PYTHON

1. Linki
   https://www.geeksforgeeks.org/python-programming-language/?ref=shm
   https://www.geeksforgeeks.org/read-a-file-line-by-line-in-python/?ref=leftbar-rightbar – różne użyteczne z powyższej strony
   Object printing - https://www.geeksforgeeks.org/object-oriented-programming-in-python-set-2-data-hiding-and-object-printing/
   Organizacja modułów - http://introtopython.org/classes.html#Modules-and-classes
   https://towardsdatascience.com/whats-init-for-me-d70a312da583
   Environments: https://www.geeksforgeeks.org/python-virtual-environment/

   Tests: https://realpython.com/python-testing/

2. Ciekawe moduły
   https://github.com/google/textfsm
   SysArgv
   Argparse
3. Instalacja
   Wersji pythona na linuxie może być wiele. Zazwyczaj jest co najmniej dwie – 2.7 i najnowsza z rodziny 3.
   Binarki znajdują się w /usr/bin/ i są to:
   /usr/bin/python2 – link, który wskazuje najczęściej na /usr/bin/python2.7
   /usr/bin/python3 – link, który wskazuje na np. /usr/bin/python3.6
   /usr/bin/python – link, który wskazuje na /etc/alternatives/python

   Wiele skryptów/programów odwołuje się do python2 lub python3 i te linki zawsze powinny być. Ale równie wiele odwołuje się do linku python (bez numeru) i ten link również powinien być. Modyfikacja tego ostatniego jest dość ryzykowna, bo odwołują się do niego najczęściej skrypty systemowe.
   Możemy doinstalować kolejną wersję i wtedy znajdzie się ona w /usr/local/bin. Dla uporządkowania należy dodać linki w /usr/bin np.
   sudo ln -sfn /usr/local/bin/python3.9 /usr/bin/python3.9
   sudo ln -sfn /usr/local/bin/python3.9-config /usr/bin/python3.9-config
   sudo ln -sfn /usr/local/bin/pip3.9 /usr/bin/pip3.9

   Czyli nie tylko python3.9, ale też python3.9-config i pip3.9.
   Mając kilka wersji z serii 3 i chcąc z nich naprzemiennie korzystać należy te nazwy stosować tzn. /usr/bin/env python3.9 w skryptach lub pip3.9 podczas instalacji.

   Jeśli teraz chcemy, aby v 3.9 była obowiązująca dla rodziny 3, to należy wymienić link
   /usr/bin/python3 -> python3.6
   na
   /usr/bin/python3 ->/usr/local/bin/python3.9

4. Virtual Environment
   Celem jest to, aby instalacja modułów dotyczących danego projektu dotyczyła tylko tego projektu, a nie całego systemu operacyjnego. Tworzone są podkatalogi w katalogu projektowym, które zawierają interpreter python i prywatne moduły. Jeśli chodzi o interpreter, to jest to link do interpretera systemowego. Moduły systemowe też można zlinkować, ale domyślnie nie jest to robione.
   2.1 Tworzenie środowiska wirtualnego (z poziomu katalogu z projektem):
   python3 -m venv *env* *(env – przykładowa nazwa podkatalogu z interpreterem i modułami)*
   2.2 Uruchamianie środowiska wirtualnego

source env/bin/activate (w rzeczywistości zmienią się domyślne ścieżki wyszukiwania)
2.3 Gaszenie środowiska wirtualnego
Deactivate

5. Wyjątki

Najlepiej przechwytywać konkretne wyjątki. Unikać przechwytywania any oraz *Exception* (który też jest bardzo ogólny). Jeśli ustawiamy try, to wiemy, że tu może wystąpić jakiś konkretny błąd. I ten błąd obsługujemy w jakiś konkretny sposób. Jeśli *try* przechwyci wszystkie wyjątki, to może się zdarzyć, że obsłużymy go w niezamierzony sposób.
Ale możemy podejść do zagadnienia inaczej, jeśli zależy nam tylko na przerwaniu programu lub funkcji po pojawieniu się jakiegokolwiek wyjątku. Robimy to po to, żeby użytkownik nie dostał kłopotliwego w odbiorze oryginalnego komunikatu, lecz komunikat wymyślony przez nas. W takim przypadku możemy zastosować *try Exception* np.

```
except Exception as err:
  logger(type(err).__name__)
  return
```

