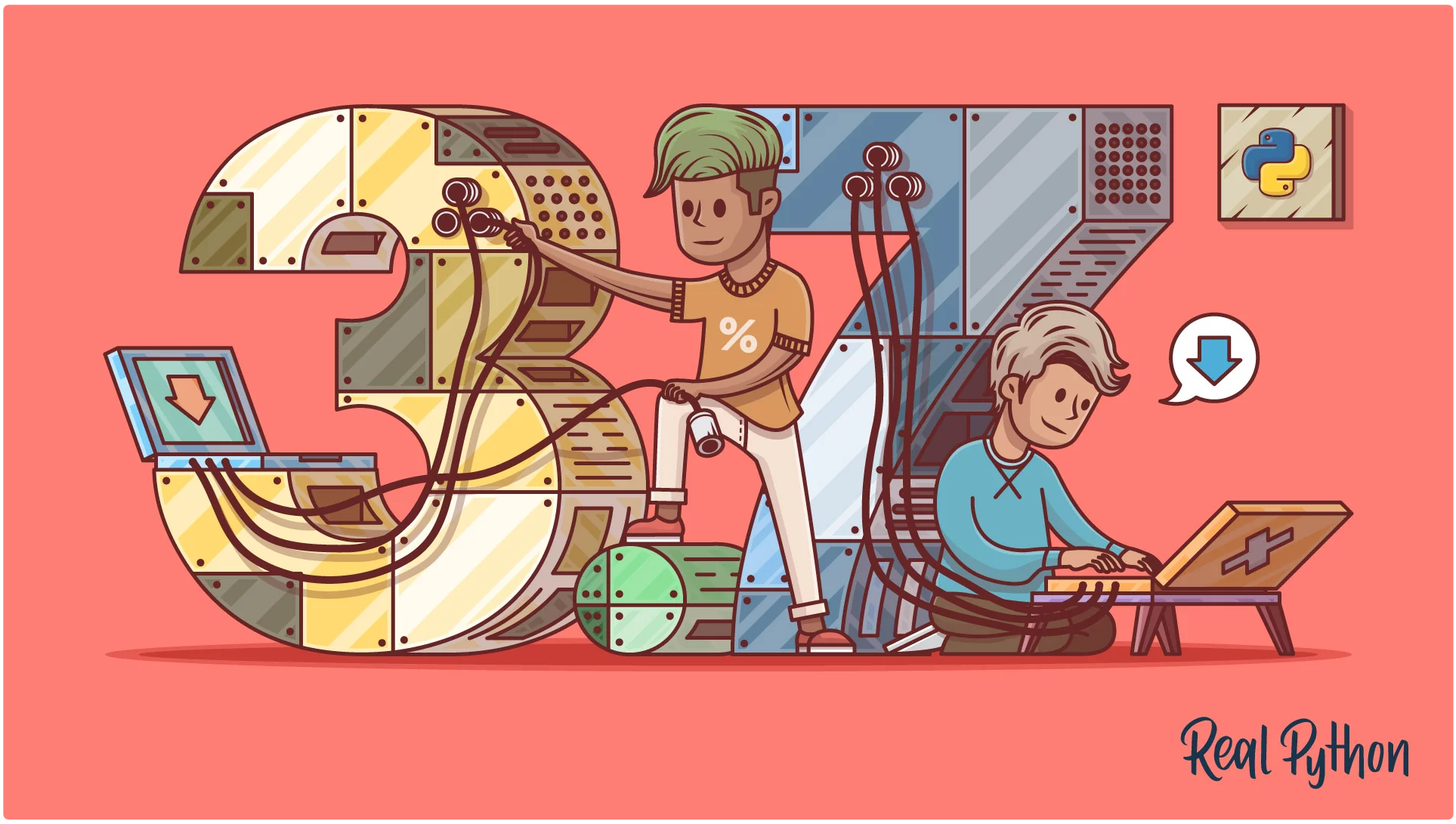

Cool New Features in Python 3.7

by Geir Arne Hjelle Jun 27, 2018 25 Comments python

Table of Contents

Python 3.7 is officially released! This new Python version has been in development since September 2016, and now we all get to enjoy the results of the core developers’ hard work.

