



LINUXTIPS

EBOOK

PYTHON MODERNO PARA DEVOPS

Um guia de ferramentas atuais para
resolver problemas práticos em
automação e DevOps

BRUNO ROCHA

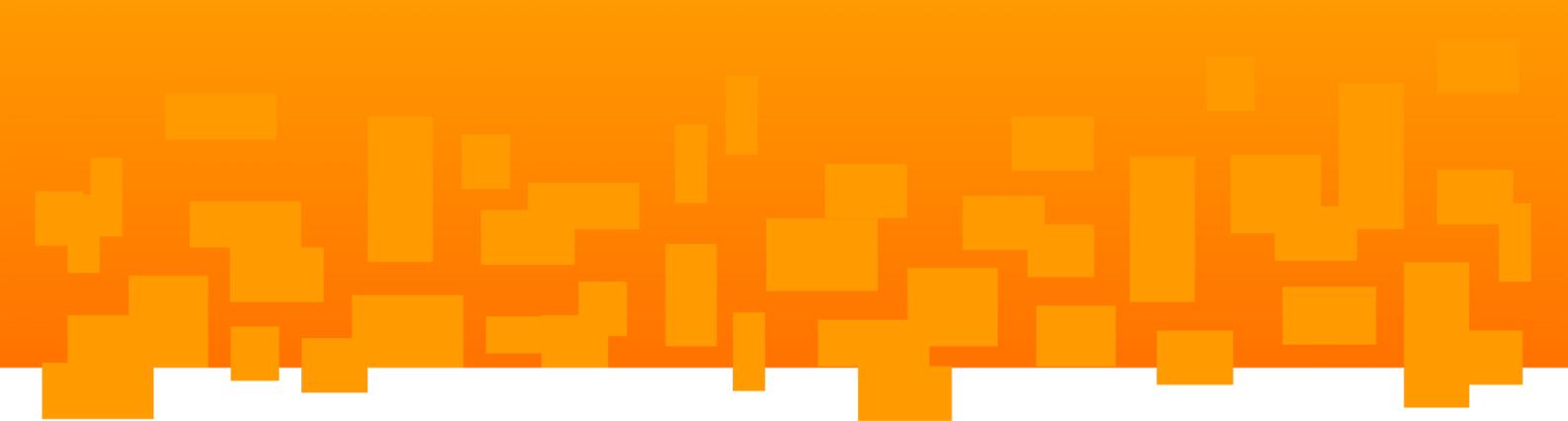


PREFÁCIO

ESTE E-BOOK, “Python Moderno para DevOps: Um guia de ferramentas atuais para resolver problemas práticos em automação e DevOps,” foi escrito por Bruno Cesar Rocha, Principal Software Engineer na Red Hat, acumulando quase 20 anos de experiência em TI em papéis como SysAdmin, Engenheiro de Qualidade, Data Engineer e Arquiteto de Software. Atualmente morando em Portugal, trabalha no desenvolvimento do Ansible. É criador do Dynaconf (biblioteca Python) e Marmite (Rust). Autor do livro “Python Web Development Cookbook”. Com grande paixão pelo ensino atua como educador na LINUXtips onde ministra treinamentos de Python, e como criador do canal CodeShow no YouTube, compartilhando conhecimento técnico e é também membro eleito da Python Software Foundation.

Neste guia, Bruno não apenas apresenta as ferramentas e técnicas, mas também oferece insights valiosos baseados em sua experiência real. O objetivo é fornecer um recurso prático e acessível para quem busca aprimorar suas habilidades em automação e DevOps com Python Moderno.





SUMÁRIO

INTRODUÇÃO

- 5 Porque Python é uma essencial ferramenta para DevOps
- 6 Visão geral do ecossistema Python para DevOps
- 7 O que é o Python Moderno?

CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

- 8 Desafios Comuns na Gestão de Ambientes Python
- 9 Instalação e gestão de ambientes com UV
- 10 Ambientes virtuais e isolamento de dependências
- 11 Podemos simplesmente usar o UV
- 12 Qualidade de Código

VALIDAÇÃO DE DADOS

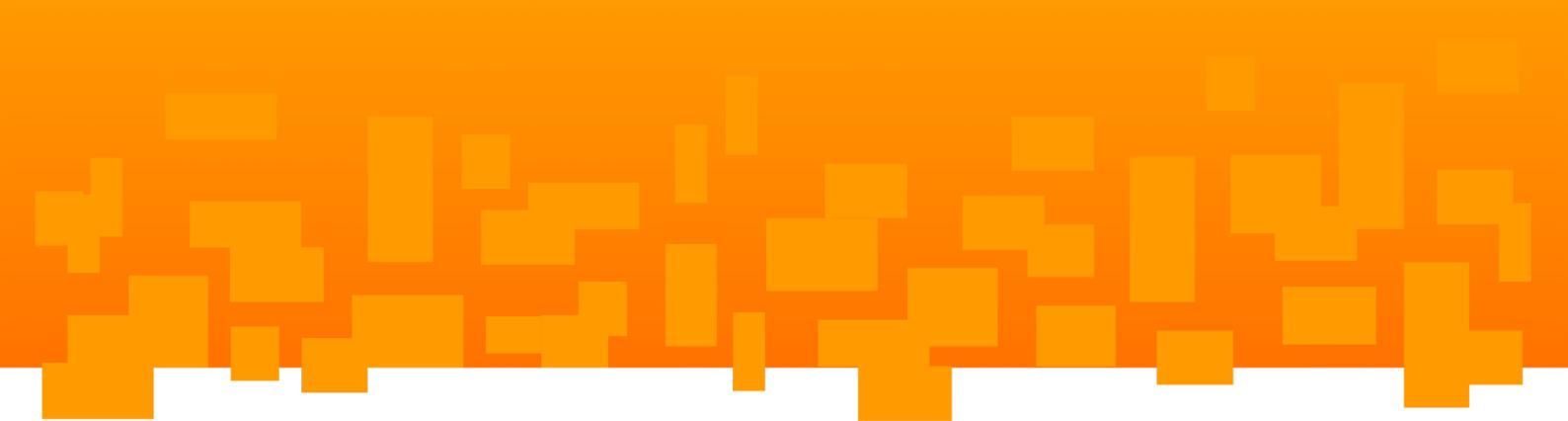
- 14 Anotação gradual de tipos
- 17 Validação de dados com Pydantic

GERENCIAMENTO DE CONFIGURAÇÕES

- 20 Dynaconf para gerenciar configurações de forma flexível
- 23 Exemplos de configurações para diferentes ambientes
- 24 Usando a CLI da Dynaconf
- 25 Desenvolvimento de APIs REST com FastAPI
- 27 Consumindo APIs REST

LINHA DE COMANDO

- 30 Criação de CLIs



TESTES E QUALIDADE

- 35 Testes com pytest
- 38 Testes de performance com Locust

DEBUGGING EFICIENTE

- 42 Debugging interativo
- 46 Debugging remoto

FERRAMENTAS DE PRODUTIVIDADE

- 49 Tarefas com Taskipy

INTERAÇÃO COM SERVIÇOS EXTERNOS

- 51 Integração com Git
- 53 Docker
- 54 Quer um desafio?
- 55 Kubernetes

CONCLUSÃO

PRÓXIMOS PASSOS

CONSIDERAÇÕES FINAIS

INTRODUÇÃO

Porque Python é uma essencial ferramenta para DevOps

A cultura DevOps, que visa integrar as equipes de desenvolvimento e operações, exige muitas ferramentas que facilitem a automação, a colaboração e a eficiência em todo o ciclo de vida do software. Nesse contexto, Python atua como uma linguagem “cola”, um canivete suíço de ferramentas prontas para resolver diversos problemas.

Uma das principais razões para a popularidade de Python em DevOps é a sua **simplicidade e legibilidade**. A sintaxe clara e concisa de Python torna o código fácil de escrever, entender e manter, o que é crucial em ambientes de ritmo acelerado e colaborativos. Equipes de desenvolvimento e operações podem trabalhar juntas de forma eficaz quando a base de código é acessível a todos.

Sua vasta biblioteca padrão, rico ecossistema de terceiros, forte comunidade e capacidade de automação em diversas áreas do ciclo de vida do software a consolidam como uma das **linguagens fundamentais para a implementação bem-sucedida de práticas DevOps**.

Dominar Python é, portanto, um investimento estratégico para qualquer profissional ou equipe que busca otimizar a entrega de software e a eficiência operacional.



Python é a segunda melhor opção para tudo. Há um módulo para praticamente qualquer coisa, e alguém já enfrentou o mesmo problema antes.”

**— Bruce Stone,
Diretor de Engenharia**

Visão geral do ecossistema Python para DevOps

A enorme e vasta coleção de bibliotecas e ferramentas do ecossistema Python abrange áreas cruciais para DevOps, aqui estão alguns exemplos:

INFRAESTRUTURA COMO CÓDIGO (IAC):

Ansible, boto3 (AWS), lambda PowerTools, google-cloud-python (GCP), Docker-py, kubernetes-client

AUTOMAÇÃO DE TESTES:

pytest, playwright, locust, freezegun

INTEGRAÇÃO E ENTREGA CONTÍNUA (CI/CD):

python-jenkins, python-gitlab, PyGithub

MONITORAMENTO E LOGGING:

logging, pandas, regex, structlog, psutil, Clientes Python para Prometheus e OTEL

Em resumo, a versatilidade que o Python oferece permite a automação de diversas tarefas e também a criação de soluções personalizadas para DevOps. É possível integrar as ferramentas e plataformas e ainda criar vários CLIs e dashboards. Além das ferramentas específicas para plataformas DevOps, o profissional de DevOps também se beneficia de toda a gama de ferramentas e construções da própria linguagem e biblioteca embutida, permitindo o desenvolvimento e gestão de projetos para CLI, TUI, API ou até mesmo para integração com IA.



83% dos desenvolvedores estão usando Python para projetos de DevOps, administração de sistemas e scripts de automação — um aumento significativo em relação aos 35% em 2017."

*— Scott Ferguson,
jornalista da DevOps.*

O que é o Python Moderno?

PYTHON MODERNO refere-se à nova forma, design, estrutura e estética das soluções desenvolvidas com Python a partir da versão 3.7. Essa modernidade se caracteriza pelo uso extensivo de **Type Annotations**, que servem como garantia de qualidade fornecendo contexto e significado para as aplicações. Essa abordagem é aplicável na criação de ferramentas CLI, APIs e na ingestão de dados externos que necessitam de validação. Esse movimento trouxe para o Python conceitos como Enum e dataclasses, apoiaram o surgimento de bibliotecas como Pydantic, FastAPI, Typer e FastMCP e também de ferramentas como Ruff e UV.

