What's new in Python 3.8?

The latest version of Python is going to be available soon, and we've compiled a list of new features you need to know about.

Insights Python By Sanket on July 23, 2019

The latest, greatest version of Python is going to be out in beta soon. While there's still some time before the final stable version is available, it is worth looking into all that's new. Python 3.8 adds some new syntax to the language, a few minor changes to existing behavior, and mostly a bunch of speed improvements — maintaining the tradition from the earlier 3.7 release. This post outlines the most significant additions and changes you should know about Python 3.8. Take a look!

1. The walrus operator

Assignment expressions have come to Python with the "walrus" operator `:=`. This will enable you to assign values to a variable as part of an expression. The major benefit of this is it saves you some lines of code when you want to use, say, the value of an expression in a subsequent condition.

So, something like this:

```
line = f.readline()
while line:
    ...  # process line
    line = f.readline()
```

Can now be written in short like this:

```
while line := f.readline():
    ... # process line
```

Yay for brevity, but some might say this affects readability of code — it can be argued that the first variant here is clearer and explicit. This discussion was the center of a major controversy in the Python community.

2. Positional-only arguments

A special marker, `/`, can now be used when defining a method's arguments to specify that the functional only accepts positional arguments on the left of the marker. Keyword-only arguments have been available in Python with the `*` marker in functions, and addition of `/` marker for positional-only arguments improves the language's consistency and allows for a robust API design. Take an example of this function:

```
def pow(x, y, z=None, /):
    r = x ** y
    if z is not None:
        r %= z
    return r
```

The `/` marker here means that passing values for `x`, `y` and `z` can only be done positionally, and not using keyword arguments. The behavior is illustrated below:

```
>>> pow(2, 10)  # valid
>>> pow(2, 10, 17)  # valid
>>> pow(x=2, y=10)  # invalid, will raise a TypeError
>>> pow(2, 10, z=17)  # invalid, will raise a TypeError
```

A more detailed explanation on the motivation and use-cases can be found in [PEP 570](#).

3. f-strings now support “=”

Python programmers often use “printf-style” debugging. In the old days this was pretty verbose:

```
print "foo=", foo, "bar=", bar
```

f-strings make this a bit nicer:

```
print(f"foo={foo} bar={bar}")
```

But you still have to repeat yourself: you have to write out the *string* “foo”, and then the *expession* “foo”.

The `=` specifier, used as `f'{expr=}'` expands to the text of the expression, an equal sign, then the repr of the evaluated expression. So now, you can simply write:

```
print(f"{foo=} {bar=}")
```

A small step for the language, but a giant leap for everyone who sprinkles `print()` statements for debugging!

4. `reversed()` now works with `dict`

Since Python 3.7, dictionaries preserve the order of insertion of keys. The `reversed()` built-in can now be used to access the dictionary in the reverse order of insertion — just like `OrderedDict`.

```
>>> my_dict = dict(a=1, b=2)
>>> list(reversed(my_dict))
['b', 'a']
>>> list(reversed(my_dict.items()))
[('b', 2), ('a', 1)]
```

5. Simplified iterable unpacking for `return` and `yield`

This unintentional behavior has existed since Python 3.2 which disallowed unpacking iterables without parentheses in `return` and `yield` statements. So, the following was allowed:

```
def foo():
    rest = (4, 5, 6)
    t = 1, 2, 3, *rest
    return t
```

But these resulted in a `SyntaxError`:

```
def baz():
    rest = (4, 5, 6)
    return 1, 2, 3, *rest
def baz():
    rest = (4, 5, 6)
    yield 1, 2, 3, *rest
```

The latest release fixes this behavior, so doing the above two approaches are now allowed.

6. New syntax warnings

The Python interpreter now throws a `SyntaxWarning` in some cases when a comma is missed before tuple or list. So when you accidentally do this:

```
data = [
    (1, 2, 3)  # oops, missing comma!
    (4, 5, 6)
]
```

Instead of showing `TypeError: 'tuple' object is not callable` which doesn't really tell you what's wrong, a helpful warning will be shown pointing out that you probably missed a comma. Pretty helpful while debugging! The compiler now also produces a `SyntaxWarning` when identity checks (`is` and `is not`) are used with certain types of literals (e.g. strings, integers, etc.). You rarely want to compare identities with literals other than `None`, and a compiler warning can help avoid a number of elusive bugs.

7. Performance improvements

This release adds a number of performance speed-ups to the interpreter, following suit from the previous 3.7 release.

