Syntaxe Python

Python Setup

Installer Python (recommandé)

Linux (Ubuntu / Debian)

```
apt update
apt install python3 python3-venv python3-pip
```

Il est également possible d'installer anaconda. Mais autant utiliser celui qui est inclus sur Linux.

Windows

- Installez Anaconda ou Miniconda

Faire un environnement virtuel dédié

Avec python-venv

Unix/GNU/XNU

```
$ python3 -m virtualenv env_name
$ . ./env_name/bin/activate
(env_name)$ pip install [dépendances]
(env_name)$ # Codez ici
(env_name)$ deactivate
```

Windows

```
> python -m virtualenv env_name
> env_name\Scripts\activate
(env_name)> pip install [dependances]
(env_name)> # Codez ici
(env_name)> deactivate
```

Conda env

Unix/GNU/XNU

```
$ conda create -n env_name
$ conda activate env_name
(env_name)$ conda install [dépendances]
(env_name)$ # Codez ici
(env_name)$ conda deactivate
```

Windows

```
> conda create -n env_name
> conda activate env_name
(env_name)> conda install [dependances]
(env_name)> # Codez ici
(env_name)> conda deactivate
```

Bonus : Docker et VS Code Remote Container

Installations

- Installez Docker (si Windows, utilisez les Linux containers)

```
# GNU/Linux
sudo apt update
sudo apt install git curl sed
curl -fsSL https://get.docker.com -o get-docker.sh
sh get-docker.sh
sudo usermod -aG docker $USER
```

- Installez VS Code ou VS Code Insiders
- Installez le Remote Development extension pack

Usage (setup) (il ne sera pas nécessaire pour la formation)

- `CTRL+P`
- `Remote-Containers: Add Development Container Configuration Files...`
- `Python:3`

Usage (reopen)

- `CTRL+P`
- `Remote-Containers: Reopen Folder in Container`
- Configurez votre environnement
  - `apt update && apt install python3-pip`
  - `pip install -r requirements.txt`
  - Choisissez `/usr/local/bin/python3` en tant qu'interpréteur
  - Installez pylama/pylint et autopep8/yapf proposé par VS Code
- Ne reste plus qu'à coder !

Normes de syntaxe

Normes de documentation

- PEP 257 -- Docstring Conventions (c'est court, donc lisez complètement)
- PEP 484 -- Type Hints, que je recommande de prendre connaissance pleinement
- Google Python Style Guide, (napoleon exemple)

```
def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to
be
    included in the docstring:

    Args:
        param1 (int): The first parameter.
        param2 (str): The second parameter.

    Returns:
        bool: The return value. True for success, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """
```

```
def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    Args:
        param1: The first parameter.
        param2: The second parameter.

    Returns:
        The return value. True for success, False otherwise.

    """
```

- numpydoc docstring guide¶, (napoleon exemple)

```
def function_with_types_in_docstring(param1, param2):
    """Example function with types documented in the docstring.

    `PEP 484`_ type annotations are supported. If attribute, parameter, and
    return types are annotated according to `PEP 484`_, they do not need to
be
    included in the docstring:
```

```
    Parameters
    ----------
    param1 : int
        The first parameter.
    param2 : str
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    .. _PEP 484:
        https://www.python.org/dev/peps/pep-0484/

    """
```

```
def function_with_pep484_type_annotations(param1: int, param2: str) -> bool:
    """Example function with PEP 484 type annotations.

    The return type must be duplicated in the docstring to comply
    with the NumPy docstring style.

    Parameters
    ----------
    param1
        The first parameter.
    param2
        The second parameter.

    Returns
    -------
    bool
        True if successful, False otherwise.

    """
```

Bonus : Sphinx et napoleon

Il est possible de générer des PDFs grâce aux packages Sphinx et Napoléon.

Installation :

```
pip install sphinx
```

Imaginons cette architecture classique :

```
nom_du_projet
|- LICENSE
|- README.md
|- setup.py        # Empaquetage
|- mon_package
|   |- __init__.py  # Déclaration en tant que module (contenu pouvant être vide)
|   |- main.py      # Code
|   |- submodule
|      |- __init__.py  # Déclaration en tant que module (contenu pouvant être
vide)
|      |- code.py
|      |- other_code.py
|- tests
   |- test_mon_package.py
```

Cette exemple est repris avec class_example.