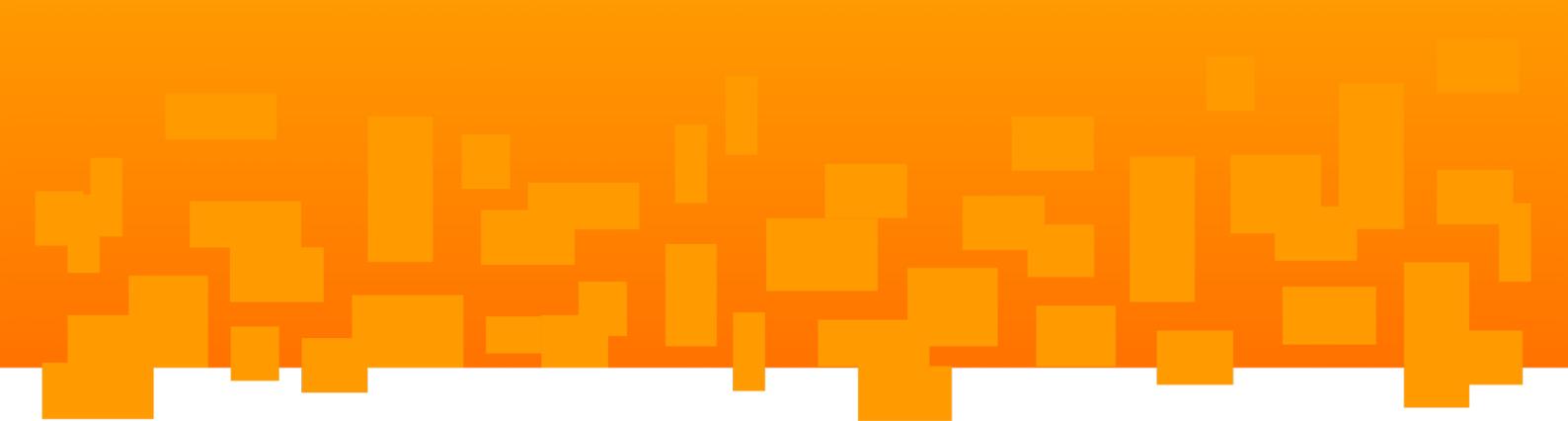
Mais recentemente, o “Python Moderno” também engloba a **implementação de iniciativas de melhoria do Python**, geralmente propostas por meio das PEPs. Essas mudanças significativas otimizam a experiência vinda do desenvolvimento, tornando a linguagem mais simples e assertiva. Exemplos dessas iniciativas incluem as PEPs para aprimoramento do sistema de builds e dependências, que resultaram em ferramentas como **UV** e Poetry; otimizações no interpretador; melhorias na intercomunicação com linguagens externas como Rust e C; otimizações de desempenho; e a consolidação de Python como uma plataforma robusta para agentes de IA.



“

A introdução das anotações de tipos no Python transformou a linguagem de uma ferramenta de script ágil para uma plataforma robusta de desenvolvimento de software. Essa evolução não apenas aprimorou a legibilidade e a manutenção do código, mas também elevou a confiança dos desenvolvedores em projetos de larga escala.”

— **Dra. Camila Andrade,**
*Engenheira de Software e Especialista
em Arquitetura de Sistemas*



CONFIGURANDO O AMBIENTE DE DESENVOLVIMENTO

Desafios Comuns na Gestão de Ambientes Python

GERENCIAR AMBIENTES Python é algo desafiador devido aos múltiplos projetos com dependências específicas e incompatíveis, dificultando a reprodutibilidade entre ambientes de desenvolvimento, teste e produção. A gestão de diferentes versões do Python e o ciclo de vida das dependências, incluindo segurança e compatibilidade, também são complexos.

Os problemas geralmente manifestam-se em conflitos de versões, instalações corrompidas e na impossibilidade de reproduzir o mesmo ambiente de forma determinística. Durante muitos anos, este foi o principal desafio do ecossistema Python. Recentemente, com a PEP-621, o Python passou a ter um sistema de *build* e gestão de dependências mais bem especificado, possibilitando o surgimento de ferramentas como o Poetry e o UV, sendo o UV a ferramenta mais moderna para gerenciamento de projetos Python.



O pip, o gerenciador de pacotes padrão, não consegue lidar com duas versões diferentes do mesmo pacote, o que leva ao 'inferno de dependências', fazendo com que instalações inteiras falhem."

— **Sagar Sharma,**
jornalista da Analytics India Magazine

Instalação e gestão de ambientes com UV

A EMPRESA Astral (também criadora da ferramenta Ruff) criou uma ferramenta inspirada no Cargo (gestor de projetos da linguagem Rust), a proposta foi a de fornecer uma Visão Unificada (Unified Vision) para todas as tarefas de gestão de um projeto, seja uma biblioteca, uma aplicação ou simplesmente scripts em Python que precisam ser executados sem a preocupação com a gestão de um ambiente. O UV abstrai toda a complexidade, é retrocompatível com PIP e virtualenv, e super-rápido, pois seu core é escrito em Rust.

CLIQUE AQUI para instalar siga as instruções no repositório do projeto
<https://github.com/astral-sh/uv>

ALGUNS EXEMPLOS DO QUE PODE SER FEITO COM UV

Instalar uma versão do interpretador Python:

```
uv python install 3.13
```

Iniciar um novo projeto:

```
uv init meu-projeto
```

Adicionar uma dependência:

```
uv add dynaconf
```

Executar um script:

```
uv run programa.py ou uv run dynaconf list
```

Compatibilidade com PIP e pip-tools:

```
uv pip install ou uv pip compile
```

CLIQUE AQUI para acessar a documentação oficial que possui guias para todas as funcionalidades do projeto

<https://docs.astral.sh/uv/>



instalador de pacotes Python moderno e de alto desempenho, escrito em Rust. Ele serve como um substituto para as ferramentas tradicionais de gerenciamento de pacotes Python, como o pip, oferecendo melhorias significativas em velocidade, confiabilidade e resolução de dependências.”

- Equipe da DataCamp,
em artigo sobre o UV

Ambientes virtuais e isolamento de dependências

A RECOMENDAÇÃO continua sendo a de ter um ambiente virtual para cada projeto, isso pode ser feito diretamente no Python com o uso de `python -m venv .venv` este comando invoca o módulo `venv` e cria um ambiente virtual nomeado `.venv` que é a convenção mais comum para nome de ambiente virtual. A inconveniência de usar este modo, é que sempre que precisar executar o projeto ou tarefas relacionadas a gestão do projeto, terá que antes ativar o ambiente virtual com: `source .venv/bin/activate` ou então executar os binários diretamente de dentro da pasta do `.venv` e além de

ser inconveniente isso também é um ponto de falha, pois é bastante comum executar um comando assumindo estar num ambiente, mas, na verdade, estar a usar o Python do sistema, é realmente chato quando isso acontece.

O **UV** resolve este problema, pois com UV não precisamos nos preocupar com o ambiente virtual, a ferramenta abstrai essa administração, basta nós sempre lembrarmos de executar os comandos com o prefixo do **UV**, portanto.

Ao invés de ter que ativar a virtual env da forma complicada:

```
Shell ▾
> python -m venv .venv          # Crie a virtualenv se
ela não existir
> source .venv/bin/activate     # Não esqueça de ativar
> pip install requests         # Se esqueceu de
ativar, usará o Python do sistema
> python programa.py           # Será que esse
`python` é mesmo o da .venv ou do sistema?
```

Podemos simplesmente usar o UV

A FERRAMENTA se encarrega de habilitar a virtualenv, e caso a virtualenv não exista, o UV cria uma para o projeto, e não tem mais esse problema de usar erroneamente o Python global do sistema.

Shell

```
> uv init          # necessário apenas uma vez  
> uv add requests  
> uv run programa.py
```

Em casos onde o seu projeto é apenas um único arquivo, um script apenas, o UV facilita ainda mais, pois nem é necessário que o UV crie e gerencie a virtualenv, neste caso o UV pode usar a PEP-723 que permite adiciona as dependências dentro do próprio script.

Shell

```
> uv add --script programa.py dynaconf  
> cat programa.py  
# /// script  
# requires-python = ">=3.13"  
# dependencies = [  
#     "dynaconf",  
# ]  
# ///  
  
> uv run programa.py  
Installed 1 package in 10ms
```

Qualidade de Código

MANTER A QUALIDADE de código é algo essencial para que o projeto seja mais fácil de manter, existem algumas práticas e ferramentas que podem ajudar nesta tarefa, os principais objetivos na qualidade de código são: **Estilo de Código coeso e padronizado, Baixa complexidade ciclomática, Organização de Módulos, Validação de tipos.**

O ecossistema Python possui diversas ferramentas que podem resolver esses problemas. Algumas delas são: black, mccabe, pylint, isort e mypy.

No Ecossistema de Python Moderno, a maior parte dessas funcionalidades foram todas agregadas em uma única ferramenta, o **ruff**

O Ruff consegue ajustar o estilo do seu código seguindo as diretrizes da PEP-8 e do Black, organizar os imports, remover variáveis não utilizadas e atualizar o seu código para versões recentes do Python. Juntamente com o UV o Ruff é outra ferramenta essencial para um projeto Python Moderno.

Shell

```
uvx ruff check # verifica o seu projeto  
uvx ruff format # formata o seu código nos padrões da  
pep8
```

INFO

uvx é o comando que usa o uv para executar ferramentas globais do sistema, assim o ruff pode estar disponível para todos os seus projetos não precisando instalar individualmente em cada um.

CLIQUE AQUI para acessar o site

<https://astral.sh/ruff>

Além disso o ruff tem um LSP que se integra com o seu editor (VSCode, Zed, Neovim, Pycharm) e indica em tempo real os problemas detectados em seu código. O que o **ruff** ainda não faz é a validação estática da tipagem do projeto. A Astral está desenvolvendo uma ferramenta chamada **ty** que já pode ser usada em modo beta.

CLIQUE AQUI para acessar a ferramenta

<https://github.com/astral-sh/ty>

Shell

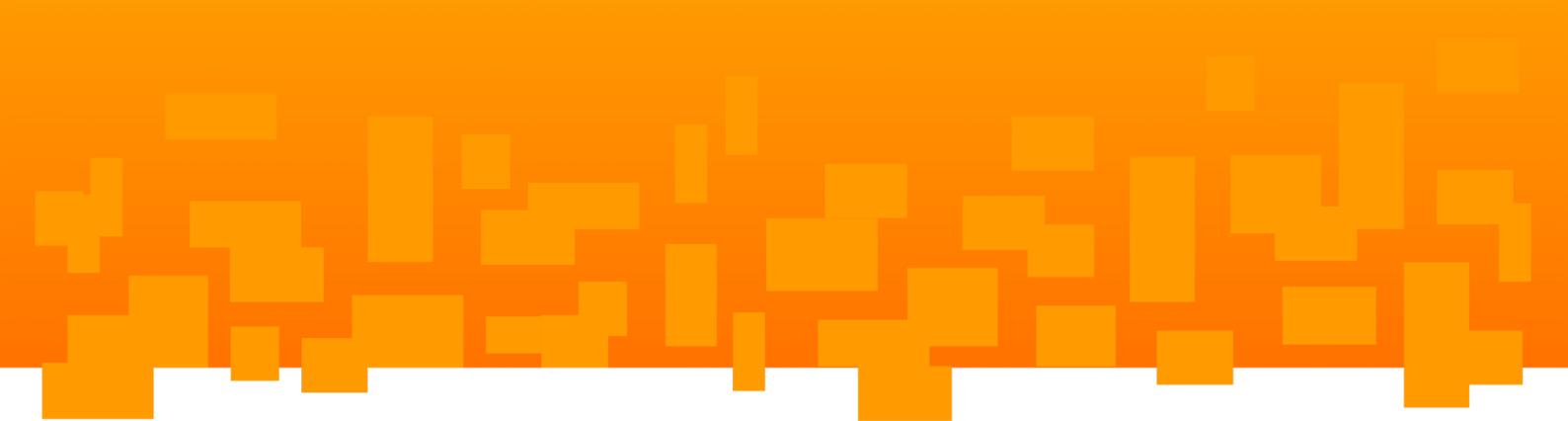
```
uvx ty check # Verifica a tipagem do projeto
```

