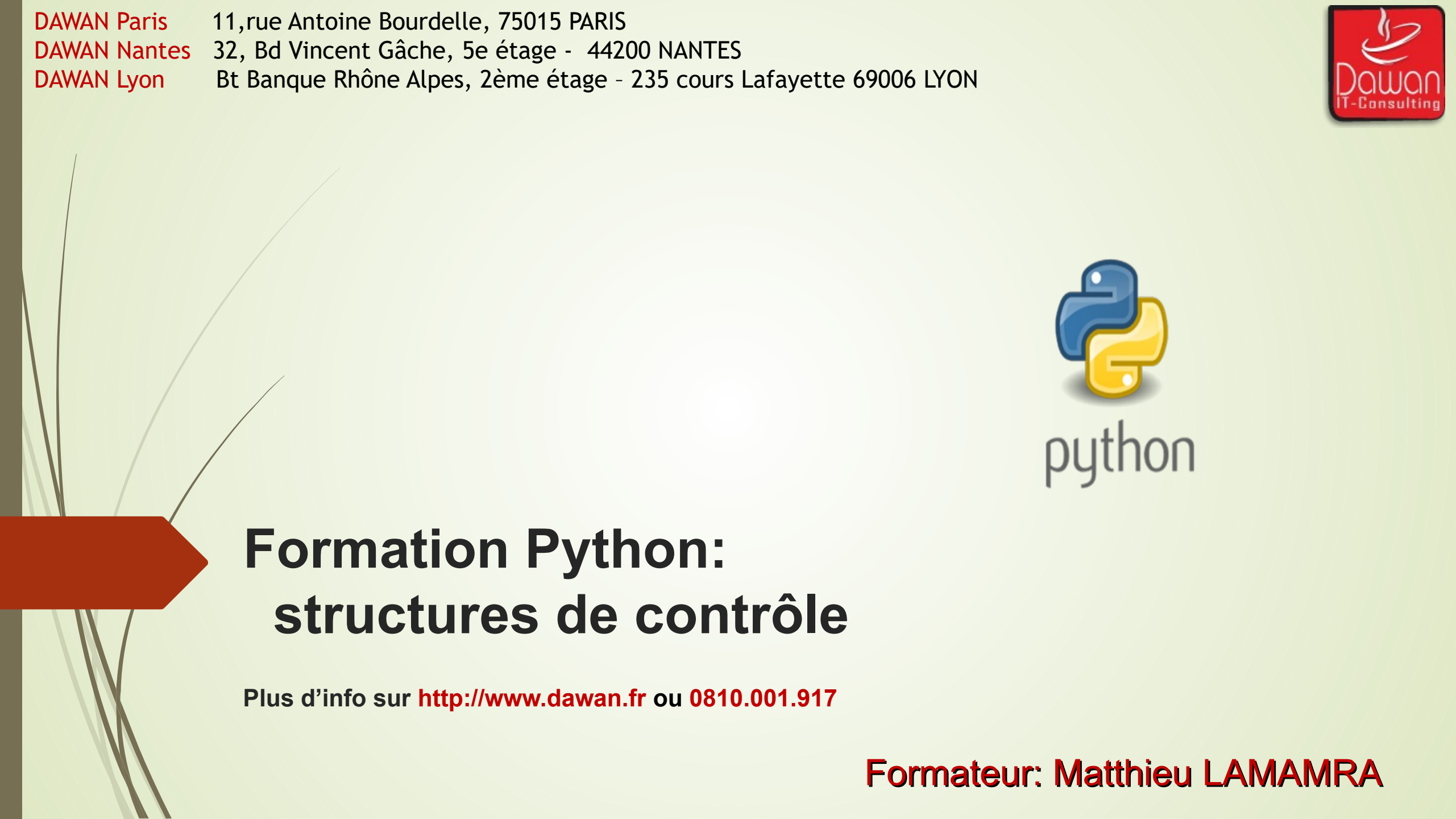
DAWAN Paris 11,rue Antoine Bourdelle, 75015 PARIS
DAWAN Nantes 32, Bd Vincent Gâche, 5e étage - 44200 NANTES
DAWAN Lyon Bt Banque Rhône Alpes, 2ème étage – 235 cours Lafayette 69006 LYON



