Generer des ip information

Mockaroo traitement daddresse ip

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Alter the value of this field using Mockaroo formula syntax. Use this to refer to the value of this field.

**Examples**

Add 1 to the value of this field:

this + 1

Change the value of this field to upper case:

upper(this)

Transform the value of this field based on custom logic:

if this == 'January' then 'cold'

elsif this == 'July' then 'hot'

else 'mild' end

**Mockaroo Formula Reference**

Formulas allow you to use [Ruby](https://www.ruby-lang.org/) code to generate data based on custom logic. For example:

times\_reached\_base / at\_bats + slugging

**Operators**

+ - \* / %

**Logic**

< > <= >= == != and or

**Conditional Statements**

if my\_num % 2 == 0 then 'even' else 'odd' end

if score > 10 then "high" elsif score > 5 then "medium" else "low" end

**Functions**

base64(str) => encodes a string as base64

bcrypt('some password') => Returns a bcrypt hash of the provided string.

code("CURRENT\_TIMESTAMP") => returns a value that is will not be wrapped in quotes in the downloaded file. Use this to inject code into your data.

concat(first\_name, " ", last\_name) => concatenates all arguments into a single string.

date('7/4/2015') => July 4, 2015. You can optionally pass a format string as a second argument. The default format is "%m/%d/%Y". See [Ruby Date.strptime](https://apidock.com/ruby/Date/strptime/class) for more info on formats.

date\_diff('days', my\_date, date('2015-01-01')) => The numbers of days between my\_date and Jan 1, 2015. 'days', 'hours', 'minutes', and 'seconds' are also supported.

day(my\_date\_field) => The day of my\_date\_field as an integer. Optionally specify true as a second argument to return the day as string in 2 digit format.

digest(str, 'MD5|HMAC|RMD160|SHA1|SHA256|SHA384|SHA512', 'hex|base64') => digest of str with specific algorithm and encoding

epoch(datetime) => Converts a datetime field to an epoch.

field("My Field Name") => gets the value of a field. Use this to access fields that start with an uppercase letter or contain non-alphanumeric characters.

from\_dataset("dataset", "column", join\_criteria) => Fetches a value from a dataset.  
For example, from\_dataset("People", "name", id: person\_id) returns the value of the "name" column from the "People" dataset where the id column matches the person\_id column in the current schema.

generate(data\_type, \*\*options) => generates a value using one of Mockaroo's built-in datatypes. See [Types](http://localhost:3000/docs#Types) for a list of available data types and their parameters. For example, generate("First Name") generates a random first name. generate('Number', min: 0, max: 100, decimals: 0) generates a random integer between 0 and 100.

hex(str) => encodes a string as hex

lower("XYZ") => xyz

mongo\_object\_id(my\_id) => Output the value of my\_id in MongoDB ObjectId JSON format

month(my\_date\_field) => The month of my\_date\_field as an integer. Optionally specify true as a second argument to return the month as a string in 2 digit format.

my\_date + years(2) => the date 2 years after my\_date. months, days, hours, minutes, and seconds are also available.

naughty(my\_field, 10) => returns a naughty string 10% of the time and the value of my\_field 90% of the time. Use with hidden fields (name starting with "\_\_") to turn nice fields into naughty fields!

now() => The current date and time.

normal\_dist(mean, std\_dev, decimals) => Generates a random number using a normal disribution. Mean and std\_dev are required. Decimals is optional and defaults to 2.

pad(field, length, fill\_string, "left|right|center") => pads a string to a fixed length with filler.

format(number\_value, decimals) => formats a number as a string with a fixed number of decimal places. For example: format(1.2, 3) => "1.200"

random(min, max) => generates a random number between min and max.

round(x / y, 2) => rounds to 2 decimal places

time(my\_datetime\_field => Returns only the time part of a datetime field.

upper("xyz") => XYZ

uuid\_v5("dns", "www.mockaroo.com") => generates a v5 UUID  
The first parameters should be "dns", "url", "oid", or "x500". The second parameter is any string

year(my\_date\_field) => The year of my\_date\_field as an integer

**Accessing Request Parameters when Used in a Mock API**

request\_params['name\_of\_parameter'] => returns the value of a URL or query string parameter specified in the request

request\_entity['name\_of\_parameter'] => returns the value of a key in the JSON request entity body