What does the new Python version bring? While the documentation gives a good overview of the new features, this article will take a deep dive into some of the biggest pieces of news. These include:

- Easier access to debuggers through a new `breakpoint()` built-in
- Simple class creation using data classes
- Customized access to module attributes
- Improved support for type hinting
- Higher precision timing functions

More importantly, Python 3.7 is fast.

In the final sections of this article, you'll read more about this speed, as well as some of the other cool features of Python 3.7. You will also get some advice on upgrading to the new version.

The `breakpoint()` Built-In

While we might strive to write perfect code, the simple truth is that we never do. Debugging is an important part of programming. Python 3.7 introduces the new built-in function `breakpoint()`. This does not really add any new functionality to Python, but it makes using debuggers more flexible and intuitive.

Assume that you have the following buggy code in the file `bugs.py`:

Python

```
def divide(e, f):
    return f / e

a, b = 0, 1
print(divide(a, b))
```

Running the code causes a `ZeroDivisionError` inside the `divide()` function. Let's say that you want to interrupt your code and drop into a debugger right at the top of `divide()`. You can do so by setting a so called "breakpoint" in your code:

Python

```
def divide(e, f):
    # Insert breakpoint here
    return f / e
```

A breakpoint is a signal inside your code that execution should temporarily stop, so that you can look around at the current state of the program. How do you place the breakpoint? In Python 3.6 and below, you use this somewhat cryptic line:

Python

```
def divide(e, f):
    import pdb; pdb.set_trace()
    return f / e
```

Here, `pdb` is the Python Debugger from the standard library. In Python 3.7, you can use the new `breakpoint()` function call as a shortcut instead:

Python

```
def divide(e, f):
    breakpoint()
    return f / e
```

In the background, `breakpoint()` is first importing `pdb` and then calling `pdb.set_trace()` for you. The obvious benefits are that `breakpoint()` is easier to remember and that you only need to type 12 characters instead of 27. However, the real bonus of using `breakpoint()` is its customizability.

Run your `bugs.py` script with `breakpoint()`:

Shell

```
$ python3.7 bugs.py
> /home/gahjelle/bugs.py(3)divide()
-> return f / e
(Pdb)
```

The script will break when it reaches `breakpoint()` and drop you into a PDB debugging session. You can type `c` and hit `Enter ↩` to continue the script. Refer to Nathan Jennings' PDB guide if you want to learn more about PDB and debugging.

Now, say that you think you've fixed the bug. You would like to run the script again but without stopping in the debugger. You could, of course, comment out the `breakpoint()` line, but another option is to use the `PYTHONBREAKPOINT` environment variable. This variable controls the behavior of `breakpoint()`, and setting `PYTHONBREAKPOINT=0` means that any call to `breakpoint()` is ignored:

Shell

```
$ PYTHONBREAKPOINT=0 python3.7 bugs.py
ZeroDivisionError: division by zero
```

Oops, it seems as if you haven't fixed the bug after all…

Another option is to use `PYTHONBREAKPOINT` to specify a debugger other than PDB. For instance, to use PuDB (a visual debugger in the console) you can do:

Shell

```
$ PYTHONBREAKPOINT=pudb.set_trace python3.7 bugs.py
```

For this to work, you need to have `pudb` installed (`pip install pudb`). Python will take care of importing `pudb` for you though. This way you can also set your default debugger. Simply set the `PYTHONBREAKPOINT` environment variable to your preferred debugger. See this guide for instructions on how to set an environment variable on your system.

The new `breakpoint()` function does not only work with debuggers. One convenient option could be to simply start an interactive shell inside your code. For instance, to start an IPython session, you can use the following:

Shell

```
$ PYTHONBREAKPOINT=IPython.embed python3.7 bugs.py
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print(e / f)
0.0
```

You can also create your own function and have `breakpoint()` call that. The following code prints all variables in the local scope. Add it to a file called `bp_utils.py`:

Python

```
from pprint import pprint
import sys

def print_locals():
    caller = sys._getframe(1)  # Caller is 1 frame up.
    pprint(caller.f_locals)
```

To use this function, set `PYTHONBREAKPOINT` as before, with the `<module>.<function>` notation:

Shell

```
$ PYTHONBREAKPOINT=bp_utils.print_locals python3.7 bugs.py
{'e': 0, 'f': 1}
ZeroDivisionError: division by zero
```

Normally, `breakpoint()` will be used to call functions and methods that do not need arguments. However, it is possible to pass arguments as well. Change the line `breakpoint()` in `bugs.py` to:

Python

```
breakpoint(e, f, end="<-END\n")
```

Note: The default PDB debugger will raise a `TypeError` at this line because `pdb.set_trace()` does not take any positional arguments.

