

[Curso python]

Python para administradores de sistemas

<https://github.com/APSL/curso-python-sistemas>

Bernardo Cabezas Serra
bcabezas@apsl.net
[@bercab](https://twitter.com/bercab)



Agenda - día 3

Contexto: Revisión trabajo sysadmin, DevOps, SRE. Automatización

Python sistemas/SRE. Casos uso. Puntos fuertes y débiles.

Taller fabric

Procesos y comunicación. Scripts python.

Estructura scripts potente: click, python-sh.

Distribución y despliegue scripts

Bonus: Interfaces DevOps rundeck. Monitoring check-mk.

Objetivos - día 3

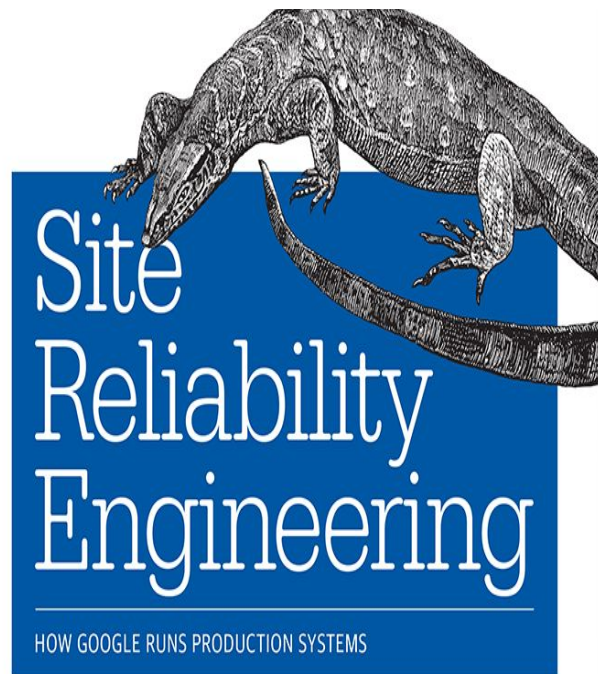
- Contexto: Nuestra visión sobre el papel de programación y python en SRE
- Ver ejemplo práctico administración con fabric
- Entender comunicación entre sistemas vía procesos.
- De scripts con stdlib a click y python-sh
 - stdlib: sys (stdin, stderr, status_code)
- Ejemplos rápidos de algunas librerías:
 - requests, psycpg2, ldap, email, ftp
 - logs y trazas: logging, sentry
- Entender distribución software sistemas.
- Crear script monitorización NRPE
- Crear interfaz usuario final DevOps

¿Qué hacemos en sistemas? Visión APSL

Site Reliability Engineer

- Ingeniería de **confiabilidad** de la plataforma
- Incorpora aspectos de ingeniería del software a **infraestructura y operaciones**
- Define muy bien nuestra profesión.
- Programamos para automatizar y orquestar.