**Handling Blanks**

my\_num + 1 if my\_num => returns my\_num + 1, or blank if my\_num is blank

a + b if a and b => returns a + b, or blank if either is blank

(my\_num || 0) + 1 => returns my\_num + 1, or 1 if my\_num is blank

if my\_field.nil? then 'blank' else 'not blank' end => "blank" when my\_field is blank, otherwise "not blank"

**Accessing Nested JSON Objects**

When a formula field is inside of a JSON array it has access to all fields in that array as well as all fields on parent objects. Both fields in the array and parent fields are accessed **without a qualifier**. For example, if a formula is in an array named "myArray" that has nested field named "myArray.myField", the formula can access that field as "myField" NOT "myArray.myField". When formulas need to access fields in nested objects (not in an array), however, the fully qualified field name must be used.

**Generating Data using Regular Expressions**

You can use Ruby's Regexp.gen method to generate data based on regular expressions. For example:/(A|B|C)-\d3/.gen

**Using built-in Ruby classes and methods**

For security purposes, not all of the Ruby language is available in Mockaroo Formulas. All methods of the following modules are whitelisted whitelisted:

* Array
* BasicObject
* BigDecimal
* BSON::ObjectId
* Date
* DateAndTime::Calculations
* DateTime
* Enumerable
* Fixnum
* Float
* Hash
* Integer
* Math
* NilClass
* Range
* Regexp
* String
* Time

Additionally you can also declare and call lambdas. Formulas use Ruby 2.7.3.

Faker’s documentation

A screenshot of a computer

Description automatically generated

<https://faker.readthedocs.io/en/master/>

Pour automatiser l'attribution d'adresses IP dans un réseau Packet Tracer, vous pouvez utiliser Python pour générer une adresse IP disponible sur une plage d'adresses donnée. Voici un script qui pourrait vous aider à accomplir cela :

### Script Python pour Trouver une Adresse IP Disponible

Supposons que vous avez une plage d'adresses IP et que vous souhaitez attribuer une adresse IP disponible à un PC. Le script ci-dessous vérifie les adresses IP utilisées dans le réseau et trouve une adresse IP disponible.

1. \*\*Configuration de base\*\* : Définir la plage d'adresses IP.

2. \*\*Vérification des adresses utilisées\*\* : Simuler ou obtenir la liste des adresses IP utilisées.

3. \*\*Trouver une adresse IP disponible\*\* : Trouver une adresse IP qui n'est pas dans la liste des adresses utilisées.

### Exemple de Script

```python

import ipaddress

def get\_used\_ips():

# Simuler une liste d'IP utilisées dans le réseau (peut être remplacée par une méthode réelle)

return [

'192.168.1.1',

'192.168.1.2',

'192.168.1.3',

# Ajouter plus d'IP utilisées

]

def find\_available\_ip(network):

used\_ips = get\_used\_ips()

for ip in network.hosts():

if str(ip) not in used\_ips:

return str(ip)

return None

# Définir la plage d'adresses IP du réseau

network = ipaddress.ip\_network('192.168.1.0/24')

available\_ip = find\_available\_ip(network)

if available\_ip:

print(f"Adresse IP disponible trouvée : {available\_ip}")

else:

print("Aucune adresse IP disponible trouvée")

```

### Explications

1. \*\*Fonction get\_used\_ips()\*\* :

- Simule une liste d'adresses IP déjà utilisées. Vous pouvez remplacer cette fonction pour obtenir dynamiquement les adresses IP utilisées via Packet Tracer ou une autre source.

2. \*\*Fonction find\_available\_ip(network)\*\* :

- Parcourt les adresses IP du réseau spécifié et retourne la première adresse IP disponible (non utilisée).