- `operator.itemgetter()` is now 33% faster. This was made possible by optimizing argument handling and adding a fast path for the common case of a single non-negative integer index into a tuple (which is the typical use case in the standard library).
- Field lookups in `collections.namedtuple()` are now more than two times faster, making them the fastest form of instance variable lookup in Python.
- The `list` constructor does not over-allocate the internal item buffer if the input iterable has a known length (the input implements len). This makes the created list 12% smaller on average.
- Class variable writes are now twice as fast: when a non-dunder attribute was updated, there was an unnecessary call to update slots, which is optimized.
- Invocation of some simple built-ins and methods are now 20-50% faster. The overhead of converting arguments to these methods is reduced.
- `uuid.UUID` now uses slots to reduce it's memory footprint.

Summary

The upcoming release of Python adds some great new features to the language and significantly improves the performance with fundamental speed-up fixes. There is a small number behavior changes that might require modifying existing code while upgrading to Python 3.8, but the performance gains and new syntax make it totally worth the effort. A detailed change log of all that's new can be found here.

Cool New Features in Python 3.8

by Geir Arne Hjelle Oct 14, 2019 6 Comments intermediate python Tweet Share Email

Table of Contents

The newest version of Python is released! Python 3.8 has been available in beta versions since the summer, but on October 14th, 2019 the first official version is ready. Now, we can all start playing with the new features and benefit from the latest improvements.

What does Python 3.8 bring to the table? The documentation gives a good overview of the new features. However, this article will go more in depth on some of the biggest changes, and show you how you can take advantage of Python 3.8.

In this article, you'll learn about:

- Using assignment expressions to simplify some code constructs
- Enforcing positional-only arguments in your own functions
- Specifying more precise type hints
- Using f-strings for simpler debugging

With a few exceptions, Python 3.8 contains many small improvements over the earlier versions. Towards the end of the article, you'll see many of these less attention-grabbing changes, as well as a discussion about some of the optimizations that make Python 3.8 faster than its predecessors. Finally, you'll get some advice about upgrading to the new version.

The Walrus in the Room: Assignment Expressions

The biggest change in Python 3.8 is the introduction of **assignment expressions**. They are written using a new notation (`:=`). This operator is often called the **walrus operator** as it resembles the eyes and tusks of a walrus on its side.

Assignment expressions allow you to assign and return a value in the same expression. For example, if you want to assign to a variable and print its value, then you typically do something like this:

```
>>> walrus = False
>>> print(walrus)
False
```

In Python 3.8, you're allowed to combine these two statements into one, using the walrus operator:

```
>>> print(walrus := True)
True
```

The assignment expression allows you to assign `True` to `walrus`, and immediately print the value. But keep in mind that the walrus operator does *not* do anything that isn't possible without it. It only makes certain constructs more convenient, and can sometimes communicate the intent of your code more clearly.

One pattern that shows some of the strengths of the walrus operator is `while` loops where you need to initialize and update a variable. For example, the following code asks the user for input until they type `quit`:

```
inputs = list()
current = input("Write something: ")
while current != "quit":
    inputs.append(current)
    current = input("Write something: ")
```

This code is less than ideal. You're repeating the `input()` statement, and somehow you need to add `current` to the list *before* asking the user for it. A better solution is to set up an infinite `while` loop, and use `break` to stop the loop:

```
inputs = list()
while True:
    current = input("Write something: ")
    if current == "quit":
        break
    inputs.append(current)
```

This code is equivalent to the one above, but avoids the repetition and somehow keeps the lines in a more logical order. If you use an assignment expression, you can simplify this loop further:

```
inputs = list()
while (current := input("Write something: ")) != "quit":
    inputs.append(current)
```

This moves the test back to the `while` line, where it should be. However, there are now several things happening at that line, so it takes a bit more effort to read it properly. Use your best judgement about when the walrus operator helps make your code more readable.

PEP 572 describes all the details of assignment expressions, including some of the rationale for introducing them into the language, as well as several examples of how the walrus operator can be used.

Positional-Only Arguments

The built-in function `float()` can be used for converting text strings and numbers to `float` objects. Consider the following example:

```
>>> float("3.8")
3.8

>>> help(float)
class float(object)
 |  float(x=0, /)
 |
 |  Convert a string or number to a floating point number, if possible.

[...]
```

Look closely at the signature of `float()`. Notice the slash (`/`) after the parameter. What does it mean?

Note: For an in-depth discussion on the `/` notation, see PEP 457 - Notation for Positional-Only Parameters.

