What's new in Python ~~2~~3.7?

by Stéphane Wirtel

EuroPython 2018 - Edinburgh - July 25th 2018

I am Stéphane

I live in Belgium
@matrixise

python™ python SOFTWARE FOUNDATION

- **PythonFOSDEM**
- **CPython contributor**
- **#fellow member** of @ThePSF

PEP 537

- PEP 553, Built-in breakpoint()
- PEP 557, Data Classes
- PEP 562, Module `__getattr__` and `__dir__`
- PEP 563, Postponed Evaluation of Annotations
- PEP 564, Time functions with nanosecond resolution
- PEP 565, Show DeprecationWarning in `__main__`
- PEP 567, Context Variables
- PEP 540, UTF-8 mode

https://www.python.org/dev/peps/pep-0537/

PEP 553: Built-in breakpoint()

A wonderful function

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

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

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

A wonderful function

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

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

and of course, you have a crash...

```
Traceback (most recent call last):
  File "bugs.py", line 5, in <module>
    print(divide(a, b))
  File "bugs.py", line 2, in divide
    return f / e
ZeroDivisionError: division by zero
```

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

So, how can I debug it? maybe with `debugger` (hello JS)?

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

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

So, how can I debug it? maybe with `debugger` (hello JS)?

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

No, it is not `debugger`, then what is the name of my debugger and how to add a breakpoint?

- pdb
- ipdb
- pudb
- pdbpp
- ...

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

for example, with pdb

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

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

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

for example, with pdb

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

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

or with pudb

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

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

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

or just with `breakpoint()`

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

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

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

or just with `breakpoint()`

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

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

```
$ python bugs.py
> /tmp/bugs.py(3)divide()
-> return f / e
(Pdb)
```

By default, breakpoint will execute `pdb.set_trace()`

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

but `breakpoint()` can be configured with

- `PYTHONBREAKPOINT=0` disables debugging
- `PYTHONBREAKPOINT=''` or `PYTHONBREAKPOINT=` uses the default pdb.set_trace() function
- `PYTHONBREAKPOINT=callable` uses the specified callable.
  - PYTHONBREAKPOINT=pudb.set_trace

```
$ PYTHONBREAKPOINT=0 python bugs.py
$ PYTHONBREAKPOINT='' python bugs.py
$ PYTHONBREAKPOINT=pudb.set_trace python bugs.py
$ PYTHONBREAKPOINT=IPython.embed python bugs.py
...
```

https://www.python.org/dev/peps/pep-0553/

PEP 553: Built-in breakpoint()

or you can use an other callable: `print()`

```
def divide(e, f):
    breakpoint(e, f, end=' <-END\n')
    return f / e
```

```
PYTHONBREAKPOINT=print python bugs.py
1 0 <-END
Traceback (most recent call last):
  File "bugs.py", line 6, in <module>
    print(divide(a, b))
  File "bugs.py", line 3, in divide
    return f / e
ZeroDivisionError: division by zero
```

https://www.python.org/dev/peps/pep-0553/

PEP 557: Data Classes

Numerous attempts to define classes which exist primarily to store values

Examples

- `collections.namedtuple`
- The `attrs` project (Hi Hynek!)
- with python classes
- ...

PEP 557: Data Classes

With a `tuple`

```
>>> person = ('Stephane', 'Wirtel', 37)
>>> person[0] # is it the firstname or the lastname?
```

PEP 557: Data Classes

With a `tuple`

```
>>> person = ('Stephane', 'Wirtel', 37)
>>> person[0] # is it the firstname or the lastname?
```

With a `dict`

```
>>> person = {'firstname': 'Stephane', 'lastname': 'Wirtel', 'age': 37}
>>> person['firstname'] # would prefer the . notation
```

PEP 557: Data Classes

with `collections.namedtuple`

```
from collections import namedtuple

Person = namedtuple('Person', ['firstname', 'lastname', 'age'])

person = Person('Stephane', 'Wirtel', 37)
person.age
person.firstname

person == Person('Stephane', 'Wirtel', 37)
person == ('Stephane', 'Wirtel', 37)

# but
person.age = 38 # will crash
```

