Python

- Grunder
- requests
- Environment Variables
- argparse

Grunder

Den här sidan skrivs som studiematerial för mig. Fler sidor kan byggas mer utförligt.

En bra princip i Python är att arbeta i virtuella miljöer. I en virtuell miljö installeras en specifik version av Python och tillhörande paket i en kombination som kan vara unik för just den här miljön.

Aktivera virtuell miljö (Linux) via:

```
python3 -m venv myenv
source myenv/bin/activate
```

Nu ska prompten visa `(myenv)$`.

Paket kan nu installeras i miljön via `pip`, Pythons pakethanterare. Ex. `pip install paketnamn`. Vill man alltid ha samma version av ett paketnamn slår man `paketnamn==1.1.1`, då installeas alltid version 1.1.1. Slår man `paketnamn>=1.0` installeras 1.0 eller högre.

Paket kan man söka efter på https://pypi.org/search.

För att stänga av miljön slå `deactivate`.

Bra arbetssätt är att använda en requirements.txt fil. I den filen ska alla paket man önskar använda finnas, ex:

```
jmespath==1.0.1
jq==1.7.0
jsonpath-ng==1.5.3
lxml==4.9.3
markdown-it-py==3.0.0
```

Sedan kan man installera alla paket i miljön med `pip install -r requirements.txt`.

Om man vill spara de paket man för tillfället har installerade i en venv kan man göra det genom `pip freeze > requirements.txt`.

Man kan köra ett pythonjobb genom `./jobb.py` eller `python jobb.py`. Den förstnämnda fungerar bara om man definierar vad för typ av skript det är högst upp i filen, ex. med `#!/usr/bin/env python`3. Det kallas för Shebang-line. Filen behöver även bara körbar, det gör man i Linux genom `chmod +x jobb.py`.

Datatyper

Ett objekt som är mutable kan förändras och ett objekt som är immutable kan ej förändras. En variabel kan fortfarande förändras genom att tilldela om den, men datatypen kan inte modifieras.

Namn	Typ	Mutable	Beskrivning
Integer	int	Nej	Hela nummer utan decimaler, ex. 6, 600 och 1589
Float			Ej hela nummer, ex. 1337.1337
Boolean	bool	Nej	Ett jämförbart värde, kan bara vara True eller False
String	str	Nej	Sekvens av text avgränsad av citatttecken, såsom "Cisco, "Piper" och "2000"
List	list	Ja	Strukterad sekvens av objekt, ex. [10, "DNA", 19.8]
Tuple	tup	Nej	Strukterad sekvens av immutable objects, ex. (10, "DNS", 19.8)
Dictionary	dict	Ja	Ostrukterad key:value pairs, ex. {"key1":"value1","name":"Pip"[}
Set	set	Ja	Ostrukterad samling av unika object, ex. {"a","b"}

Variabel

En variabel är ett värde i Python som är mappad till ett Pythonobjekt som är sparat i minne. Ex. `supervariabel = 1337` är då en variabel där supervariabel matchar Integer 1337.

```
[jehrlander@bash python]python3
Python 3.12.13 (main, Jul 10 2026, 00:00:00) [GCC 14.3.1 20251022 (Red Hat 14.3.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> supervariabel = 1337
>>> print(supervariabel)
1337
>>> type(supervariabel)
```

```
<class 'int'>
```

Om man istället använder supervariabel = "1337" så blir det en String och inte en Integer.

```
>>> supervariabel = "1337"
>>> type(supervariabel)
<class 'str'>
```

En integer innehåller inte decimale: har den decimaler så blir det en float.

```
>>> supervariabel = 1337
>>> type(supervariabel)
<class 'int'>

>>> supervariabel = 1337.1337
>>> type(supervariabel)
<class 'float'>
```

Listor

Python har inte arrays. I Python kan man använda sig av listor. En lista kan innehålla alla typer av dataobjekt, även andra listor.

```
>>> switchar = ['Sw1', 'Sw2', 'Sw3']
>>> switchar
['Sw1', 'Sw2', 'Sw3']

>>> type(switchar)
<class 'list'>



>>> routrar = ['R1', 'R2']

>>> natvarksprylar = [switchar, routrar]
>>> natvarksprylar
[['Sw1', 'Sw2', 'Sw3'], ['R1', 'R2']]
>>> type(natvarksprylar)
<class 'list'>
```

Listor är att föredra över tuples när ett värde kan förändras, då listor är mutable.