It turns out that while the one parameter of `float()` is called `x`, you're not allowed to use its name:

```
>>> float(x="3.8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float() takes no keyword arguments
```

When using `float()` you're only allowed to specify arguments by position, not by keyword. Before Python 3.8, such **positional-only** arguments were only possible for built-in functions. There was no easy way to specify that arguments should be positional-only in your own functions:

```
>>> def incr(x):
...     return x + 1
...
>>> incr(3.8)
4.8

>>> incr(x=3.8)
4.8
```

It's possible to simulate positional-only arguments using `*args`, but this is less flexible, less readable, and forces you to implement your own argument parsing. In Python 3.8, you can use `/` to denote that all arguments before it must be specified by position. You can rewrite `incr()` to only accept positional arguments:

```
>>> def incr(x, /):
...     return x + 1
...
>>> incr(3.8)
4.8

>>> incr(x=3.8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: incr() got some positional-only arguments passed as
           keyword arguments: 'x'
```

By adding `/` after `x`, you specify that `x` is a positional-only argument. You can combine regular arguments with positional-only ones by placing the regular arguments after the slash:

```
>>> def greet(name, /, greeting="Hello"):
...     return f"{greeting}, {name}"
...
>>> greet("Łukasz")
'Hello, Łukasz'

>>> greet("Łukasz", greeting="Awesome job")
'Awesome job, Łukasz'

>>> greet(name="Łukasz", greeting="Awesome job")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() got some positional-only arguments passed as
           keyword arguments: 'name'
```

In `greet()`, the slash is placed between `name` and `greeting`. This means that `name` is a positional-only argument, while `greeting` is a regular argument that can be passed either by position or by keyword.

At first glance, positional-only arguments can seem a bit limiting and contrary to Python's mantra about the importance of readability. You will probably find that there are not a lot of occasions where positional-only arguments improve your code.

However, in the right circumstances, positional-only arguments can give you some flexibility when you're designing functions. First, positional-only arguments make sense when you have arguments that have a natural order but are hard to give good, descriptive names to.

Another possible benefit of using positional-only arguments is that you can more easily refactor your functions. In particular, you can change the name of your parameters without worrying that other code depends on those names.

Positional-only arguments nicely complement **keyword-only** arguments. In any version of Python 3, you can specify keyword-only arguments using the star (`*`). Any argument *after* `*` must be specified using a keyword:

```
>>> def to_fahrenheit(*, celsius):
...     return 32 + celsius * 9 / 5
...
>>> to_fahrenheit(40)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: to_fahrenheit() takes 0 positional arguments but 1 was given

>>> to_fahrenheit(celsius=40)
104.0
```

`celsius` is a keyword-only argument, so Python raises an error if you try to specify it based on position, without the keyword.

You can combine positional-only, regular, and keyword-only arguments, by specifying them in this order separated by `/` and `*`. In the following example, `text` is a positional-only argument, `border` is a regular argument with a default value, and `width` is a keyword-only argument with a default value:

```
>>> def headline(text, /, border="♦", *, width=50):
...     return f" {text} ".center(width, border)
...
```

Since `text` is positional-only, you can't use the keyword `text`:

```
>>> headline("Positional-only Arguments")
'♦♦♦♦♦♦♦♦♦♦♦ Positional-only Arguments ♦♦♦♦♦♦♦♦♦♦♦♦'

>>> headline(text="This doesn't work!")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: headline() got some positional-only arguments passed as
           keyword arguments: 'text'
```

`border`, on the other hand, can be specified both with and without the keyword:

```
>>> headline("Python 3.8", "=")
'=================== Python 3.8 ==================='

>>> headline("Real Python", border=":")
':::::::::::::::::: Real Python :::::::::::::::::::'
```

Finally, `width` must be specified using the keyword:

```
>>> headline("Python", "🐍", width=38)
'🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍 Python 🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍🐍'

>>> headline("Python", "🐍", 38)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: headline() takes from 1 to 2 positional arguments
           but 3 were given
```

You can read more about positional-only arguments in PEP 570.

More Precise Types

Python's typing system is quite mature at this point. However, in Python 3.8, some new features have been added to `typing` to allow more precise typing:

- Literal types
- Typed dictionaries
- Final objects
- Protocols

Python supports optional **type hints**, typically as annotations on your code:

```
def double(number: float) -> float:
    return 2 * number
```

In this example, you say that `number` should be a `float` and the `double()` function should return a `float`, as well. However, Python treats these annotations as *hints*. They are not enforced at runtime:

```
>>> double(3.14)
6.28

>>> double("I'm not a float")
"I'm not a floatI'm not a float"
```

`double()` happily accepts `"I'm not a float"` as an argument, even though that's not a `float`. There are libraries that can use types at runtime, but that is not the main use case for Python's type system.

Instead, type hints allow static type checkers to do type checking of your Python code, without actually running your scripts. This is reminiscent of compilers catching type errors in other languages like Java, Rust, and Crystal. Additionally, type hints act as documentation of your code, making it easier to read, as well as improving auto-complete in your IDE.

