Advanced Python Programming Course

Lecture 4.

Linters. OOP in Python: Classes and Instances

Assoc. Prof. Kovalenko S.M.
Department of software engineering and intelligent management technologies,
NTU “KhPI”

Black

- *Black* is uncompromising the Python code formatter.
- Installing Black

```
pip install black
```

- To format Jupyter Notebooks, install with

```
pip install "black[jupyter]“
```

- Using

```
black {source_file_or_directory}
```

Code for testing `Black`

```
import math
def Some_Function(arg1, arg2 = 3):
    b=math.cos(math.pi)+10
    c=b*arg1
    return c/arg2
def Some_Function2(a1):
    return a1+a1*2

Some_Function(4,5)
Some_Function2(5)

>>> black black_test.py
```

Code for testing `Black`. Result

```
import math


def Some_Function(arg1, arg2=3):
    b = math.cos(math.pi) + 10
    c = b * arg1
    return c / arg2


def Some_Function2(a1):
    return a1 + a1 * 2


Some_Function(4, 5)
Some_Function2(5)
```

Black test with Jupyter notebooks

In [45]:
```
import math
def Some_Function(arg1, arg2 = 3):
    b=math.cos(math.pi)+10
    c=b*arg1
    return c/arg2
def Some_Function2(a1):
    return a1+a1*2
Some_Function(4,5)
Some_Function2(5)
```

Out[45]: 15

```
>>> black black_test.ipynb
```

```
D:\Света\Python\Advanced\2023\2>black examples.ipynb
reformatted examples.ipynb

All done! □ □□ □
1 file reformatted.
```

In [45]:
```
import math


def Some_Function(arg1, arg2=3):
    b = math.cos(math.pi) + 10
    c = b * arg1
    return c / arg2


def Some_Function2(a1):
    return a1 + a1 * 2


Some_Function(4, 5)
Some_Function2(5)
```

Out[45]: 15

Linters

- Linters are programs that advise about code quality by displaying warnings and errors. They can detect your Python code mistakes, notice invalid code patterns and find elements that do not follow your conventions. Python linters have a number of advantages, such as:
  - Preventing bugs in a project.
  - Making code readable for any python programmer.
  - Catching unneeded pieces of code.
  - Making code cleaner and less complex.
- Of course, every approach has its disadvantages:
  - Linters can return false positives.
  - This process can be time-consuming.
  - Some errors could be overlooked.

Linters

- Pylint (version 2.12.1)
- Flake8 (version 4.0.1)
- PyFlakes (version 2.4.0)