E as alternativas são o **mypy** e o **pyright** que são ferramentas já estabelecidas para este fim e que também podem ser executadas com o **uvx**

Shell

```
uvx mypy  
uvx pyright
```





VALIDAÇÃO DE DADOS

Anotação gradual de tipos

PYTHON POR ser uma linguagem dinâmica, faz inferencia dos tipos das variáveis e parâmetros de função, sendo assim podemos simplesmente escrever uma função assim:

```
Python
def soma(x, y):
    return x + y
```

Parece que está tudo certo com a função acima né? e está! aliás esse é um dos poderes do Python, ser simples e dinâmico. Mas nem tudo são flores, apesar de simples em alguns casos a função acima pode resultar em um erro de runtime

```
Python
# OK
soma(1,1)          # resultado: 2
soma("Hello", "World") # resultado "HelloWorld"

# Não OK
soma("Hello", 2)      # TypeError: unsupported operand
types
```

O erro acontece pois a função aceita qualquer tipo de dado, porém nem todos os tipos de dados são suportados na operação de soma +. A parte pior é que, devido a natureza dinâmica dos dados, **este erro pode só acontecer em runtime, ou seja, em produção!**

Uma das características mais importantes do Python Moderno é anotação gradual de tipos, ela permite que a gente descreva os tipos esperados em nossa função, ajudando o código a ser mais legível, permitindo a análise estática de tipos e até empoderando ferramentas como o Pydantic que veremos a seguir.

```
Python ▾  
def soma(x: int, y: int) -> int:  
    return x + y|
```

Adicionamos **nome: tipo** na assinatura de parâmetros da função e **-> tipo** na assinatura de retorno da função, durante o runtime o interpretador Python simplesmente coleta estas informações e as deixa disponível para que ferramentas externas analizem. Exemplo:

```
Python  
# programa.py  
  
def soma(x: int, y: int) -> int:  
    return x + y  
  
soma(1, 1)  
soma(1, "Batata")
```

E agora, podemos usar um analisador estático de tipos como o `mypy` ou o `ty`

```
Shell
> uvx ty check
error[invalid-argument-type]: Argument to function
`soma` is incorrect
--> programa.py:5:9
|
4 | soma(1, 1)
5 | soma(1, "Batata")
|           ^^^^^^^^^ Expected `int`, found
`Literal["Batata"]`
|
info: Function defined here
--> programa.py:1:5
|
1 | def soma(x: int, y: int) -> int:
|   ^^^^          ----- Parameter declared here
2 |     return x + y
|
Found 1 diagnostic
```

Ao executar o `ty` ou qualquer outro analisador de tipos, podemos detectar estes problemas durante a nossa esteira de testes e evitar que o problema aconteça em runtime.

Validação de dados com Pydantic

APESAR DA checagem estática de tipos ser interessante, ela não é suficiente pois alguns erros serão inevitáveis, imagine uma aplicação onde você depende de dados externos, fazendo request de um **JSON** de uma **API** ou lendo um arquivo que o usuário faz upload, ou até mesmo carregando dados salvos em um banco de dados.

Para este caso teremos que lidar com o erro, tomar uma decisão com base no erro, informar o usuário da aplicação que o erro aconteceu, e é ai que o Pydantic nos ajuda.

Apesar do Python ter um suporte nativo a **dataclasses**, elas não fornecem validação em tempo de execução.

Vamos ver um exemplo com Pydantic. Imagine que você tem um sistema que lê um JSON de uma API ou de um arquivo externo, exemplo user.json

```
JSON
{
    "nome": "Player 1",
    "pontos": "vinte"
}
```

Agora você usa a biblioteca `json` para carregar o arquivo e também incrementar a pontuação do usuário.

Python

```
import json
user = json.load(open("user.json"))
user["pontos"] = user["pontos"] + 1 # TypeError
```

Este programa irá falhar com `TypeError` pois não é possível somar `"vinte"` a 1, e isso se deve ao fato de Python ser uma linguagem com tipagem dinâmica mas ao mesmo tempo tipagem forte, isso significa que Python adere ao protocolo dos objetos e não tenta fazer coerção automaticamente, evitando assim erros que poderiam ser silenciados, em outras linguagens o resultado seria: `"vinte1"` mas Python não deixa isso acontecer.

Para garantir a validação destes dados, vamos usar o `Pydantic`. Primeiro, nós **definimos o esquema de validação dos dados e o modelo**, e, só então, **carregamos os dados do JSON** e os passamos para o esquema de validação.

Python

```
# programa.py
import json
from pydantic import BaseModel

class User(BaseModel):
    nome: str
    pontos: int

user = User(**json.load(open("user.json")))
```

Shell

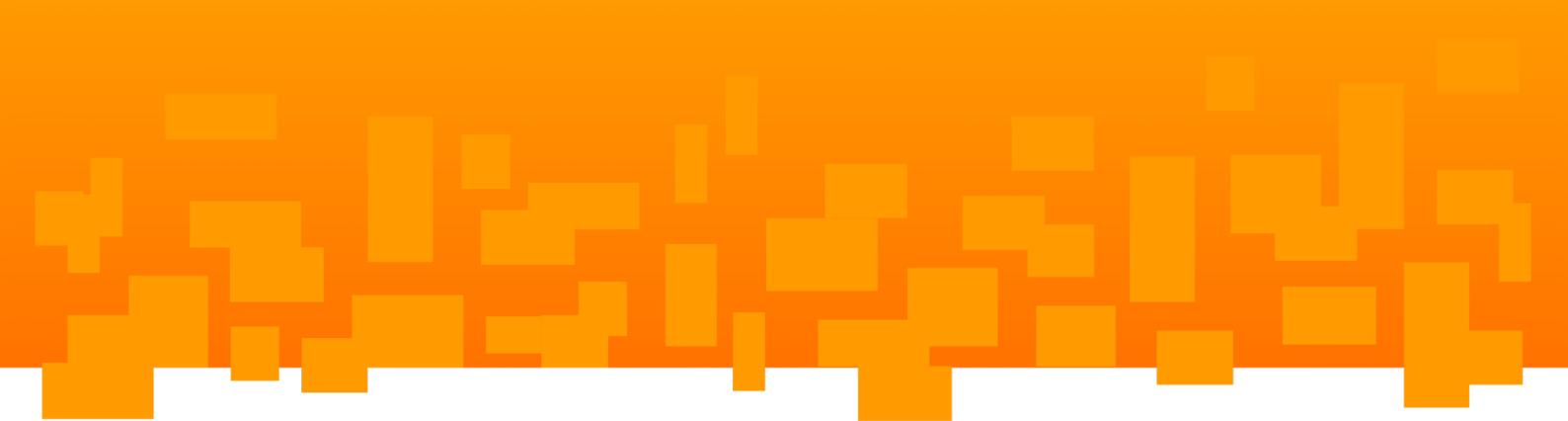
```
> uv add --script programa.py pydantic
```

Shell

```
> uv run programa.py
Traceback (most recent call last):
  File "/pydantic_ex/programa.py", line 17, in <module>
    user = User(**json.load(open("user.json")))
pydantic_core._pydantic_core.ValidationError: 1
validation error for User
pontos
  Input should be a valid integer, unable to parse
  string as an integer [type=int_parsing,
  input_value='vinte', input_type=str]
  For further information visit
  https://errors.pydantic.dev/2.11/v/int_parsing
```

Apesar do erro acontecer em runtime, podemos usar o Pydantic para ajudar a definir regras de validação, por exemplo, redirecionando o usuário, efetuando uma conversão de dados ou mostrando uma mensagem mais interessante. A recomendação inicial é que, qualquer dado que seja carregado de fontes externas seja validado com um modelo definido com Pydantic.





GERENCIAMENTO DE CONFIGURAÇÕES

Dynaconf para gerenciar configurações de forma flexível

TODA APLICAÇÃO precisa de algum tipo de configuração, seja através de um arquivo de settings ou via a leitura de variáveis de ambiente, a biblioteca Dynaconf fornece uma interface para definição, validação e leitura de settings de forma padronizada.

O terceiro item da metodologia **12-Factor App** enfatiza a importância de armazenar configurações no ambiente, separando-as do código-fonte. Isso significa que informações como credenciais de banco de dados, chaves de API e variáveis específicas de ambiente devem ser mantidas fora do código, geralmente em variáveis de ambiente. Essa prática facilita a modificação das configurações entre diferentes ambientes (desenvolvimento, teste, produção) sem a necessidade de alterar o código-fonte, promovendo segurança, flexibilidade e escalabilidade.

Tenho usado o Dynaconf há cerca de 4 anos e realmente aprecio como o módulo é projetado para resolver desafios recorrentes de forma simples e direta."

– **Carlos Neto,**
desenvolvedor e autor do blog Carlos Neto's Tech Blog

A Dynaconf permite que os desenvolvedores definam variáveis de ambiente com prefixo personalizado (por exemplo, MYAPP_) e as utilize para configurar parâmetros da aplicação. Além disso, o Dynaconf suporta a leitura de arquivos .env e diversos outros formatos para automatizar a exportação de variáveis de ambiente, mantendo a flexibilidade e a segurança das configurações.

Shell

```
# Instale dynaconf direto do repo para incluir a preview
# do uso de typing
uv add git+https://github.com/dynaconf/dynaconf
```

Python

```
from typing import Annotated
from dynaconf.typed import Dynaconf, Options
from dynaconf.typed.validators import In, Gt

class Settings(Dynaconf):
    hostname: Annotated[str, In(["localhost",
"server.com"])]
    port: Annotated[int, Gt(5000)] = 5001

    dynaconf_options = Options(
        envvar_prefix="MYAPP",
        settings_files=["settings.yaml",
".secrets.yaml"],
    )

settings = Settings()
```

AVISO

No momento da escrita deste e-book, a funcionalidade de tipos da Dynaconf está em modo de preview, algumas características estão sujeitas a mudança e está prevista a estabilização no lançamento da Dynaconf 4.0.0

A última configuração cria um objeto `settings` a partir do esquema definido, este objeto irá carregar as chaves de configuração `hostname` com a validação definida para que seja do tipo `str` e o valor seja um entre “localhost” ou “`server.com`”. E também uma chave `port` que deve ser do tipo int, ser maior que `5000` e por default terá o valor `5001`

A Dynaconf vai carregar toda variável de ambiente que tenha o prefixo `MYAPP_` e também os arquivos `settings.yaml` e `.secrets.yaml` caso existam.

As configurações também podem ser definidas em nos arquivos, isso é útil para ter valores default ou para arquivos que são copiados durante o processo de deploy.

Textproto

```
# settings.yaml
hostname: "localhost"
port: 8001
```

E o arquivo `.secrets.yaml` é uma convenção, pode adicioná-lo ao `.gitignore` e nele definir suas senhas locais ao ambiente.