Run this code with `breakpoint()` masquerading as the `print()` function to see a simple example of the arguments being passed through:

Shell

```
$ PYTHONBREAKPOINT=print python3.7 bugs.py
0 1<-END
ZeroDivisionError: division by zero
```

See PEP 553 as well as the documentation for `breakpoint()` and `sys.breakpointhook()` for more information.

Data Classes

The new `dataclasses` module makes it more convenient to write your own classes, as special methods like `.__init__()`, `.__repr__()`, and `.__eq__()` are added automatically. Using the `@dataclass` decorator, you can write something like:

Python

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Country:
    name: str
    population: int
    area: float = field(repr=False, compare=False)
    coastline: float = 0

    def beach_per_person(self):
        """Meters of coastline per person"""
        return (self.coastline * 1000) / self.population
```

These nine lines of code stand in for quite a bit of boilerplate code and best practices. Think about what it would take to implement `Country` as a regular class: the `.__init__()` method, a `repr`, six different comparison methods as well as the `.beach_per_person()` method. You can expand the box below to see an implementation of `Country` that is roughly equivalent to the data class:

Alternate implementation of the "Country" class Show/Hide

After creation, a data class is a normal class. You can, for instance, inherit from a data class in the normal way. The main purpose of data classes is to make it quick and easy to write robust classes, in particular small classes that mainly store data.

You can use the `Country` data class like any other class:

Python

```
>>> norway = Country("Norway", 5320045, 323802, 58133)
>>> norway
Country(name='Norway', population=5320045, coastline=58133)

>>> norway.area
323802

>>> usa = Country("United States", 326625791, 9833517, 19924)
>>> nepal = Country("Nepal", 29384297, 147181)
>>> nepal
Country(name='Nepal', population=29384297, coastline=0)

>>> usa.beach_per_person()
0.06099946957342386

>>> norway.beach_per_person()
10.927163210085629
```

Note that all the fields `.name`, `.population`, `.area`, and `.coastline` are used when initializing the class (although `.coastline` is optional, as is shown in the example of landlocked Nepal). The `Country` class has a reasonable `repr`, while defining methods works the same as for regular classes.

By default, data classes can be compared for equality. Since we specified `order=True` in the `@dataclass` decorator, the `Country` class can also be sorted:

Python

```
>>> norway == norway
True

>>> nepal == usa
False

>>> sorted((norway, usa, nepal))
[Country(name='Nepal', population=29384297, coastline=0),
 Country(name='Norway', population=5320045, coastline=58133),
 Country(name='United States', population=326625791, coastline=19924)]
```

The sorting happens on the field values, first `.name` then `.population`, and so on. However, if you use `field()`, you can customize which fields will be used in the comparison. In the example, the `.area` field was left out of the `repr` and the comparisons.

Note: The country data are from the CIA World Factbook with population numbers estimated for July 2017.

Before you all go book your next beach holidays in Norway, here is what the Factbook says about the Norwegian climate: “temperate along coast, modified by North Atlantic Current; colder interior with increased precipitation and colder summers; rainy year-round on west coast.”

Data classes do some of the same things as `namedtuple`. Yet, they draw their biggest inspiration from the `attrs` project. See our full guide to data classes for more examples and further information, as well as PEP 557 for the official description.

Customization of Module Attributes

Attributes are everywhere in Python! While class attributes are probably the most famous, attributes can actually be put on essentially anything—including functions and modules. Several of Python’s basic features are implemented as attributes: most of the introspection functionality, doc-strings, and name spaces. Functions inside a module are made available as module attributes.

Attributes are most often retrieved using the dot notation: `thing.attribute`. However, you can also get attributes that are named at runtime using `getattr()`:

Python

```
import random

random_attr = random.choice(("gammavariate", "lognormvariate", "normalvariate"))
random_func = getattr(random, random_attr)

print(f"A {random_attr} random value: {random_func(1, 1)}")
```

Running this code will produce something like:

Shell