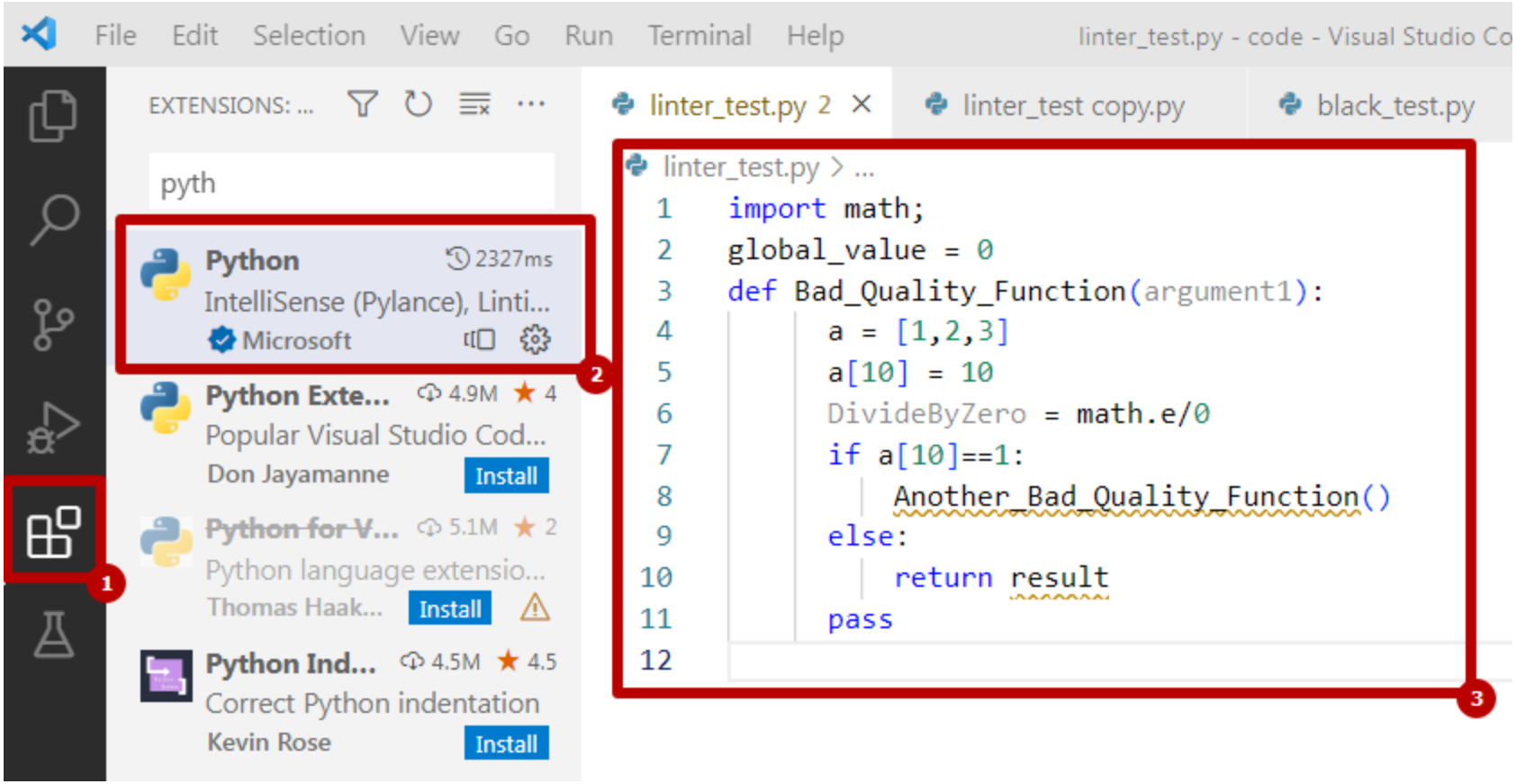
Example of bad code

```
import math;
global_value = 0
def Bad_Quality_Function(argument1):
      a = [1,2,3]
      a[10] = 10
      DivideByZero = math.e/0
      if a[10]==1:
          Another_Bad_Quality_Function()
      else:
          return result
      pass
```

Problems

- unnecessary semicolon in **import math**;
- missing newlines and whitespaces
- missing docstring
- unused **global_value** and **argument1**
- incorrect naming style
- selecting element from the list with incorrect index
- dividing by zero expression
- calling method that doesn’t exist
- returning undefined value
- unnecessary **pass** statement

Using default Python linter



Using pylint

- `pip install pylint`
- `pylint <file_name.py>`

```
------------------------------------------------------------------
Your code has been rated at 0.00/10 (previous run: 0.00/10, +0.00)
```

Using pylint

linter_test.py 2 × | linter_test copy.py | black_test.py | class1.py | black_test copy