Exemplos de configurações para diferentes ambientes (dev, testing, prod)

A DYNACONF também permite a definição de múltiplos ambientes em camadas quando usando arquivos de configuração, isto é útil para configurações default.

Textproto

```
# settings.toml

[default]
hostname = "localhost"

[development]
port = 5001

[testing]
port = 6032

[production]
port = 8082
```

E ao executar sua aplicação, por default ela estará sempre no modo **development** (o primeiro da lista) e você pode alternar com a variável **MYAPP_MODE**

Usando a CLI da Dynaconf

PARA FACILITAR a inspeção das configurações a Dynaconf possui uma ferramenta CLI que permite algumas operações:

Shell

```
> export INSTANCE_FOR_DYNACONF=programa.settings
> uv run dynaconf list
PORT<int> 9999
HOSTNAME<str> 'localhost'

> uv run dynaconf get hostname
localhost

> uv run dynaconf inspect -m debug -f yaml -v
versions:
  dynaconf: 3.3.0.dev0
  root_path: /home/rochacbruno/Projects/uv_123/dyna
  validators:
    - Validator('hostname', type=str, required=True)
    - Validator('hostname', operations={'is_in':
      ['localhost', 'server.com']})
    - Validator('port', type=int)
    - Validator('port', operations={'gt': 5000})
  loaded_files:
    - /....settings.yaml
  history:
    - loader: yaml
      identifier: settings.yaml
      data:
        - HOSTNAME
        - PORT
```

Além disso, a Dynaconf ainda suporta acessar storages externos como Redis, Vault, Bancos de Dados e várias estratégias para sobrecarga e combinação de settings.

CLIQUE AQUI para acessar o site

<https://dynaconf.com>

CRIANDO E CONSUMINDO APIs REST

Desenvolvimento de APIs REST com FastAPI

APESAR DE não ser este o trabalho central de uma pessoa de DevOps, muitas vezes a solução de problemas pode envolver publicar uma API, seja para experimentos ou para simular uma API real para testes.

O FastAPI é o framework mais moderno do ecossistema Python, ele é baseado no Pydantic, usa as anotações de tipo em sua definição de API e entrega praticamente tudo pronto, desde a validação até a documentação, é a ferramenta perfeita para API rápidas, leves e escritas com pouco código. Vamos a um exemplo prático que fica mais didático:

Imagine que sua tarefa é servir uma API REST que retorna métricas do sistema. 15 linhas de código são suficientes para uma API completa, pode executar com:



Python

```
# main.py

from fastapi import FastAPI
from pydantic import BaseModel
import psutil

app = FastAPI()

class SystemMetrics(BaseModel):
    memoria: float # Uso de memória em %
    cpu: float      # Uso de CPU em %
    disk: float     # Uso de disco em %

@app.get("/metrics", response_model=SystemMetrics)
def get_system_metrics():
    memory = psutil.virtual_memory().percent
    cpu = psutil.cpu_percent(interval=1)
    disk = psutil.disk_usage('/').percent
    return SystemMetrics(memoria=memory, cpu=cpu,
disk=disk)
```

Acesse no seu browser <http://localhost:8000/docs>

Shell

```
> uv run --with "fastapi,uvicorn,psutil" uvicorn
main:app
INFO:     Uvicorn running on http://127.0.0.1:8000
(Press CTRL+C to quit)
```

Consumindo APIs REST

FastAPI 0.1.0 OAS 3.1

[/openapi.json](#)

default ^

GET /metrics Get System Metrics

Métricas do Sistema

Parameters

No parameters

Try it out

Responses

Code	Description	Links
200	Successful Response	<i>No links</i>

Media type

application/json ▾

Controls Accept header.

Example Value | Schema

```
{  
    "memoria": 0,  
    "cpu": 0,  
    "disk": 0  
}
```

Schemas ^

SystemMetrics ^ Collapse all object

```
memoria* number  
cpu* number  
disk* number
```

PRA QUEM trabalha com DevOps é imprescindível saber usar a ferramenta cURL, é praticamente um padrão para acesso a URLs e APIs.

Essa mesma operação pode ser feito dentro de um script Python com a biblio-

Shell

```
> curl http://localhost:8000/metrics
{"memoria":59.9, "cpu":5.4, "disk":53.5}
```

teca requests, a vantagem é que estando no Python podemos usar o Pydantic para efetuar a validação dos dados recebidos.

LINHA DE COMANDO

Python

```
# programa.py

import requests
from pydantic import BaseModel, ValidationError

class SystemMetrics(BaseModel):
    memoria: float
    cpu: float
    disk: float

def fetch_metrics():
    try:
        response =
requests.get("http://localhost:8000/metrics")
        response.raise_for_status()
        metrics = SystemMetrics(**response.json())

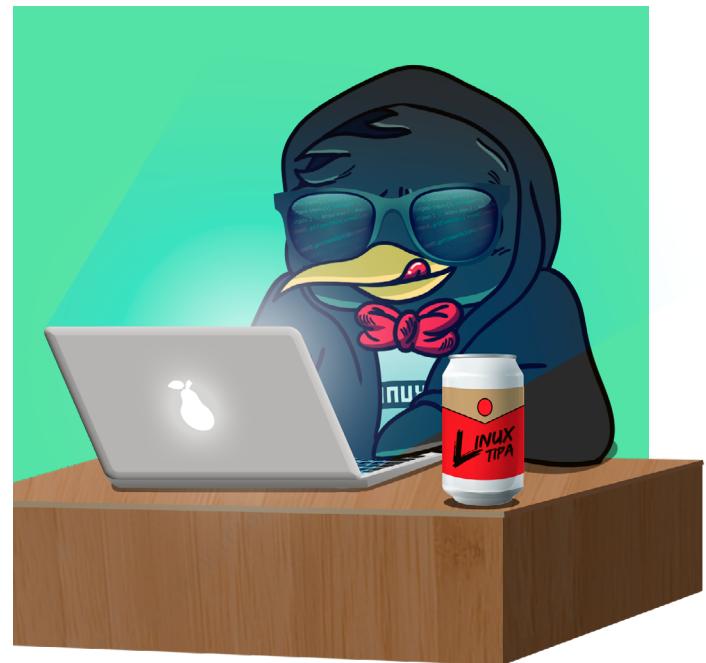
        print("✅ Métricas recebidas com sucesso:")
        print(f"Memória: {metrics.memoria}%")
        print(f"CPU: {metrics.cpu}%")
        print(f"Disco: {metrics.disk}%")
    except requests.RequestException as e:
```

```
        print(f"X Erro na requisição: {e}")
    except ValidationError as ve:
        print("X Erro na validação dos dados
recebidos:")
        print(ve.json())

if __name__ == "__main__":
    fetch_metrics()
```

Shell

```
> uv run --with "pydantic,requests" programa.py
✓ Métricas recebidas com sucesso:
Memória: 60.4%
CPU: 6.7%
Disco: 53.5%
```



Criação de CLIs

TAREFAS REPETITIVAS devem ser automatizadas, e existem várias ferramentas que ajudam com essas tarefas, como por exemplo: editores de texto, executores de comandos como Make ou Just, e suites de automação como o Ansible. Mas mesmo com essas ferramentas todas, pode sempre ter aquele caso onde você quer criar a ferramenta para ter o controle total de seu uso e validação e também para permitir que outras pessoas utilizem a ferramenta.

Os problemas que você vai resolver com CLI vão variar de acordo com a necessidade do seu projeto, vamos fazer aqui um exemplo simples só para te dar uma

ideia de como a ferramenta funciona. Existem excelentes ferramentas dentro do Python Moderno para criação de CLI, entre as mais famosas estão Typer, Pydantic-Typer, Fire, Arguably e o Cyclops, eu pessoalmente gosto mais do Cyclops pela sua simplicidade.

Suponhamos que você tenha um template de **Dockerfile** e quer criar uma ferramenta CLI que irá gerar o arquivo para você através da parametrização.

AVISO

Cuidado! A criação de CLI pode se tornar um vício, e é bastante recomendado!

Inic平ize um projeto do tipo package com o UV

Shell

```
> uv init --package dfgen
> cd dfgen
> uv add jinja2 cyclops
```

Edite o programa criado pelo UV adicionando a lógica da sua aplicação

Python

```
# src/dfgen/__init__.py

DOCKERFILE_TEMPLATE = """\
FROM {{ base_image }}
{% if user -%}
RUN useradd -ms /bin/bash {{ user }}
USER {{ user }}
{%- endif -%}
{%- if workdir %}
WORKDIR {{ workdir }}
{%- endif -%}
{%- for cmd in run_commands -%}
RUN {{ cmd }}
{%- endfor -%}
{%- if expose %}
EXPOSE {{ expose }}
{%- endif -%}
{%- if cmd %}
CMD ["{{ cmd }}"]
{%- endif -%}
"""

from cyclops import App
from jinja2 import Template
from typing import Literal

main = App()

@main.default
def generate(
    from_: Literal["alpine", "ubuntu", "centos"] =
"alpine",
    user: str = "root",
    expose: int = 80,
    cmd: str = "sh",
    cwd: str = "/",
```

```

    run: list[str] = [],
    output: str = "Dockerfile",
):
"""
    Gera um Dockerfile com base nos parâmetros
fornecidos.

Parameters
-----
from
    Imagem Base
user
    Usuário a ser criado
expose
    Porta para expor
cmd
    Comando de inicialização
cwd
    Diretório de trabalho
run
    Comandos RUN adicionais
output
    Arquivo de saída
"""

template = Template(DOCKERFILE_TEMPLATE)
dockerfile_content = template.render(
    base_image=from_,
    user=user,
    expose=expose,
    cmd=cmd,
    workdir=cwd,
    run_commands=run,
)
with open(output, "w") as f:
    f.write(dockerfile_content)
print(f"Dockerfile gerado com sucesso em: {output}")

if __name__ == "__main__":
    main()

```

Instale o seu pacote como um tool global do UV (desta forma não vai precisar usar `uv run`)

```
Shell  
> uv tool install . -e  
Installed 1 executable: dfgen
```

Agora pode executar seu binário diretamente

```
Shell  
> dfgen --help  
Usage: dfgen [ARGS] [OPTIONS]
```

Gera um Dockerfile com base nos parâmetros fornecidos.