En étant à la racine :

```
$ mkdir docs
$ cd docs
$ sphinx-quickstart

> Separate source and build directories (y/n) [n]: y

> Project name: mon_package

> Author name(s): Prenom NOM <exemple@exemple.com>
> Project release []: X.X.X

> Project language [en]: <ENTRER>
```

Modifiez conf.py :

```
cd source
nano conf.py
```

Remplacez ces lignes :

```
# import os
# import sys
# sys.path.insert(0, os.path.abspath('.'))
```

par :

```
import os
import sys
sys.path.insert(0, os.path.abspath('../..'))  # Précisément ici
```

Et ajoutez Napoleon en extension :

```
extensions = [
    'sphinx.ext.napoleon',
]
```

Remontez au niveau de **docs** et générez un template apidoc

```
cd ..
sphinx-apidoc -o ./source -f ../mon_package
```

Et générez enfin la documentation html

```
make html
```

Pour voir la documentation, ouvrez dans `build/html/index.html`.

Il ne reste plus qu'à le publier !

Normes de code

- **PEP 8 -- Style Guide for Python Code** (si t'es chaud, go !)
- PEP 257 -- Docstring Conventions
- Google Python Style Guide

Outillage

Je ne mettrais, ici, que les instructions pour installez les outils nécessaires pour le codestyle et docstyle.

Instructions (Pycharm ou IntelliJ IDEA)

- Installez Pycharm ou IntelliJ IDEA.
- (IntelliJ IDEA) Installez le plugin Python (ou Python Community, su vous utilisez IntelliJ IDEA Community)

Instructions (VS Code)

- Installez VS Code ou VS Code Insiders
- Installez l'extension Python
- Installez pylama ou pylint avec yapf ou autopep8

```
pip install pylama
pip install pylint
pip install yapf
pip install autopep8
```

- (pylama) Ouvrez les settings (`CTRL + ,`) et recherchez `python.linting.pylamaEnabled`, et **activez-le**.
- (optionnel) Mode brutal : Ouvrez les settings (`CTRL + ,`) et recherchez `python.linting.pylintUseMinimalCheckers`, et **désactivez-le.**
- (optionnel) Réglez votre formatter : Ouvrez les settings (`CTRL + ,`) et recherchez `python.formatting.provider` et **choisissez**.

Autre : Kite (ML autocomplete)

kite.com

Language Tour

Imports

```
from math import sqrt  # ♡
import math  # Oui
import matplotlib.pyplot as plt  # Oui
# from matplotlib import *  # Non
```

Main

```
if __name__ == "__main__":
    # Ecrire le code
```

Variable et Built-in types

```
entier: int = 1_000_000
entier += 1  # += -= *= **= etc.
calc: int = entier**2  # Puissance
calc: int = entier // 2  # Division Euclidienne
print(calc)
# Bitwise operators également supportés
# << shift left
# >> shift right
# & and
# | or
# ~ not
# ^ xor

float_var: float = 1.0
float_var: float = 5e6

complex_var: complex = complex(1, 2)  # 1 + 2j
complex_var: complex = 1 + 2j

bool_var: bool = True
bool_var: bool = False

string: str = "hey\nman"

print("Culcule {} est fait !".format(0.1))
print("First letter: {first}. Last letter: {last}.".format(
    last='Z',
    first='A',
))
# Voir doc python pour + d'info
raw_string: str = r"hey\nman"
print(raw_string)  # => hey\nman

# Multi-line code
long_string: str = "Lorem ipsum dolor sit amet, consectetur adipiscing elit.\
Aenean eget aliquam risus. Sed ac quam vitae enim aliquet auctor. Aliquam\
hendrerit dictum eros nec sagittis. Quisque eleifend vitae arcu congue\
```

```
commodo. Suspendisse convallis ex non euismod hendrerit. Phasellus sit amet\
nulla sodales, scelerisque dolor eget, scelerisque urna. Class aptent taciti\
sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.\
Morbi posuere ac neque a tincidunt. Phasellus maximus quam nec urna sagittis\
lobortis. Nullam non pharetra augue. Donec sagittis orci eu massa finibus\
semper. Donec nisi lorem, laoreet in arcu a, scelerisque euismod dui."

# => print 1 line

multi_line_string: str = """Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Aenean eget aliquam risus. Sed ac quam vitae enim aliquet auctor.
Aliquam hendrerit dictum eros nec sagittis. Quisque eleifend vitae arcu
congue commodo. Suspendisse convallis ex non euismod hendrerit. Phasellus
sit amet nulla sodales, scelerisque dolor eget, scelerisque urna. Class
aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos
himenaeos. Morbi posuere ac neque a tincidunt. Phasellus maximus quam nec
urna sagittis lobortis. Nullam non pharetra augue. Donec sagittis orci eu
massa finibus semper. Donec nisi lorem, laoreet in arcu a, scelerisque
euismod dui."""
# => print multi-line

# Multi-line code (math)
nombre: int = (
    1 + 2 + 3 +  # Commentaire pour couper la ligne en deux
    4 + 5 + 6)

list_var: List[int] = [1, 2, 3, 4, 5]  # mutable (contenu modifiable)

# Note: Notation List[type] introduit via le package typing depuis la 3.5

list_dynamic_var: list = [1, 'two', 3.14, [0, 3, 5]]
last: int = list_var[-1]  # 5
between: List[int] = list_var[0:1]  # [1], end excluded
from_start: List[int] = list_var[:1]  # [1]
from_last: List[int] = list_var[1:]  # [2, 3, 4, 5]
every_x_step: List[int] = list_var[::2]  # [1, 3, 5]
reverse_magic: List[int] = list_var[::-1]  # [5, 4, 3, 2, 1]

tuple_var: Tuple[int] = (1, 2)  # immutable (ex: coordonnées fixe)

dict_var: Dict[str, str] = {
    'foo': 'bar',
    'd3ad': 'cOde',
}  # dict(), mutable

dict_var['new_key'] = 'new_value'

set_var: Set[int] = {
    1, 2, 3
}  # Sans ordre, de toute façon personne l'utilise, mutable
```