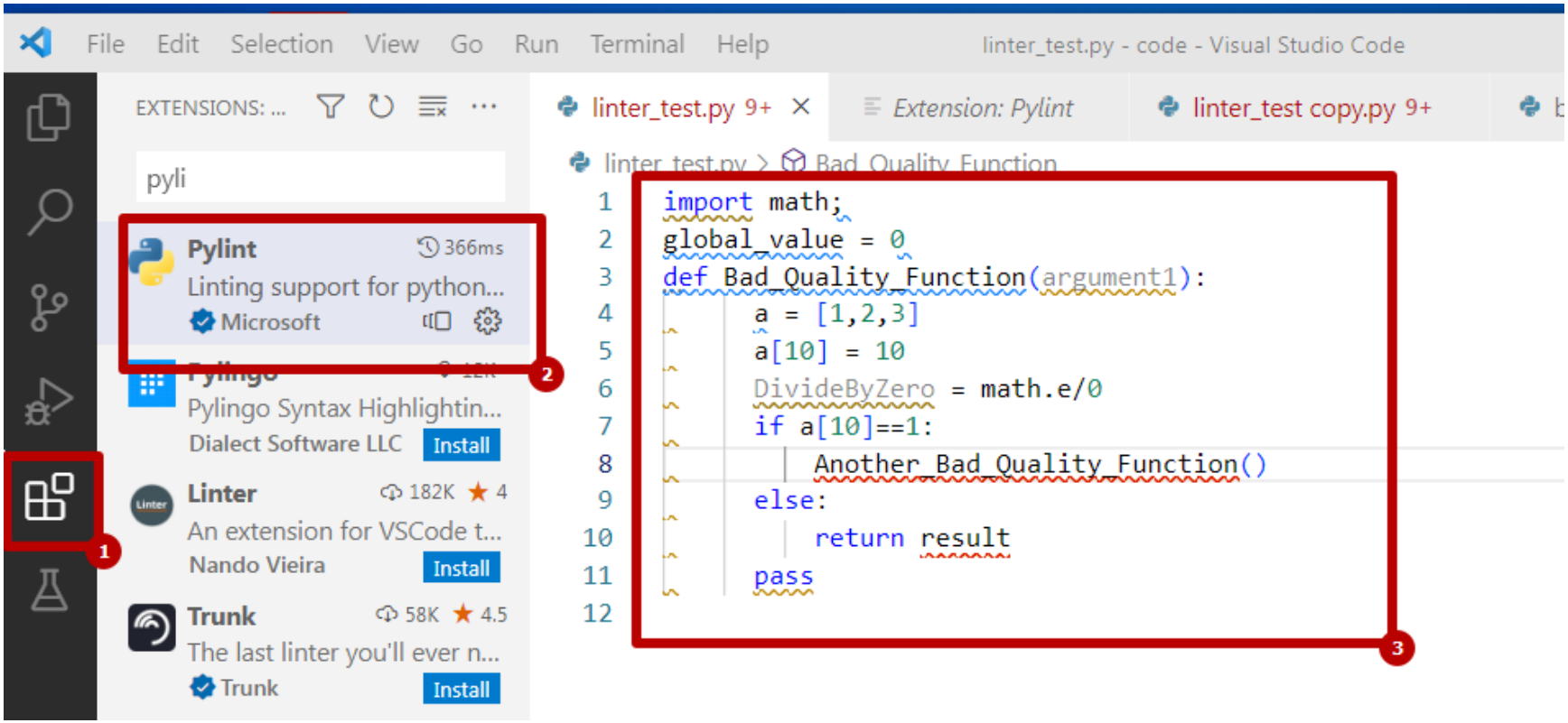
linter_test.py > Bad_Quality_Function

```python
import math;
global_value = 0
def Bad_Quality_Function(argument1):
    a = [1,2,3]
    a[10] = 10
    DivideByZero = math.e/0
    if a[10]==1:
        Another_Bad_Quality_Function()
    else:
```

TERMINAL | OUTPUT | DEBUG CONSOLE | PROBLEMS 2 | cmd

```
D:\Света\Python\Advanced\code>pylint linter_test.py
************* Module linter_test
linter_test.py:1:12: C0303: Trailing whitespace (trailing-whitespace)
linter_test.py:1:0: W0301: Unnecessary semicolon (unnecessary-semicolon)
linter_test.py:2:16: C0303: Trailing whitespace (trailing-whitespace)
linter_test.py:4:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:5:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:6:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:7:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:8:0: W0311: Bad indentation. Found 10 spaces, expected 8 (bad-indentation)
linter_test.py:9:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:10:0: W0311: Bad indentation. Found 10 spaces, expected 8 (bad-indentation)
linter_test.py:11:0: W0311: Bad indentation. Found 6 spaces, expected 4 (bad-indentation)
linter_test.py:1:0: C0114: Missing module docstring (missing-module-docstring)
linter_test.py:2:0: C0103: Constant name "global_value" doesn't conform to UPPER_CASE naming style
```

Using pylint in VSCode



pylint with Jupyter notebook

- `pip install nbqa pylint`
- `nbqa pylint my_notebook.ipynb`

```
D:\Света\Python\Advanced\2023\2>nbqa pylint examples.ipynb
************* Module examples
examples.ipynb:cell_2:4:0: C0103: Function name "Some_Function" doesn't conform
to snake_case naming style (invalid-name)
examples.ipynb:cell_2:8:4: C0103: Variable name "b" doesn't conform to snake_cas
e naming style (invalid-name)
examples.ipynb:cell_2:9:4: C0103: Variable name "c" doesn't conform to snake_cas
e naming style (invalid-name)
examples.ipynb:cell_2:13:0: C0103: Function name "Some_Function2" doesn't confor
m to snake_case naming style (invalid-name)
examples.ipynb:cell_7:4:0: C0116: Missing function or method docstring (missing-
function-docstring)
examples.ipynb:cell_9:1:0: C0116: Missing function or method docstring (missing-
function-docstring)
examples.ipynb:cell_9:1:0: C0103: Function name "Mult_by_two" doesn't conform to
 snake_case naming style (invalid-name)
examples.ipynb:cell_10:1:0: E0602: Undefined variable 'mult_by_two' (undefined-v
ariable)
examples.ipynb:cell_12:1:0: C0116: Missing function or method docstring (missing
-function-docstring)

------------------------------------
Your code has been rated at 3.81/10
```