Tuples

Tuples är lika listor, men de är inte skrivbara (immutable). Råkar man göra såhär så blir det en tuple istället:

```
>>> natvarksprylar = switchar,routrar
>>> natvarksprylar (['Sw1', 'Sw2', 'Sw3'], ['R1', 'R2'])
>>> type(natvarksprylar) <class 'tuple'>
```

Men regeln är att för att skapa en tuple använder man parantes istället för hakparantes. Så korrekt sätt att gskapa tuple är:

```
>>> natvarksprylar = (switchar, routrar)
>>> type(natvarksprylar)
<class 'tuple'>
```

En tuple är bra och effektiv att använda när värdet aldrig förändras, då tuple är immutable.

Dictionary

En dictionary är ett till sätt att skapa en samling av värden. Dictionary används när man behöver data som är ihopkopplat med ett visst värde, key:value.

```
>>> R1 = { "os":"Cisco IOS-XE", "hardware":"IR1101", "location":"Mordor" }
>>> type(R1)
<class 'dict'>
>>> R1["hardware"]
'IR1101'
>>> R1["location"]
'Mordor'
```

Det går att lägga till en dictionary till sin redan existerande dictionary:

```
>>> R1["installed_date"] = {"year":"2026", "month":"07", "day":"21"}
>>> R1["installed_date"]
{'year': '2026', 'month': '07', 'day': '21'}


>>> R1
```

```
{'os': 'Cisco IOS-XE', 'hardware': 'IR1101', 'location': 'Mordor', 'installed_date': {'year': '2026', 'month': '07', 'day': '21'}}
```

En bra tumregel för att hålla koll på listor/tuples/dictionary är:

- `[]` → skapar en **list**
- `{}` → skapar ett **dict** (eller ett **set** om det bara innehåller värden)
- `()` används ofta för gruppering, men **kommatecknet** är det som faktiskt skapar en **tuple**
  .

Sets

Sets är mutable. Man kan göra sets till frozen sets vilket gör de immutable.

```
>>> nummer = {1, 2, 3, 4, 5}
>>> type(nummer)
<class 'set'>

>>> udda_nummer = {1, 3, 5, 7}

# Lägg ihop två sets
>>> nummer | udda_nummer
{1, 2, 3, 4, 5, 7}

# Se vilka värden som finns i båda sets
>>> nummer & udda_nummer
{1, 3, 5}
```

Input och output

input() är en funktion som tar det man skriver och gör det möjligt att spara det till en variabel. Med funktionen print kan man sedan visa det i terminalen. Exempel:

```
>>> inpt = input('Skriv ditt namn: ')
Skriv ditt namn: Jehrlander
>>> print(inpt)
Jehrlander
>>> type(inpt)
<class 'str'>
```

Det som skriv ut med print-funktionen har en implicit newline character (\n) på slutet.

print-funktionen använder en separera melan element. Det kan göra att en print inte ser jättesnygg ut, ex:

```
>>> numbs = {1, 2, 4, 5, 6, 7, 10 }
>>> print('Numbers in set', 1, ':', numbs )
Numbers in set 1 : {1, 2, 4, 5, 6, 7, 10}
```

Slänger man på en `sep=''` (två enkelcitattecken) så tas automatiska mellanrum bort och man kan lägga till egna på ett snyggt sätt:

```
>>> print('Numbers in set ', 1, ': ', numbs, sep='' )
Numbers in set 1: {1, 2, 4, 5, 6, 7, 10}
```