Loops

```python
# Loops
# Loops
for i in range(10):
    if i < 1:
        print("do if")
    else:
        print("else")
        break  # Skip else.
else:
    print("I run when for is entirely finished.")

for idx in range(len(list_var)):  # NON
    print(idx)

for value in list_var:  # Oui
    print(value)

for idx, value in enumerate(list_var):  # ♡
    print(value == list_var[idx])  # => True

L = [2, 4, 6, 8, 10]
R = [3, 6, 9, 12, 15]
for lval, rval in zip(L, R):  # Parcourir 2 listes en même temps
    print(lval, rval)
# 2 3
# 4 6
# 6 9
# 8 12
# 10 15

for value in tuple_var:
    print(value)

for key in dict_var:
    print(key)

for value in dict_var.values():
    print(value)

for key, value in dict_var.items():
    print(key, value)

# while, break, continue
while False:
    if 2 is not int:
        print("get out of this hell")
        break  # Termine la boucle
    if 1 is int:
        continue  # Termine l'itération
    print("i guess i'm skipped if 1 is int")

# Compare
a: int = 16
if 15 < a < 30:  # in between
```

```
    print("true")

# or, and, is
if (15 < a < 30) or (a == "16" and a is not str):
    print("do")

# in
if a in [15, 16, 17]:
    print("do")

# False condition
# 0 == False
# None == False
# "" == False
# [] == False
```

Fonctions

```
def function(
        dynamic,
        type_hint_decimal: float,
        *args,  # Surplus d'argument
        optionnel="moi",
        **kwargs) -> int:  # Surplus d'argument key=word
    """
    documentation: description rapide


    Description longue.
    Cette function fait que dalle.
    De toute façon personne l'aime.

    """  # Ceci est stocké dans une variable __doc__ (privé)
print(dynamic)
print(type_hint_decimal)
print(optionnel)
for arg in args:
    print(arg)
for name, kwarg in kwargs.items():
    print(name, kwarg)
return int(dynamic)  # Parce que dynamic est dynamic, faut parser en int


MY_KWARGS = {"arg6": 6, "arg7": "sept", "arg8": 8}
MY_ARGS = ('args var 4', 'args var 5')  # ou list()
print(function('1', 2, *MY_ARGS, optionnel="Le 3e", **MY_KWARGS))
```

Se rappeler de l'ordre : (ordonné et sans nom en 1er, ensuite le unpack *args, keyword et optionnel en 3e et enfin le unpack keyword **kwargs)

Lambda et list map, filter et reduce

Un lambda est une fonction qui n'a pas de nom. Il est possible de lui donner un nom avec une attribution `=` mais, en terme de code style, ce n'est pas conseillé. Généralement, on utilise un lambda pour des opérations nécessitant une `function` ou `Callable` en argument.

```
add_lambda: function = lambda x, y: x + y  # Avec le type hint
add_lambda: Callable[..., int] = lambda x, y: x + y
add_lambda: Callable[[int, int], int] = lambda x, y: x + y

# NOTE: Ici le type hint est [..., ReturnType]
# NOTE 2: Ici, ... ou communement Ellipsis litéral permet de représenter une
# partie d'une list qui n'est pas représentable.
print(add_lambda(1, 2))  # Fonction sans nom (x, y) => x + y


# Equivalent
def add(x: int, y: int) -> int:
    """Littéralement, ajouter"""
    return x + y  # Même fonction avec un nom
```

`map` est un opérateur permettant de générer un `Itérable` nommé `map` en appliquant une `function` sur la liste cible.