Note: There are several static type checkers available, including Pyright, Pytype, and Pyre. In this article, you'll use Mypy. You can install Mypy from PyPI using `pip`:

```
$ python -m pip install mypy
```

In some sense, Mypy is the reference implementation of a type checker for Python, and is being developed at Dropbox under the lead of Jukka Lehtasalo. Python's creator, Guido van Rossum, is part of the Mypy team.

You can find more information about type hints in Python in the original PEP 484, as well as in Python Type Checking (Guide).

There are four new PEPs about type checking that have been accepted and included in Python 3.8. You'll see short examples from each of these.

PEP 586 introduce the **`Literal`** type. `Literal` is a bit special in that it represents one or several specific values. One use case of `Literal` is to be able to precisely add types, when string arguments are used to describe specific behavior. Consider the following example:

```
# draw_line.py

def draw_line(direction: str) -> None:
    if direction == "horizontal":
        ...  # Draw horizontal line

    elif direction == "vertical":
        ...  # Draw vertical line

    else:
        raise ValueError(f"invalid direction {direction!r}")

draw_line("up")
```

The program will pass the static type checker, even though `"up"` is an invalid direction. The type checker only checks that `"up"` is a string. In this case, it would be more precise to say that `direction` must be either the literal string `"horizontal"` or the literal string `"vertical"`. Using `Literal`, you can do exactly that:

```
# draw_line.py

from typing import Literal

def draw_line(direction: Literal["horizontal", "vertical"]) -> None:
    if direction == "horizontal":
        ...  # Draw horizontal line

    elif direction == "vertical":
        ...  # Draw vertical line

    else:
        raise ValueError(f"invalid direction {direction!r}")

draw_line("up")
```

By exposing the allowed values of `direction` to the type checker, you can now be warned about the error:

```
$ mypy draw_line.py
draw_line.py:15: error:
    Argument 1 to "draw_line" has incompatible type "Literal['up']";
    expected "Union[Literal['horizontal'], Literal['vertical']]"
Found 1 error in 1 file (checked 1 source file)
```

The basic syntax is `Literal[<literal>]`. For instance, `Literal[38]` represents the literal value 38. You can express one of several literal values using `Union`:

```
Union[Literal["horizontal"], Literal["vertical"]]
```

Since this is a fairly common use case, you can (and probably should) use the simpler notation `Literal["horizontal", "vertical"]` instead. You already used the latter when adding types to `draw_line()`. If you look carefully at the output from Mypy above, you can see that it translated the simpler notation to the `Union` notation internally.

There are cases where the type of the return value of a function depends on the input arguments. One example is `open()` which may return a text string or a byte array depending on the value of `mode`. This can be handled through [overloading](https://www.python.org/dev/peps/pep-0484/#function-method-overloading).

The following example shows the skeleton of a calculator that can return the answer either as regular numbers (`38`), or as [roman numerals](https://en.wikipedia.org/wiki/Roman_numerals) (`XXXVIII`):

```
# calculator.py

from typing import Union

ARABIC_TO_ROMAN = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                   (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                   (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")]

def _convert_to_roman_numeral(number: int) -> str:
    """Convert number to a roman numeral string"""
    result = list()
    for arabic, roman in ARABIC_TO_ROMAN:
        count, number = divmod(number, arabic)
        result.append(roman * count)
    return "".join(result)

def add(num_1: int, num_2: int, to_roman: bool = True) -> Union[str, int]:
    """Add two numbers"""
    result = num_1 + num_2

    if to_roman:
        return _convert_to_roman_numeral(result)
    else:
        return result
```

The code has the correct type hints: the result of `add()` will be either `str` or `int`. However, often this code will be called with a literal `True` or `False` as the value of `to_roman` in which case you would like the type checker to infer exactly whether `str` or `int` is returned. This can be done using `Literal` together with `@overload`:

```
# calculator.py

from typing import Literal, overload, Union

ARABIC_TO_ROMAN = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
                   (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
                   (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")]

def _convert_to_roman_numeral(number: int) -> str:
    """Convert number to a roman numeral string"""
    result = list()
    for arabic, roman in ARABIC_TO_ROMAN:
        count, number = divmod(number, arabic)
        result.append(roman * count)
    return "".join(result)

@overload
```

```
def add(num_1: int, num_2: int, to_roman: Literal[True]) -> str: ...
@overload
def add(num_1: int, num_2: int, to_roman: Literal[False]) -> int: ...

def add(num_1: int, num_2: int, to_roman: bool = True) -> Union[str, int]:
    """Add two numbers"""
    result = num_1 + num_2

    if to_roman:
        return _convert_to_roman_numeral(result)
    else:
        return result
```

The added `@overload` signatures will help your type checker infer `str` or `int` depending on the literal values of `to_roman`. Note that the ellipses (`...`) are a literal part of the code. They stand in for the function body in the overloaded signatures.