Formation Python: structures de contrôle

Plus d'info sur http://www.dawan.fr ou 0810.001.917

Formateur: Matthieu LAMAMRA

Structures de contrôle

- Principe

- Structurer l’exécution du code selon :
- En-têtes terminés par ‘ : ‘
- Blocs de code exécutés de manière conditionnelle ou en itération (boucle)

=> En Python, les blocs sont définis par l'indentation (2 ou 4 espaces)

Structures de contrôle

- IF : structures conditionnelles

- if *condition* : définit l'en-tête conditionnelle
- elif *condition* : pour une alternative
- else : pour les cas en dehors des conditions précédentes

  => utilisation des opérateurs logiques et de comparaison

  => *condition* est évaluée comme booléen

```
if x ≥ 2 and y != x :
    bloc de code……
```

Structures de contrôle

- IF : Évaluations booléennes

Type	FAUX
bool	False
int	0
float	0.0
string	‘’
tuple	()
list	[]
dict	{}
None type	None

Structures de contrôle

- IF : exemple complet

```
x = input("entier naturel: ")

if not x.isnumeric():
    print("pas un entier naturel!")
elif not x:
    print("x vaut 0!")
else:
    print("x est positif!")
```

```
status = "retard" if delay < 10 else "annulé"  # opérateur ternaire
```

Structures de contrôle

- La boucle FOR

- En python, la boucle FOR parcourt des séquences, dites **itérables**
- En-tête : **for** *element* **in** *sequence*:

```
for letter in "abracadabrantesque":
    print(letter)

trainings = ("java", "python", "c#")
for t in trainings:
    print(t, len(t))

trainees = {"java": "bob", "python": "Jean", "C#": "henry"}

for lang, name in trainees.items():
    print(f"{ lang.capitalize() }: { name.upper() }"
```

Structures de contrôle

- La boucle FOR : enumerate()

- Énumérate transforme une séquence en liste de tuples (index, elem)
- Cela permet d'itérer à la fois sur les indices et les éléments

```
seq  = ["a", "b", "c"]

for i, elem in enumerate(seq):
    seq[ i ] = elem.upper()
```

Structures de contrôle

- Boucle FOR : la fonction range()

- Génère une séquence d'entiers sur lesquels on va boucler
- **range**(*début_compris*, *fin_non_compris*, *pas*)
- Dans une boucle : for i in range(…) :

```
r1, r2, r3 = range(5), range(5, 10), range(1, 11, 2)

print(list(r1))
print(list(r2))
print(list(r3))

r4 = range(10, 5, -1)
print(tuple(r4))
```

Structures de contrôle

- La boucle WHILE

- En python, la boucle WHILE s'exécute tant qu'une condition est vraie
- En-tête : **while *condition* :**

```
a, b = 0, 1

while b < 100:
    print(b)
    a, b = b, a + b
```

Structures de contrôle

- Casser la boucle : break, continue, else

  - **break :** interrompt l'exécution de la boucle et la quitte
  - **continue :** interrompt l'exécution de la boucle et passe à l'itération suivante
  - **else :** s'exécute à la fin de la boucle, sauf si celle ci est interrompue par break
  - **pass :** instruction tampon, bloc par défaut, ne fait strictement rien

```
do_break = True # ou False

for i in range( 10 ): # ou while i < 10
    if do_break:
        break
    if i == 6:
        continue
    print( i )              # i +=1
else:
    Print( "else  is executed !" )
```

```
for i in range(5):
    pass
```

Structures de contrôle

- Les listes en intension

- Ou encore « Comprehension lists », définissent la transformation d'une liste
- En logique, l'intension d'un concept est sa définition

```
sequence, new_sequence  = ["a", "b", "c"], []

for element in sequence:
    new_sequence.append(element.upper())

comprehension = [ element.upper() for element in sequence ]

print(new_sequence, comprehension)
```

Structures de contrôle

- Les listes en intension : condition

- Structure générale

  [transformation for élément in sequence if condition]

```
odd_squares = []

for i in range(10):
    if not i % 2:
        odd_squares.append(i ** 2)

comprehension = [ i ** 2 for i in range(10) if not i % 2 ]
```