Från Python3.6 och senare kan man använda sig av fstrings, vilket gör formattering och användning av print-funktionen väldigt enkel:

```
>>> value1 = 'Jehrlander'
>>> value2 = 1337
>>> print(f'{value1} gillar {value2} väldigt mycket')
Jehrlander gillar 1337 väldigt mycket
```

Vill man printa en integer med en string så använder man sig av kommatecken. Ex:

```
print("Antal enheter i DNAC:", device_count_result)
```

Indenteringar

I Python används indenteringar för att dela upp kod. I exempelvis if statemens krävs detta för att koden ska fungera. Det går att göra lite hur man vill, bara man är konsekvent, exempelvis fungerar det att bara använda ett mellanslag. Rekommendationen/best practice är att använda fyra mellanslag.

Exempel på om man glömmer indentering:

```
>>> n = 20
>>> if n == 20:
```

```
... print('N är 20')
  File "<stdin>", line 2
    print('N är 20')
    ^^^^^
IndentationError: expected an indented block after 'if' statement on line 1
```

Doc-strings

Doc-strings används för att dokumentera vad ett Python-jobb, eller en funktion, gör. Placeras högst upp i filen eller i funktionen inom trippla dubbla citattecken, se exempel:

```
#!/usr/bin/python3
"""
Förklaring till jobbet
Koperingsrätt wiki.jehrlander.net 2026
"""


def superfunktion():
  """En superbra funktion.
  """
```

Vilkorssatser och loopar

Eller conditional and loops. Tre primära sätt finns:

- if - värden jämförs och beslut tas därefter
- for - en räknande loop som går igenom data ett specifikt antal gånger
- while - en loop som utförs så länge vissa förhållanden nås

if

Ett if-statement börjar med if och utför en jämförelse för att se om ett förhållande är sannt eller ej. Här är ett väldigt enkelt exempel:

```
>>> n = 20
>>> if n == 20:
...     print('N är 20.')
...
N är 20.
```

Det är alltså booleansk logik, True eller False.

Om siffran är annorlunda, alltså att n inte är 20, kan man använda sig av anna logik för att hantera det fallet, elif (else if).

```
>>> n = 3
>>> if n == 17:
...     print('N är 17.')
... elif n < 10:
...     print('N är under 10.')
... elif n > 10:
...     print('N är över 10.')
...
N är under 10.
```

Och om inget matchar if eller elif kan man avsluta med else.

```
n = 10
if n == 17:
    print('N är 17.')
elif n < 10:
    print('N är under 10.')
elif n > 10:
    print('N är över 10.')
else:
    print('Ingen match ovan, N måste vara 10.')
```

Ett praktiskt exempel, i Pythonjobb med Nornir jag har byggt appliceras viss konfiguration beroende på om switchen är med i en grupp. Då har jag använt if-statements såhär för att hoppa över de switchar som ej är med i gruppen:

```
    if "leaves" not in task.host.groups:
        log_step(f"=== {task.host} === Not a leaf - not applying leaf iBGP neighbor config")
```

```
    return Result(host=task.host, result="Not in 'leaves' group")
```

return avslutar då jobbet för de switchar som ej är med i gruppen leaves.

for

Med foor kan man skapa en loop som iterar genom koden ett specifikt antal gånger. Kan även kallas för en counting loop. Används ofta för parsning av data.

```
>>> data=(1, 2, 3, 4, 5)
>>> type(data)
<class 'tuple'>
>>> for variable in data:
...     print(variable)
...
1
2
3
4
5
```

variable är då en variabel som skapas av for loopen. Om man printar variable efter att loopen är kvar är det enbart det senaste värdet kvar:

```
>>> print(variable)
```

while

Med while kan man skapa en loop som loopar så länge ett värde stämmer med definitionen, som med if. Så man kan använda ex. while True och while False.

```
>>> count = 1
>>> while (count < 5):
...     print("Loop count is:", count)
...     count = count + 1
... else:
...     print("Loop is finished.")
...
Loop count is: 1
```

```
Loop count is: 2
Loop count is: 3
Loop count is: 4
Loop is finished.
```

Man kan använda sig av break för att ta sig ur loopen.