3. \*\*Utilisation de la bibliothèque ipaddress\*\* :

- `ipaddress.ip\_network` est utilisé pour définir la plage d'adresses IP du réseau. Vous pouvez spécifier n'importe quelle plage (par exemple, '192.168.1.0/24').

4. \*\*Parcours des adresses IP\*\* :

- `network.hosts()` retourne toutes les adresses IP hôtes dans la plage définie (excluant les adresses réseau et de diffusion).

### Comment l'utiliser avec Cisco Packet Tracer

1. \*\*Intégration avec Packet Tracer\*\* : Si vous utilisez Cisco Packet Tracer, vous pouvez automatiser la configuration en utilisant les fonctionnalités de script de Packet Tracer (par exemple, via l'API Cisco ou des scripts Tcl).

2. \*\*Exécution du Script\*\* : Exécutez ce script pour trouver une adresse IP disponible, puis configurez le PC dans Packet Tracer avec cette adresse IP.

### Conclusion

Ce script peut être étendu et modifié en fonction de vos besoins spécifiques. Par exemple, si vous avez une méthode pour obtenir dynamiquement les adresses IP utilisées directement depuis Packet Tracer, vous pouvez intégrer cette méthode à la place de la fonction `get\_used\_ips()`.

Si vous avez besoin d'aide pour une partie spécifique ou pour l'intégration avec Packet Tracer, n'hésitez pas à le préciser !

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

Pour automatiser la configuration des bureaux avec les informations données dans votre tableau, nous allons lire les données directement depuis le tableau et générer des fichiers de configuration pour chaque bureau. Voici un exemple de script Python qui pourrait être utilisé pour accomplir cette tâche.

Nous allons supposer que nous avons converti le tableau en une structure de données Python (par exemple, une liste de dictionnaires), ce qui nous permet de traiter facilement les informations.

**Script Python pour Configurer les Bureaux à Partir des Données du Tableau**

python

import csv

# Exemples de données du tableau, à remplacer par la lecture depuis un fichier si nécessaire

data = [

{"Bureau": 1, "VLAN": "Wi-Fi", "Sous-réseau": "192.168.0.0/26", "Plage d'adresses IP": "192.168.0.0 - 192.168.0.63", "Adresses IP utilisables": "192.168.0.1 - 192.168.0.62"},

{"Bureau": 1, "VLAN": "Ethernet", "Sous-réseau": "192.168.0.64/26", "Plage d'adresses IP": "192.168.0.64 - 192.168.0.127", "Adresses IP utilisables": "192.168.0.65 - 192.168.0.126"},

{"Bureau": 2, "VLAN": "Wi-Fi", "Sous-réseau": "192.168.0.128/26", "Plage d'adresses IP": "192.168.0.128 - 192.168.0.191", "Adresses IP utilisables": "192.168.0.129 - 192.168.0.190"},

# Ajouter le reste des données...

]

def parse\_ip\_range(ip\_range):

start\_ip, end\_ip = ip\_range.split(' - ')

return start\_ip.strip(), end\_ip.strip()

def generate\_ip\_list(start\_ip, end\_ip):

start = ipaddress.ip\_address(start\_ip)

end = ipaddress.ip\_address(end\_ip)

return [str(ip) for ip in ipaddress.summarize\_address\_range(start, end)]

def create\_config\_files(data):

for entry in data:

bureau = entry["Bureau"]

vlan = entry["VLAN"]

subnetwork = entry["Sous-réseau"]

ip\_range = entry["Adresses IP utilisables"]

start\_ip, end\_ip = parse\_ip\_range(ip\_range)

ip\_list = generate\_ip\_list(start\_ip, end\_ip)

config\_content = f"Bureau {bureau} - {vlan} configuration:\nSous-réseau: {subnetwork}\n"

for i, ip in enumerate(ip\_list, start=1):

config\_content += f"PC{i} IP address: {ip}\nSubnet mask: 255.255.255.0\n"

filename = f"Bureau{bureau}\_{vlan}\_config.txt"

with open(filename, "w") as config\_file:

config\_file.write(config\_content)