Jeśli wyjątek ma przerwać program, to sprawa jest bardzo uproszczona. Ale często zdarza się, że definiujemy wyjątek charakterystyczny dla aplikacji i chcemy go obsłużyć w jakiś sposób bez przerywania aplikacji. W dodatku mamy wielopoziomową strukturę obiektów lub funkcji. Wtedy, jeśli wyjątek zdarzy się w obiekcie lub funkcji najniższej w strukturze musimy przetransportować go na samą górę. Czyli w każdym pośrednim poziomie stosować try-except.

6. Obiekty

6.1 Linki:
https://realpython.com/python-class-constructor/ - wnikliwe wytłumaczenie tworzenia obiektu. Z dokumentu wynika, że instancję tworzy *__new__()* i zwraca ją w postaci *self* a *__init__()* wypełnia tę instancję wartościami. Czyli jeśli piszemy MyObject(arg), to MyObject uruchamia __new__(), która tworzy instancję (self), następnie uruchamiany jest __init__, któremu przekazywany jest self i arg.

a. Dobrym zwyczajem jest tworzenie dla obiektu metody __repr__(), która zwraca tekstową reprezentację obiektu np.

```
def __repr__(self) -> str:
    return f"{type(self).__name__}(x={self.x}, y={self.y})"
```

b. Jeśli mamy np. r = (**rcoredata), to nie zawsze oznacza, że r będzie obiektem klasy ConnectionHandler. ConnectHandle może być funkcją, która zwraca obiekt. W takim przypadku należy przeanalizować moduł w poszukiwaniu jaki obiekt zwraca taka funkcja.

6.2 Obiekty – ciekawe zastosowania

```
class Car():

        # constructor
        def __init__(self, company, modelName, price, seatingCapacity):
                self.company = company
                self.modelName = modelName
                self.price = price
                self.seatingCapacity = seatingCapacity


# list of car objects
carsList = [Car('Honda', 'Jazz', 900000, 5), Car('Suzuki', 'Alto', 450000, 4), Car('BMW', 'X5', 9000000, 5)]

# cars with price less than 10 Lakhs
economicalCars = [car for car in carsList if car.price <= 1000000]

# print those cars
for car in economicalCars:
        print(car.company+'--'+car.modelName)
```

TESTY

```
python -m unittest discover
```

- uruchomienie testowania z użyciem narzędzia unittest
  discover – dyrektywa dla unittest, uruchamia wszystkie moduły testowe, których nazwa zaczyna się do test*
  - działa jeśli struktura katalogu jest następująca:

```
project/
│
├── src/
│   ├── __init__.py
│   └── func.py
│
└── tests/
    ├── __init__.py
    └── test.py
```

I w test.py musi być coś podobnego do *„from src.func import funkcja_do_testowania”*

Palo Alto API SDK

Links:

https://github.com/PaloAltoNetworks/pan-os-python/tree/develop/examples - Kilka przykładów

https://medium.com/palo-alto-networks-developer-blog/handling-pan-os-vsys-in-pandevice-212fe892d303 - konfiguracja z uwzględnieniem VSYS

Palo Alto Networks Developer Blog (medium.com) – kopalnia wiedzy o Palo, ale sporo o SDK dla pythona

pan-os-python – rest api, następca pandevice, zainstalowany na sk2

Jeśli używamy SDK, to nie interesujemy się już szczegółami wywołań REST. Nie musimy wiedzieć nic o xpath czy elemant-value. Wszystko to jest ukryte w SDK. Należy skupić się na dokumentacji do SDK. Wyjątkiem są komendy operacyjne np. show, dla których w niektórych przypadkach musimy znaleźć reprezentację xml-ową. Taki przypadek jest opisany niżej.