Commands

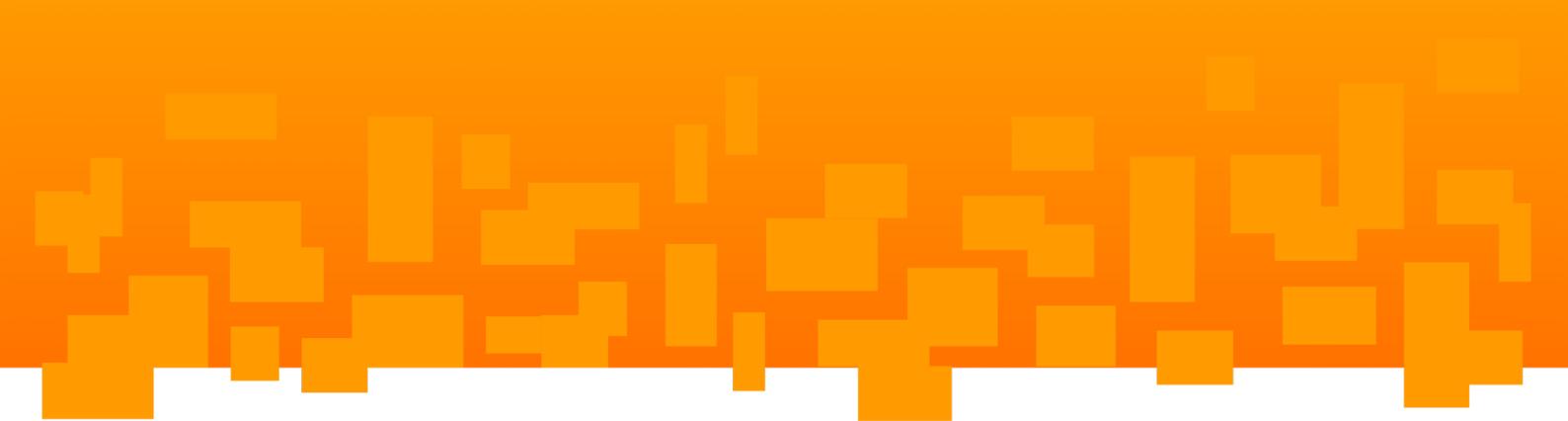
```
| --help -h Display this message and exit.  
| --version Display application version.
```

Parameters

```
| FROM --from           [choices: alpine, ubuntu,  
centos] [default: alpine]  
| USER --user           Usuário a ser criado [default:  
root]  
| EXPOSE --expose      Porta para expor [default: 80]  
| CMD --cmd             Comando de inicialização  
[default: sh]  
| CWD --cwd             Diretório de trabalho  
[default: /]  
| RUN --run --empty-run Comandos RUN adicionais  
[default: []]  
| OUTPUT --output       Arquivo de saída [default:  
Dockerfile]
```

O legal é que a entrada do CLI é validada com os tipos definidos na função do comando.

Agora, você basta usar a criatividade integrando com outras ferramentas! Quando estiver pronto, poderá rodar **uv build** e **uv publish** e, assim, outras pessoas poderão usar sua ferramenta instalando com **pip install** ou **uv add**



TESTES E QUALIDADE

Testes com pytest

O **PYTHON** tem uma biblioteca de testes embutida, o `UnitTest`, porém o Python Moderno adota o **pytest** por ser uma ferramenta mais atualizada e muito mais simples de usar.

Vamos continuar em nossa aplicação `dflen` e escrever alguns testes para ela. Primeiro, adicionamos o **pytest** como dependência, mas um detalhe importante aqui é que adicionamos apenas no grupo **test**. Desta forma, quem instalar nossa aplicação não vai precisar do pytest e só o instalaremos apenas quando formos rodar os testes.

```
Shell
> uv add --group test pytest
Installed 5 packages in 2ms
```

Agora podemos adicionar um arquivo de testes.

```
Shell
mkdir tests
touch tests/test_generate.py
```

E no arquivo de teste usamos o Pytest para definir um teste unitário que vai usar parametrização e fixtures.

```
Python
import pytest

from jinja2 import Template
from dfgen import generate, DOCKERFILE_TEMPLATE


@pytest.mark.parametrize("params", [
    {
        "from_": "ubuntu",
        "user": "app",
        "expose": 8080,
        "cmd": "python app.py",
        "cwd": "/app",
        "run": ["apt-get update", "apt-get install -y python3"],
        "output": "Dockerfile"
    },
    {
        "from_": "alpine",
        "user": "root",
        "expose": 80,
        "cmd": "sh",
        "cwd": "/",
        "run": [],
        "output": "Containerfile"
    }
])
def test_generate_custom_params(params, tmpdir):
    """Test generate function with custom parameters."""
    test_file = tmpdir.join(params["output"])
    params["output"] = str(test_file)
    generate(**params)

    with open(test_file, "r") as f:
        content = f.read()

    expected = Template(DOCKERFILE_TEMPLATE).render(
        base_image=params["from_"],
        user=params["user"],
        expose=params["expose"],
        cmd=params["cmd"],
        workdir=params["cwd"],
        run_commands=params["run"],
    )
    assert content == expected
```

Agora basta executar os testes.

```
Shell
> uv run pytest -v
=====
          test session starts
=====
collected 2 items

tests/test_generate.py::test_generate_custom_params[params0]
PASSED [ 50%]
tests/test_generate.py::test_generate_custom_params[params1]
PASSED [100%]

=====
          2 passed in 0.08s
=====
```

Repare que escrevemos apenas um teste mas o Pytest executou 2! Isso aconteceu pois o pytest irá repetir o teste para cada item da parametrização. Repare também que usamos um arquivo temporário sem precisar se preocupar com a gestão deste arquivo, graças a existência de fixtures no pytest. Neste caso usamos uma fixture embutida a `tmpdir` mas é possível criar um arquivo `conftest.py` e nele definir suas próprias fixtures para reutilizar.

Além de testes unitários – que são esses testes em que invocamos funções individualmente como fizemos com `generate` – podemos, também, escrever testes funcionais que iriam de fato executar a ferramenta `dfgen` na linha de comando e testar o seu retorno. A estrutura é a mesma, a diferença é que, neste caso, usariamos `subprocess` para executar a ferramenta CLI em um shell separado.

Testes de performance com Locust

AGORA AO INVÉS de testar unitariamente a nossa aplicação vamos ver como ela se comporta em uma carga grande de requisições, o Locust é uma ferramenta que ajuda nesta tarefa e além de poder ser usado para testes pode também ser usado para chaos-engineering, ou seja, simplesmente simular muitas requisições e ver se a infra + aplicação aguenta a carga.

Shell

```
mkdir teste_de_carga  
cd teste_de_carga  
touch locustfile.py
```

O conteúdo do locustfile é a especificação em Python de quais endpoints serão executados.

```
Python ▾  
# locustfile.py  
from locust import HttpUser, task, between  
  
class MetricsUser(HttpUser):  
    wait_time = between(1, 2)  
  
    @task  
    def metrics(self):  
        self.client.get("/metrics")
```

Para testar vamos usar a API que criamos lá no começo, lembra? a API que retorna as métricas do sistema. Execute aquela API e deixe a rodando na porta **8000** e vamos testar se ela aguenta a carga. Para poder começar, precisamos de um **locustfile.py**

Para executar, podemos utilizar o **uv** passando **--with locust** para ele garantir que o locust será instalado e então chamar o binário do **locust**

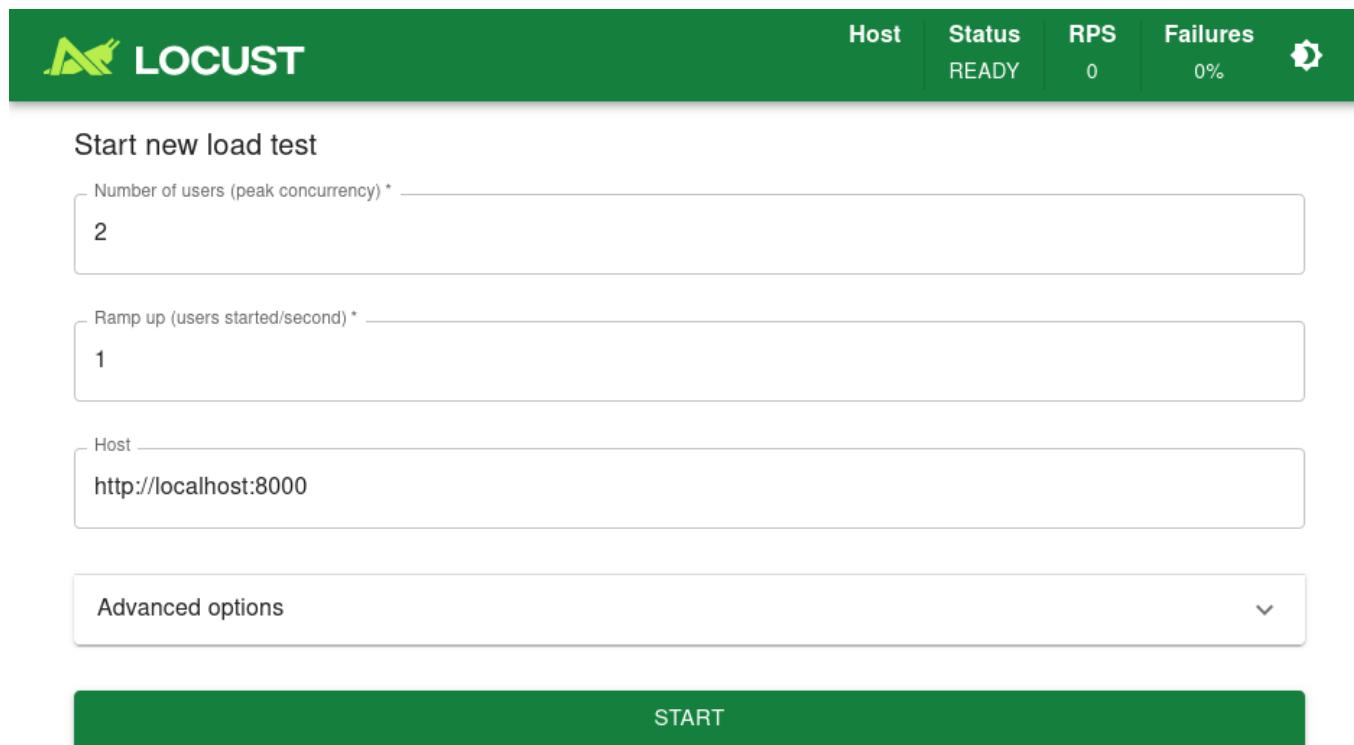
Shell

```
> uv run --with locust locust
Starting Locust 2.37.6
Starting web interface at http://0.0.0.0:8089,
press enter to open your default browser.
```

Assim que abrir no browser, a interface web do locust vai oferecer um formulário para parametrizar a sessão de teste de carga. Se a sua infra for boa, pode colocar algo como 100 usuários simultaneamente, ou até aumentar se estiver testando um ambiente altamente escalável em staging.

AVISO

Cuidado, não faça esse teste em produção, a não ser que este seja mesmo o objetivo, pois você pode causar um DoS.