```
A gammavariate random value: 2.8017715125270618
```

For classes, calling `thing.attr` will first look for `attr` defined on `thing`. If it is not found, then the special method `thing.__getattr__("attr")` is called. (This is a simplification. See this article for more details.) The `.__getattr__()` method can be used to customize access to attributes on objects.

Until Python 3.7, the same customization was not easily available for module attributes. However, PEP 562 introduces `__getattr__()` on modules, together with a corresponding `__dir__()` function. The `__dir__()` special function allows customization of the result of calling `dir()` on a module.

The PEP itself gives a few examples of how these functions can be used, including adding deprecation warnings to functions and lazy loading of heavy submodules. Below, we will build a simple plugin system that allows functions to be added to a module dynamically. This example takes advantage of Python packages. See this article if you need a refresher on packages.

Create a new directory, `plugins`, and add the following code to a file, `plugins/__init__.py`:

Python

```
from importlib import import_module
from importlib import resources

PLUGINS = dict()

def register_plugin(func):
    """Decorator to register plug-ins"""
    name = func.__name__
    PLUGINS[name] = func
    return func

def __getattr__(name):
    """Return a named plugin"""
    try:
        return PLUGINS[name]
    except KeyError:
        _import_plugins()
        if name in PLUGINS:
            return PLUGINS[name]
        else:
            raise AttributeError(
                f"module {__name__!r} has no attribute {name!r}"
            ) from None

def __dir__():
    """List available plug-ins"""
    _import_plugins()
    return list(PLUGINS.keys())

def _import_plugins():
    """Import all resources to register plug-ins"""
    for name in resources.contents(__name__):
        if name.endswith(".py"):
            import_module(f"{__name__}.{name[:-3]}")
```

Before we look at what this code does, add two more files inside the `plugins` directory. First, let's see `plugins/plugin_1.py`:

Python

```
from . import register_plugin

@register_plugin
def hello_1():
    print("Hello from Plugin 1")
```

Next, add similar code in the file `plugins/plugin_2.py`:

Python

```
from . import register_plugin

@register_plugin
def hello_2():
    print("Hello from Plugin 2")

@register_plugin
def goodbye():
    print("Plugin 2 says goodbye")
```

These plugins can now be used as follows:

Python

```
>>> import plugins
>>> plugins.hello_1()
Hello from Plugin 1

>>> dir(plugins)
['goodbye', 'hello_1', 'hello_2']

>>> plugins.goodbye()
Plugin 2 says goodbye
```

This may not all seem that revolutionary (and it probably isn't), but let's look at what actually happened here. Normally, to be able to call `plugins.hello_1()`, the `hello_1()` function must be defined in a `plugins` module or explicitly imported inside `__init__.py` in a `plugins` package. Here, it is neither!

Instead, `hello_1()` is defined in an arbitrary file inside the `plugins` package, and `hello_1()` becomes a part of the `plugins` package by registering itself using the `@register_plugin` decorator.

The difference is subtle. Instead of the package dictating which functions are available, the individual functions register themselves as part of the package. This gives you a simple structure where you can add functions independently of the rest of the code without having to keep a centralized list of which functions are available.

Let us do a quick review of what `__getattr__()` does inside the `plugins/__init__.py` code. When you asked for `plugins.hello_1()`, Python first looks for a `hello_1()` function inside the `plugins/__init__.py` file. As no such function exists, Python calls `__getattr__("hello_1")` instead. Remember the source code of the `__getattr__()` function:

Python

```
def __getattr__(name):
    """Return a named plugin"""
    try:
        return PLUGINS[name]        # 1) Try to return plugin
    except KeyError:
        _import_plugins()           # 2) Import all plugins
        if name in PLUGINS:
            return PLUGINS[name]    # 3) Try to return plugin again
        else:
            raise AttributeError(   # 4) Raise error
                f"module {__name__!r} has no attribute {name!r}"
            ) from None
```

`__getattr__()` contains the following steps. The numbers in the following list correspond to the numbered comments in the code:

1. First, the function optimistically tries to return the named plugin from the `PLUGINS` dictionary. This will succeed if a plugin named `name` exists and has already been imported.
2. If the named plugin is not found in the `PLUGINS` dictionary, we make sure all plugins are imported.
3. Return the named plugin if it has become available after the import.
4. If the plugin is not in the `PLUGINS` dictionary after importing all plugins, we raise an `AttributeError` saying that `name` is not an attribute (plugin) on the current module.

How is the `PLUGINS` dictionary populated though? The `_import_plugins()` function imports all Python files inside the `plugins` package, but does not seem to touch `PLUGINS`:

Python

