Co nowego w Pythonie 3.6?

PyGda #21

- Python 3.5.0 – 13 Wrzesień 2015
- Python 3.6.0 – 23 Grudzien 2016

Nowa implementacja słownika

- Implementacja zaczerpnięta z PyPy
- Optymalizacja zużycia pamięci
- Deklarowana oszczędność pamięci na poziomie 20-25%
- Bez zmian algorytm hashujący

“The order-preserving aspect of this new implementation is considered an implementation detail and should not be relied upon.”

```
For example, the dictionary:

    d = {'timmy': 'red', 'barry': 'green', 'guido': 'blue'}

is currently stored as:

    entries = [['--', '--', '--'],
               [-8522787127447073495, 'barry', 'green'],
               ['--', '--', '--'],
               ['--', '--', '--'],
               ['--', '--', '--'],
               [-9092791511155847987, 'timmy', 'red'],
               ['--', '--', '--'],
               [-6480567542315338377, 'guido', 'blue']]

Instead, the data should be organized as follows:

    indices =  [None, 1, None, None, None, 0, None, 2]
    entries =  [[-9092791511155847987, 'timmy', 'red'],
                [-8522787127447073495, 'barry', 'green'],
                [-6480567542315338377, 'guido', 'blue']]
```

- https://mail.python.org/pipermail/python-dev/2012-December/123028.html

Formatted string literals

- Nowy sposób formatowania stringów (kolejny?)

```
a = 10
b = 5
print(f'Value a is {a}, value b is {b}')

>>>Value a is 10, value b is 5
```

Numeric literals separation

```
some_val = 1110_1000_1000_0000
print(some_val)

>>> 1110100010000000
```

Type annotations dla zmiennych

```
primes: List[int] = []
captain: str   # no initial value!
class Starship:
        stats: Dict[str, int] = {}
```

Nowa metoda __init_subclass__

```
class Meta(type):
    def __new__(cls, clsname, superclasses, attributedict):
        print('Launching Meta __new__ ', cls.__name__)
        return type.__new__(cls, clsname, superclasses, attributedict)

    def __init__(cls, clsname, superclasses, attributedict):
        print('Launching Meta __init__ ', cls.__name__)
        super().__init__(clsname, superclasses, attributedict)

    @classmethod
    def __prepare__(metacls, name, bases):
        print('Launching Meta __preapre__', metacls.__name__)
        return {}

    def __call__(self, *args, **kwargs):
        print('Launching Meta __call__ of ', self.__class__.__name__)
        return super().__call__(*args, **kwargs)
```

```python
class PluginBase(metaclass=Meta):
    subclasses = []

    def __new__(cls, *args, **kwargs):
        print('Launching Base Class __new__')
        return super().__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        print('Launching Base Class __init__')
        super().__init__(*args, **kwargs)

    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        print('Launching Base Class __init_subclass__ ,adding value attribute')
        cls.value = None
        cls.subclasses.append(cls)

    def __call__(self, *args, **kwargs):
        print('Launching Base Class __call__')
        return super().__call__(*args, **kwargs)
```

```python
class Plugin1(PluginBase):
    def __new__(cls, *args, **kwargs):
        print('Launching __new__ of', cls.__name__)
        return super().__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        print('Launching __init__ of', self.__class__.__name__)
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        print('Launching __call__ of', self.__class__.__name__)
        return super().__call__(*args, **kwargs)
```

Launching Meta __preapre__ Meta
Launching Meta __new__ Meta
Launching Meta __init__ PluginBase
Launching Meta __preapre__ Meta
Launching Meta __new__ Meta
Launching Base Class __init_subclass__ ,adding value attribute
Launching Meta __init__ Plugin1
Launching Meta __call__ of Meta
Launching Base Class __new__
Launching Base Class __init__
Launching Meta __call__ of Meta
Launching __new__ of Plugin1
Launching Base Class __new__
Launching __init__ of Plugin1
Launching Base Class __init__

__set_name__ w deskryptorach

```
class IntField:
    def __get__(self, instance, owner):
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise ValueError(f'expecting integer in {self.name}')
        instance.__dict__[self.name] = value

    # this is the new initializer:
    def __set_name__(self, owner, name):
        """
        output: <__main__.IntField object at 0x7fe81b5dcf28> <class '__main__.Model'> int_field
        """
        print(self, owner, name)
        self.name = name


class Model:
    int_field = IntField()
```

Async generators

PEP492 introduced support for native coroutines and `async` / `await` syntax to Python 3.5. A notable limitation of the Python 3.5 implementation is that it was not possible to use `await` and `yield` in the same function body. In Python 3.6 this restriction has been lifted, making it possible to define *asynchronous generators*:

```
async def ticker(delay, to):
    """Yield numbers from 0 to *to* every *delay* seconds."""
    for i in range(to):
        yield i
        await asyncio.sleep(delay)
```

Dzieki za uwagę

info@py.gda.pl

www.facebook.com/pygda

meetup.com/PyGda-pl/