Funktioner

En function i Python är en samling kod som kan ta emot någon parameter, eller ingen alls, och returnera något typ av värde tillbaks till koden som anropade funktionen. Principen är Don't Repeat Yourself, DRY, ska något utföras flera gånger ska man anropa en funktion som gör det istället för att bygga samma sak flera gånger.

I Python finns det många färdiga funktioner via standardbiblioteket.

Man kan även bygga egna. En funktion definieras genom nyckelordet def, ett namn för funktionen, paranteser för eventuell data(argument) du vill skicka med till funktionen och ett kolon. En egen funktion får ej börja med en siffra, får ej vara ett reserverat Pyton-ord (ex. `print()`) och får innehålla vad som helst av följande: A-Z, a-z, 0-9, understreck (_) och Bindestreck-minus (-).

Information om en funktion kan man ta fram genom help(värde). I en egen funktion, om man i raden efter def placerar information i nom 6 citattecken indenterat ex. ``` ``` Information om funktion``` ``` så kan man använda help() även för egna funktioner.

```
def notify_users(task: Task, status: str = "start") -> Result:
    """
    Skickar meddelande till inloggade användare på switchen.
    """


>>> help(notify_users)
Help on function notify_users in module tasks.notify_users:


notify_users(task: nornir.core.task.Task, status: str = 'start') -> nornir.core.task.Result
    Skickar meddelande till inloggade användare på switchen.
```

För att få information tillbaka från en funktion ska funktionen avslutas med return.

```
>>> def sub(arg1, arg2):
...     result = arg1 - arg2
...     return result
...
>>> sub(37, 13)
24
```

Det går även att definiera när man åkallar funktionen vilket argument som är vilket. I exemplet ovan blev det först arg1 och den andra arg2, men definierar man såhär blir det tvärtom:

```
>>> sub(arg2=37, arg1=13)
-24
```

Classes

I Python använder man classes (klass) för att beskriva objekt. En class definieraras med nyckelordet class, ett namn och avslut med ett kolon. PEP-8 (PEP8)(stilguide för Pyton) rekommenderar att en class inleds med en stor bokstav för att särskilja från en variabel.

Klassen beskriver vilka egenskaper (attribut) och funktioner (methods/metoder) som objekten ska ha.

En klass är väldigt användbar när man har många värden. Tänk exempelvis en samling routrar, alla bör ha samma fält men olika i värden (key:value), då är det bra med en class.

Poängen med classes/objects är att göra koden enklare. När man använder classes behöver man inte repetera element som är relaterade till classen.

Inheritance

Om man redan har en class som utför något kan en child class ärva attribut och metoder från den första klassen. För att ärva i en klass skapas klassen som vanligt men parantes läggs till som hänvisar till den första klassen. Om den första klassen definierades med `class Router:` så definieras den andra med class `NotRouters(Router):`. Den första klassen måste komma före den andra i koden.

Moduler

En modul kan importeras i ett Pythonjobb med import. Det gäller både egenskapade och fördefinierade moduler.

Egenskapade moduler importeras såhär: `from tasks.configure_macsec import configure_macsec`. Tasks är då en undermapp till mappen där Pythonjobbet finns.

Fördefinierade moduler kan importeras såhär:

```
from nornir.core.filter import F
import time
```

Om man bara vill ha en del av en modul så använder man `from` såsom i ovan.

För att använda en fördefinierad modul så måste paketet installeras via pip, som beskrevs längre upp, gärna i en virtuell miljö. Det finns även andra pakethanterare, exempelvis uv.

