but remains unbound
    # until the caller executes this genexp:
    yield ((a := i) for i in range(3))
    yield lambda: a + 100
    print("done")
    try:
        print(f"`a` is bound to {a}")
        assert False
    except UnboundLocalError:
        print("`a` is not yet bound")
```

Then:

```
>>> results = list(f()) # [genexp, lambda]
done
`a` is not yet bound
# The execution frame for f no longer exists in CPython,
# but f's locals live so long as they can still be referenced.
>>> list(map(type, results))
[<class 'generator'>, <class 'function'>]
>>> list(results[0])
[0, 1, 2]
>>> results[1]()
102
>>> a
42
```

References (#id41)

[1] Proof of concept implementation (https://github.com/Rosuav/cpython/tree/assignment-expressions (https://github.com/Rosuav/cpython/tree/assignment-expressions))

[2] (#id1) Pivotal post regarding inline assignment semantics (https://mail.python.org/pipermail/python-ideas/2018-March/049409.html (https://mail.python.org/pipermail/python-ideas/2018-March/049409.html))

Copyright (#id42)

This document has been placed in the public domain.

Source: https://github.com/python/peps/blob/master/pep-0572.rst (https://github.com/python/peps/blob/master/pep-0572.rst)

Tweets by @ThePSF

Python Software
@ThePSF
If you have a few hours a week to spare to support the Python community, consider becoming a PSF Managing/Contributing Member: bit.ly/2yRCajL
15m

Python Software
@ThePSF
Python Sul 2017 lightning talks (PyVideo) - pyvideo.org/python-sul-201…
2h

Python Software
@ThePSF
Write more decorators (and fewer classes) by Antonio Verardi - pyvideo.org/europython-201…. This talk presents a pattern that can be used to design clean and pythonic interfaces for programmers based on replacing single-method classes with decorated functions.
22h

Python Software
@ThePSF
Using Machine Learning to play chess by Aubhro Sengupta - pyvideo.org/pyohio-2017/us…. Today, chess engines contain many lines of code handcrafted under the guidance of grandmasters. Find out how to create an engine that learns to improve itself.

Tweets by @ThePSF

The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Back to Top

Copyright ©2001-2018. Python Software Foundation Legal Statements Privacy Policy Powered by Rackspace