As a complement to `Literal`, [PEP 591](https://www.python.org/dev/peps/pep-0591/) introduces [**`Final`**](). This qualifier specifies that a variable or attribute should not be reassigned, redefined, or overridden. The following is a typing error:

```
from typing import Final

ID: Final = 1

...

ID += 1
```

Mypy will highlight the line `ID += 1`, and note that you `Cannot assign to final name "ID"`. This gives you a way to ensure that constants in your code never change their value.

Additionally, there is also a [**`@final`**]() decorator that can be applied to classes and methods. Classes [decorated]() with `@final` can't be subclassed, while `@final` methods can't be overridden by subclasses:

```
from typing import final

@final
class Base:
    ...

class Sub(Base):
    ...
```

Mypy will flag this example with the error message `Cannot inherit from final class "Base"`. To learn more about `Final` and `@final`, see [PEP 591]().

The third PEP allowing for more specific type hints is [PEP 589](), which introduces [**`TypedDict`**](). This can be used to specify types for keys and values in a dictionary using a notation that is similar to the typed [`NamedTuple`]().

Traditionally, dictionaries have been annotated using [`Dict`](). The issue is that this only allowed one type for the keys and one type for the values, often leading to annotations like `Dict[str, Any]`. As an example, consider a dictionary that registers information about Python versions:

```
py38 = {"version": "3.8", "release_year": 2019}
```

The value corresponding to `version` is a string, while `release_year` is an integer. This can't be precisely represented using `Dict`. With the new `TypedDict`, you can do the following:

```
from typing import TypedDict

class PythonVersion(TypedDict):
    version: str
    release_year: int

py38 = PythonVersion(version="3.8", release_year=2019)
```

The type checker will then be able to infer that `py38["version"]` has type `str`, while `py38["release_year"]` is an `int`. At runtime, a `TypedDict` is a regular `dict`, and type hints are ignored as usual. You can also use `TypedDict` purely as an annotation:

```
py38: PythonVersion = {"version": "3.8", "release_year": 2019}
```

Mypy will let you know if any of your values has the wrong type, or if you use a key that has not been declared. See PEP 589 for more examples.

Mypy has supported **Protocols** for a while already. However, the official acceptance only happened in May 2019.

Protocols are a way of formalizing Python's support for duck typing:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. (Source)

Duck typing allows you to, for example, read `.name` on any object that has a `.name` attribute, without really caring about the type of the object. It may seem counter-intuitive for the typing system to support this. Through structural subtyping, it's still possible to make sense of duck typing.

You can for instance define a protocol called `Named` that can identify all objects with a `.name` attribute:

```
from typing import Protocol

class Named(Protocol):
    name: str

def greet(obj: Named) -> None:
    print(f"Hi {obj.name}")
```

Here, `greet()` takes any object, as long as it defines a `.name` attribute. See PEP 544 and the Mypy documentation for more information about protocols.

Simpler Debugging With f-Strings

f-strings were introduced in Python 3.6, and have become very popular. They might be the most common reason for Python libraries only being supported on version 3.6 and later. An f-string is a formatted string literal. You can recognize it by the leading `f`:

```
>>> style = "formatted"
>>> f"This is a {style} string"
'This is a formatted string'
```

When you use f-strings, you can enclose variables and even expressions inside curly braces. They will then be evaluated at runtime and included in the string. You can have several expressions in one f-string:

```
>>> import math
>>> r = 3.6

>>> f"A circle with radius {r} has area {math.pi * r * r:.2f}"
'A circle with radius 3.6 has area 40.72'
```

In the last expression, `{math.pi * r * r:.2f}`, you also use a format specifier. Format specifiers are separated from the expressions with a colon.

`.2f` means that the area is formatted as a floating point number with 2 decimals. The format specifiers are the same as for [`.format()`](). See the [official documentation]() for a full list of allowed format specifiers.

