

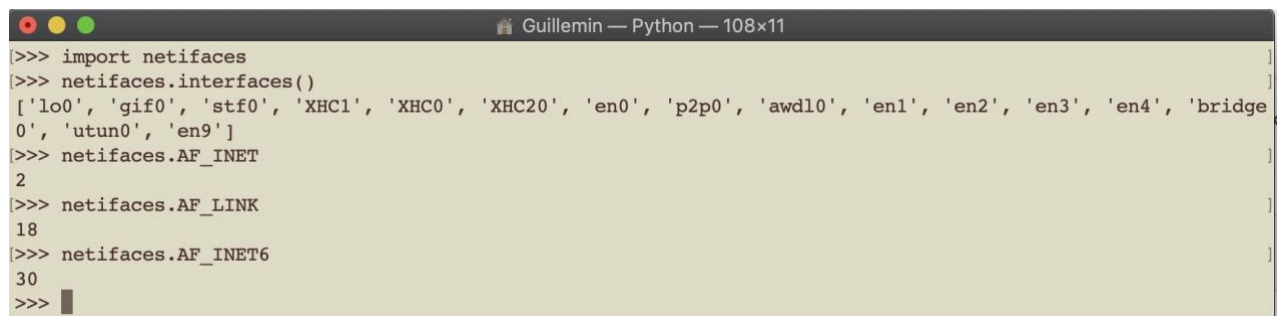
# Fiche Ressource n°6 SAÉ24

## Compléments sur Python et Scapy pour la SAÉ24

### 1. Connaitre ses interfaces avec le module netifaces

Il n'y a pas de moyen facile d'obtenir les adresses des interfaces réseau de la machine à partir de Python, il est pratiquement impossible de le faire de manière portable. Le module netifaces écrit en C est le moyen le plus simple et portable pour obtenir les paramètres des interfaces réseaux en Python.

On peut voir la liste des identifiants des interfaces de votre machine avec la commande `netifaces.interfaces()` :

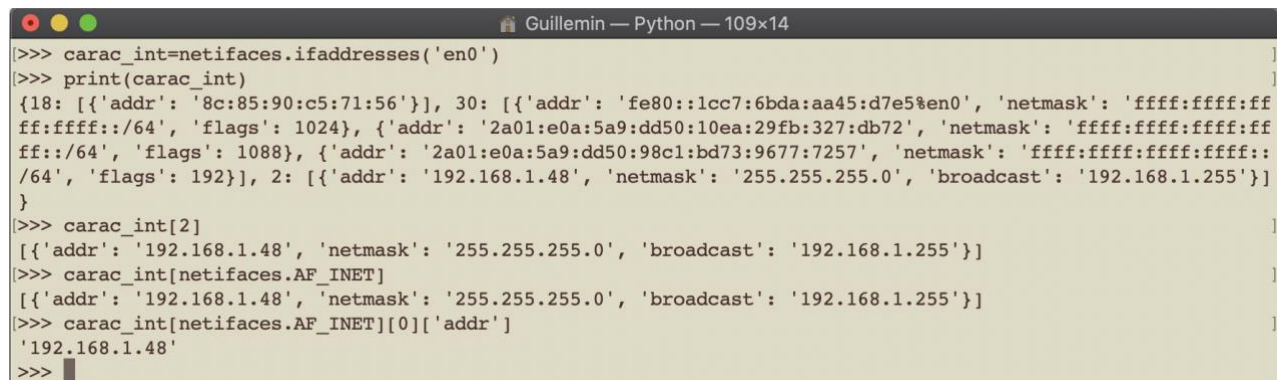


```
Guillemin — Python — 108x11
[>>> import netifaces
[>>> netifaces.interfaces()
['lo0', 'gif0', 'stf0', 'XHC1', 'XHC0', 'XHC20', 'en0', 'p2p0', 'awdl0', 'en1', 'en2', 'en3', 'en4', 'bridge0', 'utun0', 'en9']
[>>> netifaces.AF_INET
2
[>>> netifaces.AF_LINK
18
[>>> netifaces.AF_INET6
30
[>>> ]
```

Netifaces distingue 3 types d'Adresses Family AF dont les valeurs numériques dépendent du système :

