

# Python3 vs Python2

# Самое важное

Python 3	Python 2
<code>print('Hello world')</code>	<code>print 'Hello world'</code>
<code>2/3 = 0.66</code>	<code>2/3 = 0</code>
Есть типы <code>str</code> для Unicode, <code>bytes/bytearray</code>	Есть типы <code>str</code> для ASCII, <code>unicode</code>
<code>raise ValueError('error')</code>	<code>raise ValueError, 'error message'</code>
Глобальные переменные не меняются в цикле	Глобальные переменные меняются в цикле
<code>input()</code> для получения текста	<code>raw_input()</code> для получения текста
<code>zip</code> , <code>map</code> , <code>filter</code> , <code>dict.keys()</code> , <code>dict.values()</code> , <code>dict.items()</code> возвращают итераторы	<code>zip</code> , <code>map</code> , <code>filter</code> , <code>dict.keys()</code> , <code>dict.values()</code> , <code>dict.items()</code> возвращают списки

<https://docs.python.org/3/howto/pyporting.html>

## `__future__`

- `from __future__ import print_function`
- `from __future__ import division`

# `__future__`

feature	optional in	mandatory in	effect
nested_scopes	2.1.0b1	2.2	<a href="#">PEP 227</a> : <i>Statically Nested Scopes</i>
generators	2.2.0a1	2.3	<a href="#">PEP 255</a> : <i>Simple Generators</i>
division	2.2.0a2	3.0	<a href="#">PEP 238</a> : <i>Changing the Division Operator</i>
absolute_import	2.5.0a1	3.0	<a href="#">PEP 328</a> : <i>Imports: Multi-Line and Absolute/Relative</i>
with_statement	2.5.0a1	2.6	<a href="#">PEP 343</a> : <i>The “with” Statement</i>
print_function	2.6.0a2	3.0	<a href="#">PEP 3105</a> : <i>Make print a function</i>
unicode_literals	2.6.0a2	3.0	<a href="#">PEP 3112</a> : <i>Bytes literals in Python 3000</i>

# Six

## **six.advance\_iterator(*it*)**

Get the next item of iterator *it*. `StopIteration` is raised if the iterator is exhausted. This is a replacement for calling `it.next()` in Python 2 and `next(it)` in Python 3. Python 2.6 and above have a builtin `next` function, so six's version is only necessary for Python 2.5 compatibility.

## **six.callable(*obj*)**

Check if *obj* can be called. Note `callable` has returned in Python 3.2, so using six's version is only necessary when supporting Python 3.0 or 3.1.

## **six.iterkeys(*dictionary*, \*\**kwargs*)**

Returns an iterator over *dictionary*'s keys. This replaces `dictionary.iterkeys()` on Python 2 and `dictionary.keys()` on Python 3. *kwargs* are passed through to the underlying method.

## **six.itervalues(*dictionary*, \*\**kwargs*)**

Returns an iterator over *dictionary*'s values. This replaces `dictionary.itervalues()` on Python 2 and `dictionary.values()` on Python 3. *kwargs* are passed through to the underlying method.

## **six.iteritems(*dictionary*, \*\**kwargs*)**

Returns an iterator over *dictionary*'s items. This replaces `dictionary.iteritems()` on Python 2 and `dictionary.items()` on Python 3. *kwargs* are passed through to the underlying method.

## **six.iterlists(*dictionary*, \*\**kwargs*)**

Calls `dictionary.iterlists()` on Python 2 and `dictionary.lists()` on Python 3. No builtin Python mapping type has such a method; this method is intended for use with multi-valued dictionaries like Werkzeug's. *kwargs* are passed through to the underlying method.