In Python 3.8, you can use assignment expressions inside f-strings. Just make sure to surround the assignment expression with parentheses:

```
>>> import math
>>> r = 3.8

>>> f"Diameter {(diam := 2 * r)} gives circumference {math.pi * diam:.2f}"
'Diameter 7.6 gives circumference 23.88'
```

However, the real f-news in Python 3.8 is the new debugging specifier. You can now add `=` at the end of an expression, and it will print both the expression and its value:

```
>>> python = 3.8
>>> f"{python=}"
'python=3.8'
```

This is a short-hand, that typically will be most useful when working interactively or adding print statements to debug your script. In earlier versions of Python, you needed to spell out the variable or expression twice to get the same information:

```
>>> python = 3.7
>>> f"python={python}"
'python=3.7'
```

You can add spaces around `=`, and use format specifiers as usual:

```
>>> name = "Eric"
>>> f"{name = }"
"name = 'Eric'"

>>> f"{name = :>10}"
'name =       Eric'
```

The `>10` format specifier says that `name` should be right-aligned within a 10 character string. `=` works for more complex expressions as well:

```
>>> f"{name.upper()[::-1] = }"
"name.upper()[::-1] = 'CIRE'"
```

For more information about f-strings, see [Python 3's f-Strings: An Improved String Formatting Syntax (Guide)]().

The Python Steering Council

Technically, Python's **governance** is not a language feature. However, Python 3.8 is the first version of Python not developed under the **benevolent dictatorship** of Guido van Rossum. The Python language is now governed by a **steering council** consisting of five core developers:

- Barry Warsaw
- Brett Cannon
- Carol Willing
- Guido van Rossum
- Nick Coghlan

The road to the new governance model for Python was an interesting study in self-organization. Guido van Rossum created Python in the early 1990s, and has been affectionally dubbed Python's **Benevolent Dictator for Life** (BDFL). Through the years, more and more decisions about the Python language were made through **Python Enhancement Proposals** (PEPs). Still, Guido officially had the last word on any new language feature.

After a long and drawn out discussion about assignment expressions, Guido announced in July 2018 that he was retiring from his role as BDFL (for real this time). He purposefully did not name a successor. Instead, he asked the team of core developers to figure out how Python should be governed going forward.

Luckily, the PEP process was already well established, so it was natural to use PEPs to discuss and decide on a new governance model. Through the fall of 2018, several models were proposed, including electing a new BDFL (renamed the Gracious Umpire Influencing Decisions Officer: the GUIDO), or moving to a community model based on consensus and voting, without centralized leadership. In December 2018, the steering council model was chosen after a vote among the core developers.

The Python Steering Council at PyCon 2019. From left to right: Barry Warsaw, Brett Cannon, Carol Willing, Guido van Rossum, and Nick Coghlan (Image: Geir Arne Hjelle)

The steering council consists of five members of the Python community, as listed above. There will be an election for a new steering council after every major release of Python. In other words, there will be an election following the release of Python 3.8.

Although it's an open election, it's expected that most, if not all, of the inaugural steering council will be reelected. The steering council has broad powers to make decisions about the Python language, but should strive to exercise those powers as little as possible.

You can read all about the new governance model in PEP 13, while the process of deciding on the new model is described in PEP 8000. For more information, see the PyCon 2019 Keynote, and listen to Brett Cannon on Talk Python To Me and on The Changelog podcast. You can follow updates from the steering council on GitHub.

Other Pretty Cool Features

So far, you've seen the headline news regarding what's new in Python 3.8. However, there are many other changes that are also pretty cool. In this section, you'll get a quick look at some of them.

`importlib.metadata`

There is one new module available in the standard library in Python 3.8: `importlib.metadata`. Through this module, you can access information about installed packages in your Python installation. Together with its companion module, `importlib.resources`, `importlib.metadata` improves on the functionality of the older `pkg_resources`.

As an example, you can get some information about `pip`:

```
>>> from importlib import metadata
>>> metadata.version("pip")
'19.2.3'

>>> pip_metadata = metadata.metadata("pip")
>>> list(pip_metadata)
['Metadata-Version', 'Name', 'Version', 'Summary', 'Home-page', 'Author',
 'Author-email', 'License', 'Keywords', 'Platform', 'Classifier',
  'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
  'Classifier', 'Classifier', 'Classifier', 'Classifier', 'Classifier',
  'Classifier', 'Classifier', 'Requires-Python']

>>> pip_metadata["Home-page"]
'https://pip.pypa.io/'

>>> pip_metadata["Requires-Python"]
'>=2.7,!=3.0.*,!=3.1.*,!=3.2.*,!=3.3.*,!=3.4.*'

>>> len(metadata.files("pip"))
668
```

The currently installed version of `pip` is 19.2.3. `metadata()` gives access to most of the information that you can see on PyPI. You can for instance see that this version of `pip` requires either Python 2.7, or Python 3.5 or higher. With `files()`, you get a listing of all files that make up the `pip` package. In this case, there are almost 700 files.