```
def _import_plugins():
    """Import all resources to register plug-ins"""
    for name in resources.contents(__name__):
        if name.endswith(".py"):
            import_module(f"{__name__}.{name[:-3]}")
```

Don't forget that each plugin function is decorated by the `@register_plugin` decorator. This decorator is called when the plugins are imported and is the one actually populating the `PLUGINS` dictionary. You can see this if you manually import one of the plugin files:

Python

```
>>> import plugins
>>> plugins.PLUGINS
{}

>>> import plugins.plugin_1
>>> plugins.PLUGINS
{'hello_1': <function hello_1 at 0x7f29d4341598>}
```

Continuing the example, note that calling `dir()` on the module also imports the remaining plugins:

Python

```
>>> dir(plugins)
['goodbye', 'hello_1', 'hello_2']

>>> plugins.PLUGINS
{'hello_1': <function hello_1 at 0x7f29d4341598>,
 'hello_2': <function hello_2 at 0x7f29d4341620>,
 'goodbye': <function goodbye at 0x7f29d43416a8>}
```

`dir()` usually lists all available attributes on an object. Normally, using `dir()` on a module results in something like this:

Python

```
>>> import plugins
>>> dir(plugins)
['PLUGINS', '__builtins__', '__cached__', '__doc__',
 '__file__', '__getattr__', '__loader__', '__name__',
 '__package__', '__path__', '__spec__', '_import_plugins',
 'import_module', 'register_plugin', 'resources']
```

While this might be useful information, we are more interested in exposing the available plugins. In Python 3.7, you can customize the result of calling `dir()` on a module by adding a `__dir__()` special function. For `plugins/__init__.py`, this function first makes sure all plugins have been imported and then lists their names:

Python

```
def __dir__():
    """List available plug-ins"""
    _import_plugins()
    return list(PLUGINS.keys())
```

Before leaving this example, please note that we also used another cool new feature of Python 3.7. To import all modules inside the `plugins` directory, we used the new `importlib.resources` module. This module gives access to files and resources inside modules and packages without the need for `__file__` hacks (which do not always work) or `pkg_resources` (which is slow). Other features of `importlib.resources` will be highlighted later.

Typing Enhancements

Type hinting and annotations have been in constant development throughout the Python 3 series of releases. Python's typing system is now quite stable. Still, Python 3.7 brings some enhancements to the table: better performance, core support, and forward references.

Python does not do any type checking at runtime (unless you are explicitly using packages like `enforce`). Therefore, adding type hints to your code should not affect its performance.

Unfortunately, this is not completely true as most type hints need the `typing` module. The `typing` module is one of the slowest modules in the standard library. PEP 560 adds some core support for typing in Python 3.7, which significantly speeds up the `typing` module. The details of this are in general not necessary to know about. Simply lean back and enjoy the increased performance.

While Python's type system is reasonably expressive, one issue that causes some pain is forward references. Type hints—or more generally annotations—are evaluated while the module is imported. Therefore, all names must already be defined before they are used. The following is not possible:

Python

```
class Tree:
    def __init__(self, left: Tree, right: Tree) -> None:
        self.left = left
        self.right = right
```

Running the code raises a `NameError` because the class `Tree` is not yet (completely) defined in the definition of the `.__init__()` method:

Python

```
Traceback (most recent call last):
  File "tree.py", line 1, in <module>
    class Tree:
  File "tree.py", line 2, in Tree
    def __init__(self, left: Tree, right: Tree) -> None:
NameError: name 'Tree' is not defined
```

To overcome this, you would have needed to write `"Tree"` as a string literal instead:

Python

```
class Tree:
    def __init__(self, left: "Tree", right: "Tree") -> None:
        self.left = left
        self.right = right
```

See PEP 484 for the original discussion.

In a future Python 4.0, such so called forward references will be allowed. This will be handled by not evaluating annotations until that is explicitly asked for. PEP 563 describes the details of this proposal. In Python 3.7, forward references are already available as a `__future__` import. You can now write the following:

Python