<https://landing.google.com/sre/books/>

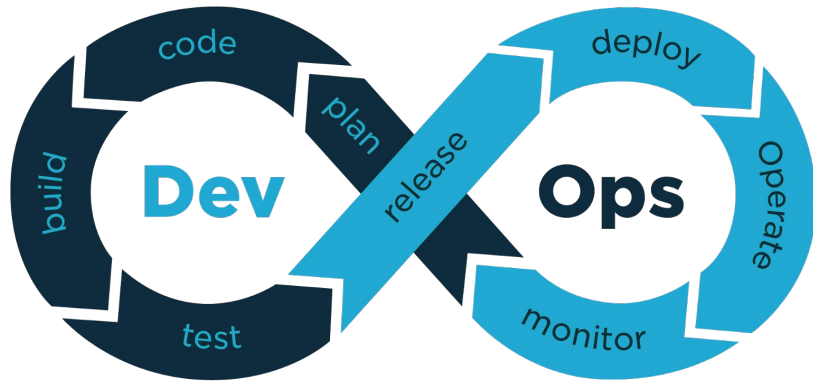


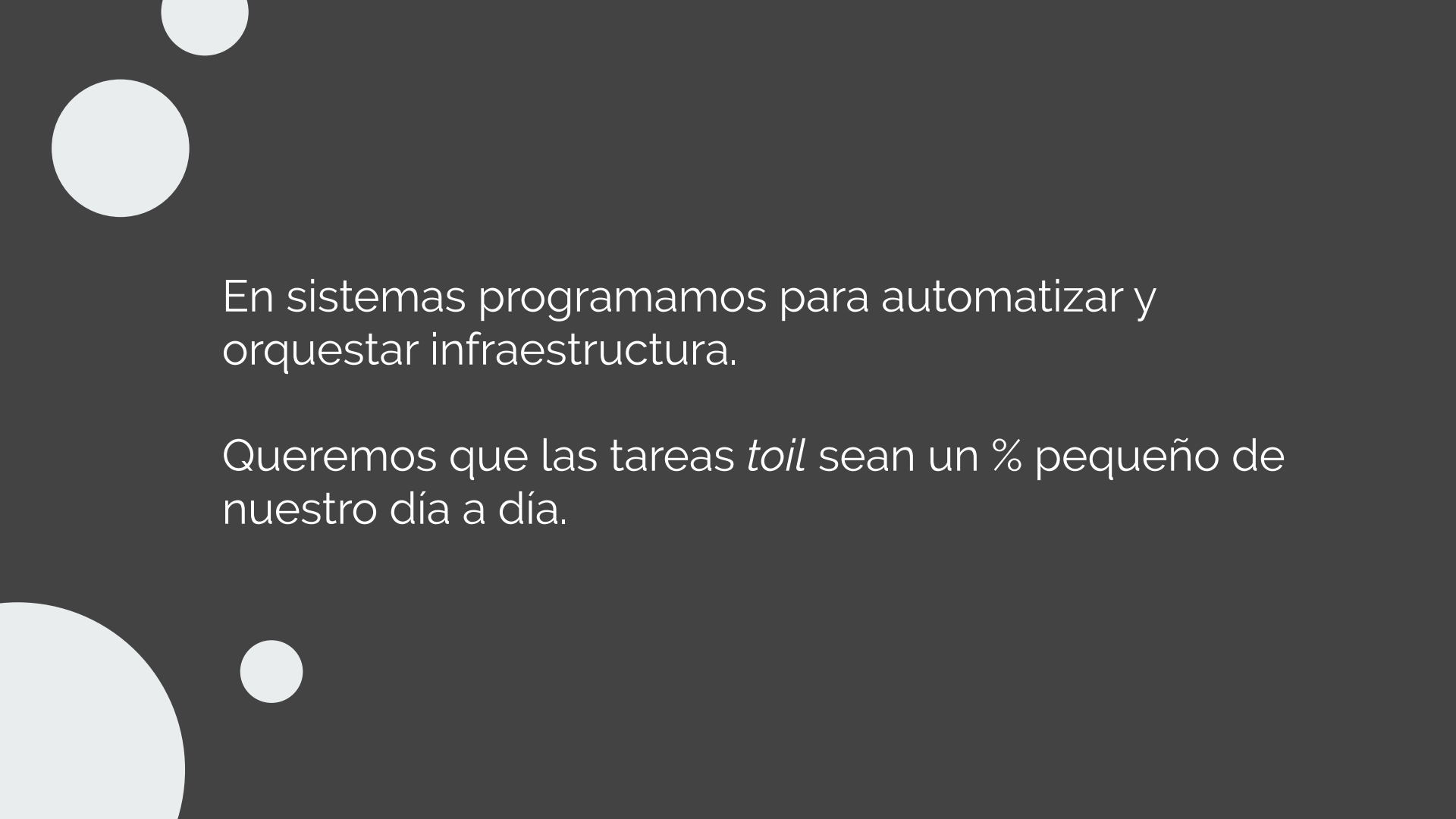
Edited by Betsy Beyer, Chris Jones,
Jennifer Petoff & Niall Murphy



1. Leverage tooling and **automation**
 - SREs have a charter to **automate menial tasks (called "toil")** away
2. Measure everything
 - SRE defines prescriptive ways to measure values
 - SRE fundamentally believes that **systems operation is a software problem**
3. Reduce organizational silos
 - SRE shares ownership with developers to create shared responsibility
 - SREs use the same tools that developers use, and vice versa
4. Accept failure as normal
 - SREs **embrace risk**
 - SRE quantifies failure and availability in a prescriptive manner using **Service Level Indicators (SLIs)** and **Service Level Objectives (SLOs)**
 - SRE mandates blameless **post mortems**
5. Implement gradual changes
 - SRE encourages developers and product owners to move quickly by reducing the cost of failure

SRE vs DevOps





En sistemas programamos para automatizar y orquestar infraestructura.