- `AF_LINK` : famille d'adresses de l'interface de la couche liaison, par ex. Ethernet (2 dans l'exemple)
- `AF_INET` : Famille d'adresses IPv4 (18 dans l'exemple)
- `AF_INET6` : famille d'adresses IPv6 (30 dans l'exemple)

Vous pouvez utiliser la commande `netifaces.ifaddresses` pour obtenir les adresses d'une interface particulière :

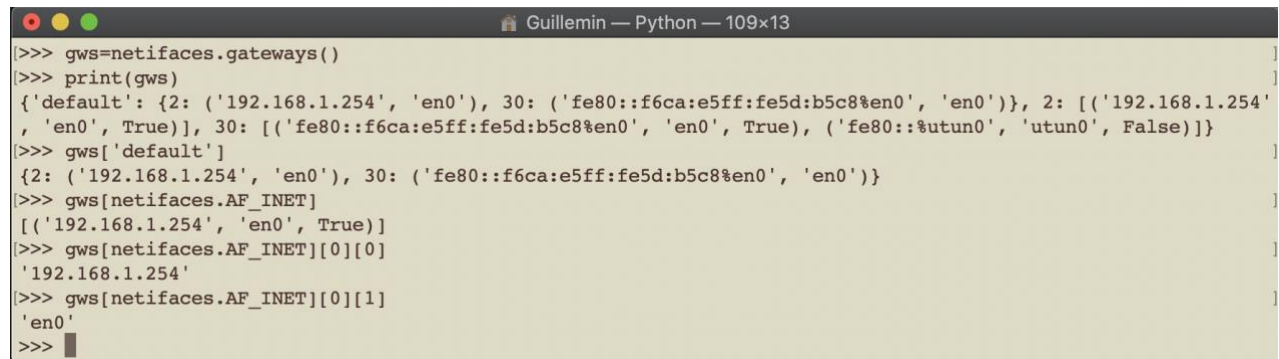


```
Guillemin — Python — 109x14
[>>> carac_int=netifaces.ifaddresses('en0')
[>>> print(carac_int)
{18: [{'addr': '8c:85:90:c5:71:56'}], 30: [{'addr': 'fe80::1cc7:6bda:aa45:d7e5%en0', 'netmask': 'ffff:ffff:ffff:ffff::/64', 'flags': 1024}, {'addr': '2a01:e0a:5a9:dd50:10ea:29fb:327:db72', 'netmask': 'ffff:ffff:ffff:ffff::/64', 'flags': 1088}, {'addr': '2a01:e0a:5a9:dd50:98c1:bd73:9677:7257', 'netmask': 'ffff:ffff:ffff:ffff::/64', 'flags': 192}], 2: [{'addr': '192.168.1.48', 'netmask': '255.255.255.0', 'broadcast': '192.168.1.255'}]}
[>>> carac_int[2]
[{'addr': '192.168.1.48', 'netmask': '255.255.255.0', 'broadcast': '192.168.1.255'}]
[>>> carac_int[netifaces.AF_INET]
[{'addr': '192.168.1.48', 'netmask': '255.255.255.0', 'broadcast': '192.168.1.255'}]
[>>> carac_int[netifaces.AF_INET][0]['addr']
'192.168.1.48'
[>>> ]
```

Le résultat de cette commande est un dictionnaire avec une entrée pour chaque format d'adresse, chaque entrée étant une liste.

La commande `netifaces.ifaddresses[netifaces.AF_INET][0]['addr']` retourne les adresses IPV4 du dictionnaire (valeur numérique 30), premier élément (index 0) de la liste qui est un dictionnaire : `{'addr' : '192.168.1.48', 'netmask' : '255.255.255.0', 'broadcast' : '192.168.1.255'}`, valeur de la clé 'addr'.

