Writing Good Python
But Python is already great!!!

- Readability
- Massive ecosystem – Libraries, Frameworks and Tools
- Vibrant community
- Many other things …
“Writing” Good Python
Hello!

I am Prashant

Currently a Software Engineer at HubSpot
Maintainability

\[ M \approx \frac{1}{(T) \times (R)} \]

- M = Maintainability
- T = Amount of time it takes a developer to make a change
- R = Risk that change will break something.
- Cannot be strictly defined
- Can be judged by readability, coupling, consistency etc.
```python
import os
import datetime
import sys

class calculator:
    def __init__(self):
        self.last_result = 0;
        pass

    def add(self, NUM1, NUM2):
        """""""""""""
        result = NUM1 + NUM2
        self.last_result = result
        return result

    def SUB(self, x, y):
        result = x - y
        self.last_result = result
        return result

    def Div(self, num1, num2):
        if not not (0 == num2):
            return 0
        result = num1 / num2
        self.last_result = result
        return result

def mul(self, num1, num2):
    # fixme
    ans = num1 * num2
    self.last_result = num1 * num2
    return num1 * num2

result_template = "Result is:
last_result_template = "Last result was:

if __name__ == "__main__":
calc = calculator()
try:
    print(result_template, calc.add(1, 2))
    print(result_template, calc.add(1, 2))
    print(result_template, calc.last_result)
    print(result_template, calc.sub(1, 2))
except:
    print("Maximum value possible or number:","", sys.maxint)
    print("Something bad happened!")
```
Python Style Guide

- Batteries included principle
- PEP 8 - https://www.python.org/dev/peps/pep-0008/
- PEP 257 - https://www.python.org/dev/peps/pep-0257/
pip install pylint

Coding standards

Error detection

Refactoring – Duplicated code

Customizable – Configure which errors/conventions are important using pylintrc file. Can write plugins to add a personal feature

https://www.pylint.org/
$ python -m pylint bad_example.py

************* Mobile bad_example
bad_example.py:23: W391: Unnecessary semicolon (unnecessary-semicolons)
bad_example.py:23: C0125: Unnecessary parentheses after 'not' keyword (superfluous-parentheses)
bad_example.py:30: W0611: 'f tame' (fame)
bad_example.py:11: C0114: Missing module docstring (missing-module-docstring)
bad_example.py:6: C0103: Class name "calculator" doesn't conform to PascalCase naming style (invalid-name)
bad_example.py:6: C0115: Missing class docstring (missing-class-docstring)
bad_example.py:9: W0107: Unnecessary pass statement (unnecessary-pass)
bad_example.py:11: C0103: Argument name "num1" doesn't conform to snake_case naming style (invalid-name)
bad_example.py:11: C0103: Argument name "num2" doesn't conform to snake_case naming style (invalid-name)
bad_example.py:11: C0112: Empty method docstring (empty-docstring)
bad_example.py:17: C0103: Method name "div" doesn't conform to make_case naming style (invalid-name)
bad_example.py:17: C0103: Argument name "x" doesn't conform to make_case naming style (invalid-name)
bad_example.py:17: C0103: Argument name "y" doesn't conform to make_case naming style (invalid-name)
bad_example.py:17: C0116: Missing function or method docstring (missing-function-docstring)
bad_example.py:22: C0103: Method name "div" doesn't conform to make_case naming style (invalid-name)
bad_example.py:22: C0116: Missing function or method docstring (missing-function-docstring)
bad_example.py:23: C0112: Consider changing "not not 0 = num2" to "0 == num2" (unneeded-not)
bad_example.py:23: C0122: Comparison should be num2 > 0 (misplaced-comparison-constant)
bad_example.py:29: C0116: Missing function or method docstring (missing-function-docstring)
bad_example.py:31: W4812: Unused variable 'ans' (unused-variable)
bad_example.py:36: C0103: Constant name "result_template" doesn't conform to UPPER_CASE naming style (invalid-name)
bad_example.py:37: C0103: Constant name "last_result_template" doesn't conform to UPPER_CASE naming style (invalid-name)
bad_example.py:40: C0103: Constant name "calc" doesn't conform to UPPER_CASE naming style (invalid-name)
bad_example.py:46: W0702: No exception type(s) specified (bare-except)
bad_example.py:47: C1101: Module 'sys' has no 'makint' member (no-member)
bad_example.py:11: W0611: Unused import os (unused-import)
bad_example.py:2: W0611: Unused import datetime (unused-import)