Queremos que las tareas *toil* sean un % pequeño de nuestro día a día.



Tipos de tareas , punto de vista de quien lo usa

- **Pegamento:** conexión entre distintos sistemas, plataformas o servicios. Extensión funcionalidad.
 - CI/CD: script final de monitorización.
 - Sistemas monitorización y alertas
- Tareas periódicas (cron)
- Scripts CLI administración para sysops
- Scripts CLI para developers
- GUI usuario final o DevOps. Proporcionar herramientas.
- Orquestación: emisión y recepción eventos distribuidos.
 - Respuesta a eventos o fallos. Reinicio servicio.
 - Tareas respuesta a recepción ficheros.
- Apps completas sistemas (inventario aplicaciones, chatops)



Programación SRE. Casos uso.

- **Pegamento entre distintos servicios / software / plataformas**
 - Control de un servicio a otro: (ej: script Despliegue continuo jenkins hacia server/cloud).
 - Plugin monitorización: Tamaño de DB, % uso servidor FTP copias.
 - Cambio formato entrada - salida para unir sistemas (parsing).
 - Notificación en monitorización
 - bots telegram, slack.
 - email
 - La interfaz del “pegamento” puede ser por CLI, pero también por API/librería o plugin.
 - Extensiones python: Plugins alerta.io
 - API / Librería: Exportadores prometheus
 - **CLI: plugin nagios NRPE**
- **Tareas puntuales de sistemas que requieren parseo, ordenación, ejecución.**
 - Conversión masiva ficheros. Cambiar encoding, nombre, resize imágenes.
 - Borrado masivo, búsqueda duplicados
 - Análisis de logs en búsqueda de algo concreto
 - Scrapping web.
- **Tareas periódicas con cierta criticidad y control.**
 - Movimiento datos para aplicaciones. dump / restore databases.
 - Anonimización
 - Sincronización de ciertos ficheros con acciones puntuales ante error y alertas:
 - Copia de FTP a S3. Parseo ficheros recibidos y realizar acción: cambio formato, ejecución condicional, notificaciones.



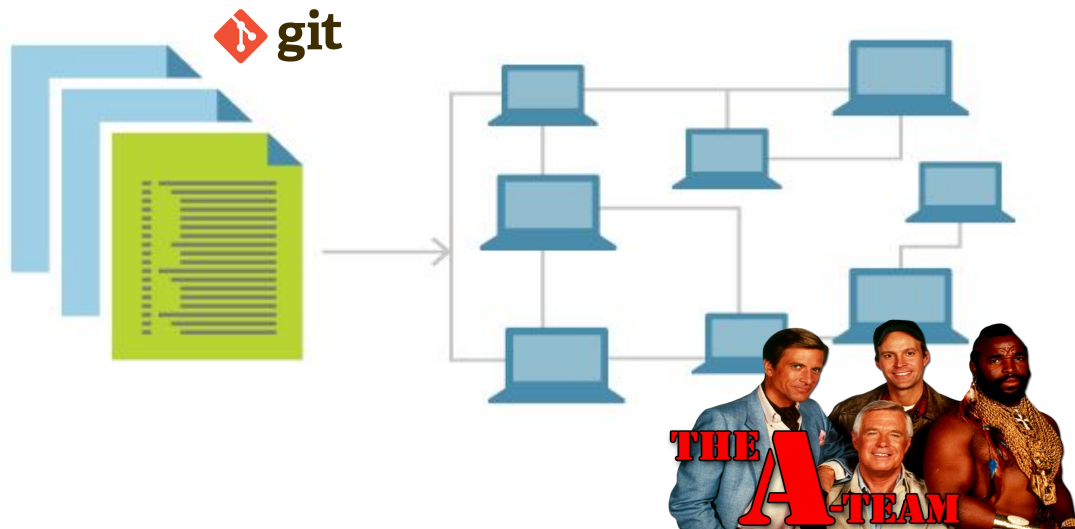
Programación SRE. Casos uso (II)