`files()` returns a list of `Path` objects. These give you a convenient way of looking into the source code of a package, using `read_text()`. The following example prints out `__init__.py` from the `realpython-reader` package:

```
>>> [p for p in metadata.files("realpython-reader") if p.suffix == ".py"]
[PackagePath('reader/__init__.py'), PackagePath('reader/__main__.py'),
```

```
 PackagePath('reader/feed.py'), PackagePath('reader/viewer.py')]

>>> init_path = _[0]  # Underscore access last returned value in the REPL
>>> print(init_path.read_text())
"""Real Python feed reader

Import the `feed` module to work with the Real Python feed:

    >>> from reader import feed
    >>> feed.get_titles()
    ['Logging in Python', 'The Best Python Books', ...]

See https://github.com/realpython/reader/ for more information
"""

# Version of realpython-reader package
__version__ = "1.0.0"

...
```

You can also access package dependencies:

```
>>> metadata.requires("realpython-reader")
['feedparser', 'html2text', 'importlib-resources', 'typing']
```

`requires()` lists the dependencies of a package. You can see that `realpython-reader` for instance uses [`feedparser`]() in the background to read and parse a feed of articles.

There is a backport of `importlib.metadata` [available on PyPI]() that works on earlier versions of Python. You can install it using `pip`:

```
$ python -m pip install importlib-metadata
```

You can fall back on using the PyPI backport in your code as follows:

```
try:
    from importlib import metadata
except ImportError:
    import importlib_metadata as metadata

...
```

See the [documentation]() for more information about `importlib.metadata`

New and Improved `math` and `statistics` Functions

Python 3.8 brings many improvements to existing standard library packages and modules. `math` in the standard library has a few new functions. `math.prod()` works similarly to the built-in `sum()`, but for multiplicative products:

```
>>> import math
>>> math.prod((2, 8, 7, 7))
784

>>> 2 * 8 * 7 * 7
784
```

The two statements are equivalent. `prod()` will be easier to use when you already have the factors stored in an iterable.

Another new function is `math.isqrt()`. You can use `isqrt()` to find the integer part of [square roots](#):

```
>>> import math
>>> math.isqrt(9)
3

>>> math.sqrt(9)
3.0

>>> math.isqrt(15)
3

>>> math.sqrt(15)
3.872983346207417
```

The square root of 9 is 3. You can see that `isqrt()` returns an integer result, while `math.sqrt()` always returns a `float`. The square root of 15 is almost 3.9. Note that `isqrt()` truncates the answer down to the next integer, in this case 3.

Finally, you can now more easily work with *n*-dimensional points and vectors in the standard library. You can find the distance between two points with `math.dist()`, and the length of a vector with `math.hypot()`:

```
>>> import math
>>> point_1 = (16, 25, 20)
>>> point_2 = (8, 15, 14)

>>> math.dist(point_1, point_2)
14.142135623730951

>>> math.hypot(*point_1)
35.79106033634656

>>> math.hypot(*point_2)
22.02271554554524
```

This makes it easier to work with points and vectors using the standard library. However, if you will be doing many calculations on points or vectors, you should check out NumPy.

The `statistics` module also has several new functions:

- `statistics.fmean()` calculates the mean of `float` numbers.
- `statistics.geometric_mean()` calculates the geometric mean of `float` numbers.
- `statistics.multimode()` finds the most frequently occurring values in a sequence.
- `statistics.quantiles()` calculates cut points for dividing data into *n* continuous intervals with equal probability.

The following example shows the functions in use:

```
>>> import statistics
>>> data = [9, 3, 2, 1, 1, 2, 7, 9]
>>> statistics.fmean(data)
4.25

>>> statistics.geometric_mean(data)
```

```
3.013668912157617

>>> statistics.multimode(data)
[9, 2, 1]

>>> statistics.quantiles(data, n=4)
[1.25, 2.5, 8.5]
```

In Python 3.8, there is a new `statistics.NormalDist` class that makes it more convenient to work with the Gaussian normal distribution.

To see an example of using `NormalDist`, you can try to compare the speed of the new `statistics.fmean()` and the traditional `statistics.mean()`:

```
>>> import random
>>> import statistics
>>> from timeit import timeit

>>> # Create 10,000 random numbers
>>> data = [random.random() for _ in range(10_000)]

>>> # Measure the time it takes to run mean() and fmean()
>>> t_mean = [timeit("statistics.mean(data)", number=100, globals=globals())
...           for _ in range(30)]
>>> t_fmean = [timeit("statistics.fmean(data)", number=100, globals=globals())
...            for _ in range(30)]

>>> # Create NormalDist objects based on the sampled timings
>>> n_mean = statistics.NormalDist.from_samples(t_mean)
>>> n_fmean = statistics.NormalDist.from_samples(t_fmean)

>>> # Look at sample mean and standard deviation
>>> n_mean.mean, n_mean.stdev
(0.825690647733245, 0.07788573997674526)

>>> n_fmean.mean, n_fmean.stdev
(0.010488564966666065, 0.0008572332785645231)

>>> # Calculate the lower 1 percentile of mean
>>> n_mean.quantiles(n=100)[0]
0.6445013221202459
```