PEP 557: Data Classes

with a `class`

```
class Person:
    def __init__(self, firstname, lastname, age):
        self.firstname = firstname
        self.lastname = lastname
        self.age = age

    def __repr__(self):
        return f'Person(firstname={self.firstname}, lastname={self.lastname})'

    def __eq__(self, o):
        return (self.firstname == o.firstname
                and self.lastname == o.lastname
                and self.age == o.age)

    def __lt__(self, o):
        return (self.firstname < o.firstname
                and self.lastname < o.lastname
                and self.age < o.age)

person = Person('Stephane', 'Wirtel', 37)
person.firstname
>>> person == Person('Stephane', 'Wirtel', 37)
True
>>> person < Person('Stephane', 'Wirtel', 37)
False
```

PEP 557: Data Classes

but now, you can replace the previous example with this dataclass

So, we can use the dataclass

```
from dataclasses import dataclass

@dataclass
class Person:
    firstname: str
    lastname: str
    age: int

person = Person('Stephane', 'Wirtel', 37)
person.firstname
>>> person == Person('Stephane', 'Wirtel', 37)
True
>>> person < Person('Stephane', 'Wirtel', 37)
False
```

PEP 557: Data Classes

default values

```
from dataclasses import dataclass

@dataclass
class Position:
    name: str
    latitude: float = 0.0
    longitude: float = 0.0

>>> europython = Position('Edinburgh')
Position(name='Edinburgh', latitude=0.0, longitude=0.0)
```

PEP 557: Data Classes

define methods/properties

```
from dataclasses import dataclass

@dataclass
class Person:
    firstname: str
    lastname: str
    age: int

    @property
    def fullname(self):
        return f'{self.firstname} {self.lastname}'
```

PEP 557: Data Classes

configuration of the `@dataclass` decorator

- `init=True` => `__init__`
- `repr=True`=> `__repr__`
- `eq=True` => `__eq__`
- `order=False` => `__lt__`, `__le__`, `__gt__`, `__ge__`
- `frozen=False` => `__setattr__`, `__getattr__` ...

PEP 557: Data Classes

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Person:
    firstname: str
    lastname: str
    age: int
```

```
>>> person = Person('Stephane', 'Wirtel', 37)
>>> person.age = 27 # dataclasses.FrozenInstanceError: cannot assign to field 'age
```

PEP 557: Data Classes

Inheritance

```
@dataclass
class Person:
    firstname: str
    lastname: str

@dataclass
class User(Person):
    username: str
    password: str

person = Person('Stephane', 'Wirtel')
user = User('Stephane', 'Wirtel', 'matrixise', 's3cr3t')
```

PEP 562: Module `__getattr__` and `__dir__`

- Customize the access to an attribute of a module, example, a deprecated function

```
# lib
from warnings import warn
deprecated_names = ['old_function']

def _deprectated_old_function(arg, other):
    ...

def __getattr__(name: str) -> Any:
    if name in deprecated_names:
        warn(f'{name} is deprecated', DeprecationWarning)
        return globlals()[f'_deprectated_{name}']
    raise AttributeError(f'module {__name__} has no attribute {name}')
```

PEP 562: Module `__getattr__` and `__dir__`

- Customize the access to an attribute of a module, example, a deprecated function

```
# lib
from warnings import warn
deprecated_names = ['old_function']

def _deprectated_old_function(arg, other):
    ...

def __getattr__(name: str) -> Any:
    if name in deprecated_names:
        warn(f'{name} is deprecated', DeprecationWarning)
        return globlals()[f'_deprectated_{name}']
    raise AttributeError(f'module {__name__} has no attribute {name}')
```

```
# main.py
from lib import old_function # Works, but emits the warning
```

PEP 562: Module `__getattr__` and `__dir__`

- Customize the listing of the items from a module

```
# lib.py
deprecated_names = ['old_function']

__all__ = ['new_function_one', 'new_function_two']

def new_function_one(arg, other):
    ...

def new_function_two(arg, other):
    ...

def __dir__() -> List[str]:
    return sorted(__all__ + deprecated_names)
```

PEP 562: Module `__getattr__` and `__dir__`

- Customize the listing of the items from a module

```
# lib.py
deprecated_names = ['old_function']

__all__ = ['new_function_one', 'new_function_two']

def new_function_one(arg, other):
    ...

def new_function_two(arg, other):
    ...

def __dir__() -> List[str]:
    return sorted(__all__ + deprecated_names)
```

```
# main.py
import lib

dir(lib) # prints ['new_function_one', 'new_function_two', 'old_function', ...]
```

PEP 563: Postponed Evaluation of Annotations

Without the annotations

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

- PEP 484 -- Type Hints
- PEP 526 -- Syntax for Variable Annotations
- https://www.python.org/dev/peps/pep-0563

PEP 563: Postponed Evaluation of Annotations

Without the annotations

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

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

- PEP 484 -- Type Hints
- PEP 526 -- Syntax for Variable Annotations
- https://www.python.org/dev/peps/pep-0563

PEP 563: Postponed Evaluation of Annotations

Without the annotations

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

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

Problem with the forward reference, for that, we have to define the class like that

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

- PEP 484 - Type Hints
- PEP 526 - Syntax for Variable Annotations
- https://www.python.org/dev/peps/pep-0563

PEP 563: Postponed Evaluation of Annotations

Now, with the annotations

```
from __future__ import annotations

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

and we do not need a forward reference.