- Despliegue apps. Interfaz DevOps
 - Update git/mercurial, reinicio, migraciones DB
 - Limpiar caché app
 - Copia DB a PRE
 - Listar ficheros
- Backup customizado. Donde no pueden llegar herramientas comunes.
- Monitorización y métricas
 - Alarmas para nagios NRPE. Ej:
 - FTP: %uso, disponibilidad.
 - DB: locks, Tamaño total, tamaño db, num rows tabla, etc.
 - Alarmas parsing logs
 - Prometheus exporters
- Reporting
 - Extracción información logs
 - Extracción información DB
- Orquestación
 - CI / CD. scripts despliegue on-premise, AWS, GCE, Azure.
 - Orientación a eventos en sistemas legacy. Revisión filesystem, recepción ficheros. Disparar eventos.
- chatops
- Scrapping



IaC: Infrastructure as Code

... is the process of **managing** and **provisioning** computer data centers through machine-readable definition





- Beneficios:

- Auto-documentación infraestructura
- Eficiencia: automatización y velocidad.
 - Minimización coste.
- Control de versiones y auditoría.
 - Procedimientos establecidos
 - Colaboración equipos.
- Reducción riesgo.
 - Rollback rápido cambios
 - Bare metal restore.

- Gestión de la configuración - IaC - Ejemplos:

- Despliegue de un entorno de PRE desde cero
- Contingencia (bare metal recovery)
- Cambios en infraestructura controlados

Herramientas:

- Config servicios en Instancias:

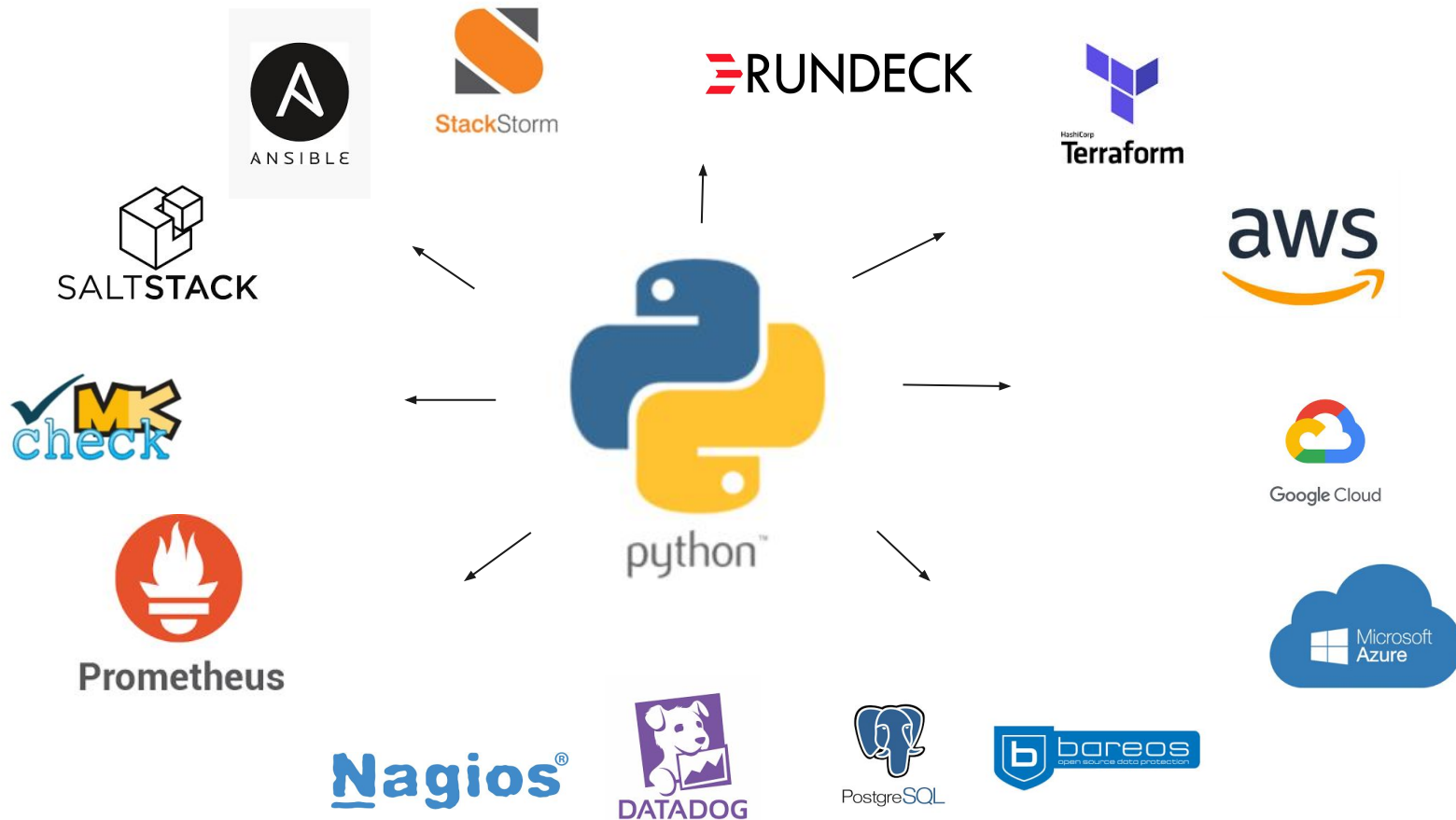
- **saltstack**
- ***fabric (no por defecto)***
- **ansible**
- **puppet**
- **chef**