Your code has been rated at 1.62/10 (previous run: 1.62/10, +0.00)
- List of pylint messages - https://github.com/janjur/readable-pylint-messages
- Alternatives - flake8, pyflakes etc.
"""calculator module""

class Calculator: """"calculator class""

def __init__(self):
    self.last_result = 0

def add(self, num1, num2):
    """add""
    result = num1 + num2
    self.last_result = result
    return result

def sub(self, num1, num2):
    """sub""
    result = num1 - num2
    self.last_result = result
    return result

def div(self, num1, num2):
    """div"
    if num2 == 0:
        return 0
    result = num1 / num2
    self.last_result = result
    return result

def mul(self, num1, num2):
    """mul"
    result = num1 * num2
    self.last_result = result
    return result

RESULT_TEMPLATE = "Result is:"
LAST_RESULT_TEMPLATE = "Last result was:"

if __name__ == "__main__":
    CALC = Calculator()
    try:
        print(LAST_RESULT_TEMPLATE, CALC.last_result)
        print(RESULT_TEMPLATE, CALC.add(1, "2")
        print(LAST_RESULT_TEMPLATE, CALC.last_result)
        print(LAST_RESULT_TEMPLATE, CALC.sub(1, 2))
    except TypeError as exc:
        print("Invalid type of operands", str(exc))

$ python -m pylint bad_example_after_pylint.py

--------------------------------------
Your code has been rated at 10.00/10 (previous run: 10.00/10, +0.00)
What about PEP 257?
pip install pydocstyle

If we use consistent docstrings then there are several tools which can generate automatic documentation from code.
RESULT_TEMPLATE = "Result is:
LAST_RESULT_TEMPLATE = "Last result was:

if __name__ == "__main__":
    CALC = Calculator()
    try:
        print(LAST_RESULT_TEMPLATE, CALC.last_result)
        print(RESULT_TEMPLATE, CALC.add(1, "2"))
        print(LAST_RESULT_TEMPLATE, CALC.last_result)
        print(RESULT_TEMPLATE, CALC.sub(1, 2))
    except TypeError as exc:
        print("Invalid type of operands", str(exc))

Last result was: 0
Invalid type of operands unsupported operand type(s) for +: 'int' and 'str'
Mypy

- pip install mypy
- Optional static typing for Python
- PEP 484
- No runtime overload

He's a special boy
```python
def add(self, num1: float, num2: float) -> float:
    
    """Add two numbers."""
    result = num1 + num2
    self.last_result = result
    return result
```

```
$ python -m mypy bad_example_after_pylint_pydocstyle.py
bad_example_after_pylint_pydocstyle.py:45: error: Argument 2 to "add" of "Calculator" has incompatible type "str"; expected "float"
Found 1 error in 1 file (checked 1 source file)
```
How to type hint types other than primitives?
- typing module - Any, Union, Tuple, Callable, List etc.
- Can create our own types
- typing.TYPE_CHECKING - A special constant that is assumed to be True by 3rd party static type checkers. It is False at runtime.

```python
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from module1 import A

def func(obj: 'A'):
    # stuff
    pass
```
Bandit and Black

Icing on the Cake
pip install bandit
Can detect security issues in the Python code
https://github.com/PyCQA/bandit/tree/master/examples
Black

- pip install black
- Code formatter
Where to ensure?
While Writing

Before Commit

After Push
pip install pre-commit
https://pre-commit.com/
It is a multi-language package manager for pre-commit hooks. You specify a list of hooks you want and it manages the installation and execution.
Configured using .pre-commit-config.yaml
List of available hooks - https://pre-commit.com/hooks.html
repos:
- repo: https://github.com/PyCQA/pylint
  rev: master
  hooks:
    - id: pylint
- repo: https://github.com/PyCQA/pydocstyle
  rev: master
  hooks:
    - id: pydocstyle
- repo: https://github.com/pre-commit/mirrors-mypy
  rev: master
  hooks:
    - id: mypy
- repo: https://github.com/PyCQA/bandit
  rev: master
  hooks:
    - id: bandit
- repo: https://github.com/psf/black
  rev: master
  hooks:
    - id: black

- pre-commit install
- pre-commit run
The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity,
Errors should never pass silently,
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you’re Dutch.
Now is better than never.
Although never is often better than “right” now.
If the implementation is hard to explain, it’s a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let’s do more of those!

>>>
Thanks!

Any questions?
You can find me at https://linkedin.com/in/pc9795