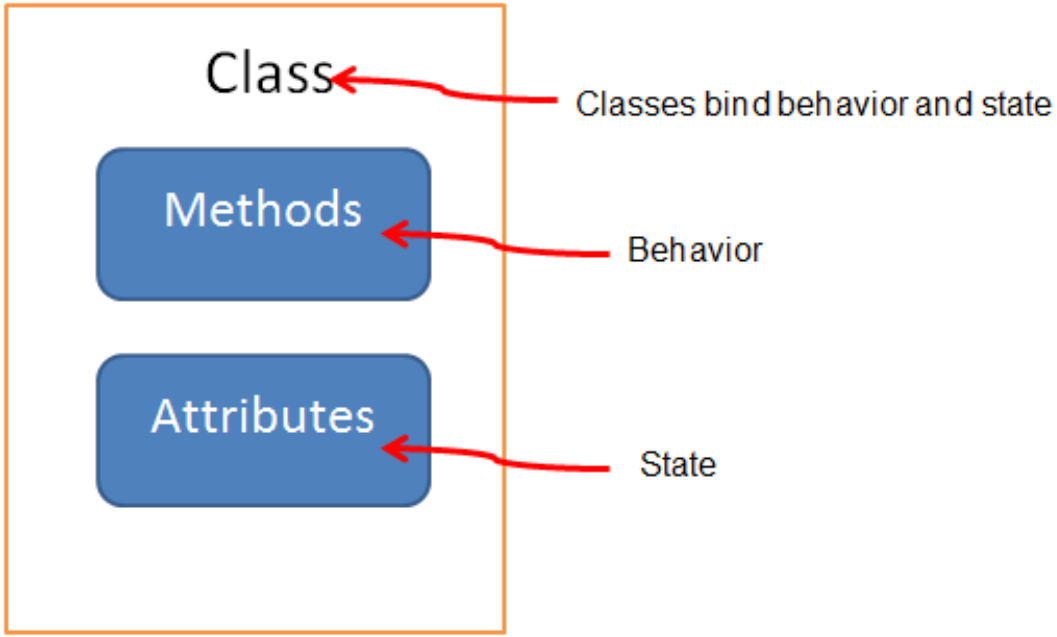
Python classes

- **Python is a multi-paradigm programming language**, which means it supports different programming approach. One different way to program in Python is *object-oriented programming (OOP)*.

- **Note:** you do not have to use OOP when programming in Python. You can still write very powerful programs using the *procedure-oriented programming (POP)*. That said, the POP is good for simple and small programs, while the OOP is better suited for large programs.

Python classes

- The **object-oriented programming** breaks the programming task into objects, which combine data (known as *attributes*) and behaviors/functions (known as *methods*).



Classes and instances

- A **class** defines a set of attributes that describe the state of the object, as well as a set of methods that define the object's behavior. The attributes and methods of a class are encapsulated together, which means that they are kept together as a single unit and can be reused in different parts of a program.
- When you create an instance of a class, you are creating a new object that has its own set of attributes and methods, but is based on the same blueprint as the original class.
- Each instance of a class is independent of the others, meaning that changes made to one instance do not affect the others.

Creating a class

```
class Person:
    pass

mark = Person()
john = Person()

print(mark)
print(john)
```

```
<__main__.Person object at 0x050110E8>
<__main__.Person object at 0x05011628>
```

Instance variables

```
class Person:

    def __init__(self, name, last_name):
        self.name = name
        self.last_name = last_name
        self.email = f'{name}.{last_name}.@khpi.edu.ua'


mark = Person('Mark', 'Darcy')
john = Person('John', 'Thornton')
mary = Person('Mary', 'Shelley')

print(mark.email)
print(john.email)
print(mary.email)
```

```
Mark.Darcy.@khpi.edu.ua
John.Thornton.@khpi.edu.ua
Mary.Shelley.@khpi.edu.ua
```

Instance methods

```
class Person:

    def __init__(self, name, last_name):
        self.name = name
        self.last_name = last_name
        self.email = f'{name}.{last_name}.@khpi.edu.ua'

    def fullname(self):
        return f'{self.name} {self.last_name}'

mark = Person('Mark', 'Darcy')
john = Person('John', 'Thornton')
mary = Person('Mary', 'Shelley')

print(mark.fullname())
print(mary.fullname())
print(john.fullname())
```

```
Mark Darcy
Mary Shelley
John Thornton
```