- PEP 484 -- Type Hints
- PEP 526 -- Syntax for Variable Annotations
- https://www.python.org/dev/peps/pep-0563

PEP 563: Postponed Evaluation of Annotations

With the annotations and the dataclasses

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Optional

@dataclass
class Node:
    left: Optional[Node] = None
    right: Optiona[Node] = None

root = Tree()
print(root)
```

- PEP 484 -- Type Hints
- PEP 526 -- Syntax for Variable Annotations
- https://www.python.org/dev/peps/pep-0563

PEP 564: Time functions with nanosecond resolution

Float type limited to 104 days

The Python `time.time()` function returns the current time as a floating-point numberself. Limited to 64 bits and in the IEEE 754 format. With this limitation, the `float` type starts to lose nanoseconds after 104 days.

-> Precision loss

Example

On Python microbenchmarks, it's common to see function calls taking less than 100ns. A difference of a few nanoseconds might become significant.

https://www.python.org/dev/peps/pep-0564

PEP 564: Time functions with nanosecond resolution

- `time.clock_gettime_ns`
- `time.clock_settime_ns`
- `time.monotonic_ns`
- `time.perf_counter_ns`
- `time.process_time_ns`
- `time.time_ns`

https://www.python.org/dev/peps/pep-0564

PEP 564: Time functions with nanosecond resolution

- `time.clock_gettime_ns`
- `time.clock_settime_ns`
- `time.monotonic_ns`
- `time.perf_counter_ns`
- `time.process_time_ns`
- `time.time_ns`

```
>>> time.monotonic()
154630.068913333
```

https://www.python.org/dev/peps/pep-0564

PEP 564: Time functions with nanosecond resolution

- `time.clock_gettime_ns`
- `time.clock_settime_ns`
- `time.monotonic_ns`
- `time.perf_counter_ns`
- `time.process_time_ns`
- `time.time_ns`

```
>>> time.monotonic()
154630.068913333
```

```
>>> time.monotonic_ns()
154631543907219
```

https://www.python.org/dev/peps/pep-0564

PEP 565: Show DeprecationWarning in **main**

Since Python 3.2, the `DeprecationWarning` was hidden by default, but there was a side effect, because these warning were only visible then running tests, which meant that developers could be surprised by breaking changes in the APIs they used.

PEP 565: Show DeprecationWarning in **main**

Since Python 3.2, the `DeprecationWarning` was hidden by default, but there was a side effect, because these warning were only visible then running tests, which meant that developers could be surprised by breaking changes in the APIs they used.

- `FutureWarning`: always displayed by default
- `DeprecationWarning`: displayed by default only in `__main__` and when running tests.
- `PendingDeprecationWarning`: diplayed by default only when running tests.

PEP 567: Context Variables

For a synchronous environment, TLS is perfect but not for an asynchronous environment, which in this case, the asynchronous tasks execute concurrently in the same OS thread.

This concept is similar to the TLS, but allows correctly keeping track of values per asynchronous task.

- one new module: `contextvars`
- two objects: `Context` and `ContextVar`

PEP 567: Context Variables

```
# demo.py
import contextvars

var = contextvars.ContextVar('var', default=42)
print('without context', var.get())

var.set('spam')

def main():
    print('inside', var.get())
    var.set('ham')
    print('inside', var.get())

# create a copy of the context (associated to the OS Thread)
ctx = contextvars.copy_context()
ctx.run(main)
print('outside', var.get())
```

```
$ python demo.py
42
inside spam
inside ham
outside spam
```

PEP 567: Context Variables

```
# in this example, asyncio will handle the context for us
import contextvars

current_user = contextsvars.ContextVar('current_user', default=None)

async def inner():
    log.debug('Current User: %s', current_user.get())