Filozofia działania panos: mamy lokalny model konfiguracji złożony z obiektów PanObjects i mamy metody służące do interakcji z urządzeniem. Najpierw budujemy drzewo konfiguracyjne lokalnie (tak jak z klocków). Mogą to być nowe, albo istniejące na urządzeniu (wtedy używamy refresh()). Konfigurujemy lokalne obiekty konfiguracyjne i na końcu wysyłamy na urządzenie za pomocą create() – jeśli istnieje obiekt na urządzeniu to łączy konfiguracje lub apply() – jeśli istnieje obiekt na urządzeniu to nadpisuje. Lokalne drzewo konfiguracyjne musimy zbudować od samej góry, czyli najpierw Firewall, potem jakiś child object np. Rulebase, potem następny child object np. SecurityRule. Ale wysyłkę na urządzenie możemy wykonać tylko dla obiektu najniższego w hierarchii.

Common configuration methods of PanObject:
Build the configuration tree: `add()`, `remove()`, `find()`, and `findall()`
Push changed configuration to the live device: `apply()`, `create()`, and `delete()`
Pull configuration from the live device: `refresh()`, `refreshall()`

Discovering an operational command's syntax

If you are trying to execute an operational command and the auto-formatting that pan-os-python performs doesn't seem to be working, SSH to your PAN-OS appliance and enable debugging to see how PAN-OS is formatting the command. Let's take the CLI command `show arp all` as an example. Let's SSH to PAN-OS and see what we get back:

```
> debug cli on
```

```
> show arp all

<request cmd="op" cookie="2801768344648204" uid="1000"><operations><show><arp><entry
name='all'/></arp></show></operations></request>
```

When taking debug CLI output and turning it into an operational command string, you'll want to take all the XML inside of the `<operations>` tag. Thus, our command to XML conversion looks like this:

- `show arp all` -> `<show><arp><entry name='all'/></arp></show>`

Operational commands that have an `<entry>` tag with an attribute (here, `name='all'`) is not a format that pan-os-python can convert to on your behalf. Thus, you will have to send in the XML yourself and instruct pan-os-python that the `cmd` argument does not need to be turned into XML:

```
ans = fw.op("<show><arp><entry name='all'/></arp></show>", cmd_xml=False)
```

Jak to działa?

```
Dodawaie nowej reguły
from panos.firewall import Firewall
from panos.policies import Rulebase, SecurityRule

fw = Firewall(.......)
# tworzy połączenie do urządzenia i tylko to, fw jest obiektem klasy Firewall, domyślnie nie ma żadnych
atrybutów oprócz połączenia.

base = Rulebase()
# Obiekt Rulebase zawiera atrybuty i funkcje charakterystyczne dla reguł

fw.add(base)
# obiekt fw zawiera teraz zarówno atrybuty połączenia jak i reguł

rule = SecurityRule("Int to Ext", .......)
base.add(rule)
# do instancji Rulebase dodajemy obiekt z nową regułą


rule.create()
# wrzucamy na urządzenie konfigurację nowej reguły
rule.opstate.audit_comment.update("initial config")
# opstate jest to namespace, który funkcjonuje obok głównego drzewa konfiguracji; zawiera parametry obiektu,
#które są inne dla każdego typu obiektu; ten obiekt ma swoją własną metodę do wysyłki na urządzenie – update()
```

```
Modyfikacja reguły
from panos.panorama import Panorama, DeviceGroup
from panos.policies import PreRulebase, SecurityRule

pano = Panorama(.........)
# tutaj jest przykład z Panorama, ale dla Firewall jest tak samo

dg = DeviceGroup("myDg")
pano.add(dg)

base = PreRulebase()
dg.add(base)

rule = SecurityRule("Int to Ext")
# "Int to Ext" jest istniejącą regułą

base.add(rule)
rule.refresh()
# ściągamy z urządzenia parametry reguły
```

```
# Update the rule description
rule.description = "My new description"

rule.apply()
# apply() nadpisuje obiekt na urządzeniu; dlatego istotny jest refresh() wcześniej
rule.opstate.audit_comment.update("ID 12345 updating rule description")
```

```
Z interaktywnego pythona:
>>> fw.children
[] – pusta lista => świeżo utworzony fw bez żadnych podłączonych child's.
>>> AddressObject.refreshall(fw, add=True)
# ściąga z urządzenia wszystkie AddressObject i od razu dodaje do lokalnego drzewa konfiguracyjnego
>>> fw.children
[<panos.objects.AddressObject object at 0x108080e90>,
 <panos.objects.AddressObject object at 0x108080f50>,
 <panos.objects.AddressObject object at 0x108080ed0>]
```