- Creación infraestructura Cloud:

- **terraform**
- **AWS CloudFormation**
- **packer**
- **saltstack -> salt-cloud**



Panorámica sistemas



¿Qué ofrece python a un administrador de sistemas? (**frente a shellscript/powershell**)

- Usar las mejores prácticas de programación en creación scripts:
 - Los scripts se convierten en programas **mantenibles**, con ciclo vida software (build, release, deploy)
 - Monitorización y trazabilidad. Logs y gestión excepciones: **logging, sentry**
 - **Independencia de sistema Operativo**: Linux, Windows, Mac OS X
 - **Separación de configuración y código** en scripts.
 - Orientación a objetos, TDD.
- Hace las tareas comunes sencillas:
 - Trabajo con ficheros (txt, json, yaml, csv, xml)
 - Trabajo con expresiones regulares muy sencillo.
 - Módulo os permite funcionalidades independientes de S.O.
 - Shutil operaciones alto nivel.
 - Parametrización scripts sencilla con **click**.
 - Iteraciones, condicionales, excepciones.
 - Tipos datos avanzados: listas, diccionarios, **sets**.
 - Métodos mágicos clases: Orientación objetos útil en sistemas.
 - Uso APIs remotas sencillo
 - Wrappers comandos sistemas ultra sencillo con **python-sh** o subprocess.



- Extensión de software existente mediante **plugins python**:
 - Alertas nagios, check-mk, NRPE, datadog, prometheus
 - Detección anomalías TICK (Telegraf + Influx + Cronograph + **Kapacitor**)
 - Plugins alerta.io
 - Extensión módulos y estados saltstack
 - Recetas fabric
- Facilidad pruebas rápidas e introspección APIS
 - ipython
 - jupyter notebook
 - ipdb
- Pilas incluidas!
 - stdlib muy potente
 - Concurrencia integrada
- Procesamiento distribuido
 - paramiko
 - celery
 - fabric
 - saltstack events & modules,
- Servicios rápidos
 - flask, django



Puntos débiles

- Lenguaje interpretado. Código no nativ. Necesita intérprete python pre-instalado ~= 240MB
 - Compilados: Go, Rust, julia
- Gestión dependencias mejorable (común a todos los lenguajes, pero rust y Go lo tienen mejor. Avances con pipenv y poetry.
- Instalación en servidores y conflicto con versiones de librerías existentes
 - virtualenv
- Paralelismo y GIL



¿Cuándo usar python en lugar de shellscript?

- Siempre buscar soluciones hechas, en la medida de lo posible.
 - rclone, rsync, sistemas backup. No reinventar la rueda
- Preferencia sobre shellscript cuando:
 - Control errores y trazabilidad necesario
 - Tratamiento fechas
 - Mas de 10 líneas :P
 - Iteraciones
 - Obvia: acceso APIs

¿Cómo elegir paquete python?

- Revisar paquete en pypi
- Revisar actividad y estrellas github
- Revisar si tiene documentación en readthedocs
- awesome python

A teal background with several white circles of varying sizes. One large circle is in the top left, another large one is in the bottom left, and several smaller ones are scattered around.

Siguientes pasos

Taller fabric

Procesos y comunicación. Scripts python.

Estructura scripts potente: click, python-sh.

Distribución y despliegue scripts

Bonus: Interfaces DevOps rundeck. Monitoring check-mk.



Fabric is a high level Python (2.7, 3.4+) library designed to execute shell commands remotely over SSH, yielding useful Python objects in return

1

Fabric 1: Orientado a devops. Ejecución sencilla comandos remotos.