@routes.get('/{name}')
async def handler(request):
    current_user.set(request.match_info['name'])
    await inner()
```

PEP 545: Python documentation translations

You can find your favourite documenation in:

- English ;-)
- French
- Japanese
- Korean

Not in PEPs

"Dict keeps insertion order" is the ruling!

Thanks to INADA Naoki

https://mail.python.org/pipermail/python-dev/2017-December/151283.html

`async` & `await`

are

keywords!

`async` & `await` are **keywords!**

For `async`

```
async = True
```

```
$ python /tmp/test_async.py
  File "/tmp/test_async.py", line 1
    async = True
          ^
SyntaxError: invalid syntax
```

and `await`

```
def await():
    pass
```

```
$ python /tmp/test_await.py
File "test_await.py", line 1
  def await():
          ^
SyntaxError: invalid syntax
```

`asyncio` has a lot of improvements

Before, when you wanted to execute a coroutine for `asyncio`, you needed to create a loop, etc...

```
# before
import asyncio

async def hello_world():
    print('hello world')

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(hello_world())
finally:
    loop.close()
```

`asyncio` has a lot of improvements

Before, when you wanted to execute a coroutine for `asyncio`, you needed to create a loop, etc...

```
# before
import asyncio

async def hello_world():
    print('hello world')

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(hello_world())
finally:
    loop.close()
```

Welcome to the `asyncio.run` function

```
# hello asyncio.run()
import asyncio

async def hello_world():
    print('hello world')

asyncio.run(hello_world())
```

`asyncio` has a lot of improvements

New functions

- `asyncio` has the support for `contextvars`
- `asyncio.create_task()`: shortcut to `asyncio.get_event_loop().create_task()`
- `loop.start_tls()`: upgrade the existing connection to TLS
- `asyncio.current_task()`: returns the current task
- `asyncio.all_tasks()`: return all the tasks of a loop
- `loop.sock_sendfile()`: use the `os.sendfile` (when possible + performance)
- ...

`asyncio` has a lot of improvements

Performance improvements

- `asyncio.get_event_loop` has been implemented in C (+- 15 times faster).
- `asyncio.Future` callback managmenet has been optimized
- `asyncio.gather()` - 15% faster
- `asyncio.sleep()` - 2x faster
- ...

https://docs.python.org/3.7/whatsnew/3.7.html#asyncio

Gifts for the developers

```
> python -X importtime -c 'import asyncio'
import time: self [us] | cumulative | imported package
import time:       381 |        381 | zipimport
import time:      2044 |       2044 | _frozen_importlib_external
import time:       197 |        197 |     _codecs
import time:      1515 |       1712 |   codecs
...
```

Gifts for the developers

```
> python -X utf8 your_script.py
> PYTHONUTF8=1 python your_script.py
```

- will enable the UTF-8 mode
- **disabled by default**
- but automatically enabled if we use the "POSIX" locale

if enabled:

- Python will use the utf-8 encoding, regardless the locale of the current platform.
- `sys.getfilesystemencoding()` returns `UTF-8`
- `locale.getpreferredencoding()` return `UTF-8`
- `sys.stdin` & `sys.stdout` error handlers to `surrogateescape` (avoid a `UnicodeDecodeError`)

Gifts for the developers

```
> python -X dev
```

- enable the "development mode"
- add additional runtime checks
- install debug hooks for the memory allocators
- enable the `faulthandler` module for a beautiful Python dump on a crash ;-)
- enable the `asyncio` debug mode

`importlib.resources`

- Allows to load a binary artifact that is shipped within a package.
- Provides functionality similar to `pkg_resources` but without all of the overhead and performance problems of `pkg_resources`
- a Resource can live on the File System, a zip file or anywhere.

`importlib.resources`

- Allows to load a binary artifact that is shipped within a package.
- Provides functionality similar to `pkg_resources` but without all of the overhead and performance problems of `pkg_resources`
- a Resource can live on the File System, a zip file or anywhere.

My package

```
email/
├── __init__.py
└── tests
    ├── data
    │   ├── __init__.py
    │   └── message.eml
    └── __init__.py
```

`importlib.resources`

Get the content of `email/tests/data/message.eml` without `importlib.resources`

```
data_dir = os.path.join(os.path.dirname(__file__), 'tests', 'data')
data_path = os.path.join(data_dir, 'message.eml')
with open(data_path, encoding='utf-8') as fp:
    eml = fp.read()
```

`importlib.resources`

Get the content of `email/tests/data/message.eml` without `importlib.resources`

