What's New in Python 3.8?

Adam Forsyth

What's not in the talk?

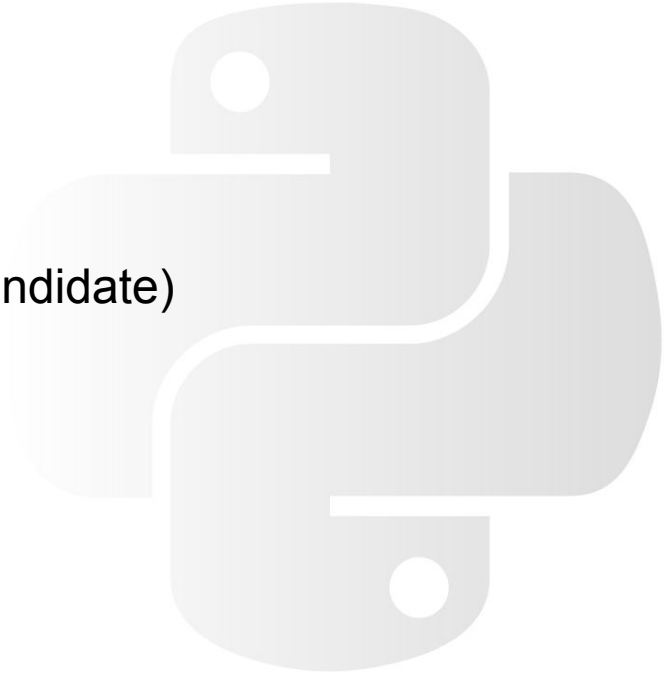
- Every single change
- Subinterpreters (PEP 554)
- Insider Info
- C code

PEP 569 - Release Schedule

- Candidate 1: 2019-10-01

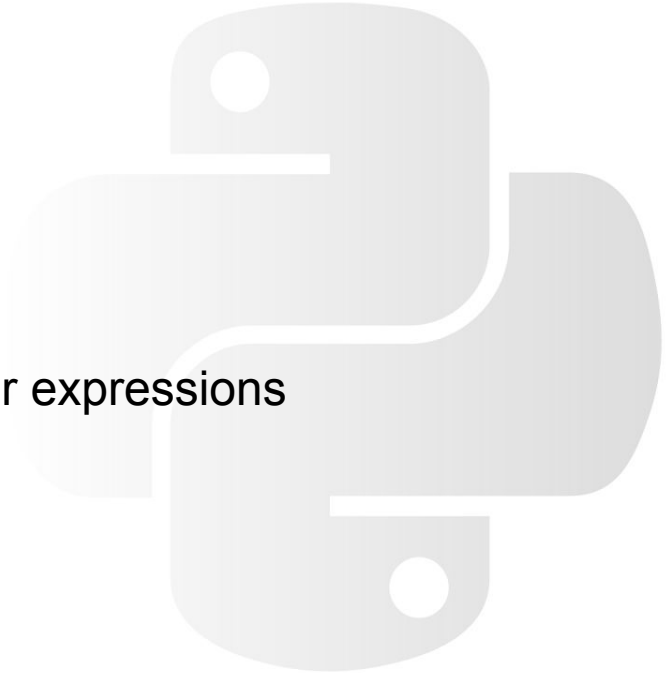
Future:

- Final: 2019-10-14 (assuming a single release candidate)



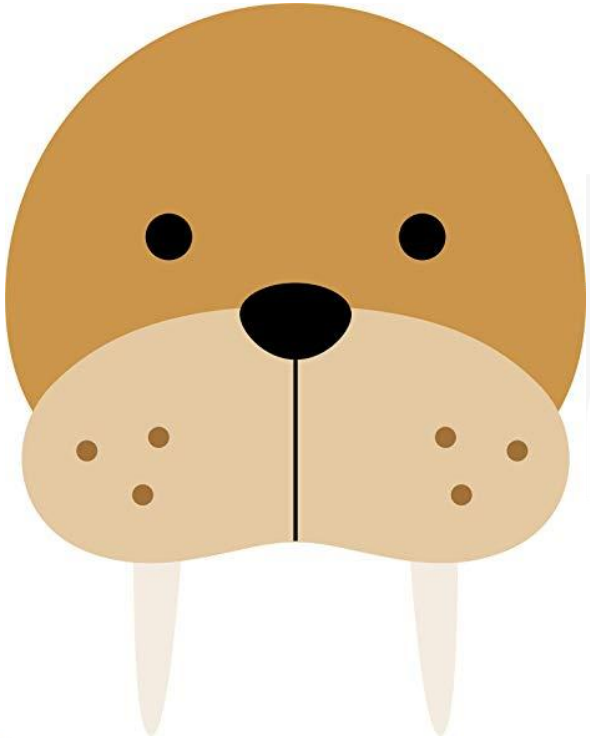
PEP 572 - Assignment Expressions

- Uses the `:=` operator
- Assign to a variable (almost) everywhere
- Eliminates extra lines for assignments
- Some places you can’t use it
- Some things it doesn’t do
- New ways to use comprehensions and generator expressions