När en modul är importerad kan man kontrollera alla metor i modulen med dir().

Vill man av någon anledning kan man byta namn på en modul lokalt, ex. `import calendar as cal`, då anropar man cal i jobbet.

Arbeta med filer i Python

Textfiler

Att arbeta med textfiler i python är lätt. Två funktioner används, open() och close(). Nedan exempel innehåller en testfil i samma mapp som koden.

```
>>> readdata = open("testfil.txt", "r")
>>> print(readdata.read())
Jag är en testfil!


# Tar man inte med .read() så blir det såhär
>>> print(readdata)
<_io.TextIOWrapper name='testfil.txt' mode='r' encoding='UTF-8'>
```

De olika flaggorna efter "testfil.txt", ex. "r", gör olika saker:

- r: Öppna för att läsa (standard)
- w: Öppna för att skriva
- x: Öppna för exklusivt skapande, misslyckas om filen redan finns
- a: Öppna för att skriva, lägger till längst ned i filen om den finns
- b: Öppna i binärt läga
  t: Öppna i textläge (standard)

- +: Öppna för att uppdatera (läsa och skriva)

När en fil har öppnats med open() så kan den vara låst av Python från resten av operativsystemet. Så se till att stänga den när man är klar, ex. med `readdata.close()`.

```
>>> readdata.close()


>>> print(readdata.read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Istället för att öppna en fil genom att tilldela en variabel kan man använda with (context manager). with innehåller bättre undantagshantering och stänger automatiskt filen när man är klar med läsning eller skrivning. Här i exemplet nedan ser vi att printningen ej fungerar när funktionen är avslutad:

```
>>> with open("testfil.txt", "r") as data:
...     print(data.read())
...
Jag är en testfil!

>>> print(data.read())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

Lägg till text i filen (a för append):

```
>>> with open("testfil.txt", "a") as data:
...     data.write('\nEn till rad till testfilen!')
...
28
>>> with open("testfil.txt", "r") as data:
...     print(data.read())
...
Jag är en testfil!

En till rad till testfilen!
```

CSV

För att läsa CSV-filer behövs först csv-modulen, skapa ett reader objekt at läsa CSV filen till, läs filen till en filhanterare, kör sedan CSV läsfunktionen och skicka resutlatet till reader-objektet.

```
import csv
samplefile = open('routerlist.csv')
samplereader = csv.reader(samplefile)
sampledata = list (samplereader)
sampledata
```

Beroende på hur CSV-filen är uppdelad kan man behöver lägga till delimiter. Är den semikolonseparerade fungerar nedan kod:

```
>>> with open("export_2026-07-23.csv") as routerdata:
...     routerlist = csv.reader(routerdata, delimiter=";")
...     for row in routerlist:
...         routernamn = row[1]
...         print(routernamn)
```

Här är ett exempel på ett skript som hittar en fil som matchar ett visst mönster i samma mapp, tar informationen från en viss rad, lägger till fqdn och printar:

```
from pathlib import Path
import csv


script_dir = Path(__file__).parent


for fil in script_dir.iterdir():
    if fil.is_file() and "router" in fil.name:
        print(f"Hittade: {fil.name}")
        with open(fil, "r") as routerdata:
            routerlist = csv.reader(routerdata, delimiter=";")
            for row in routerlist:
                routernamn=row[10]
                print(routernamn, ".jehrlander.net", sep='')
        break
```

JSON

JavaScript Object Notation (JSON) är en datastruktur som kommer från programmeringsspråket Java. Väldigt använt och ganska trevligt i mitt tycke, lätt att läsa.

Det finns en inbyggd JSON modul till python. Med den kan man konvertera JSON till listor eller dictionaries. Fyra funktioner finns för att utföra detta:

- load(): Importera JSON och kovnertera till Python dictionary.
- loads(): Importera JSON data från en string för parsning och manipulering.
- dump(): Skriv JSON data från Pythonobjekt tll en fil.
- dumps(): Ta JSON dictionary data och konvertera till en serialiserad sträng för parsning och manipulering i Python.

s:et på slutet av loads och dumps står för string, dump string.