- Creamos un fabfile.py
- Declaramos comandos
- Ejecutamos fab <comando> en el mismo directorio

2

Fabric 2: Mejorado y extendido para ser usado como API

```
pip2 install "fabric<2"
```

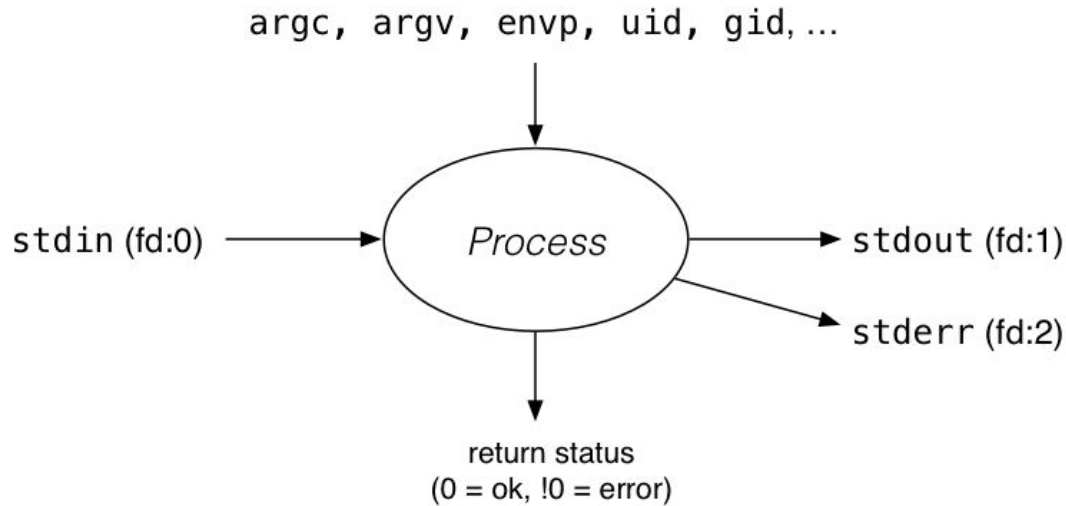


```
from fabric.api import task, run, sudo, env, get
from fabric.context_managers import cd, prefix, settings
from fabric.contrib.files import upload_template, exists
from fabric.utils import abort, puts
from fabric.contrib.console import confirm
import warnings
warnings.filterwarnings(action='ignore', module='*.paramiko.*')
```

```
env.hosts = ['kiwi7.apsl.net']
env.port = 2232
#env.user = "ubuntu"
```

```
@task()
def uname():
    out = run('uname -a')
    puts(out)
```

```
$ fab -l
uname
```



- Usamos la comunicación de procesos unix como “**API**” estándar al crear scripts
 - el mencionado pegamento :). ej: jenkins CI-CD con GCE (k8s), plugins NRPE
- El proceso puede recibir información vía argumentos, variables de entorno, stdin o signals.
- El proceso devuelve información via stdout, stderr y return status



```
#!/bin/sh
```

```
echo "Todos los argumentos: $0"
```

```
echo "arg 1: $1"
```

```
echo "arg 2: $2"
```

```
echo "Salida stdout"
```

```
echo "Salida stderr" 1>&2
```

```
echo "VAR: $VAR"
```

```
exit 8
```

```
#!/usr/bin/env python
```

```
import sys
```

```
import os
```

```
if not sys.stdin.isatty():
```

```
    print("Las dos primeras lineas de tu stdin son:")
```

```
    lines = ["{} -> {}".format(n, line.strip()) for n,line in enumerate(sys.stdin)]
```

```
    lines = lines[:2]
```

```
    print("\n".join(lines))
```

```
sys.stdout.write('Salida Stdout\n')
```

```
sys.stderr.write('Salida Stderr\n')
```

```
print(sys.argv)
```

```
print("VAR: {}".format(os.environ.get("VAR")))
```

```
sys.exit(8)
```



Estructura básica script NRPE: ([gist link](#))

```
#!/usr/bin/env python3
import sys

NAGIOSCODES = {
    'OK': 0,
    'WARNING': 1,
    'CRITICAL': 2,
    'UNKNOWN': 3,
    'DEPENDENT': 4,
}

def nagios_return(code, msg):
    """ prints the response message
    and exits the script with one
    of the defined exit codes
    DOES NOT RETURN
    """
    print("{} : {}".format(code, msg))
    sys.exit(NAGIOSCODES[code])

def test_stuff(warr=10, crit=20):
    try:
        1/0 #example something fails
    except Exception as e:
        return("UNKNOWN", str(e))

    # test stuff
    return(OK, "all ok")

if __name__ == '__main__':
    code, msg = test_stuff(10, 20)
    nagios_return(code, msg)
```