Pour obtenir l'adresse des passerelles de la machine, on utilise la commande `netifaces gateways()` :



```
>>> gws=netifaces.gateways()
>>> print(gws)
{'default': {2: ('192.168.1.254', 'en0'), 30: ('fe80::f6ca:e5ff:fe5d:b5c8%en0', 'en0')}, 2: [('192.168.1.254', 'en0', True)], 30: [('fe80::f6ca:e5ff:fe5d:b5c8%en0', 'en0', True), ('fe80::%utun0', 'utun0', False)]}
>>> gws['default']
{2: ('192.168.1.254', 'en0'), 30: ('fe80::f6ca:e5ff:fe5d:b5c8%en0', 'en0')}
>>> gws[netifaces.AF_INET]
[('192.168.1.254', 'en0', True)]
>>> gws[netifaces.AF_INET][0][0]
'192.168.1.254'
>>> gws[netifaces.AF_INET][0][1]
'en0'
>>>
```

## 2. [Émettre et recevoir des paquets avec Scapy](#)

La fonction `send()` envoie les paquets au niveau de la couche 3 c'est-à-dire que Scapy va gérer le routage (en choisissant l'interface de sortie connectée à la passerelle par défaut) et l'encapsulation dans la couche 2.

```
>>> help(send)
```

Help on function send in module scapy.sendrecv:

```
send(x, inter=0, loop=0, count=None, verbose=None, realtime=None,
return_packets=False, socket=None, *args, **kargs)
```

- `x` : les paquets
- `inter` : temps (en s) entre deux paquets (0 par défaut)
- `loop` : permet d'envoyer le paquet indéfiniment avec `loop=1` (valeur par défaut 0)
- `count` : nombre de paquets à envoyer (par défaut `Aucun=1`)
- `verbose` : mode verbeux (par défaut `None=conf.verbose`)
- `realtime` : vérifie qu'un paquet a bien été envoyé avant d'envoyer le suivant
- `return_packets` : retourne les paquets envoyés
- `socket` : le socket à utiliser (la valeur par défaut est `conf.L3socket(kargs)`)
- `iface` : l'interface sur laquelle envoyer les paquets
- `moniteur` : (pas sur linux) envoyer en mode moniteur
- `retours` : aucun

```
>>> pkt6=IP(dst="www.iut-velizy.uvsq.fr")/ICMP()/ "IUT de velizy"
```

```
>>> send(pkt6)
```

```
.
Sent 1 packets.
```

1	...	192.168.1.48	193.51.27.3	ICMP	Echo (ping) request	id=0x0000, seq=0/0, ttl=64 (reply in 2)
2	...	193.51.27.3	192.168.1.48	ICMP	Echo (ping) reply	id=0x0000, seq=0/0, ttl=54 (request in 1)

▶ Frame 1: 55 bytes on wire (440 bits), 55 bytes captured (440 bits) on interface 0

▶ Ethernet II, Src: Apple\_c5:71:56 (8c:85:90:c5:71:56), Dst: FreeboxS\_5d:b5:c8 (f4:ca:e5:5d:b5:c8)

▶ Internet Protocol Version 4, Src: 192.168.1.48, Dst: 193.51.27.3

▼ Internet Control Message Protocol

Type: 8 (Echo (ping) request)

Code: 0

Checksum: 0x8de7 [correct]

[Checksum Status: Good]

Identifier (BE): 0 (0x0000)

Identifier (LE): 0 (0x0000)

Sequence number (BE): 0 (0x0000)

Sequence number (LE): 0 (0x0000)

[\[Response frame: 2\]](#)

▼ Data (13 bytes)

Data: 4955542064652056656c697a79

[Length: 13]

0000	f4	ca	e5	5d	b5	c8	8c	85	90	c5	71	56	08	00	45	00	...	]	...	...	q	V	...	E	...
0010	00	29	00	01	00	00	40	01	dc	c4	c0	a8	01	30	c1	33	..)	...	@	...	...	0	...	3	...
0020	1b	03	08	00	8d	e7	00	00	00	00	49	55	54	20	64	65	.....	...	I	U	T	...	d	e	...
0030	20	56	65	6c	69	7a	79										...								

Velizy

```
>>> send(pkt6, inter=0.1, count=3)
...
Sent 3 packets.
```

La fonction `sendp()` fonctionne au niveau de la couche 2. C'est à vous de spécifier le protocole de couche 2, et si vous ne le faites pas, aucun paquet ne sera envoyé même si vous voyez « sent 1 packet » dans la console.

```
>>> pkt7=Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst='192.168.1.254')
>>> sendp(pkt7)
.
Sent 1 packets.
```

La fonction `sr()` sert à envoyer des paquets et à recevoir des réponses. Elle renvoie un tuple avec deux listes. Le premier élément est une liste avec le couple (**paquet envoyé, réponse**), et le second élément est la liste des **paquets sans réponse**. Ces deux éléments sont des listes, mais ils sont « enveloppés » (wrapper en anglais) dans un objet pour mieux les présenter, et pour leur fournir quelques méthodes qui effectuent les actions les plus fréquemment nécessaires.