Fonctions Python

- Déclaration

- Définissent des blocs de code qu'on peut paramétrer et appeler
- Retournent une valeur précédée de return ou None sinon
- Structure générale :

```
def nom_fonction([param1], [param2=default], [args*], [kwargs**]) : #signature
    bloc…
    [ return ]

    nom_fonction(param1, param2, […], {…}) # appel
```

Fonctions Python

- Premier exemple

- Fonction factorielle

```
def factorielle( n ):
    acc = 1
    for i in range(1, n + 1):
        acc *= i
    return acc

print(factorielle(5), factorielle(10))
```

Fonctions Python

- Paramètres et retour

- Ne sont pas typés

```
def fois3( n ):
    return 3 * n

fois3(3) # 9
fois3("rien ") # rien rien rien
```

- Valeurs de retour multiples affectables par les tuples « Unpacking »

```
def mini_maxi( liste ):
    return min( liste ), max( liste )

mini, maxi = mini_maxi( [ 1, 2, 3 ] )

tup = mini_maxi( [ 1, 2, 3 ] )
tup[ 0 ] , tup[ 1 ]
```

Fonctions Python

- Gestion des Paramètres

  - Par défaut les paramètres d'appels sont obligatoires et positionnels: dans l'ordre de la signature

```
def create_account( id, name ):
    return i, name

create_account( 1, "henry" )
```

  - En utilisant les noms des paramètres à l'appel, on peut modifier l'ordre des paramètres

```
def create_account( id, name ):
    return i, name

create_account( id=1, name="henry" )

create_account( name="henry", id=1 )
```

Fonctions Python

- Gestion des Paramètres

  - Un paramètre est optionnel s'il lui est affecté une valeur par défaut dans la signature

```
def create_account( id, name, mode="user" ):
    return i, name, mode

create_account( 1, "henry" )

create_account( 1, "henry",  "admin" )
```

  - Les paramètres optionnels doivent être placés derrière les paramètres positionnels dans la signature

```
def create_account( id, name, mode="user", priority=10 ):
    return i, name, mode, priority

create_account( 1, "henry" )

create_account( name="henry", id=1, priotrity=1 )
```

Fonctions Python

- « variadics » : *args

- Dans la signature : permet de définir un nombre quelconque de paramètres optionnels non nommés
- Accessibles dans le corps de la fonction sous forme de tuple

```
def test_args( id, *args ):
    for arg in args:
        print( arg )
        return i, args

test_args( 1 )

test_args( 1, "two", 3.0, "four" )
```

Fonctions Python

- « variadics » : *args
  - A l'appel : dispose les éléments d'un tuple ou d'une liste comme arguments d'une fonction

```
def call_args( arg1, arg2, arg3 ):
    return arg1, arg2, arg3

params = ( "hi", "my name", "is")

call_args( *params )

call_args( "hi", *params[0:2] )
```

  - « unpacking »

```
first, second,  *others = [1, 2, 3, 4, 5]
```

Fonctions Python

- « variadics » : **kwargs

- Dans la signature : permet de définir un nombre quelconque de paramètres optionnels nommés
- Accessibles dans le corps de la fonction sous forme de dictionnaire

```
def test_kwargs( id, **kwargs ):
    for key, value in kwargs.items():
        print( key, value )
    return i, kwargs

test_kwargs( 1 )

test_args( 1, two=2, three=3.0, four="four" )
```

Fonctions Python

- « variadics » : **kwargs
  - A l'appel : dispose les paires « clé / valeur » de dictionnaires comme paramètres d'une fonction

```
def call_kwargs( arg1, arg2, arg3 ):
    return arg1, arg2, arg3

params = { "arg1": "hi", "arg2": "my name", "arg3": "is" }

call_args( **params )

del params[ "arg1" ]

call_args( "hi", **params )
```

Fonctions Python

- « / et * » dans les signatures
  - À gauche de / , les paramètres doivent être appelés de façon positionnelle
  - A droite de * , les paramètres doivent être appelés de façon nommée

```
def call( arg1, /, arg2="dflt", *, arg3="dflt" ):
    return arg1, arg2, arg3

call( arg1="val" ) # ERROR
call( "val1", "val2", "val3" ) # ERROR
```