PEP 572 - Assignment Expressions

:= is



PEP 572 - Assignment Expressions

Python 3.8:

```
if y:= foo(x):
    print(y)

while (y := foo(x)) < 1000:
    print(y)

[y:= foo(x), y**2, y**3]
```

Python 3.7:

```
>>> if y = foo(x):
  File "<stdin>", line 1
    if y = foo(x):
       ^
SyntaxError: invalid syntax
>>> if y := foo(x):
  File "<stdin>", line 1
    if y := foo(x):
       ^
SyntaxError: invalid syntax
```

PEP 572 - Assignment Expressions

Python 3.7:

```
if foo(x) > 4:
    print(foo(x))
```

OR

```
y = foo(x)
if y > 4:
    print(y)
```

Python 3.8:

```
if (y:= foo(x)) > 4:
    print(y)
```

PEP 572 - Assignment Expressions

```
>>> y := 1
      ^

>>> x = y := 1
          ^

>>> foo(x = y := 1)
               ^

>>> def foo(x = y := 1):
                    ^

>>> def foo(x: y := 1):
                   ^
```

```
>>> lambda: y := 1
                ^


>>> x = 1
>>> f'{x:=10}'
'          1'
```

PEP 572 - Assignment Expressions

Assignment Statements Only:

- Assigning to anything other than a name `y[1] = 2`
- Multiple assignment `x = y = z = 1`
- Iterable Packing `y = 1, 2`
- Iterable Unpacking `x, y, z = something`
- Inline Type Annotations `y: int = 1`
- Augmented assignment `y += 1`

PEP 572 - Assignment Expressions

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


>>> names = "PyBay", "Adam Forsyth", "Monty Python"
>>> {(n := name.lower()): n.split() for name in names}
{'pybay': ['pybay'], 'adam forsyth': ['adam', 'forsyth'],
 'monty python': ['monty', 'python']}
```

PEP 572 - Assignment Expressions

```
if y := re.match("1", x):
    print("1st:", y.match(0))
elif y := re.match("2", x):
    print("2nd:", y.match(0))
elif y := re.match("3", x):
    print("3rd:", y.match(0))
else:
    print("No match")
```

```
y = re.match("1", x)
if y:
    print("1st:", y.match(0))
else:
    y = re.match("2", x):
    if y:
        print("2nd:", y.match(0)
    else:
        y = re.match("3", x)
        if y:
```

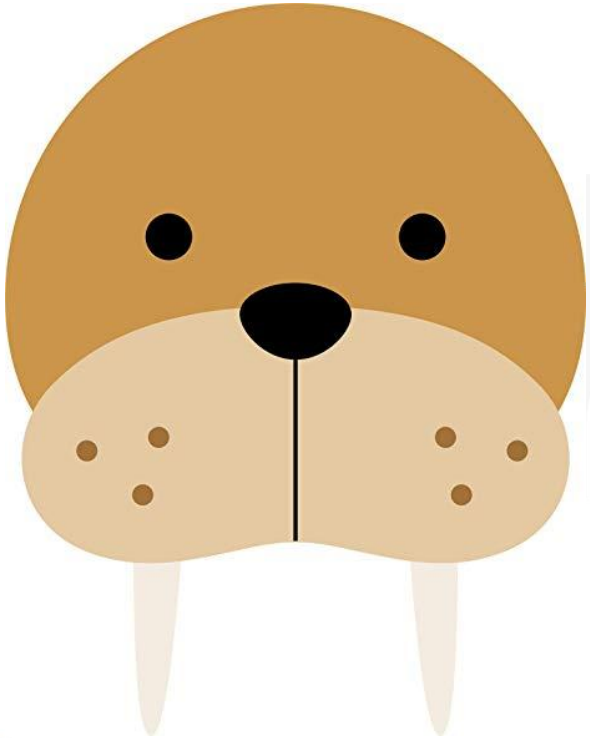
PEP 572 - Assignment Expressions

```
>>> values = 1, 2, 3, 4
>>> total = 0
>>> [total := total + v for v in values]
[1, 3, 6, 10]
```

```
>>> [y for i in range(10) if (y := i % 4)]
[1, 2, 3, 1, 2, 3, 1]
```

PEP 572 - Assignment Expressions

:= is



PEP 570 - Python Positional-Only Parameters

```
>>> def foo(a, b, *, c):
...     return a, b, c
>>> foo(1, 2, c=3) == foo(c=3, b=2, a=1) == foo(1, b=2, c=3)
True

>>> foo(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes 2 positional arguments
            but 3 were given
```

PEP 570 - Python Positional-Only Parameters

```
>>> def foo(a, /, b, *, c):
...     return a, b, c
>>> foo(c=3, b=2, a=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got some positional-only arguments
           passed as keyword arguments: 'a'
```

Debug support for f-strings