Här har vi ett exempel på en enkel kod som hämtar devices från LibreNMS och sedan printar JSON-outputen på ett snyggt sätt:

```
#!/usr/bin/python3



import requests
import json
import signal
import sys

signal.signal(signal.SIGPIPE, signal.SIG_DFL) # IOError: Broken pipe
signal.signal(signal.SIGINT, signal.SIG_DFL) #Keyboardinterrupt: Ctrl-C



token = "randomtoken1337"
headers = {'Authorization': "Bearer {}".format(token)}

def main():
    api_get_devices = requests.get("https://librenms.jehrlander.net/api/v0/devices", headers=headers)
    text_api_get_devices = json.loads(api_get_devices.text)
    print(json.dumps(text_api_get_devices, indent=1))

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

indent=1 gör JSON-printingen mer läsbar. Om man inte har med `.text` i printing eller i json.loads så kommer HTTP koden att printas istället.

XML

Extensible Markup Language (XML) är ett annat vanligt dataformat. Ser ut lite som HTML då det desingades för att arbeta med HTML, så start taggar (<>) och sluttaggar </> används, ex. <name>GigabitEthernet0/0/1</name>.

En modul som heter xmltodict finns som kan konvertera XML till dictionary i Python. pretty=True gör läsbarheten mycket bättre.

```
import xmltodict

with open ("xml_sample.xml" as data:
    xml_example = data.read()

xml_dict = xmltodict.parse

print(xmltodict.unparse(xml_dict, pretty=True))
```

YAML

YAML Ain't Markup Language är ett format för att bygga konfigurationsfiler och spara data. YAML har en enkel läsbar syntax och struktur. En YAML-fil börjar alltid med ---.

Ex:

```
---
device_hostname: c9300
snmp_location: Leetland

trunk_interfaces:
  - name: Port-channel1337
    description: c9500 .1q lacp Te1/1/1, Te2/1/1
```

För att jobba med YAML behövs pyyaml-modulen. För att konvertera mellan YAML och Python används yaml.load, för att konvertera från YAML till Python, och yaml.dump för att konvertera Python till YAML.

```
import yaml

with open ("yaml_samplte.yaml") as data:
    yaml_sample = data.read()

yaml_dict = yaml.load(yamle_sample, Loader=yaml.FullLoader)
! Nu är yaml_dict en Python dictionary innehållande datat från YAML-filen
```

Felhantering

Något kommer att gå fel i koden. När det går fel behövs hantering för det, logiken man bygger kallas try-except-else-finally.
I en loop så har man först en try, om ett felmeddelande uppstår och matchar en except-logik så används den, om try misslyckas och inte matchar en except så sker else. Sedan kan man även lägga till finally efter loopen, finally körs oavsett om try/except lyckades eller ej.

Snygga till kod enligt PEP8

För att snygga till kod enligt PEP8-standarden kan man använda ex. `flake8`. Se exempeloutput:

```
flake8 run.py
run.py:3:1: F401 'nornir_scrapli.tasks.send_configs' imported but unused
run.py:3:1: F401 'nornir_scrapli.tasks.send_commands' imported but unused
run.py:3:1: F401 'nornir_scrapli.tasks.send_command' imported but unused
```

`flake8` gör inte förändringarna åt än. Vill man att något gör det automatiskt kan man använda `black` eller `white`. white förändrar inte längden på en rad i skriptet (PEP-8 standard är max 79 tecken) medans black gör det.