Ver API NRPE:

<https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/3/en/pluginapi.html>





```
#!/usr/bin/env python3
```

```
import argparse
```

```
def main():
```

```
    parser = argparse.ArgumentParser(description='Process some integers.')
```

```
    parser.add_argument("-v", "--verbose", help="increase output verbosity",  
                        action="store_true")
```

```
    parser.add_argument("-n", "--num", help="float number",  
                        type=float, default=0.1)
```

```
    args = parser.parse_args()
```

```
    if args.verbose:
```

```
        print("verbosity turned on")
```

```
    print("num: {}".format(args.num))
```

```
if __name__ == '__main__':
```

```
    main()
```

<https://docs.python.org/3/library/argparse.html>


```
#!/usr/bin/env python3
import click

@click.command()
@click.option('--count', '-c', default=1, help='Number of greetings.')
@click.option('--name', '-n', prompt='Your name',
              help='The person to greet.')
@click.argument("foo")
def hello(count, name, foo):
    """Simple program that greets NAME for a total of COUNT times."""
    for x in range(count):
        click.echo('Hello %s!' % name)

if __name__ == '__main__':
    hello()
```

<https://click.palletsprojects.com/en/7.x/>



```
@click.group()
@click.option('--debug/--no-debug', default=False)
def cli(debug):
    click.echo('Debug mode is %s' % ('on' if debug else 'off'))

@cli.command()  # @cli, not @click!
def sync():
    click.echo('Syncing')
```

Ejemplo con configuración yaml, versión y subcomandos:

[Ver código en gist](#)



click: tipos en opciones y argumentos

```
@click.command()
@click.argument('input', type=click.File('rb'))
@click.argument('output', type=click.File('wb'))
def inout(input, output):
    """Copy contents of INPUT to OUTPUT."""
    while True:
        chunk = input.read(1024)
        if not chunk:
            break
        output.write(chunk)
```

<https://click.palletsprojects.com/en/7.x/parameters/>

```
@click.command()
@click.argument('filename', type=click.Path(exists=True))
def touch(filename):
    """Print FILENAME if the file exists."""
    click.echo(click.format_filename(filename))
```



```
@click.option('-c', '--config-file',  
              default=expanduser("~/config/test.yml"),  
              help="Config file. Defaults to ~/.config/test.yml",  
              type=click.File('r'),  
              callback=get_config  
            )
```

Ejemplo con configuración yaml, versión y subcomandos:

[Ver código en gist](#)



- Debemos orientar esfuerzo a la facilidad de distribución.
 - Ciclo de vida: release, build, deploy
 - Git / Mercurial
- Setuptools, pip, virtualenv, virtualenvwrapper, pyenv, pipenv
- Adaptarnos al S.O. destino, o bien virtualenv.
- Siempre Separamos configuración de código.
- setuptools: es la librería que usa pip, pipenv, easy_install para instalar software en el python path.



```
#setup.py
from setuptools import setup, find_packages

setup(
    name='yourpackage',
    version='0.1',
    packages=find_packages(),
    include_package_data=True,
    install_requires=[
        'Click',
    ],
    entry_points='''
        [console_scripts]
        yourscript=yourpackage.scripts.yourscript:cli
    ''',
)
```

<https://gist.githubusercontent.com/bercab/a39426517d6fcddaae2c311b9bb891e1/raw/09b2ab0b0d51de615cbde03a6dddd0a6df251c1a/sharesync-setup.py>



- Opciones:
 - virtualenv: básico
 - virtualenvwrapper
 - pyenv virtualenv
 - pipenv
 - poetry
- Qué hace virtualenv:
 - PATH
 - python sys.path
- ¿cómo lo podemos usar en producción?
 - Directamente ejecutable python, añadiendo PATH
 - workon (virtualenvwrapper)
 - script activate (source)
 - pyenv exec
 - pipenv run
 - uwsgi soporta directamente



<https://amoffat.github.io/sh/index.html>

```
from sh import ifconfig
print(ifconfig("wlan0"))
```




http://talks.apsl.net/deploy_python/



[Gracias]