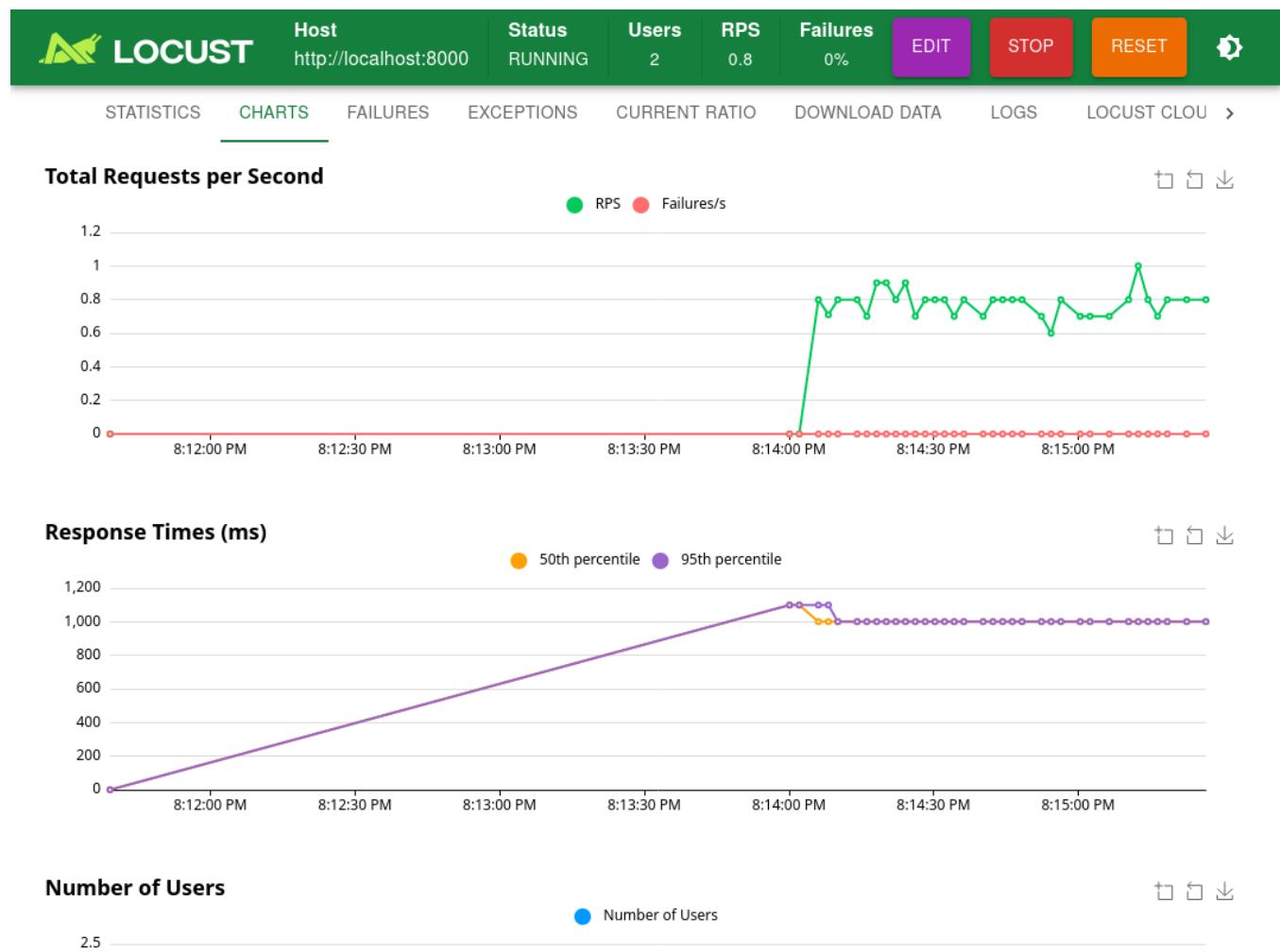


The screenshot shows the Locust web interface for starting a new load test. At the top, there's a green header bar with the Locust logo and some status metrics: Host (READY), Status (READY), RPS (0), Failures (0%), and a gear icon. Below the header, the main form has the following fields:

- Start new load test**
- Number of users (peak concurrency) ***: A text input field containing the value **2**.
- Ramp up (users started/second) ***: A text input field containing the value **1**.
- Host**: A text input field containing the URL **http://localhost:8000**.
- Advanced options**: A dropdown menu.
- START**: A large green button at the bottom.

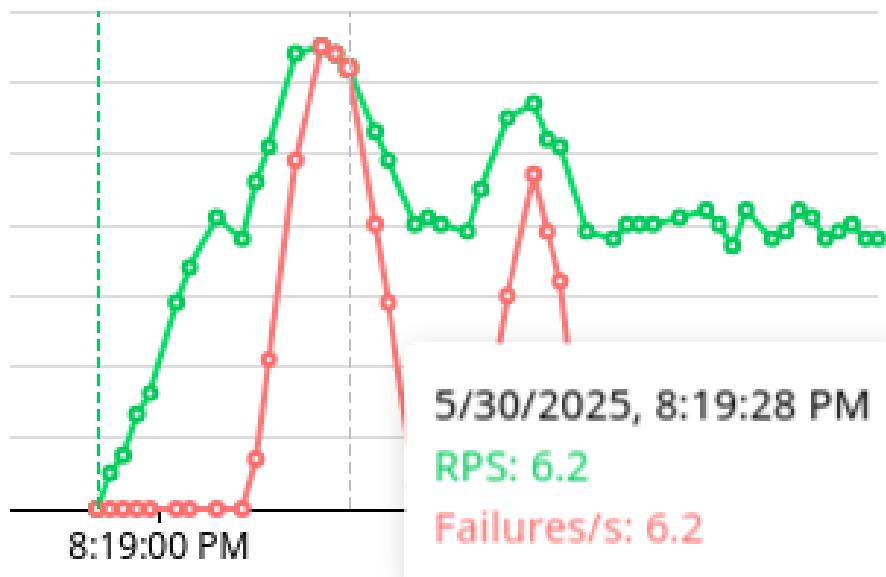
Ao clicar em start o locust começa a orquestrar as requisições simultaneamente, lembre-se que isso vai consumir recursos também da máquina onde o locust está sendo executado.

Ele vai executar até ser parado no botão de STOP ou automaticamente se você configurar o tempo limite da sessão em "advanced options". durante a execução ele vai atualizando gráficos em tempo real para monitorar o comportamento da aplicação.



Um dos principais objetivos deste tipo de teste é verificar o rácio entre sucessos e falhas, e, então, saber qual é o SLA e qual a carga que sua aplicação aguenta, além de testar se os load balancers estão funcionando corretamente.

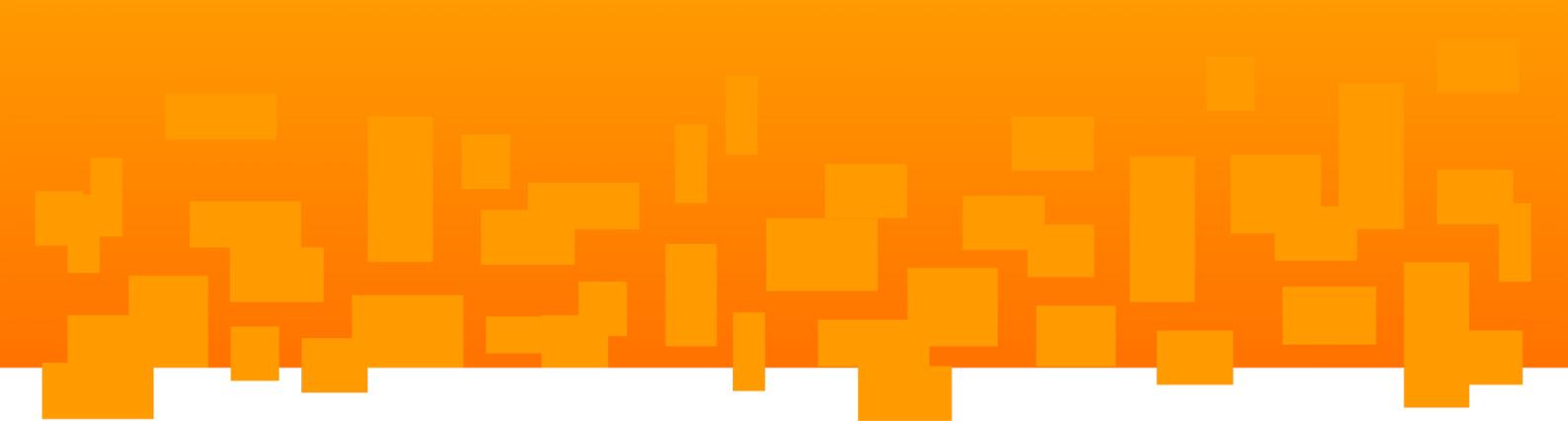
Run #2



5/30/2025, 8:19:28 PM
RPS: 6.2
Failures/s: 6.2

ATENÇÃO

O locust executa localmente, provavelmente em ambientes que estão atrás de proxy ou CDN, será efetuado um throttling, ou seja, limitação de acessos simultâneos, neste caso o Locust permite estratégias de randomizar user-agent, rotear saída por interfaces de rede diferentes e até rodar em cluster.



DEBUGGING EFICIENTE

Debugging interativo

SUPONHA QUE você tenha que resolver um bug no `dfgen` aquele programa CLI que gera Dockerfiles que criamos ali no capítulo sobre CLI, vamos assumir que o bug é que o arquivo está sendo escrito de forma incorreta e você quer inspecionar o file descriptor e o conteúdo.

```
Python
def generate(
    ...
):
    ...

    with open(output, "w") as f:
        # O que está acontecendo de errado aqui?
        f.write(dockerfile_content)

    ...
```

Você poderia simplesmente colocar um `print` mas não seria interativo e Python possui um debugger embutido, o `pdb` e o seu uso é bastante simples:

Executando o script em modo interativo faz com que o **pdb** pause a execução na primeira linha de execução e abre um prompt onde espera um comando, para listar todos os comandos digite **?**

```
Shell
> uv run python -m pdb src/dfgen/__init__.py
>
/home/rochacbruno/Projects/uv_123/dfgen/src/dfgen/__init__
--.py(1)<module>()
-> from cyclops import App
(Pdb)
```

Por exemplo, o comando **l** mostra qual é a linha atual em que o interpretador se encontra:

```
Shell
(Pdb) l
 1 -> from cyclops import App
 2     from jinja2 import Template
 3     from typing import Literal
```

Para seguir para a próxima linha digite **n** ou **next**, para ir até uma linha específica digite **until 74**

```
Shell
(Pdb) until 74
>
/home/rochacbruno/Projects/uv_123/dfgen/src/dfgen/__init__
--.py(74)<module>()
-> if __name__ == "__main__":
```

E neste momento podemos interativamente mudar a execução do programa, por exemplo, alterando o valor da variável **__name__** e usando **s** (step in), **c** (continue), ou **n** (next).

Este modo interativo é útil para quando queremos debugar um script sem a necessidade de editar o arquivo, porém, se pudermos editar o arquivo, tem um jeito mais fácil! Abra o `src/dfgen/__init__.py` e lá na linha 74 coloque um `breakpoint()`

```
Python
def generate(
    ...
):
    ...

    with open(output, "w") as f:
        breakpoint() # O que está acontecendo de errado
        aqui?
        f.write(dockerfile_content)

    ...
```

Agora execute o comando normalmente: `dfgen`

```
Shell
> dfgen
> ./.../dfgen/src/dfgen/__init__.py(69)generate()
-> breakpoint()
(Pdb) l
 64          workdir=cwd,
 65          run_commands=run,
 66      )
 67
 68      with open(output, "w") as f:
 69  ->      breakpoint()
 70          f.write(dockerfile_content)
 71
 72      print(f"Dockerfile gerado com sucesso em:
{output}")
 73
 74
(Pdb)
```

Pronto, agora o programa está pausado exatamente no ponto onde você pode interativamente inspecionar o `f` para ver se está tudo certo com o descritor de arquivos, inclusive pode até ir olhar lá no sistema operacional em `/proc` para ver se o FS está lá mesmo, ou até simular um `OSError` ou um `Race Condition`, e o mais importante, pode simplesmente digitar `dockerfile_content` para ver se o conteúdo foi gerado com sucesso.