Elle prend les mêmes paramètres que la fonction `send` avec en plus l'option `retry` qui permet de renvoyer les paquets sans réponse par exemple 3 fois si `retry = 3`.

```
Guillemin — Scapy v2.4.4 — scapy — 109x15

>>> sr(IP(dst=['www.iut-velizy.uvsq.fr', 'www.uvsq.fr'])/ICMP())
Begin emission:
Finished sending 2 packets.
....**
Received 6 packets, got 2 answers, remaining 0 packets
(<Results: TCP:0 UDP:0 ICMP:2 Other:0>,
 <Unanswered: TCP:0 UDP:0 ICMP:0 Other:0>)
>>> ans, unans = _
>>> ans.summary()
IP / ICMP 192.168.1.48 > 193.51.27.3 echo-request 0 ==> IP / ICMP 193.51.27.3 > 192.168.1.48 echo-reply 0
IP / ICMP 192.168.1.48 > 193.51.31.90 echo-request 0 ==> IP / ICMP 193.51.31.90 > 192.168.1.48 echo-reply 0
>>> unans.summary()
>>>
```

Note : en Python « \_ » signifie le dernier résultat.

On peut faire par exemple une connexion TCP avec le serveur Web (193.51.27.3) de l'IUT de Vélizy et tester si les ports FTP(20), HTTP(80) et HTTPS(443) sont ouverts :

```
>>> ans, noans=sr(IP(dst="193.51.27.3")/TCP(sport=RandShort(),
dport=[20,80,443]), timeout=1)
Begin emission:
..Finished sending 3 packets.
.*.*...
Received 9 packets, got 2 answers, remaining 1 packets
```

Comme nous pouvons le voir dans la Wireshark ci-dessous, le serveur web accepte la connexion avec un TCP avec les flags [SYN, ACK] pour http et https mais pas pour ftp. Scapy envoie également un TCP [RST] pour réinitialiser les connexions http et https.

1	→	192.168.1.48	8.8.8.8	DNS	Standard query 0x4cd4 A www.iut-velizy.uvsq.fr
2	←	8.8.8.8	192.168.1.48	DNS	Standard query response 0x4cd4 A www.iut-velizy.uvsq.fr CNAME bourde.cri.uvsq.fr A 193.51.27.3
5	→	192.168.1.48	193.51.27.3	TCP	48134 → ftp-data(20) [SYN] Seq=0 Win=8192 Len=0
6	→	192.168.1.48	193.51.27.3	TCP	21986 → http(80) [SYN] Seq=0 Win=8192 Len=0
7	→	192.168.1.48	193.51.27.3	TCP	14658 → https(443) [SYN] Seq=0 Win=8192 Len=0
8	←	193.51.27.3	192.168.1.48	TCP	http(80) → 21986 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
9	→	192.168.1.48	193.51.27.3	TCP	21986 → http(80) [RST] Seq=1 Win=0 Len=0
10	←	193.51.27.3	192.168.1.48	TCP	https(443) → 14658 [SYN, ACK] Seq=0 Ack=1 Win=14600 Len=0 MSS=1460
11	→	192.168.1.48	193.51.27.3	TCP	14658 → https(443) [RST] Seq=1 Win=0 Len=0

Dans l'objet ans on peut voir les paquets émis et leur réponse :

```
>>> ans.summary()
IP / TCP 192.168.1.48:9958 > 193.51.27.3:http S ==> IP / TCP
193.51.27.3:http > 192.168.1.48:9958 SA
IP / TCP 192.168.1.48:rdb_dbs_disp > 193.51.27.3:https S ==> IP /
TCP 193.51.27.3:https > 192.168.1.48:rdb_dbs_disp SA
```

Dans l'objet noans packet on peut voir les paquets sans réponse, ici pour ftp :

```
>>> noans.summary()
IP / TCP 192.168.1.48:9058 > 193.51.27.3:ftp_data S
```