# Créer les fichiers de configuration à partir des données du tableau

create\_config\_files(data)v # Exemples de données du tableau, à remplacer par la lecture depuis un fichie(config\_content) # Créer les fichiers de configuration à partir des données du tableau create\_config\_files(data)

**Explications**

1. **Données d'entrée** :
   * Les données du tableau sont représentées sous forme de liste de dictionnaires. Chaque dictionnaire contient les informations pour un bureau spécifique.
2. **Fonction parse\_ip\_range(ip\_range)** :
   * Prend une chaîne de caractères représentant une plage d'adresses IP et retourne les adresses IP de début et de fin.
3. **Fonction generate\_ip\_list(start\_ip, end\_ip)** :
   * Génère une liste d'adresses IP à partir de l'adresse IP de début et de fin. Utilise la bibliothèque **ipaddress** pour cela.
4. **Fonction create\_config\_files(data)** :
   * Parcourt les données, extrait les informations nécessaires, génère la liste des adresses IP utilisables, et crée un fichier de configuration pour chaque bureau et VLAN.
5. **Écriture des fichiers de configuration** :
   * Pour chaque bureau et VLAN, crée un fichier de configuration contenant les adresses IP et le masque de sous-réseau.

**Utilisation**

1. **Exécution du Script** :
   * Exécutez le script pour générer les fichiers de configuration à partir des données du tableau.
2. **Configuration des PC dans Cisco Packet Tracer** :
   * Importez les fichiers de configuration générés dans Cisco Packet Tracer ou configurez manuellement chaque PC en utilisant les adresses IP fournies dans les fichiers.

Ce script est flexible et peut être ajusté en fonction de vos besoins spécifiques. Vous pouvez ajouter des fonctionnalités supplémentaires, comme la validation des adresses IP ou l'intégration avec d'autres outils, selon vos exigences.

Pour que le script propose une adresse IP à l'utilisateur et lui demande ensuite s'il souhaite que le script configure cette adresse IP directement sur son ordinateur, nous devons inclure une partie de configuration réseau. Cependant, la modification de l'adresse IP d'un ordinateur via un script Python nécessite des privilèges administratifs et dépend du système d'exploitation.