DICA

A interface do `pdb` é bastante primitiva. Existe um sucessor dele chamado `ipdb` que oferece um terminal baseado no `ipython`, portanto, a experiência é bem melhor para usar o `ipdb`

Shell

```
(Pdb) dockerfile_content
'FROM alpine\nRUN useradd -ms /bin/bash root\nUSER
root\nWORKDIR /\nEXPOSE 80\nCMD ["sh"]\n'
```

Python

```
> uv add --dev ipdb
> export PYTHONBREAKPOINT=ipdb.set_trace
> dfgen
>
/home/rochacbruno/Projects/uv_123/dfgen/src/dfgen/__init
---py(69)generate()
    68      with open(output, "w") as f:
---> 69          breakpoint()
    70      f.write(dockerfile_content)

ipdb>
```

Agora você tem um debugger com bem mais funcionalidades e suporte a syntax highlight.

DICA BÔNUS

Se você gosta de ferramentas mais visuais pode testar o `pudb` ou o `winpdb` o funcionamento é praticamente o mesmo, a diferença é que eles iniciam interfaces mais amigáveis, além disso você pode estender o ``pdb`` a partir da instalação de um plugin, o `pdb++` com `uv add --dev pdbpp` assim que é instalado ele se injeta no `pdb` do sistema, basta usar o `pdb` normalmente que ele terá superpoderes.

Debugging remoto

ANTERIORMENTE FALAMOS sobre debugar um programa que está rodando na sua máquina local, mas **e se o programa estiver sendo executado remotamente?** Dentro de um container docker, por exemplo, ou em um pod do seu cluster K8S. Neste caso podemos executar o **pdb** mas faze-lo ser acessível através de uma porta TCP.

Para simular este caso precisaremos de um **Dockerfile** e um programa Python para debugarmos e um debugger remoto chamado **remote-pdb**

Textproto

```
FROM python:3.13-slim-bookworm
RUN apt-get update && apt-get install -y
--no-install-recommends curl ca-certificates
ADD https://astral.sh/uv/install.sh /uv-installer.sh
RUN sh /uv-installer.sh && rm /uv-installer.sh
ENV PATH="/root/.local/bin/:$PATH"
WORKDIR /app
COPY script.py /app/script.py
EXPOSE 4444

ENV PYTHONBREAKPOINT=remote_pdb.set_trace
ENV REMOTE_PDB_HOST=0.0.0.0
ENV REMOTE_PDB_PORT=4444

CMD [ "uv", "run", "script.py" ]
```

O programa `script.py`

```
Python
# /// script
# requires-python = ">=3.13"
# dependencies = [
#     "remote-pdb",
# ]
# ///

def triangle_area(base, height):
    breakpoint()
    return (base * height) / 2

if __name__ == "__main__":
    print(triangle_area(3, 4))
```

E agora fazemos o build da imagem e executamos

```
Shell
> docker build -t triangle -f Dockerfile .

> docker run --net=host --rm -it triangle
Installed 1 package in 1ms
RemotePdb session open at 0.0.0.0:4444, waiting for
connection ...
```

Assim, abrimos um segundo terminal e conectamos ao debugger, a conexão pode ser feita a partir de qualquer cliente `telnet`

Shell

```
> telnet 127.0.0.1 4444
# com a inconvenicencia de que ao terminar precisa
digitar ctrl-d para sair

# ou o netcat que permite simplesmente usar `exit` para
sair:
> netcat 127.0.0.1 4444
> /app/script.py(9)triangle_area()
-> breakpoint()
(Pdb) exit
```

As alternativas são ainda o **nc** e o **socat** e se você uma IDE como VSCode ou NeoVIM, pode usar o **debugpy** que é um debugger com suporte a acesso remoto que implementa o protocolo DAP (Debug Adapter Protocol), dentro dessas IDEs provavelmente não será preciso configurar muita coisa.

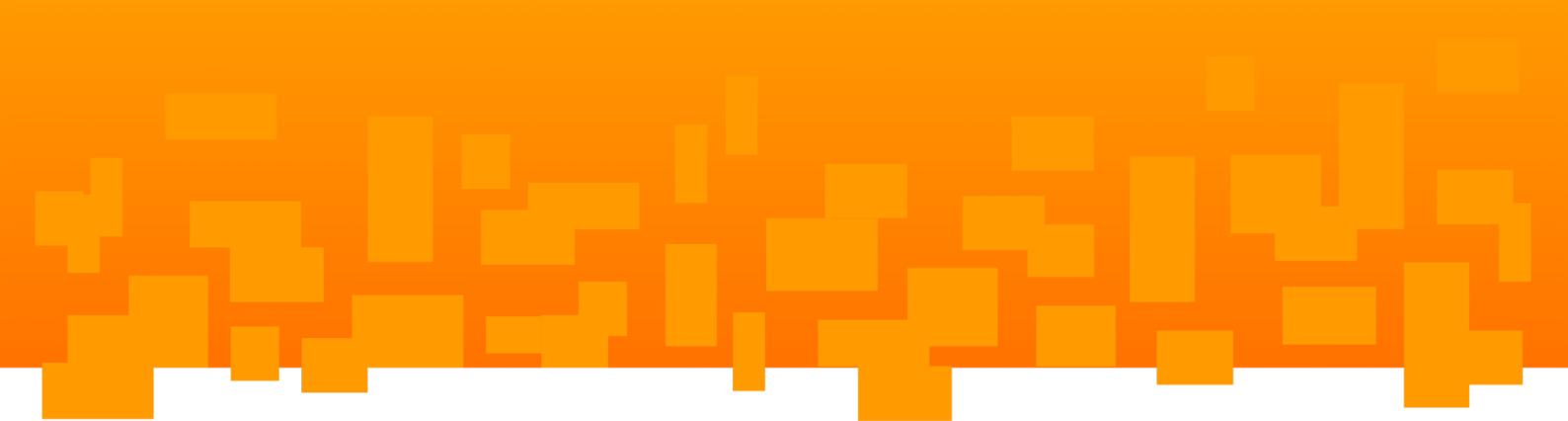
DICA

Mesmo que você seja usuário de IDE, recomendo saber usar um debugger na linha de comando pois existem casos onde o debugging tem que acontecer em ambientes limitados, ou, casos onde a própria IDE causa problemas de execução que queremos evitar.



Por meio do pdb, a depuração se transforma em uma jornada de entendimento e aprendizado. Ele nos permite mergulhar nas profundezas do nosso código, explorando seu comportamento, dissecando os bugs e iluminando os caminhos que nosso programa percorre.”

– Sunscrapers



FERRAMENTAS DE PRODUTIVIDADE

Tarefas com Taskipy

TODO PROJETO tem tarefas administrativas que são repetitivas, vamos pegar por exemplo nosso projeto `dfgen` queremos garantir que a qualidade de código está correta, que os testes executem, que um arquivo de requirements possa ser gerado, que os arquivos temporários sejam limpos, etc etc etc...

Já é conhecido de quem trabalha com linux o uso da ferramenta `make` com `Makefile` para declarar essas tarefas, **mas você sabia que isso é uma das maiores gambiarras da computação?** O make não foi feito para rodar seus testes! ele foi feito para automatizar a criação de arquivos, por isso o nome.

No Python Moderno existe uma solução bastante simples, o `taskipy`.

```
None  
› cd dfgen  
› uv add --dev taskipy
```

Agora podemos editar o `pyproject.toml` e lá no final declarar as nossas tarefas.

```
Textproto
[tool.taskipy.tasks]
check = "uvx ruff check"
fix = "uvx ruff fix"
lint = "uvx ruff lint"
format = "uvx ruff format"
clean = "find . -name '*.pyc' -delete && find . -name
'__pycache__' -delete"
test = "uvx pytest -v"
```

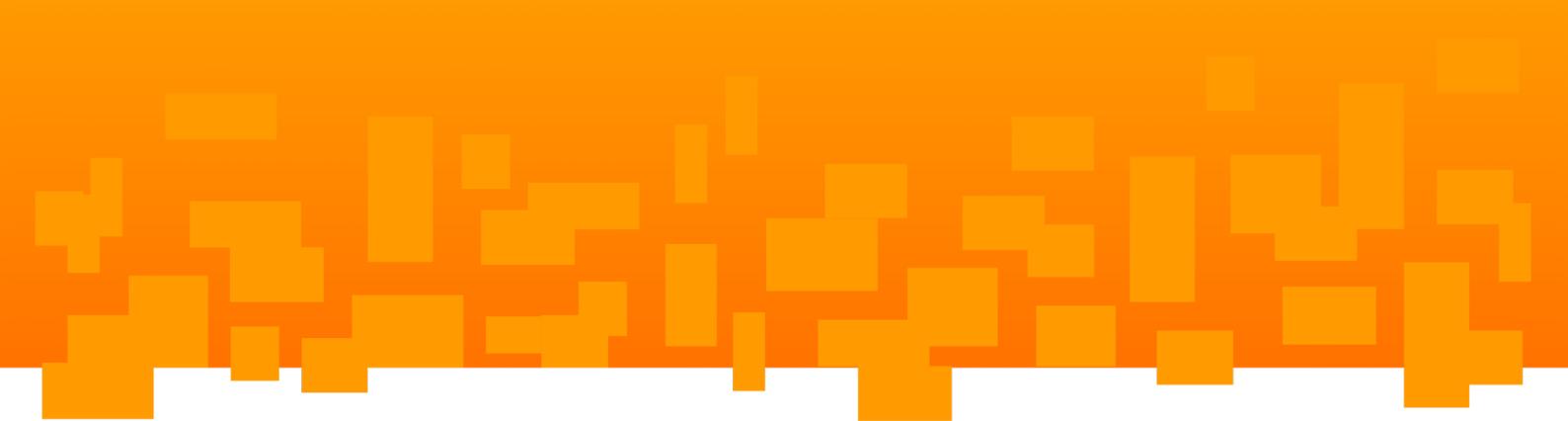
Agora basta executar os comandos

```
None
> uv run task -l
check  uvx ruff check
fix    uvx ruff check --fix
format uvx ruff format
clean   find . -name '*.pyc' -delete && find . -name
'__pycache__' -delete
test    uv run pytest -v

> uv run task check
All checks passed!

> uv run task format
1 file reformatted, 2 files left unchanged
```

A grande vantagem nesse caso é que não é mais preciso lembrar a sequência exata de comandos para executar, estará sempre disponível no `task -l`



INTERAÇÃO COM SERVIÇOS EXTERNOS

Integração com Git

CONSIDERANDO UM repositório Git que esteja rodando localmente (não estamos falando de Github nem Gitlab, apenas git), você pode usar Python para efetuar ações no repositório. Por exemplo: automatizar as tarefas recorrentes no repositório ou criar um CLI alternativo para o GIT ou, até mesmo, usar informações do repositório em seu script de deploy.

Este exemplo cria uma nova branch, adiciona todos os arquivos, adiciona um commit e no final, gera um patch. Isso pode ser muito útil para criar uma ferramenta de auditoria ou até mesmo para usar essas informações em uma integração com IA.