```
>>> f"math.pi={math.pi:.2f}"
'math.pi=3.14'
```

```
>>> f"{math.pi=:.2f}"
'math.pi=3.14'
```

PEP 586, 589, and 591 - Additions to typing

```
>>> from typing import Optional, Union
>>> def foo(x: Optional[int], y: Union[str, bytes]) -> int:
...     if x:
...         z: int = x * 2 + 1
...         return x * len(y)
...     else:
...         return 0
```

PEP 586, 589, and 591 - Additions to typing

```
from typing import Literal, Final, TypedDict

class Movie(TypedDict):
    name: str
    year: Literal[1982]

movie: Final[Movie] = {'name': 'Blade Runner', 'year': 1982}
movie = {'name': 1984, 'year': 1984}
```

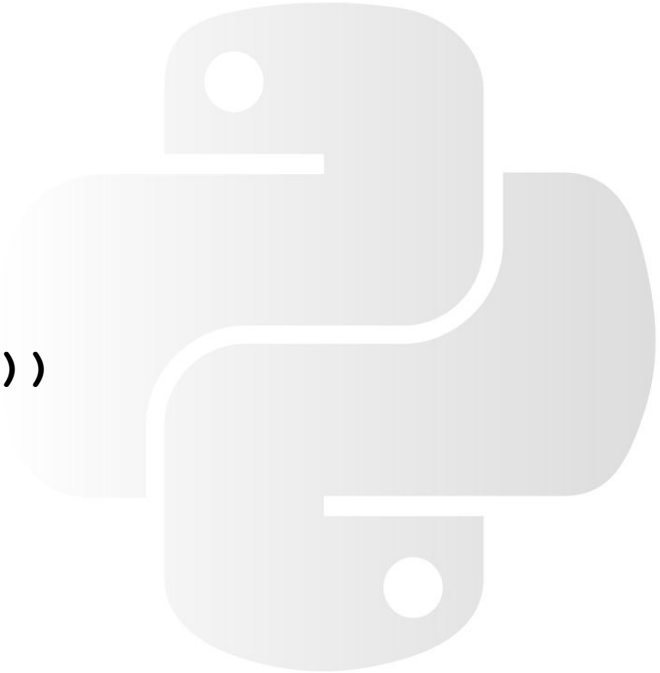
PEP 586, 589, and 591 - Additions to typing

```
$ mypy typing_example.py
typing_example.py:8: error: Cannot assign to final name
"movie"

typing_example.py:8: error: Incompatible types (expression
has type "int", TypedDict item "name" has type "str")

typing_example.py:8: error: Incompatible types (expression
has type "Literal[1984]", TypedDict item "year" has type
"Literal[1982]")
```

dicts and dictviews work with reversed()

```
>>> list(reversed({1: 2, 3: 4}))
[3, 1]
```

```
>>> list(reversed({1: 2, 3: 4}.values()))
[4, 2]
```



Async REPL

```
>>> async def foo(): pass
>>> await foo()
  File "<stdin>", line 1
    await foo()
              ^
SyntaxError: invalid syntax
```

```
>>> async def foo(): pass
>>> await foo()
  File "<stdin>", line 1
SyntaxError: 'await' outside
  function
```

Async REPL

```
$ python3.8 -masyncio
asyncio REPL 3.8.0b3
Use "await" directly instead of "asyncio.run()".
>>> import asyncio

>>> async def foo(x):
...     return x + 1
>>> await foo(2)
3
```

Behind the Scenes

PEP 578 -- Python Runtime Audit Hooks

```
>>> sys.addaudithook(lambda event, args: print(event, args))
compile (None, '<stdin>')

>>> sys.audit("event_name", ["list", "of", "details"])
exec (<code object <module> at 0x7f7b6dcb7030, file
  "<stdin>", line 1>,)
event_name (['list', 'of', 'details'],)
compile (None, '<stdin>')
```

PEP 578 -- Python Runtime Audit Hooks

Copyright

PEP 574 -- Pickle protocol 5 with out-of-band data

```
>>> import pickle
>>> class Foo:
...     attr = 'attribute'
>>> pickled = pickle.dumps(Foo)
>>> pickle.loads(pickled).attr
'attribute'
```

PEP 574 -- Pickle protocol 5 with out-of-band data

```
class bytearray:
   def __reduce_ex__(self, protocol):
      if protocol >= 5:
         return type(self), (PickleBuffer(self),), None

data = pickle.dumps(bytearray([1, 2]), buffer_callback=cb)

pickle.loads(data, buffers=buffers_from_cb)
```

PEP 590 -- Vectorcall for CPython

Calling Conventions

- Flexible = Slow = **`tp_call`**
- Creates intermediate objects (**`tuples`** and **`dicts`**)
- Specific = Fast = **`fastcall`** (but only inside CPython)
- Doesn't create intermediate objects
- **`vectorcall`** = **`fastcall`** for classes and C extensions!
- Enables speed, doesn't provide it
- Does make non-**`fastcall`** calls *slightly* slower

Thank You!

github.com/agfor