### Exemple de Script avec Configuration Réseau (Pour Windows)

Pour Windows, nous pouvons utiliser la commande `netsh` pour configurer l'adresse IP. Voici un script Python complet qui inclut cette fonctionnalité :

```python

import ipaddress

import os

import subprocess

# Exemple de données du tableau

data = [

{"Bureau": 1, "VLAN": "Wi-Fi", "Sous-réseau": "192.168.0.0/26", "Plage d'adresses IP": "192.168.0.0 - 192.168.0.63", "Adresses IP utilisables": "192.168.0.1 - 192.168.0.62"},

{"Bureau": 1, "VLAN": "Ethernet", "Sous-réseau": "192.168.0.64/26", "Plage d'adresses IP": "192.168.0.64 - 192.168.0.127", "Adresses IP utilisables": "192.168.0.65 - 192.168.0.126"},

{"Bureau": 2, "VLAN": "Wi-Fi", "Sous-réseau": "192.168.0.128/26", "Plage d'adresses IP": "192.168.0.128 - 192.168.0.191", "Adresses IP utilisables": "192.168.0.129 - 192.168.0.190"},

{"Bureau": 2, "VLAN": "Ethernet", "Sous-réseau": "192.168.0.192/26", "Plage d'adresses IP": "192.168.0.192 - 192.168.0.255", "Adresses IP utilisables": "192.168.0.193 - 192.168.0.254"},

{"Bureau": 3, "VLAN": "Wi-Fi", "Sous-réseau": "192.168.1.0/26", "Plage d'adresses IP": "192.168.1.0 - 192.168.1.63", "Adresses IP utilisables": "192.168.1.1 - 192.168.1.62"},

{"Bureau": 3, "VLAN": "Ethernet", "Sous-réseau": "192.168.1.64/26", "Plage d'adresses IP": "192.168.1.64 - 192.168.1.127", "Adresses IP utilisables": "192.168.1.65 - 192.168.1.126"},

# Ajouter le reste des données...

]

# Simuler une liste d'IP utilisées, à remplacer par une méthode réelle pour obtenir dynamiquement les IP utilisées

used\_ips = set([

'192.168.0.1', '192.168.0.2', '192.168.0.65', '192.168.0.66',

'192.168.0.129', '192.168.0.130', '192.168.0.193', '192.168.0.194',

'192.168.1.1', '192.168.1.2', '192.168.1.65', '192.168.1.66'

])

def parse\_ip\_range(ip\_range):

start\_ip, end\_ip = ip\_range.split(' - ')

return start\_ip.strip(), end\_ip.strip()

def generate\_ip\_list(start\_ip, end\_ip):

start = ipaddress.ip\_address(start\_ip)

end = ipaddress.ip\_address(end\_ip)

return [str(ip) for ip in ipaddress.summarize\_address\_range(start, end)]

def find\_available\_ip(bureau, vlan):

for entry in data:

if entry["Bureau"] == bureau and entry["VLAN"].lower() == vlan.lower():

start\_ip, end\_ip = parse\_ip\_range(entry["Adresses IP utilisables"])

ip\_list = generate\_ip\_list(start\_ip, end\_ip)

for ip in ip\_list:

if ip not in used\_ips:

used\_ips.add(ip)

return ip

return None

def configure\_ip\_address(ip, subnet\_mask="255.255.255.0"):

interface = "Ethernet" # Nom de l'interface réseau, à adapter si nécessaire

command = f'netsh interface ip set address name="{interface}" static {ip} {subnet\_mask}'

try:

subprocess.run(command, check=True, shell=True)

print(f"L'adresse IP {ip} a été configurée avec succès.")

except subprocess.CalledProcessError as e:

print(f"Erreur lors de la configuration de l'adresse IP : {e}")

# Demander à l'utilisateur de sélectionner un bureau et un VLAN

bureau = int(input("Entrez le numéro du bureau : "))

vlan = input("Entrez le type de VLAN (Wi-Fi ou Ethernet) : ")

# Trouver une adresse IP disponible

available\_ip = find\_available\_ip(bureau, vlan)

if available\_ip:

print(f"Adresse IP disponible pour Bureau {bureau} ({vlan}) : {available\_ip}")

configure = input(f"Voulez-vous configurer cette adresse IP ({available\_ip}) sur votre ordinateur ? (oui/non) : ").strip().lower()

if configure == 'oui':

configure\_ip\_address(available\_ip)

else:

print(f"Aucune adresse IP disponible pour Bureau {bureau} ({vlan})")

```

### Explications

1. \*\*Données d'entrée\*\* :

- Les données du tableau sont représentées sous forme de liste de dictionnaires.

2. \*\*Fonctions parse\_ip\_range() et generate\_ip\_list()\*\* :

- Ces fonctions restent les mêmes que précédemment pour traiter les plages d'adresses IP.

3. \*\*Fonction find\_available\_ip()\*\* :

- Cherche une adresse IP disponible pour un bureau et un VLAN spécifiques.

4. \*\*Fonction configure\_ip\_address(ip, subnet\_mask)\*\* :

- Utilise la commande `netsh` pour configurer l'adresse IP sur l'interface réseau spécifiée. L'interface réseau par défaut est "Ethernet", mais vous pouvez la modifier selon vos besoins.

5. \*\*Interaction utilisateur\*\* :

- Demande à l'utilisateur de sélectionner un bureau et un VLAN, puis propose une adresse IP disponible.

- Demande à l'utilisateur s'il souhaite configurer cette adresse IP sur son ordinateur.

6. \*\*Configuration de l'adresse IP\*\* :

- Si l'utilisateur accepte, le script exécute la commande `netsh` pour configurer l'adresse IP.

### Utilisation

1. \*\*Exécution du Script\*\* :

- Exécutez le script avec des privilèges administratifs (nécessaire pour changer l'adresse IP).

- Saisissez le numéro du bureau et le type de VLAN lorsque vous y êtes invité.

- Le script proposera une adresse IP disponible et vous demandera si vous souhaitez la configurer.