# Singleton

- Singleton – шаблон проектирования, описывающий объект, существующий в единственном экземпляре.
- Пример: конфигурация приложения или логгер

# Singleton

- Singleton – шаблон проектирования, описывающий объект, существующий в единственном экземпляре.
- Примеры: конфигурация приложения, логгер

# Метаклассы!

```
class SingletonMeta(type): # не thread-safe реализация синглтона
    _instance = None

    def __call__(self):
        if self._instance is None:
            self._instance = super().__call__()
        return self._instance

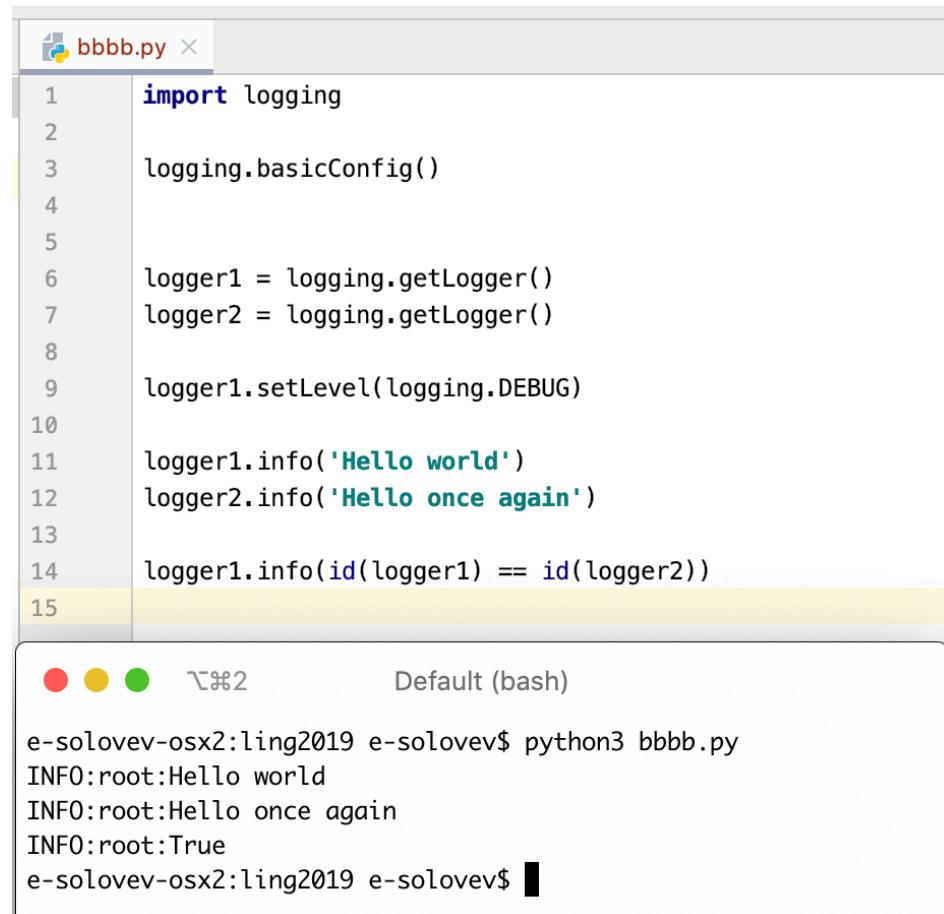
class Config(metaclass=SingletonMeta):
    def __init__(self):
        # здесь вообще-то нужно загружать конфиг из файла или откуда-то ещё
        self.__postgres_url = 'postgresql://my_user:verystrongpassword@postgres.superstartup.com:5432/production'

config1 = Config()
config2 = Config()

print(config1 == config2)
```



# Логгер



The image shows a code editor window with a file named `bbbb.py` and a terminal window below it. The code in the editor demonstrates the use of the `logging` module. It imports `logging`, configures basic logging, creates two logger objects (`logger1` and `logger2`), sets the level of `logger1` to `DEBUG`, and then logs messages from both loggers. The terminal window shows the output of running `python3 bbbb.py`, which displays the log messages as expected.

```
1 import logging
2
3 logging.basicConfig()
4
5
6 logger1 = logging.getLogger()
7 logger2 = logging.getLogger()
8
9 logger1.setLevel(logging.DEBUG)
10
11 logger1.info('Hello world')
12 logger2.info('Hello once again')
13
14 logger1.info(id(logger1) == id(logger2))
15
```

Terminal output:

```
e-solovev-osx2:ling2019 e-solovev$ python3 bbbb.py
INFO:root:Hello world
INFO:root:Hello once again
INFO:root:True
e-solovev-osx2:ling2019 e-solovev$
```