What's New in Python 3.7

Okay, here we go.
The short, short version!

— **Minister, Spaceballs (1987)**

dataclasses

- A nice addition for representing data.
- Nicer than namedtuples and shorter than stub classes.
- No `__init__` needed!

```python
from dataclasses import dataclass

@dataclass
class Author:
    name: str

@dataclass
class Book:
    title: str
    author: Author
```

```
>>> from example_dataclasses import Book
>>> Book()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 2 required positional arguments: 'title' and 'author'
```

```
author = Author("H.P. Lovecraft")
book = Book("At the Mountains of Madness", author=author)

assert dataclasses.asdict(book) == {
    "author": {"name": "H.P. Lovecraft"},
    "title": "At the Mountains of Madness",
}

assert dataclasses.astuple(book) == (
    "At the Mountains of Madness",
    ("H.P. Lovecraft",),
)
```

Where you might have seen them before

- Kinda like structs in other languages
- attrs: Classes Without Boilerplate
  - http://www.attrs.org/
  - I ❤ attrs
  - Strong inspiration for dataclasses

breakpoint() built in

- `pdb.set_trace()` by any other name
- Magic trick: Can override behaviour using `PYTHONBREAKPOINT` env var.
  - You can even disable breakpoints or convert them to logging actions
  - Makes it easier to plug the same library into an async app, web app or console library with full debugging

```python
def foo(bar):
    ham = {"ham": bar}
    breakpoint()
    return ham

if __name__ == "__main__":
    foo("spam")
```

```
python example_breakpoint.py  # Normal pdb

export PYTHONBREAKPOINT=0
python example_breakpoint.py  # No breakpoints!

export PYTHONBREAKPOINT=pudb.set_trace
python example_breakpoint.py  # uses pudb
```

Module level dunder methods

- Now you can `__getattr__` just like a class 😄
- Great for plugins and decorators which update a module level registry
  - Much more dynamic possibilities too, can enumerate plugins on the fly

```python
def funky(name):
    def func(x):
        return f"{name}: {x}"
    return func

def __getattr__(name):
    return funky(name)

def __dir__():
    return ["ham", "spam", "eggs"]
```

```
>>> import example_dunder_modules
>>> dir(example_dunder_modules)
['eggs', 'ham', 'spam']
>>> example_dunder_modules.ham("foo")
'ham: foo'
>>> example_dunder_modules.spam("foo")
'spam: foo'
```

Typing Speedups and Enhancements

- For all your `def foo(ham: str) -> int` needs.
- `from __future__ import annotations` allows for recursive references (and forward references).
- Speedups too, so imports and usage should be much faster.

This used to go 💥 with a `NameError`

```
from __future__ import annotations  # Implements new behaviour

@dataclass
class Tree:
    left: Tree
    right: Tree

leaf = Tree(None, None)
root = Tree(leaf, None)

assert dataclasses.asdict(root) == {
    "left": {"left": None, "right": None},
    "right": None,
}
```

datetime.fromisoformat()

```
from datetime import datetime

d1 = datetime.utcnow()
# d1 is datetime.datetime(2018, 7, 16, 21, 14, 46, 315714)

s = d1.isoformat()
# d1's isoformat is '2018-07-16T21:14:46.315714'

d2 = datetime.fromisoformat(s)
# parsed into d2: datetime.datetime(2018, 7, 16, 21, 14, 46, 315714)
# d1 == d2? True
```

(Close to my heart: pyiso8601 😜)

context variables, async and await

- `async` and `await` are keywords now (yay!), will break some libraries (boo!).
- context variables are like thread locals for your stack, magically handle asyncio coroutines.
- Combined with enhanced `asyncio` and the new context variables you can make more compact async code.
  - You'll have to take my word for it, I didn't have time to create the nasty pre-3.7 code.

```python
import asyncio

async def hello_world():
    """World's most async Hello World"""
    print("Hello World!")

# Starts event loop, schedules task,
# waits for result, closes loop.
asyncio.run(hello_world())
```

```python
import asyncio, contextvars

message = contextvars.ContextVar("message")

async def read_message(i, msg):
    message.set(msg)  # very contrived example
    await asyncio.sleep(i)  # pretend I'm a HTTP request
    print(f"Message for {i} is {message.get()!r} (originally {msg!r})")

async def main():
    await asyncio.gather(
        read_message(1, "ham"), read_message(3, "spam"),  # Note the order
        read_message(2, "eggs"), read_message(0, "foo"),
    )
    print("main done")

asyncio.run(main())
print("asyncio done")
```

```
$ python example_asyncio.py
Message for 0 is 'foo' (originally 'foo')
Message for 1 is 'ham' (originally 'ham')
Message for 2 is 'eggs' (originally 'eggs')
Message for 3 is 'spam' (originally 'spam')
main done
asyncio done
```

```python
import asyncio, contextvars

client_addr_var = contextvars.ContextVar("client_addr")

async def handle_request(reader, writer):
    client_addr_var.set(writer.transport.get_extra_info("socket").getpeername())

    while True:
        line = await reader.readline()
        if not line.strip():
            break
        writer.write(f"Sleeping for {line}\n".encode())
        print(f"Sleeping for {line} seconds for client {client_addr_var.get()}")
        await asyncio.sleep(float(line))
        writer.write("Awake again\n".encode())

    print(f"Goodbye to client {client_addr_var.get()}")
    writer.write(return f"Good bye, client @ {client_addr_var.get()}\n".encode())
    writer.close()

async def main():
    srv = await asyncio.start_server(handle_request, "127.0.0.1", 8081)
    async with srv:
        await srv.serve_forever()

asyncio.run(main())
```

```
# client 1
$ nc localhost 8081
2
Sleeping for b'2\n'
Awake again

Good bye, client @ ('127.0.0.1', 56967)

# client 2
$ nc localhost 8081
3
Sleeping for b'3\n'
Awake again

Good bye, client @ ('127.0.0.1', 56969)

# server
$ python example_asyncio_tcp.py
Sleeping for b'2\n' seconds for client ('127.0.0.1', 56967)
Sleeping for b'3\n' seconds for client ('127.0.0.1', 56969)
Goodbye to client ('127.0.0.1', 56967)
Goodbye to client ('127.0.0.1', 56969)
```

More Stuff

- More UTF-8:
  - PEP 538: Legacy C Locale Coercion
  - PEP 540: Forced UTF-8 Runtime Mode (`-X utf-8`)
- PEP 539: New C API for Thread-Local Storage
- PEP 564: New Time Functions With Nanosecond Resolution
  - `time.*_ns()`, e.g. `time.time_ns() -> 1531775257989279000`
- PEP 565: Show DeprecationWarning in `__main__`

And More

- Package resources `resources.open_text("module", "filename.txt") -> file`
- PEP 560: Core Support for typing module and Generic Types
- PEP 552: Hash-based .pyc Files
- PEP 545: Python Documentation Translations (en, jp, fr, ko)
- Development Runtime Mode: -X dev
- And lots of module improvements …

Further Reading

- What's New In Python 3.7 — Python 3.7.0 documentation
  - https://docs.python.org/3/whatsnew/3.7.html
- Cool New Features in Python 3.7 – Real Python
  - https://realpython.com/python37-new-features/
- Python 3.7's new builtin breakpoint — a quick tour
  - https://hackernoon.com/python-3-7s-new-builtin-breakpoint-a-quick-tour-4f1aebc444c