2. \*\*Configuration des PC\*\* :

- Si vous acceptez de configurer l'adresse IP, le script exécutera la commande pour configurer l'adresse IP sur votre ordinateur.

Ce script est un point de départ et peut être ajusté pour répondre à des besoins spécifiques ou pour fonctionner sur d'autres systèmes d'exploitation (par exemple, macOS ou Linux) en adaptant la commande de configuration réseau appropriée.

Il semble que vous recherchiez un script qui attribue automatiquement des adresses IP à chaque ordinateur portable ou imprimante dans votre réseau, en divisant une adresse IP fournie par vous en 10 sous-réseaux pour chaque bureau et 2 VLAN par bureau. Voici un exemple de script Python qui pourrait vous aider à réaliser cela :

Python

import ipaddress

# Fonction pour diviser une adresse IP en sous-réseaux pour chaque bureau et VLAN

def diviser\_adresse\_ip(adresse\_ip, nombre\_bureaux):

try:

adresse = ipaddress.IPv4Network(adresse\_ip)

except ValueError:

print("Adresse IP invalide.")

return

sous\_reseaux\_par\_bureau = 10

vlan\_par\_bureau = 2

for i in range(nombre\_bureaux):

print(f"Bureau

mtn je veux que ce script pose les question a la place de deja savoir combien de bureaux et combien de vlan par bureau

ChatGPT

D'accord, pour rendre le script interactif et permettre à l'utilisateur de spécifier le nombre de bureaux et de VLAN par bureau, vous pouvez utiliser la fonction **input()** pour demander ces informations. Voici comment vous pourriez modifier le script :

python

lesimport ipaddress

# Fonction pour diviser une adresse IP en sous-réseaux pour chaque bureau et VLAN

def diviser\_adresse\_ip(adresse\_ip, nombre\_bureaux, vlan\_par\_bureau):

try:

adresse = ipaddress.IPv4Network(adresse\_ip)

except ValueError:

print("Adresse IP invalide.")

return

sous\_reseaux\_par\_bureau = 10

for i in range(nombre\_bureaux):

print(f"Bureau {i + 1}:")

sous\_reseaux = list(adresse.subnets(new\_prefix=adresse.prefixlen + 3))

for j in range(vlan\_par\_bureau):

print(f"VLAN {j + 1}: {sous\_reseaux[j \* sous\_reseaux\_par\_bureau]}")

# Demander à l'utilisateur l'adresse IP du fournisseur

adresse\_fournisseur = input("Veuillez entrer l'adresse IP du fournisseur: ")

# Demander à l'utilisateur le nombre de bureaux

nombre\_bureaux = int(input("Veuillez entrer le nombre de bureaux: "))

# Demander à l'utilisateur le nombre de VLAN par bureau

vlan\_par\_bureau = int(input("Veuillez entrer le nombre de VLAN par bureau: "))

# Appel de la fonction avec les paramètres fournis par l'utilisateur

diviser\_adresse\_ip(adresse\_fournisseur, nombre\_bureaux, vlan\_par\_bureau) paramètres fournis par l'utilisateur diviser\_adresse\_ip(adresse\_fournisseur, nombre\_bureaux, vlan\_par\_bureau)

Ce script demandera à l'utilisateur d'entrer l'adresse IP du fournisseur, le nombre de bureaux et le nombre de VLAN par bureau, puis il affichera les adresses IP résultantes pour chaque VLAN dans chaque bureau.

**C'est en effet plus logique de demander l'adresse du fournisseur dès le début du script. Voici comment vous pouvez déplacer cette question vers le début du code :**

import ipaddress

# Fonction pour diviser une adresse IP en sous-réseaux pour chaque bureau et VLAN

def diviser\_adresse\_ip(adresse\_ip, nombre\_bureaux, vlan\_par\_bureau, ip\_bloquees\_par\_bureau):

try:

adresse = ipaddress.IPv4Network(adresse\_ip)

except ValueError:

print("Adresse IP invalide.")