```
from __future__ import annotations

class Tree:
    def __init__(self, left: Tree, right: Tree) -> None:
        self.left = left
        self.right = right
```

Note that in addition to avoiding the somewhat clumsy `"Tree"` syntax, the postponed evaluation of annotations will also speed up your code, since type hints are not executed. Forward references are already supported by `mypy`.

By far, the most common use of annotations is type hinting. Still, you have full access to the annotations at runtime and can use them as you see fit. If you are handling annotations directly, you need to deal with the possible forward references explicitly.

Let us create some admittedly silly examples that show when annotations are evaluated. First we do it old-style, so annotations are evaluated at import time. Let `anno.py` contain the following code:

Python

```
def greet(name: print("Now!")):
    print(f"Hello {name}")
```

Note that the annotation of `name` is `print()`. This is only to see exactly when the annotation is evaluated. Import the new module:

Python

```
>>> import anno
Now!

>>> anno.greet.__annotations__
{'name': None}

>>> anno.greet("Alice")
Hello Alice
```

As you can see, the annotation was evaluated at import time. Note that `name` ends up annotated with `None` because that is the return value of `print()`.

Add the `__future__` import to enable postponed evaluation of annotations:

Python

```
from __future__ import annotations

def greet(name: print("Now!")):
    print(f"Hello {name}")
```

Importing this updated code will not evaluate the annotation:

Python

```
>>> import anno

>>> anno.greet.__annotations__
{'name': "print('Now!')"}

>>> anno.greet("Marty")
Hello Marty
```

Note that `Now!` is never printed and the annotation is kept as a string literal in the `__annotations__` dictionary. In order to evaluate the annotation, use `typing.get_type_hints()` or `eval()`:

Python

```
>>> import typing
>>> typing.get_type_hints(anno.greet)
Now!
{'name': <class 'NoneType'>}

>>> eval(anno.greet.__annotations__["name"])
Now!

>>> anno.greet.__annotations__
{'name': "print('Now!')"}
```

Observe that the `__annotations__` dictionary is never updated, so you need to evaluate the annotation every time you use it.

Timing Precision

In Python 3.7, the `time` module gains some new functions as described in PEP 564. In particular, the following six functions are added:

- **`clock_gettime_ns()`:** Returns the time of a specified clock
- **`clock_settime_ns()`:** Sets the time of a specified clock
- **`monotonic_ns()`:** Returns the time of a relative clock that cannot go backwards (for instance due to daylight savings)
- **`perf_counter_ns()`**: Returns the value of a performance counter—a clock specifically designed to measure short intervals
- **`process_time_ns()`:** Returns the sum of the system and user CPU time of the current process (not including sleep time)
- **`time_ns()`**: Returns the number of nanoseconds since January 1st 1970

In a sense, there is no new functionality added. Each function is similar to an already existing function without the `_ns` suffix. The difference being that the new functions return a number of nanoseconds as an `int` instead of a number of seconds as a `float`.

For most applications, the difference between these new nanosecond functions and their old counterpart will not be appreciable. However, the new functions are easier to reason about because they rely on `int` instead of `float`. Floating point numbers are by nature inaccurate:

Python

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004

>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

This is not an issue with Python but rather a consequence of computers needing to represent infinite decimal numbers using a finite number of bits.

A Python `float` follows the IEEE 754 standard and uses 53 significant bits. The result is that any time greater than about 104 days (2^{53} or approximately 9 quadrillion nanoseconds) cannot be expressed as a float with nanosecond precision. In contrast, a Python `int` is unlimited, so an integer number of nanoseconds will always have nanosecond precision independent of the time value.

As an example, `time.time()` returns the number of seconds since January 1st 1970. This number is already quite big, so the precision of this number is at the microsecond level. This function is the one showing the biggest improvement in its `_ns` version. The resolution of `time.time_ns()` is about 3 times better than for `time.time()`.

> What is a nanosecond by the way? Technically, it is one billionth of a second, or `1e-9` second if you prefer scientific notation. These are just numbers though and do not really provide any intuition. For a better visual aid, see Grace Hopper's wonderful demonstration of the nanosecond.

As an aside, if you need to work with datetimes with nanosecond precision, the `datetime` standard library will not cut it. It explicitly only handles microseconds:

Python

```
>>> from datetime import datetime, timedelta
>>> datetime(2018, 6, 27) + timedelta(seconds=1e-6)
datetime.datetime(2018, 6, 27, 0, 0, 0, 1)