Fonctions Python

- Portée d'une variable

- Une variable est reconnue et accessible dans le bloc dans lequel elle a été créée
- Elle est accessible **globalement** dans les fonctions subalternes si elle n'y a pas été redéfinie
- Sinon, la variable redéfinie est dite **locale** à la fonction – elle est indépendante en mémoire

```
x = 10

def do_thing() :
    return x

do_thing() # 10
```

```
x = 10
print( f"x global : { id( x ) } )"

def do_local_thing() :
    x = 5
    print( f"x local : { id( x ) } )"
    return x

do_local_thing() # 5
x                # 10
```

Fonctions Python

- Portée d'une variable

- Une variable globale est modifiable dans les blocs subalternes si elle y figure précédée de **global**

```
x, y = 10, 15
print( f"x global : { id( x ) } )"

def do_global_thing() :
    global x, y
    x = 5
    print( f"x in fct : { id( x ) } )"
    return x

do_global_thing() # 5
x                 # 5
```

Fonctions Python

- Fonction comme variable et paramètre

➢ Si **« def ma_fonction() : »** est la signature, et **« ma_fonction() »** l'appel / valeur de retour

=> « ma_fonction » est une variable qui référence la fonction
=> cette variable peut utiliser **l'opérateur d'appel « () »**

```
def my_map( func, lst ):
    return func( elem for elem in lst )

def square( x ):
    return x ** 2

my_map( square, [ 1, 2, 3 ] ) # [ 1, 4, 9 ]
```

Fonctions Python

- Fonctions lambda

- Fonctions éphémères :
  - sans nom, donc à usage unique
  - utilisant les même paramètres que les fonctions nommées
  - constituées d'une seule expression

```
def my_map( func, lst ):
    return func( elem for elem in lst )

my_map( lambda x: x ** 2, [ 1, 2, 3 ] ) # [ 1, 4, 9 ]
```

Fonctions Python

- Documentation, introspection

➢ **Docstring :** triple-double-quotes de documentation sous la signature, accessible via **help()**

```
def func():
    """fonction vide !"""
    pass

help( func )
```

➢ **Introspection :** la fonction **dir()** affiche les attributs internes des fonctions

```
dir( func )
# [ "__name__", ...  ]
```

Modules Python

- Principe

- **Un module** est un fichier contenant du code python, suffixé par **« .py »**
- On peut injecter le contenu du module dans un autre fichier grâce à **import**
- **L'espace de nom** du module importé permet de reconnaître et d'utiliser ce contenu
- Un module importé est **exécuté** donc attention aux affichages

```
# my_module.py
print("coucou")
n0 = 3

def factorielle( n ):
    acc = 1
    for i in range(1, n + 1):
        acc *= i
    return acc
```

```
# other_module.py

import my_module # coucou

my_module.factorielle(my_module.n0)
```

Modules Python

- bibliothèques

- **Une bibliothèque** est un module uniquement composé de fonctions à importer
- On importe directement les fonctions avec **from** *module* **import** funcA, funcB, *
- On renomme les fonctions importées avec **from** *module* **import** funcA **as** fA

```
# other_module.py

from my_module import factorielle as fact, n0
fact(n0)

# ou
from my_module import *
factorielle(n0)
```

Modules Python

- Introspection et exécution

- L'espace de nom est une **variable** de type module (**dir, help)**
- **Une docstring** peut être placée en haut du module
- L'attribut « **__name__** » d'un module vaut
  - « **__main__** » si le module est directement exécuté par l'interpréteur (programme principal)
  - le nom du fichier, sans l'extension « .py » si le module est importé
- Cela permet de définir un bloc de code qui ne s'exécutera pas à l'import

```
# my_module.py

def square( x ):
    return x ** 2

If __name__ == "__main__":
    print("coucou", square(2))
```

```
# other_module.py

import my_module

my_module.__name__
# "my_module"
```

```
$ python3 my_module.py

coucou 4
```