In this example, you use `timeit` to measure the execution time of `mean()` and `fmean()`. To get reliable results, you let `timeit` execute each function 100 times, and collect 30 such time samples for each function. Based on these samples, you create two `NormalDist` objects. Note that if you run the code yourself, it might take up to a minute to collect the different time samples. `NormalDist` has many convenient attributes and methods. See the documentation for a complete list. Inspecting `.mean` and `.stdev`, you see that the old `statistics.mean()` runs in 0.826 ± 0.078 seconds, while the new `statistics.fmean()` spends 0.0105 ± 0.0009 seconds. In other words, `fmean()` is about 80 times faster for these data. If you need more advanced statistics in Python than the standard library offers, check out `statsmodels` and `scipy.stats`.

Warnings About Dangerous Syntax

Python has a `SyntaxWarning` which can warn about dubious syntax that is typically not a `SyntaxError`. Python 3.8 adds a few new ones that can help you during coding and debugging.

The difference between `is` and `==` can be confusing. The latter checks for equal values, while `is` is `True` only when objects are the same. Python 3.8 will try to warn you about cases when you should use `==` instead of `is`:

```
>>> # Python 3.7
>>> version = "3.7"
>>> version is "3.7"
False

>>> # Python 3.8
>>> version = "3.8"
>>> version is "3.8"
<stdin>:1: SyntaxWarning: "is" with a literal. Did you mean "=="?
False

>>> version == "3.8"
True
```

It’s easy to miss a comma when you’re writing out a long list, especially when formatting it vertically. Forgetting a comma in a list of tuples will give a confusing error message about tuples not being callable. Python 3.8 additionally emits a warning that points toward the real issue:

```
>>> [
...   (1, 3)
...   (2, 4)
... ]
<stdin>:2: SyntaxWarning: 'tuple' object is not callable; perhaps
           you missed a comma?
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'tuple' object is not callable
```

The warning correctly identifies the missing comma as the real culprit.

Optimizations

There are several optimizations made for Python 3.8. Some that make code run faster. Others reduce the memory footprint. For example, looking up fields in a `namedtuple` is significantly faster in Python 3.8 compared with Python 3.7:

```
>>> import collections
>>> from timeit import timeit
>>> Person = collections.namedtuple("Person", "name twitter")
>>> raymond = Person("Raymond", "@raymondh")

>>> # Python 3.7
>>> timeit("raymond.twitter", globals=globals())
0.05876131607996285

>>> # Python 3.8
>>> timeit("raymond.twitter", globals=globals())
0.0377705999400132
```

You can see that looking up `.twitter` on the `namedtuple` is 30-40% faster in Python 3.8. Lists save some space when they are initialized from iterables with a known length. This can save memory:

```
>>> import sys
```

```
>>> # Python 3.7
>>> sys.getsizeof(list(range(20191014)))
181719232

>>> # Python 3.8
>>> sys.getsizeof(list(range(20191014)))
161528168
```

In this case, the list uses about 11% less memory in Python 3.8 compared with Python 3.7.

Other optimizations include better performance in `subprocess`, faster file copying with `shutil`, improved default performance in `pickle`, and faster `operator.itemgetter` operations. See the official documentation for a complete list of optimizations.

So, Should You Upgrade to Python 3.8?

Let's start with the simple answer. If you want to try out any of the new features you have seen here, then you do need to be able to use Python 3.8. Tools like `pyenv` and Anaconda make it easy to have several versions of Python installed side by side. Alternatively, you can run the official Python 3.8 Docker container. There is no downside to trying out Python 3.8 for yourself.

Now, for the more complicated questions. Should you upgrade your production environment to Python 3.8? Should you make your own project dependent on Python 3.8 to take advantage of the new features?

You should have very few issues running Python 3.7 code in Python 3.8. Upgrading your environment to run Python 3.8 is therefore quite safe, and you would be able to take advantage of the optimizations made in the new version. Different beta-versions of Python 3.8 have already been available for months, so hopefully most bugs are already squashed. However, if you want to be conservative, you might hold out until the first maintenance release (Python 3.8.1) is available.

Once you've upgraded your environment, you can start to experiment with features that are only in Python 3.8, such as assignment expressions and positional-only arguments. However, you should be conscious about whether other people depend on your code, as this will force them to upgrade their environment as well. Popular libraries will probably mostly support at least Python 3.6 for quite a while longer.

See Porting to Python 3.8 for more information about preparing your code for Python 3.8.