La fonction sr1() est une variante qui ne renvoie que le paquet répondu (ou l'ensemble de paquets). La fonction renvoie le paquet de réponse ou None s'il n'y a pas eu de réponse.



```

Guillemin — Scapy v2.4.4 — scapy — 109x19
>>>
>>> reply=sr1(IP(dst='192.168.1.254')/ICMP(), timeout=1)
Begin emission:
Finished sending 1 packets.
.*
Received 2 packets, got 1 answers, remaining 0 packets
>>> reply
<IP version=4 ihl=5 tos=0x0 len=28 id=25231 flags= frag=0 ttl=64 proto=icmp chksum=0x93d3 src=192.168.1.254
dst=192.168.1.48 |<ICMP type=echo-reply code=0 chksum=0xffff id=0x0 seq=0x0 |>>
>>> reply=sr1(IP(dst='192.168.1.23')/ICMP(), timeout=1)
Begin emission:
WARNING: Mac address to reach destination not found. Using broadcast.
Finished sending 1 packets.
.....
Received 17 packets, got 0 answers, remaining 1 packets
>>> print(reply)
None
>>>

```

`srp()` et `srp1()` font la même chose que `sr()` et `sr1()` mais il faut configurer la couche 2.

### 3. Décodage des paquets RTP

RTP est décodé par Scapy mais Scapy ne peut pas savoir quels sont les paquets RTP car il n'y a pas de « well-known port » UDP pour RTP car le numéro de port UDP est négocié dynamiquement dans le message SIP. Connaissant ce numéro de port, on peut cependant forcer l'interprétation de la charge UDP d'un paquet nommé « packet » comme étant du RTP avec le code suivant :

```
packet[UDP].payload = RTP(packet[Raw].load)
```

```

Wireshark — Scapy v2.4.4 — scapy — 95x18
>>> packets[13][UDP]
<UDP sport=19038 dport=remoteware_cl len=180 chksum=0xf204 |<Raw load='\x80\x08\xae\xa0\xff\x
dba\x98^\x99\x0e\x04\xdf\x05\x0d\x09T\x06\xdb\x04\x01\x02T\x03V@\x01SD\x04F]\x07ZP\x06VP\x00]\x
d4\xdeD\x03\xdd_\x02XV\x0ddDU\x05ST\x04^\x0d_UTVT\x0d\x07S\xcdP\x01\x04S\x0P\x07\x06[\x02[VPQT\x
01\x02^\xc6\x06\x01\x09W\x0c4\xdd\x01PQ\xdb\G\x02R\x04\x02@\xc4\x08_\x03\x00\\\x03\x05M\xddEP\x
02w\xde\x03_\x0d\x04\x07\x02TZ\x03F^Yw\x02EA\xdeG\x01\x05E\xde\x0c1G\xdd\xdfF\x05QZXQ^E\x02\x04U
\x08U\x08\x02\x04\xdd\x07\x02\x06^\x04\x05PT\xdc\' |>>
>>> packets[13][UDP].payload=RTP(packets[13][Raw].load)
>>> packets[13][UDP]
<UDP sport=19038 dport=remoteware_cl len=180 chksum=0xf204 |<RTP version=2 padding=0 extensio
n=0 numsync=0 marker=0 payload_type=PCMA sequence=44704 timestamp=4292567448 sourcesync=1587088
900 |<Raw load='\xdf\x05\x0d\x09T\x06\xdb\x04\x01\x02T\x03V@\x01SD\x04F]\x07ZP\x06VP\x00]\x
d4\xdeD\x03\xdd_\x02XV\x0ddDU\x05ST\x04^\x0d_UTVT\x0d\x07S\xcdP\x01\x04S\x0P\x07\x06[\x02[VPQT\x
01\x02^\xc6\x06\x01\x09W\x0c4\xdd\x01PQ\xdb\G\x02R\x04\x02@\xc4\x08_\x03\x00\\\x03\x05M\xddEP\x
02w\xde\x03_\x0d\x04\x07\x02TZ\x03F^Yw\x02EA\xdeG\x01\x05E\xde\x0c1G\xdd\xdfF\x05QZXQ^E\x02\x04U
\x08U\x08\x02\x04\xdd\x07\x02\x06^\x04\x05PT\xdc\' |>>>
>>>