return

sous\_reseaux\_par\_bureau = 10

for i in range(nombre\_bureaux):

print(f"Bureau {i + 1}:")

sous\_reseaux = list(adresse.subnets(new\_prefix=adresse.prefixlen + 3))

for j in range(vlan\_par\_bureau):

debut = j \* sous\_reseaux\_par\_bureau + ip\_bloquees\_par\_bureau \* j

fin = (j + 1) \* sous\_reseaux\_par\_bureau + ip\_bloquees\_par\_bureau \* j

print(f"VLAN {j + 1}: {sous\_reseaux[j].subnet[debut + 1:fin - 1]}")

# Demander à l'utilisateur s'il veut configurer le réseau

reponse = input("Voulez-vous configurer le réseau ? (oui/non): ").lower()

if reponse == "oui":

auto\_config = True

# Appel de la fonction avec les paramètres fournis par l'utilisateur

print("Configuration du réseau en cours...")

# Ici, vous pouvez appeler une fonction pour configurer automatiquement le réseau

elif reponse == "non":

print("Le script est terminé.")

else:

print("Réponse invalide. Veuillez répondre par 'oui' ou 'non'.")

# Demander à l'utilisateur l'adresse IP du fournisseur

adresse\_fournisseur = input("Veuillez entrer l'adresse IP du fournisseur: ")

# Demander à l'utilisateur le nombre de bureaux

nombre\_bureaux = int(input("Veuillez entrer le nombre de bureaux: "))

# Demander à l'utilisateur le nombre de VLAN par bureau

vlan\_par\_bureau = int(input("Veuillez entrer le nombre de VLAN par bureau: "))

# Demander à l'utilisateur le nombre d'IP à bloquer par bureau

ip\_bloquees\_par\_bureau = int(input("Veuillez entrer le nombre d'IP à bloquer par bureau: "))

# Appel de la fonction avec les paramètres fournis par l'utilisateur

diviser\_adresse\_ip(adresse\_fournisseur, nombre\_bureaux, vlan\_par\_bureau, ip\_bloquees\_par\_bureau)

Maintenant, la question demandant l'adresse IP du fournisseur est positionnée au début du script, avant toutes les autres questions**. Cela rend le flux de travail plus logique et cohérent.**

Pour automatiser la configuration des adresses IP dans Packet Tracer avec ce script, vous devrez ajouter des instructions pour configurer les adresses IP sur les périphériques du réseau, tels que les ordinateurs, les routeurs ou les commutateurs. Voici comment vous pouvez le faire pour quelques périphériques courants :

1. **Pour les ordinateurs :** Vous pouvez utiliser des commandes similaires à celles-ci pour configurer une adresse IP sur un ordinateur dans Packet Tracer :

python

Copy code

# Supposons que "pc" est une instance d'un ordinateur dans Packet Tracer pc.configure(ip="192.168.1.2", subnet\_mask="255.255.255.0", gateway="192.168.1.1")

1. **Pour les routeurs :** Pour configurer une adresse IP sur une interface de routeur dans Packet Tracer, vous pouvez utiliser des commandes comme celles-ci :

python

## Supposons que "router" est une instance d'un routeur dans Packet Tracer

router.interface("GigabitEthernet0/0").ip = "192.168.1.1"

router.interface("GigabitEthernet0/0").subnet\_mask = "255.255.255.0"

1. **Pour les commutateurs :** Les commutateurs Layer 2 dans Packet Tracer n'ont pas d'adresses IP configurables directement. Ils travaillent au niveau de la couche 2 (MAC) et ne nécessitent généralement pas de configuration IP. Cependant, si vous souhaitez effectuer des actions spécifiques sur les commutateurs, vous pouvez les intégrer dans le script en fonction de vos besoins spécifiques.
2. **Pour les serveurs :** Si vous avez des serveurs dans votre réseau, vous pouvez également configurer leurs adresses IP de la même manière que pour les ordinateurs.

Vous devrez ajouter ces instructions dans le script Python pour qu'il configure automatiquement les adresses IP sur les périphériques de votre réseau dans Packet Tracer. Assurez-vous que le script utilise les bons noms de périphériques et d'interfaces pour les configurer correctement.