`filter` est un opérateur permettant de générer un `Itérable` nommé `filter` en filtrant la liste cible via une condition `bool`.

`reduce`, accessible uniquement via `from functools import reduce`, est un opérateur qui combine les éléments d'une collection en utilisant un fonction. La fonction doit être de la forme `(valeur accumulé, nouvelle valeur) ⇒ nouvelle valeur accumulé`

```
list_seconde: List[int] = [60, 267, 472859]
# => [1, 4, 7880]
list_minutes: List[int] = list(map(lambda val: val // 60, list_seconde))

for val in map(lambda val: val**2, list_seconde):  # ♡
    print(val, end=' ')  # => 3600 71289 223595633881

for val in filter(lambda val: (val % 2) == 0, list_seconde):  # ♡
    print(val, end=' ')  # => 60

sum_var: int = reduce(lambda a, b: a + b, [1, 2, 3, 4])  # => 10
# reduce est uniquement disponible via le package functools
```

Unpack list :

```
print(32, *list_seconde)
print(32, 60, 267, 472859)  # Même chose
```

Error handling

Capturer les erreurs d'exceptions.

```
try:
    # Code avec erreurs
    pass
except:  # Super illegal => catch n'importe quoi
    print("any error")

try:
    raise Exception("my error message")
except Exception as error:  # Pas conseillé => trop générique
    print(error)
else:  # S'execute si sans problème
    pass
finally:  # S'execute après le try/catch/else
    pass


class MyException(Exception):
    """This is my exception error."""


try:
    type_of_error: int = 1
    if type_of_error == 1:
        raise MyException("My Exception")  # ♡
    else:
        raise Exception("Unknow case")
except MyException as error:
    print(error)
else:
    pass
finally:
    pass
```

Generators

Les `generators` sont des `Iterable`, c'est-à-dire, parcourable via boucle `for`.

Avant de commencer, voici des exemples connus d'utilisation de génération de `list/set/dict` par syntaxe ternaires.

```python
# Ternary compare
val: int = 32
print(val if val >= 0 else -val)

# List
var_list: List[int] = [i for i in range(20) if i % 3 > 0]
# => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
var_list: List[Tuple[int]] = [(i, j) for i in range(2) for j in range(3)]
# => [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)]

# Set
var_set: Set[int] = {n**2 for n in range(12)}

# Dict
var_set: Dict[int, int] = {n: n**2 for n in range(6)}
# => {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Donc pour les `generators` :

```python
G: Generator[int, None, None] = (n**2 for n in range(12))
G: Iterable[int] = (n**2 for n in range(12))  # Implique
G: Iterator[int] = (n**2 for n in range(12))  # Equivalent
```

Un générateur est utilisable qu'une seule fois ! Comme dans un boucle `for`.

```python
list(G)  # => [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
list(G)  # => []  # Car, itérable qu'une seule fois !
```

La syntaxe complète d'un générateur est :

```python
def gen() -> Iterable[int]:
    """Generates x^2 from x=0 to x=11."""
    for idx in range(12):
        yield idx**2  # A la place de retourner une seule valeur,
        # on en retourne plusieurs
```

`yield` permet de retourner plusieurs valeurs !

Une belle utilisation d'un générateur est, par exemple, pour les nombres premiers :

```
def gen_primes(max_range: int) -> Iterable[int]:
    """Generate primes up to max_range"""
    primes = set()
    for idx in range(2, max_range):
        if all(idx % p > 0 for p in primes):
            primes.add(idx)
            yield idx
```

```
>>> for prime in gen_primes(100):
...     print(prime)
...
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
```

Classes et les 4 piliers de l'OOP

Encapsulation

```
class Complex:
    """A class supposed to be a complex, but lost his meaning of life.

    Parameters
    ----------
    realpart : float
        Partie réelle
    imagpart : float
        Partie imaginaire

    Attributes
    ----------
    public_property : str
        Propriété publique, accessible partout
    _protected_property : str
        Propriété protégé, accessible pour ses subclasses
    __encrypted_password : str
        Propriété privée, accessible uniquement sur soi
    """
    public_property = "Hey i'm public"  # Read and write everywhere
    _protected_property = "Hey i'm protected"  # Read and write from subclass
    __private_property = "Hey i'm private"  # Read and write self

    def __init__(self, realpart: float, imagpart: float):  # Constructor
        if realpart == 0:
            raise ValueError("This is retarded but 0 is not allowed")
        self.realpart = realpart  # instance var à initialiser
        self.imagpart = imagpart
```

Pour rappel, une propriété **publique** est accessible partout, une propriété **protégée** est accessible uniquement aux méthodes et aux classes héritières. Une propriété **privée** est accessible uniquement dans la classe.

__init__ est le constructeur par défaut, les propriétés variables peuvent être initialisé, à l'instance, ici.

En dehors de ça, on initialise les **attributs statique**, c'est-à-dire, des attributs unique à la classe.

Il est possible de faire des méthodes similairement à des fonctions.