>>> datetime(2018, 6, 27) + timedelta(seconds=1e-9)
datetime.datetime(2018, 6, 27, 0, 0)
```

Instead, you can use the `astropy` project. Its `astropy.time` package represents datetimes using two `float` objects which guarantees "sub-nanosecond precision over times spanning the age of the universe."

Python

```
>>> from astropy.time import Time, TimeDelta
>>> Time("2018-06-27")
<Time object: scale='utc' format='iso' value=2018-06-27 00:00:00.000>

>>> t = Time("2018-06-27") + TimeDelta(1e-9, format="sec")
>>> (t - Time("2018-06-27")).sec
9.976020010071807e-10
```

The latest version of `astropy` is available in Python 3.5 and later.

Other Pretty Cool Features

So far, you have seen the headline news regarding what's new in Python 3.7. However, there are many other changes that are also pretty cool. In this section, we will look briefly at some of them.

The Order of Dictionaries Is Guaranteed

The CPython implementation of Python 3.6 has ordered dictionaries. (PyPy also has this.) This means that items in dictionaries are iterated over in the same order they were inserted. The first example is using Python 3.5, and the second is using Python 3.6:

Python

```
>>> {"one": 1, "two": 2, "three": 3}  # Python <= 3.5
{'three': 3, 'one': 1, 'two': 2}

>>> {"one": 1, "two": 2, "three": 3}  # Python >= 3.6
{'one': 1, 'two': 2, 'three': 3}
```

In Python 3.6, this ordering was just a nice consequence of that implementation of `dict`. In Python 3.7, however, dictionaries preserving their insert order is part of the language specification. As such, it may now be relied on in projects that support only Python >= 3.7 (or CPython >= 3.6).

“`async`” and “`await`” Are Keywords

Python 3.5 introduced coroutines with `async` and `await` syntax. To avoid issues of backwards compatibility, `async` and `await` were not added to the list of reserved keywords. In other words, it was still possible to define variables or functions named `async` and `await`.

In Python 3.7, this is no longer possible:

Python

```
>>> async = 1
  File "<stdin>", line 1
    async = 1
          ^
SyntaxError: invalid syntax

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

“`asyncio`” Face Lift

The `asyncio` standard library was originally introduced in Python 3.4 to handle concurrency in a modern way using event loops, coroutines and futures. Here is a gentle introduction.

In Python 3.7, the `asyncio` module is getting a major face lift, including many new functions, support for the context variables (see below), and performance improvements. Of particular note is `asyncio.run()`, which simplifies calling coroutines from synchronous code. Using `asyncio.run()`, you do not need to explicitly create the event loop. An asynchronous Hello World program can now be written:

Python

```
import asyncio

async def hello_world():
    print("Hello World!")

asyncio.run(hello_world())
```

Context Variables

Context variables are variables that can have different values depending on their context. They are similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread. The main use case for context variables is keeping track of variables in concurrent asynchronous tasks.

The following example constructs three contexts, each with their own value for the value `name`. The `greet()` function is later able to use the value of `name` inside each context:

Python

```
import contextvars

name = contextvars.ContextVar("name")
contexts = list()

def greet():
    print(f"Hello {name.get()}")

# Construct contexts and set the context variable name
for first_name in ["Steve", "Dina", "Harry"]:
    ctx = contextvars.copy_context()
    ctx.run(name.set, first_name)
    contexts.append(ctx)

# Run greet function inside each context
for ctx in reversed(contexts):
    ctx.run(greet)
```

Running this script greets Steve, Dina, and Harry in reverse order:

Shell

```
$ python3.7 context_demo.py
Hello Harry
Hello Dina
Hello Steve
```

Importing Data Files With "`importlib.resources`"

One challenge when packaging a Python project is deciding what to do with project resources like data files needed by the project. A few options have commonly been used:

- Hard-code a path to the data file.
- Put the data file inside the package and locate it using `__file__`.
- Use `setuptools.pkg_resources` to access the data file resource.

Each of these have their shortcomings. The first option is not portable. Using `__file__` is more portable, but if the Python project is installed it might end up inside a zip and not have a `__file__` attribute. The third option solves this problem, but is unfortunately very slow.