```

### 4. Quelques méthodes utiles sur les chaînes de caractères pour la SAÉ24

Dans Scapy le contenu d'un paquet correspond à des octets « bytes » notés `b'...'` comme illustré ci-dessous :

```
3]: 1 paquets[10][Raw].load
: b'SIP/2.0 200 OK\r\nVia: SIP/2.0/UDP 192.168.35.42;branch=z9hG4bK1elad74c6645c9c76;received=192.168.35.42;rport=5060\r\nFrom: "Amiral-Motti" <sip:Amiral-Motti@com-galactiques.dark-empire.universe:5060>;tag=a7e015c9f8\r\nTo: <sip:103@com-galactiques.dark-empire.universe:5060;user=phone>;tag=as02c39d55\r\nCall-ID: 18c818eb8f6ecble\r\nCSeq: 1286641179 INVITE\r\nServer: softswitch Asterisk\r\nAllow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH\r\nSupported: replaces, timer\r\nContact: <sip:103@192.168.51.234:5060>\r\nContent-Type: application/sdp\r\nContent-Length: 280\r\n\r\nv=0\r\no=root 1114046580 1114046580 IN IP4 192.168.51.234\r\ns=Asterisk PBX 1.8.13.1-dfsg1-3+deb7u8\r\nnc=IN IP4 192.168.51.234\r\nnt=0 0\r\nnm=audio 19022 RTP/AVP 8 0 101\r\na=rtpmap:8 PCMA/8000\r\na=rtpmap:0 PCMU/8000\r\na=rtpmap:101 telephone-event/8000\r\na=fmtp:101 0-16\r\na=ptime:20\r\na=sendrecv\r\n'
```

Même si on a des bytes Le notebook décode en UTF8 ces bytes pour l’affichage.

Pour un protocole en mode texte comme SIP, chaque byte correspondant à un caractère selon un loi d'encodage. Pour parser le protocole SIP il faut décoder avec la méthode **decode** (loi d’encodage UTF8 par défaut dans la méthode decode) les bytes pour obtenir la chaîne chaîne de caractères :

```
: 1 paquets[10][Raw].load.decode()
'SIP/2.0 200 OK\r\nVia: SIP/2.0/UDP 192.168.35.42;branch=z9hG4bK1elad74c6645c9c76;received=192.168.35.42;rport=5060\r\nFrom: "Amiral-Motti" <sip:Amiral-Motti@com-galactiques.dark-empire.universe:5060>;tag=a7e015c9f8\r\nTo: <sip:103@com-galactiques.dark-empire.universe:5060;user=phone>;tag=as02c39d55\r\nCall-ID: 18c818eb8f6ecble\r\nCSeq: 1286641179 INVITE\r\nServer: softswitch Asterisk\r\nAllow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH\r\nSupported: replaces, timer\r\nContact: <sip:103@192.168.51.234:5060>\r\nContent-Type: application/sdp\r\nContent-Length: 280\r\n\r\nv=0\r\no=root 1114046580 1114046580 IN IP4 192.168.51.234\r\ns=Asterisk PBX 1.8.13.1-dfsg1-3+deb7u8\r\nnc=IN IP4 192.168.51.234\r\nnt=0 0\r\nnm=audio 19022 RTP/AVP 8 0 101\r\na=rtpmap:8 PCMA/8000\r\na=rtpmap:0 PCMU/8000\r\na=rtpmap:101 telephone-event/8000\r\na=fmtp:101 0-16\r\na=ptime:20\r\na=sendrecv\r\n'
```

