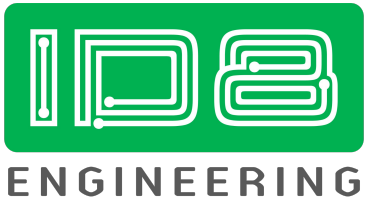


Taking Python beyond scripting

foss-north 2025

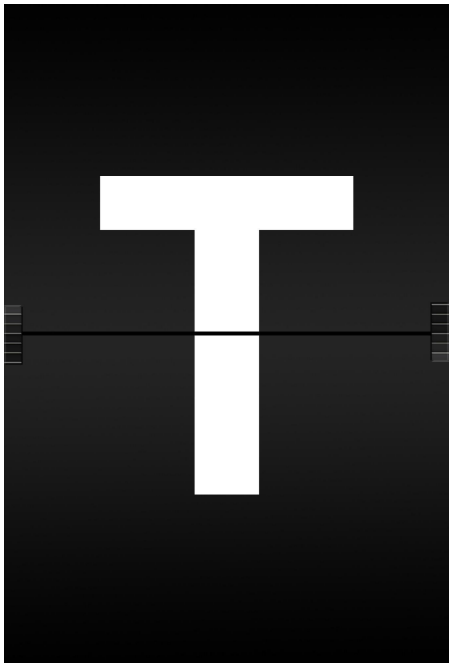
Agenda

- Introduction
- Linting (Static analysis)
- Package and project management
- Demo project
- Q/A

Introduction

Mirza Krak

- 15+ years in software and hardware engineering
- T-shaped skillset
  - expertise in embedded systems (predominantly Linux)
  - internet of things
  - cloud architecture
- Open source contributor
- CTO @ ID8 Engineering AB



ENGINEERING

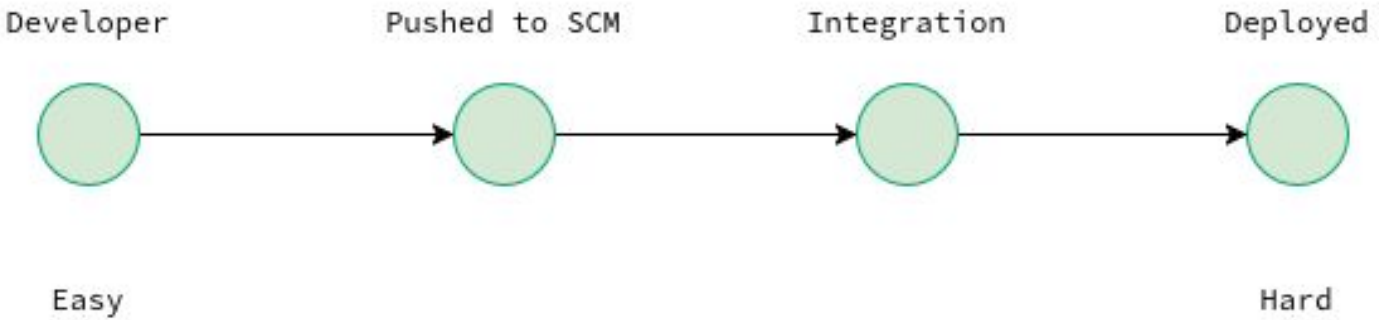
Python

- Python is a big tent, multi-paradigm interpreted language that generally allows you to do things in the way that best suits your needs, as best determined by you.
- 20 February 1991; 34 years ago
- Common in:
  - AI / Data
  - Backend / Web
  - Internet of Things
  - Automation / scripting
  - Infrastructure as Code

Beyond scripting

- Not measured by Lines of Code (LOC)
- Turning a program in to a product
- Codebase is
  - maintained over a longer period of time
  - collaborative development
  - distributed (via pip or other means)
- Types
  - command line interface (CLI)
  - modules
  - system daemons
  - microservices

Catching things early



Pitfalls moving beyond scripting

- Difficulty maintaining readability and consistency
- Insufficient testing leading to reliability issues
- Inefficient error handling
- Collaboration challenges and inconsistent coding standards
- Increased complexity in dependency management

Linting

Linting

“Lint is the computer science term for a static code analysis tool used to flag programming errors, bugs, stylistic errors and suspicious constructs”

Static analysis and linting tools

- coding standards / style guides (PEP 8)
- syntax
- conventions
- code smells
- refactor
- type validation
- security

Usage

- From command line
- Integrated with IDE/code editor
- As a pre-commit hook
- Continuous integration

Static analysis - Message types

- (C) convention, for programming standard violation
  - C0116: Missing function or method docstring (missing-function-docstring)
- (R) refactor, for bad code smell
  - R1705: Unnecessary "else" after "return", remove the "else" and de-indent the code inside it (no-else-return)
- (W) warning, for python specific problems
  - W1510: 'subprocess.run' used without explicitly defining the value for 'check'. (subprocess-run-check)
- (E) error, for much probably bugs in the code
    - E0001: Parsing failed: 'expected ':' (module.name, line 63)' (syntax-error)

What linters are there?