```
white run.py
Warning: Python 3.12 cannot parse code formatted for Python 3.15. To fix this: run Black with Python 3.15, set --target-version to py312, or use --fast to skip the safety check. Black's safety check verifies equivalence by parsing the AST, which fails when the running Python is older than the target version.
reformatted run.py

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

White kräver att black finns för att man ska kunna köra den:

```
Traceback (most recent call last):
  File "/home/jehrlander/.local/bin/white", line 3, in <module>
    from white import main
  File "/home/jehrlander/.local/lib/python3.12/site-packages/white.py", line 3, in <module>
    import black
```

Licens

Om man postar sin kod på nätet ska man ha med en licensfil. Är ingen licensfil med så är koden ej klassad som open source, trots att den kan vara tillgänglig på Internet. Licens kan man få hjälp att välja på https://choosealicense.com/.

Paket (packages)

Ett Python-paket är en mapp. Paket kan användas för att organisera sin egna kod. Kräver en `__init__.py`-fil i mappen.
Filen ska innehålla from `.fil import funktion`. Punkten innebär att den kollar i samma mapp.

exit()

Om man avslutar en funktion med exit() så är det bra att ge det ett värde. Gör man inte det blir värdet 0 vilket indikerar att jobbet har gått bra. Men om man i en try-snurra vill avsluta ett jobb när något har gått dåligt så blir det inte helt rätt. Använd exit(1) när ett jobb avslutas om något har gått fel.

requests

requests-modulen används för att kommunicera med resurser över HTTP/S. Ex:

```
#!/usr/bin/python3
import requests
import json

token = "t00k1n"
headers = {
    "Authorization": "Bearer " + token
}

def getorg():
    getorgjson = requests.get("https://api.meraki.com/api/v1/organizations", headers=headers, timeout=10)
    getorg = json.loads(getorgjson.text)
    getorg_result = (json.dumps(getorg, indent=1))
    return getorg_result

def main():
    result = getorg()
    print(result)

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

timeout är bra att använda så att requests.get inte fastnar.

Använd verify=True för att se till att SSL är ordentlig hela vägen.

Det kan vara bra att verifiera HTTP-koder i en request.

Ex:

```
if response.status_code != 200:
    print(f"Login failed. Status code: {response.status_code}")
    exit(1)
```

Environment Variables

Det går att hänvisa till operativsystemets egna ENvironment Variables i ett skript.

Först måste variabeln finnas skapad på systemet. Det skapas, på Linux, med kommandot export VARIABEL=värde.
Sedan kan man kontrollera variabeln med echo $VARIABEL.

Exempelvis:

```
[jehrlander@ubuntu ~]$ export WIKI=lander
[jehrlander@ubuntu ~]$ echo $WIKI
lander
```

Man kan se alla variablar genom att slå `env`.

Nu när variabeln finns på OSet behöver man importera os i Pythonskriptet.
Där kan man sedan tilldela värdet till en variabel genom ex:

```
if wikiuser is None:
    wikiuser = os.getenv("WIKI")
```

argparse

argparse är en inbyggd funktion som gör att man kan skicka med variablar från CLIt när man kör ett jobb.

Ex:

```
import argparse

def main():
    # Argumenthantering
    parser = argparse.ArgumentParser(
        description="Kör Nornir-jobb med filtrering"
    )

    parser.add_argument(
        "--limit",
        help="Host eller grupp att köra mot",
        required=False
    )

    parser.add_argument(
        "--debug",
        help="Visa detaljerad output vid fel",
        action="store_true"
    )

    parser.add_argument(
        "--user",
        help="Användarnamn att logga in med"
    )

    args = parser.parse_args()

    debug_print = args.debug

    # Fråga efter credentials
```

```
    if args.user:
        username = args.user
    else:
        username = input("Ange användarnamn: ")
```

Det som står vid help= kan man sedan få fram i CLIt:

```
uv run run.py --help
usage: run.py [-h] [--limit LIMIT] [--debug] [--user USER]

Kör Nornir-jobb med filtrering

options:
  -h, --help      show this help message and exit
  --limit LIMIT   Host eller grupp att köra mot
  --debug         Visa detaljerad output vid fel
  --user USER     Användarnamn att logga in med
```