A better solution is the new `importlib.resources` module in the standard library. It uses Python's existing import functionality to also import data files. Assume you have a resource inside a Python package like this:

```
data/
│
├── alice_in_wonderland.txt
└── __init__.py
```

Note that `data` needs to be a Python package. That is, the directory needs to contain an `__init__.py` file (which may be empty). You can then read the `alice_in_wonderland.txt` file as follows:

Python

```
>>> from importlib import resources
>>> with resources.open_text("data", "alice_in_wonderland.txt") as fid:
...     alice = fid.readlines()
...
>>> print("".join(alice[:7]))
CHAPTER I. Down the Rabbit-Hole

Alice was beginning to get very tired of sitting by her sister on the
bank, and of having nothing to do: once or twice she had peeped into the
book her sister was reading, but it had no pictures or conversations in
it, ‘and what is the use of a book,’ thought Alice ‘without pictures or
conversations?’
```

A similar `resources.open_binary()` function is available for opening files in binary mode. In the earlier "plugins as module attributes" example, we used `importlib.resources` to discover the available plugins using `resources.contents()`. See Barry Warsaw's PyCon 2018 talk for more information.

It is possible to use `importlib.resources` in Python 2.7 and Python 3.4+ through a backport. A guide on migrating from `pkg_resources` to `importlib.resources` is available.

Developer Tricks

Python 3.7 has added several features aimed at you as a developer. You have already seen the new `breakpoint()` built-in. In addition, a few new `-X` command line options have been added to the Python interpreter.

You can easily get an idea of how much time the imports in your script takes, using `-X importtime`:

Shell

```
$ python3.7 -X importtime my_script.py
import time: self [us] | cumulative | imported package
import time:      2607 |       2607 | _frozen_importlib_external
...
import time:       844 |      28866 |   importlib.resources
import time:       404 |      30434 | plugins
```

The `cumulative` column shows the cumulative time of import (in microseconds). In this example, importing `plugins` took about 0.03 seconds, most of which was spent importing `importlib.resources`. The `self` column shows the import time excluding nested imports.

You can now use `-X dev` to activate "development mode." The development mode will add certain debug features and runtime checks that are considered too slow to be enabled by default. These include enabling `faulthandler` to show a traceback on serious crashes, as well as more warnings and debug hooks.

Finally, `-X utf8` enables UTF-8 mode. (See PEP 540.) In this mode, `UTF-8` will be used for text encoding regardless of the current locale.

Optimizations

Each new release of Python comes with a set of optimizations. In Python 3.7, there are some significant speed-ups, including:

- There is less overhead in calling many methods in the standard library.
- Method calls are up to 20% faster in general.
- The startup time of Python itself is reduced by 10-30%.
- Importing `typing` is 7 times faster.

In addition, many more specialized optimizations are included. See this list for a detailed overview.

The upshot of all these optimizations is that Python 3.7 is fast. It is simply the fastest version of CPython released so far.

So, Should I Upgrade?

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 3.7. Using tools such as `pyenv` or Anaconda makes it easy to have several versions of Python installed side by side. There is no downside to installing Python 3.7 and trying it out.

Now, for the more complicated questions. Should you upgrade your production environment to Python 3.7? Should you make your own project dependent on Python 3.7 to take advantage of the new features?

With the obvious caveat that you should always do thorough testing before upgrading your production environment, there are very few things in Python 3.7 that will break earlier code (`async` and `await` becoming keywords is one example though). If you are already using a modern Python, upgrading to 3.7 should be quite smooth. If you want to be a little conservative, you might want to wait for the release of the first maintenance release—Python 3.7.1—tentatively expected some time in July 2018.

Arguing that you should make your project 3.7 only is harder. Many of the new features in Python 3.7 are either available as backports to Python 3.6 (data classes, `importlib.resources`) or conveniences (faster startup and method calls, easier debugging, and `-X` options). The latter, you can take advantage of by running Python 3.7 yourself while keeping your code compatible with Python 3.6 (or lower).

The big features that will lock your code to Python 3.7 are `__getattr__()` on modules, forward references in type hints, and the nanosecond `time` functions. If you really need any of these, you should go ahead and bump your requirements. Otherwise, your project will probably be more useful to others if it can be run on Python 3.6 for a while longer.

See the Porting to Python 3.7 guide for details to be aware of when upgrading.

About **Geir Arne Hjelle**

Geir Arne is an avid Pythonista and a member of the Real Python tutorial team.

» More about Geir Arne

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Dan

Joanna