```
    def method(self):
        """do"""
```

Les méthodes statiques, **indépendante le l'instance**, s'initialise avec le décorateur `@staticmethod`. On n'utilise pas `self`, vu que `self` peut ne pas être instanciais. Il faut appeler soi-même pour accéder à une variable statique, exemple : `Complex.__private_property`

```
    @staticmethod
    def static_func(arg: int) -> int:  # static
        """doc"""
        print(Complex.__private_property)
        return arg
```

Les `@classmethod` permettent de s'instancier à partir de soi-même, généralement pour faire des **constructeurs nommés**, comme par exemple les célèbre, `json.parse()` ou `Model.from_json()`.

```
    @classmethod
    def named_contructor(
            cls,
            complex_var: complex,
    ):  # Constructor with a name
        """doc"""
        return cls(complex_var.real, complex_var.imag)
```

Pour faire des **getters et setters**:

```
    # Getter : Read only use case
    # getter
    @property
    def readme(self) -> str:
        """doc"""
        return self.__private_property

    # Getter, Setter: Calculus use case
    # getter
    @property
    def password(self) -> str:
        """doc"""
        print("Decrypting...")
        return self.__encrypted_password

    # setter
    @password.setter
    def password(self, value: str) -> str:
        """doc"""
        print("Encrypting... Get some salt...")
        self.__encrypted_password = value
```

Héritage

L'héritage permet d'obtenir les méthodes et attributs du parent. Autrement dit,

```
class Person:
    age = 18

    def parler(self, voix: str):
        print(voix)


class Moi(Person):
    pass
```

```
>>> p = Moi()
>>> p.parler("Hey")
Hey
>>> p.age
18
```

Abstraction

Les fonctions **abstraites** sont des fonctions ne pouvant être utilisé à l'instance. Pour pouvoir les utiliser, il faut les `override`, c'est-à-dire, hériter de la classe et écraser la fonction. Cela permet de donner un template pour les héritiers. Il est possible d'annoter la fonction avec `@abc.abstractmethod` via le package `abc`. Pour les getter abstraites, `@abc.abstractproperty`.

Il est également possible de faire des classes **abstraites** (classes ne pouvant être instanciais) avec le package `abc`. Cela permet que **toutes** les méthodes sont abstraites.

Exemple :

```
import abc

class Person(abc.ABC):
    @abc.abstractmethod
    def parler(self, voix: str):
        """doc"""
        raise NotImplementedError

    @abc.abstractproperty
    def age(self):
        raise NotImplementedError
```

```
class Monsieur(Person):
    def parler(self, voix: str):
        """Le monsieur parle fort"""
        print("FORT " + voix)

class Mademoiselle(Person):
    def parler(self, voix: str):
        """La mademoiselle parle doucement"""
        print("douce " + voix)
```

Si jamais vous faites :

```
p = Person()
```

Vous aurez :

```
>>> p = Person()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Person with abstract methods age,
parler
```

Notez que dans cette exemple, je n'ai pas défini `age`.

Polymorphisme

Le **polymorphisme** permet l'utilisation de la même méthode malgré que la signature de l'objet est différente.

Exemple :

```
class PersonDroitier():
    def status(self):
        print("Je suis droitier.")


class PersonGaucher():
    def status(self):
        print("Je suis gaucher.")


droitier = PersonDroitier()
gaucher = PersonGaucher()
for person in (droitier, gaucher):
    person.status()
```

```
Je suis droitier.
Je suis gaucher.
```

Grâce à l'**abstraction** et à l'**héritage**, le polymorphisme devient un des pilier les plus importants de l'OOP.

En utilisant un `override`, on obtient :

```python
from abc import ABC, abstractmethod


class Person(ABC):
    @abstractmethod
    def status(self):
        raise NotImplementedError


class PersonDroitier(Person):
    def status(self):
        print("Je suis droitier.")


class PersonGaucher(Person):
    def status(self):
        print("Je suis gaucher.")


droitier = PersonDroitier()
gaucher = PersonGaucher()
for person in (droitier, gaucher):
    person.status()
```

```
Je suis droitier.
Je suis gaucher.
```