- It is complicated...
- Many standalone tools, with few aggregators
  - black - code formatter
  - isort - isort your imports, so you don't have to.
  - bandit - security oriented checks
  - autoflake - removes unused imports and unused variables from Python code
  - pyupgrade - automated upgrade to newer python syntax
  - pydocstringformatter - automated pep257.

What linters are there?

- It is complicated...
- Example 2:
  - Flake8, a style checker
  - Has a vast plugin ecosystem, pick and choose to build a competent linter
    - flake8-bugbear
    - flake8-bandit
    - flake8-builtins
    - flake8-docstrings
    - ...

A selected few

	Pylint	Flake8	Ruff	wemake-python-styleguide
License	GPLv2	MIT	MIT	MIT
Focus	Style, logic errors, complexity	Style guide enforcement (PEP8)	Ultra-fast linter + formatter	Strict, opinionated code style
Speed	Slower	Fast	Extremely fast	Fast (built on Flake8)
Extensible	Yes	Yes	No	No
Auto fix	No	No	Yes	No

Static analysis and linting tools

- coding standards / style guides (PEP 8)
- syntax
- conventions
- code smells
- refactor
- type validation
- security

Static analysis and linting tools

- coding standards / style guides (PEP 8)
- syntax
- conventions
- code smells
- refactor
- **type validation**
- security

Python typing

- Python is a dynamically typed language
- Gradual type system (opt-in) introduced in PEP 484 (2014)
- Has seen much development since it was introduced
- https://typing.python.org (official specification)

Python variable type annotations

```python
# No variable type annotations
x = 1
x = 1.0
x = True
x = "test"

# With variable type annotations
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
```

Python function type annotations

```python
# No function type annotations
def stringify(num):
    return str(num)

# With function type annotations
def stringify(num: int) -> str:
    return str(num)
```

Python function type annotations

```python
def stringify(num: int) -> str:
    return str(num)

# Call with wrong type
stringify("10")

# Error message (pyright)
Argument of type "Literal['10']" cannot be assigned to parameter
"num" of type "int" in function "stringify"
  "Literal['10']" is not assignable to "int"  (reportArgumentType)
```

Why Python typing?

- Type annotations have no runtime impact
- Help others understand their code more easily
- Catch typing problems early in the development process
- Integration with IDE
  - code completions
  - refactoring

Type checkers

	mypy	pyright	Pyre (*)	pytype
License	MIT	MIT	MIT	Apache 2.0
Developed by	Community	Microsoft	Meta	Google
LSP	Community	Native	Native	No
Language	Python	TypeScript	Python	Python
Github stars	19k	14k	7k	4k

Pitfalls moving beyond scripting

- Difficulty maintaining readability and consistency
- Insufficient testing leading to reliability issues
- Inefficient error handling
- Collaboration challenges and inconsistent coding standards
- Increased complexity in dependency management

Pitfalls moving beyond scripting

- Difficulty maintaining readability and consistency
- Insufficient testing leading to reliability issues
- Inefficient error handling
- **Collaboration challenges and inconsistent coding standards**
- Increased complexity in dependency management

Use a code formatter

- Resolve inconsistent coding standards and improve readability
- Examples:
  - Black + isort + pycodestyle
  - Ruff

Pitfalls moving beyond scripting

- Difficulty maintaining readability and consistency
- Insufficient testing leading to reliability issues
- Inefficient error handling
- Collaboration challenges and inconsistent coding standards
- **Increased complexity in dependency management**

Dependency management

Dependency management

- Using venv (basic)

```
python3 -m venv <DIR>
source <DIR>/bin/activate

# install packages needed for project

# save installed packages
pip freeze -l > requirements.txt

# install project deps
pip install -r requirements.txt
```

Dependency management

- More advanced project management
  - Poetry - Python packaging and dependency management made easy
  - uv - An extremely fast Python package and project manager, written in Rust.

Demo project using uv

Q / A

Contact


mirza@id8-engineering.io


+4730280622


www.linkedin.com/in/mirzakrak

ENGINEERING