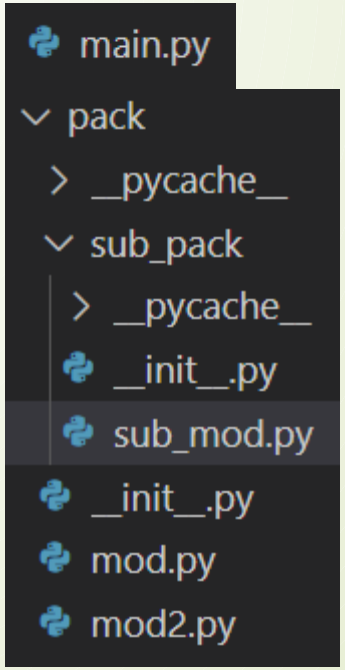
Modules Python

- Structuration en Paquets

- Les paquets sont des **répertoires** contenant des modules et un fichier **« __init__.py »**
- Les paquets peuvent se décomposer en sous paquets (sous répertoires)
- Les imports de modules dans les paquets se font sous les formes ci dessous :
- **On utilise toujours les packages depuis l'extérieur (ici main.py)**

```
# chemins d'imports absolus, dans main.py
import pack.mod
from pack.mod import func
from pack.sub_pack.sub_mod import sub_func

# chemins imports relatifs
# mod.py
from .sub_pack.sub_mod import kont
# sub_mod.py
from ..mod2 import funk
```



Modules Python

- La Bibliothèque standard

- La bibliothèque standard contient les modules installés avec python

- Ses modules variés donnent au langage son caractère « multi-usage »

  https://docs.python.org/fr/3/library/index.html

- Exemples :

  - Expressions régulières : **re**
  - Fonctions mathématiques : **math**
  - Gestion des dates et heures : **datetime**
  - Internationalisation : **locale, gettext**
  - Intéraction avec l'OS : **sys, os, io**
  - Formats de fichiers : **csv, xml, json**
  - Formats de compression : **zlib, zipfile**
  - Interfaces graphiques : **tkinter**
  - ...

Modules Python

- Exemple : Datetime

- Permet de manipuler les dates

```
from datetime import datetime

datetime(annee, mois, jour, heure, minute, seconde, microseconde, fuseau horaire)
```

- Seuls l'année, le mois et le jour sont obligatoires

Modules Python

- Datetime : créations

```
from datetime import datetime

# à partir d'une chaine de caractères (strptime: "p" pour parse)
dt = datetime.strptime("2021-12-17 10:14:20", "%Y-%m-%d %H:%M:%S")

# maintenant
now = datetime.now()

# nb de secondes depuis le 1er janvier 1970
dt = datetime.fromtimestamp(1639732460.0)
```

Modules Python

- Datetime : utilisations

```
from datetime import datetime

dt.year, dt.month, dt.hour, dt.date(), dt.time() # ...

# 0 => lundi, 6 => dimanche
dt.weekday()

# timestamp: nb de scondes depuis 1er janv. 1970
dt.timestamp()

# formater la date en chaine (strftime: "f" pour format)
dt.strftime("%Y / %m / %d"))
```

Modules Python

- Datetime : durées (Timedelta)

```
# arithmétique de date
premier_confinement = datetime.strptime("14/03/2020", "%d/%m/%Y")
aujourdhui = datetime.now()

duree_covid = aujourdhui - premier_confinement

# objet timedelta
duree_covid.days, type(duree_covid)

cuisson_oeuf_coque = timedelta(minutes=3)
atable = datetime.now() + cuisson_oeuf_coque
atable.time()
```

Modules Python

- Exemple : re

- Permet de manipuler expressions régulières

- Les expressions régulières sont des chaines de caractères
- qui décrivent des modèles de chaines de caractères
- Via une syntaxe particulière
- **https://cheatography.com/davechild/cheat-sheets/regular-expressions/**

```
import re

# est ce que le modèle matche la chaine depuis le début ?
re.match( "(\w+) weighs (\d+(\.\d+)?) (kg|g|lbs)", "peter weighs 144 lbs")

# est ce que la chaine contient un float ?
re.search( "(\d+(\.\d+)?)",  "peter weighs 144.5 lbs")

# chercher et remplacer
re.sub( "(\d+(\.\d+)?)", "*****" , "peter weighs 144.5 lbs")
# peter weighs ***** lbs
```

Modules Python

- Calendar