Jeśli mamy zamiar konfigurować tylko obiekty, które da się skonfigurować jako shared, to możemy użyć:

```
fw = firewall.Firewall("10.0.0.1", "admin", "mypassword", vsys="shared")
```

Configuration Methods

Modify the configuration tree or the live device with these methods.

- **C:** Changes the pan-os-python configuration tree
- **L:** Connects to a live device (firewall or Panorama) via the API
- **M:** Modifies the live device by making a change to the device's configuration
- **B:** Bulk operation modifies more than one object in a single API call

Method	C	L	M	B	Description
`add()`	✅				Add an object as a child of this object
`extend()`	✅				Add a list of objects as children
`insert()`	✅				Insert an object as a child at an index
`pop()`	✅				Remove a child object at an index
`remove()`	✅				Remove a child object from this object
`remove_by_name()`	✅				Remove a child object by its name
`removeall()`	✅				Remove all children of this object
`refresh()`		✅			Set params of object from live device
`refreshall()`		✅			Pull all children from the live device
`refresh_variable()`		✅			Set a single param from the live device
`create()`		✅	✅		Push object to the live device (nd)
`apply()`		✅	✅		Push object to the live device (d)
`update()`		✅	✅		Push single object param to live device
`delete()`	✅	✅	✅		Delete from live device and config tree
`rename()`	✅	✅	✅		Rename on live device and config tree
`move()`	✅	✅	✅		Reorder on live device and config tree
`create_similar()`		✅	✅	✅	Push objects of this type to live device (nd)
`apply_similar()`		✅	✅	✅	Push objects of this type to live device (d)

Method	C	L	M	B	Description
`delete_similar()`		☑	☑	☑	Delete objects of this type from live device

- (d): Destructive - Method *overwrites* an object on the live device with the same name
- (nd): Non-destructive - Method *combines* object with one on live device with the same name

Navigation Methods

These methods help you locate objects and information in an existing configuration tree. These are commonly used when you have used `refreshall` to pull a lot of nested objects and you're either looking for a specific object or aggregate stats on the objects.

Method	Description
`find()`	Return object by name and type
`findall()`	Return all objects of a type
`find_index()`	Return the index of a child object
`find_or_create()`	Return object by name and type, creates object if not in config tree
`findall_or_create()`	Return all objects of type, creates an object if none exist
`nearest_pandevice()`	Return the nearest parent Firewall or Panorama object in tree
`panorama()`	Return the nearest parent Panorama object
`devicegroup()`	Return the nearest parent DeviceGroup object
`vsys`	Return the vsys that contains this object

Informational Methods

These methods provide information about an object in the configuration tree.

Method	Description
`about()`	Return all the params set on this object and their values
`equal()`	Test if two objects are equal and return a boolean
`xpath()`	Return the XPath of this object
`element()`	Return the XML of this object as an ElementTree

Method	Description
`element_str()`	Return the XML of this object as a string
`hierarchy_info()`	Return hierarchical information about this object

Device Methods

These methods can be called on a PanDevice object (a Firewall or Panorama), but not on any other PanObject.

Method	Description
`refresh_system_info()`	Return and retain important information about the device
`commit()`	Trigger a commit on a Firewall or Panorama
`commit_all()`	Trigger a configuration push from Panorama to the Firewalls
`syncjob()`	Wait for a job on the device to finish
`refresh_devices()`	Pull all the devices attached to Panorama as Firewall objects
`op()`	Execute an operational command
`watch_op()`	Same as ‘op’, then watch for a specific result

There are many other convenience methods available. They’re all documented in the `PanDevice` class.