```
data_dir = os.path.join(os.path.dirname(__file__), 'tests', 'data')
data_path = os.path.join(data_dir, 'message.eml')
with open(data_path, encoding='utf-8') as fp:
    eml = fp.read()
```

or

```
dirname = pathlib.Path(__file__).parent
data_path = dirname / 'tests' / 'data' / 'message.eml'
eml = data_path.read_bytes()
```

`importlib.resources`

with `importlib.resources`

```
from importlib import resources
eml = resources.read_binary('email.tests.data', 'message.eml')
```

`importlib.resources`

with `importlib.resources`

```
from importlib import resources
eml = resources.read_binary('email.tests.data', 'message.eml')
```

or with a context manager

```
from importlib import resources
with resources.path('email.tests.data', 'message.eml') as eml:
    print(eml.read_binary())
```

`importlib.resources`

with `importlib.resources`

```
from importlib import resources
eml = resources.read_binary('email.tests.data', 'message.eml')
```

or with a context manager

```
from importlib import resources
with resources.path('email.tests.data', 'message.eml') as eml:
    print(eml.read_binary())
```

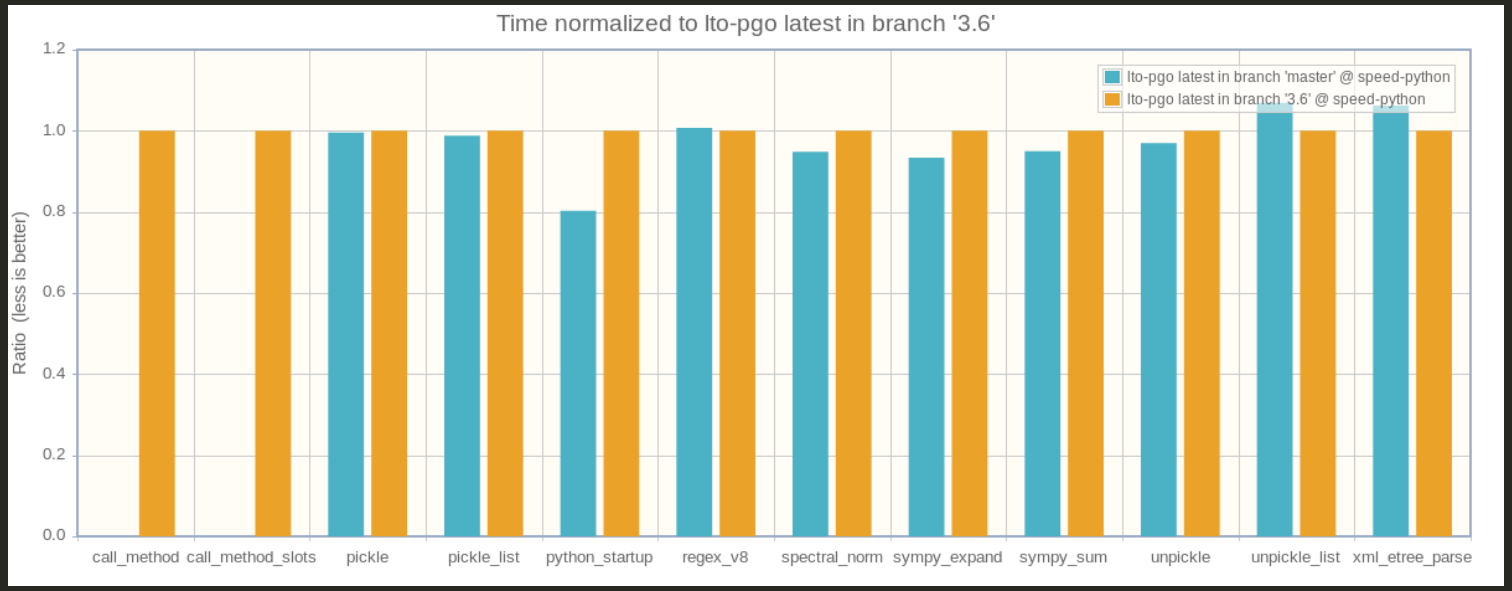
or

```
import email.tests.data
with resources.parth(email.tests.data, 'message.eml') as eml:
    print(eml.read_binary())
```

- https://docs.python.org/fr/3.7/library/importlib.html
- https://www.youtube.com/watch?v=ZsGFU2qh73E

and many improvements, bugfixes...

Performances



Conclusion

Python 3.7 is a great vintage!

[mgx.io]

DEVELOPMENT EXPERT

stephane.wirtel@mgx.io
@matrixixe