Python

```
import os
from git import Repo

# Caminho para o repositório Git local
repo_path = '/caminho/para/seu/repositorio' # Substitua
pelo caminho real
repo = Repo(repo_path)

# 1. Criar uma nova branch
new_branch = 'minha_nova_branch'
repo.git.checkout('HEAD', b=new_branch) # Cria e muda
para a nova branch

# 2. Adicionar arquivos modificados ao stage
repo.git.add('--all') # Adiciona todas as alterações
```

```

# 3. Realizar um commit
commit_message = 'Adicionando alterações na nova branch'
repo.index.commit(commit_message)

# 4. Gerar um patch com as alterações
# Obtém o hash do commit anterior
previous_commit = repo.git.rev_list('--max-parents=0',
'HEAD').split('\n')[-1]

# Gera o patch entre o commit anterior e o atual
patch = repo.git.diff(f'{previous_commit}..HEAD')

# Salva o patch em um arquivo
patch_file_path = os.path.join(repo_path,
'alteracoes.patch')
with open(patch_file_path, 'w') as patch_file:
    patch_file.write(patch)

print(f'Patch gerado e salvo em: {patch_file_path}')

```

Para executar o exemplo

```

None
> uv run --with GitPython autogit.py
Patch gerado e salvo em: ./alteracoes.patch

```

DICA

Algumas pessoas confundem git com github ou gitlab sendo que esses 2 últimos são na verdade serviços de hospedagem, além da biblioteca GitPython que acabamos de usar, também existem as libs PyGithub e a PyGitlab que podem ser usadas em conjunto com o GitPython para automatizar tarefas no serviço, por exemplo, listando issues ou criando pull requests.

Docker

DOCKER É uma das ferramentas mais úteis para quem trabalha com DevOps, a biblioteca **docker-py** fornece uma interface para interagir com docker em programas Python, sendo possível obter informações, útil por exemplo para criar uma ferramenta de monitoramento e também

permite efetuar ações no docker como criar containers, fazer build de imagens etc, o que é útil para criar um painel de controle ou um CLI customizado. Veja um exemplo simples de como listar imagens docker do seu sistema usando Python:

Python

```
# meudocker.py
import docker

def listar_imagens():
    client = docker.from_env()

    try:
        imagens = client.images.list()
        if not imagens:
            print("Nenhuma imagem Docker encontrada.")
            return

        print("Imagens Docker disponíveis:\n")
        for imagem in imagens:
            tags = imagem.tags or ["<sem tag>"]
            print(f"ID: {imagem.short_id} | Tags: {''.join(tags)}")
    except docker.errors.DockerException as e:
        print(f"Erro ao conectar com o Docker: {e}")

if __name__ == "__main__":
    listar_imagens()
```

E para executar podemos informar ao uv que precisamo da lib com **--with docker** ou se preferir execute **uv add -script meudocker.py docker**

Shell

```
> uv run --with docker meudocker.py
```

Imagens Docker disponíveis:

```
ID: sha256:a3a19805164f | Tags: triangle:latest
ID: sha256:7e0ca7ab287e | Tags: <sem tag>
ID: sha256:94c11a9d40e3 | Tags: nginx:latest
ID: sha256:f47b47288272 | Tags: postgres:13
ID: sha256:99ee9af2b6b1 | Tags: redis:5
```

Quer um desafio?

Crie já um CLI (como fizemos com o dfgen) que:

1. Recebe argumentos para executar containers docker, mas usa um prompt de texto ao invés de parâmetros.

meudocker “roda o nginx na porta 8080”

Isso pode ser feito de diversas maneiras, como usar regex para extrair os padrões, pode dar um `.split()` no prompt e então ir construindo o comando final iterativamente ou pode se aventurar em usar um agente de AI com MCP.

Kubernetes

PARA FINALIZAR, vamos agora falar sobre a ferramenta mais importante atualmente para DevOps, o Kubernetes, e sim, é possível usar Python para automatizar tarefas ou até mesmo criar um painel ou uma CLI para esta ferramenta.

Para facilitar, vamos usar o GIRUS, uma ferramenta da LINUXtips para facilitar o aprendizado com Kubernetes.

CLIQUE AQUI para instalar o GIRUS

<https://linuxtips.io/girus-labs/>

Shell

```
> curl -fsSL https://girus.linuxtips.io | bash  
...
```

 PRÓXIMOS PASSOS:

1. Para criar um novo cluster Kubernetes e instalar o Girus:

```
$ girus create cluster
```

2. Após a criação do cluster, acesse o Girus no navegador:

```
http://localhost:8000
```

3. No navegador, inicie o laboratório Linux de boas-vindas para conhecer a plataforma e começar sua experiência com o Girus!

O próprio instalador vai te indicar os próximos passos, então, siga todos eles. E também pode demorar um pouco para criar o cluster.

Shell

```
> kubectl config get-contexts
CURRENT      NAME           CLUSTER      AUTHINFO
NAMESPACE
*            aula            aula          aula          default
                  kind-girus     kind-girus   kind-girus

> kubectl get deployments
No resources found in default namespace.
```

Vamos fazer o deploy do **nginx**

Textproto

```
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:latest
        ports:
          - containerPort: 80
```

Salve o arquivo e execute

Shell

```
> kubectl apply -f deployment.yaml
deployment.apps/nginx-deployment created

> kubectl get deployments
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2     2           0          4s

> kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
nginx-deployment-544dc8b7c4-5l7kz   1/1     Running   0          104s
nginx-deployment-544dc8b7c4-7cwv8   1/1     Running   0          104s
```

Maravilha! Temos um 2 pods rodando o NGINX. É o suficiente para nosso experimento. Vamos usar Python para obter as mesmas informações.

Python

```
# list_pods.py
from kubernetes import client, config
config.load_kube_config()
v1 = client.CoreV1Api()

print("Listing pods with their IPs:")
pods = v1.list_pod_for_all_namespaces(watch=False)
for pod in pods.items:

    print(f'{pod.status.pod_ip}\t{pod.metadata.namespace}\t{pod.metadata.name}')
```

Shell

```
> uv run --with kubernetes list_pods.py
Listing pods with their IPs:
10.244.0.7 default    nginx-deployment-544dc8b7c4-517kz
10.244.0.8 default    nginx-deployment-544dc8b7c4-7cwv8
10.244.0.3 girus      girus-backend-6cf55844b-qhbd6
10.244.0.5 girus      girus-frontend-64f7649747-mbqcb
```

Este exemplo foi propositalmente bem simples, pois as possibilidades de uso do kubernetes-client com Python são imensas e o repositório oficial oferece exemplos mais completos

CLIQUE AQUI para acessar o site

<https://github.com/kubernetes-client/python>

Quer mais um desafio?

Crie uma ferramenta CLI ou API, que:

1. Recebe os argumentos do `deployment.yaml`
2. Salva o arquivo temporariamente.
3. Aplica o deployment.
4. Remove o arquivo temporário
5. E também exibe o status dos pods deste deployment:

```
meuk8s -name postgres -image
postres:latest -replicas 3 -
port 5432.
```

Ou, se quiser usar mais a criatividade, regex, parsing ou AI:

```
meuk8s "faz deploy do nginx com
2 replicas na porta 80"
```

CONCLUSÃO

NESTE GUIA, exploramos o poder do Python Moderno para DevOps, focando em ferramentas e técnicas práticas. Vimos como configurar ambientes com UV, garantir qualidade de código com Ruff, validar dados com Pydantic e gerenciar configurações com Dynaconf. Aprendemos a criar e consumir APIs REST com FastAPI, automatizar tarefas com CLIs e Taskipy, e interagir com serviços externos como Git, Docker e Kubernetes. Além disso, abordamos testes com Pytest e Locust, debugging eficiente com PDB e ferramentas de produtividade. Dominar essas habilidades transforma o Python em um aliado essencial para a automação e eficiência em DevOps.



PRÓXIMOS PASSOS

Python para Devops

CHEGAMOS AO fim da nossa conversa, mas para você, é apenas o começo de uma jornada incrível no mundo da tecnologia. Se você leu até aqui, é porque entende a importância de se manter atualizado, de buscar conhecimento prático e de se destacar em um mercado que não para de evoluir. O cenário de DevOps exige profissionais completos, que não só entendem os conceitos, mas que sabem aplicá-los na prática, otimizando processos e construindo soluções robustas. É por isso que dominar ferramentas como o Python se tornou um diferencial inegável. Mas como levar esse conhecimento para o próximo nível e aplicá-lo de verdade em um ambiente DevOps? A resposta está no novo treinamento exclusivo **Python para DevOps** da LINUXtips!

Este treinamento é uma experiência prática, focada em te dar as ferramentas e o conhecimento para ser um profissional de ponta. Desenvolvido por especialistas que vivem o DevOps, o treinamento vai te capacitar a usar o Python para automatizar tarefas repetitivas, provisionar infraestrutura, gerenciar configurações e construir pipelines de CI/CD que realmente funcionam.

O que você vai aprender e colocar em prática?

FUNDAMENTOS DO PYTHON PARA AUTOMAÇÃO entenda a estrutura e as melhores práticas para escrever scripts eficientes.

NETWORKING E SEGURANÇA como o Python pode te ajudar a gerenciar redes e garantir a segurança dos seus ambientes.

MANIPULAÇÃO DE DADOS E OBSERVA-BILIDADE trabalhe já com diferentes formatos de dados e aprenda como monitorar suas aplicações e sistemas.

AUTOMAÇÃO COM FERRAMENTAS DEVOPS aprenda a integrar o Python com ferramentas como Ansible para orquestrar e automatizar processos complexos.

CONSTRUÇÃO DE CLIs E DE APIs Desenvolva as próprias ferramentas de linha de comando e APIs para interagir com seus sistemas.

Seja você um Engenheiro DevOps que busca otimizar rotinas, um Desenvolvedor Backend querendo elevar suas APIs, ou um profissional de TI que deseja entrar de vez no mundo da automação, o "Python para DevOps" foi feito para você. Prepare-se para projetos reais, desafios que simulam o dia a dia do mercado e um aprendizado que te coloca à frente.

CLIQUE AQUI para garantir a sua vaga e saber mais sobre o novo treinamento Python para Devops da LINUXtips

<https://linuxtips.io/python-para-devops/>



Outros treinamentos

A LINUXtips oferece uma gama completa de treinamentos para DevOps e desenvolvimento de software que podem complementar o seu conhecimento.

Se você tem interesse em entender tudo sobre a linguagem Python em detalhes e explorar cada um dos elementos por trás da linguagem, o treinamento **Python Base** oferece 64 horas + aulas ao vivo de conteúdo que não tem nada de básico!

CLIQUE AQUI para acessar o treinamento Python Base da LINUXtips

<https://linuxtips.io/treinamento/python-base/>

Agora, se você já conhece a base de Python e quer se aprofundar ainda mais em desenvolvimento web, o treinamento **Python Web API** aborda desde os princípios básicos da internet, comunicação em rede, criação do seu próprio web framework do zero e o uso dos principais frameworks e ferramentas de mercado como Flask, Django e FastAPI em projetos reais.

CLIQUE AQUI para acessar o treinamento Python Web API da LINUXtips

<https://linuxtips.io/treinamento/python-web-api/>

CONSIDERAÇÕES FINAIS

Agradecimento especial para todas as pessoas que acompanham o meu conteúdo nos canais e na LINUXtips, a toda a equipe da LINUXtips por fazer a revolução no ensino de tecnologia acontecer e principalmente a minha família, minha esposa Karla e meu filho Erik.

E é claro a você que leu isso tudo até o final! Obrigado!