Ensuite on peut utiliser la méthode **split** pour séparer la chaîne de caractère dans une liste en fonction d’un caractère séparateur. Ici on voit par exemple que le caractère séparateur des différents champs est `\r\n`.

```

: 1 paquets[10][Raw].load.decode().split("\r\n")

['SIP/2.0 200 OK',
'Via: SIP/2.0/UDP 192.168.35.42;branch=z9hG4bK1elad74c6645c9c76;received=192.168.35.42;rport=5060',
'From: "Amiral-Motti" <sip:Amiral-Motti@com-galactiques.dark-empire.universe:5060>;tag=a7e015c9f8',
'To: <sip:103@com-galactiques.dark-empire.universe:5060;user=phone>;tag=as02c39d55',
'Call-ID: 18c818eb8f6ecble',
'CSseq: 1286641179 INVITE',
'Server: softswitch Asterisk',
'Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY, INFO, PUBLISH',
'Supported: replaces, timer',
'Contact: <sip:103@192.168.51.234:5060>',
'Content-Type: application/sdp',
'Content-Length: 280',
'',
'v=0',
'o=root 1114046580 1114046580 IN IP4 192.168.51.234',
's=Asterisk PBX 1.8.13.1~dfsg1-3+deb7u8',
'c=IN IP4 192.168.51.234',
't=0 0',
'm=audio 19022 RTP/AVP 8 0 101',
'a=rtpmap:8 PCMA/8000',
'a=rtpmap:0 PCMU/8000',
'a=rtpmap:101 telephone-event/8000',
'a=fmtp:101 0-16',
'aptime:20',
'a=sendrecv',
'']

```

La méthode **startswith** permet de voir si une chaîne de caractères commence par des caractères spécifiques comme illustré ci-dessous :

```

: 1 chaine = "SIP invite From R&T"
2 if chaine.startswith("SIP") :
3     print("yes")
4 else :
5     print("no")

```

yes

---

La méthode **find** permet de donner la position d'une suite de caractères dans une chaîne de caractères comme illustré ci-dessous :

```

: 1 chaine = "SIP invite From R&T"
2 start = chaine.find("From")
3 print(start)

```

Si la suite de caractère n'est pas trouvée, la méthode renvoie -1 :

```
: 1 chaine = "SIP invite From R&T"
  2 start = chaine.find("GEII")
  3 print(start)
```

-1

## 5. Exécuter des programmes externes dans le code Python

Le module subprocess est le module Python 3 recommandé pour exécuter des programmes externes et éventuellement lire leurs sorties dans votre code Python.

Subprocess comprend plusieurs classes et fonctions, mais on présente ici seulement la fonction popen(). Cette fonction ouvre le programme externe dans nouveau processus ce qui permet au programme Python de continuer son exécution. La commande externe doit être présentée sous forme d'une liste comme illustré ci-dessous pour un ping :

```
1 ping = subprocess.Popen(['ping', '-c 2', '192.168.1.254'])
2
3 #Visualisation du PID du processus lancé par subprocess
4 print(f"Le PID du processus lancé pour effectuer les ping est {ping.pid}\n")
```

Le PID du processus lancé pour effectuer les ping est 9172

Si besoin, on peut récupérer la sortie standard de la commande stdout en créant un pipe :

```
1 ping = subprocess.Popen(['ping', '-c 2', '192.168.1.254'], stdout = subprocess.PIPE, encoding = 'utf8')
2
3 #Impression des données dans la sortie standard
4 for line in ping.stdout :
5     print(line)
```

PING 192.168.1.254 (192.168.1.254): 56 data bytes

64 bytes from 192.168.1.254: icmp\_seq=0 ttl=64 time=2.359 ms

64 bytes from 192.168.1.254: icmp\_seq=1 ttl=64 time=25.562 ms

--- 192.168.1.254 ping statistics ---

2 packets transmitted, 2 packets received, 0.0% packet loss

round-trip min/avg/max/stddev = 2.359/13.960/25.562/11.602 ms

Si la commande pour exécuter le programme externe est longue, on peut utiliser le module shlex pour la mettre sous forme d'une liste comme illustré ci-dessous :



```
: 1 import shlex
  2
  3 a = 'ping -c 2 192.168.1.254'
  4 command1 = shlex.split(a)
  5 print(command1)
  6
  7 monobjet = subprocess.Popen(command1, stdout = subprocess.PIPE, encoding = 'utf8')
  8 for line in monobjet.stdout :
  9     print(line)
```

```
['ping', '-c', '2', '192.168.1.254']
```

```
PING 192.168.1.254 (192.168.1.254): 56 data bytes
```

```
64 bytes from 192.168.1.254: icmp_seq=0 ttl=64 time=1.811 ms
```

```
64 bytes from 192.168.1.254: icmp_seq=1 ttl=64 time=2.001 ms
```

```
--- 192.168.1.254 ping statistics ---
```

```
2 packets transmitted, 2 packets received, 0.0% packet loss
```

```
round-trip min/avg/max/stddev = 1.811/1.906/2.001/0.095 ms
```