

Séance du mardi 9/06

* API \Leftrightarrow modularité du code

* toujours se poser la question :
quelles sont les briques élémentaires
dont j'ai besoin \Rightarrow chercher un
module et consulter son API

- point d'entrée public des
utilisateurs d'une bibliothèque

\hookrightarrow API liste des fonctions
proposées dans le module

- le point d'entrée est la doc / vers
service web

Ex. : lifop5.univ-lyon1.fr / users d'accès
au code
source

\hookrightarrow service web / API Rest-

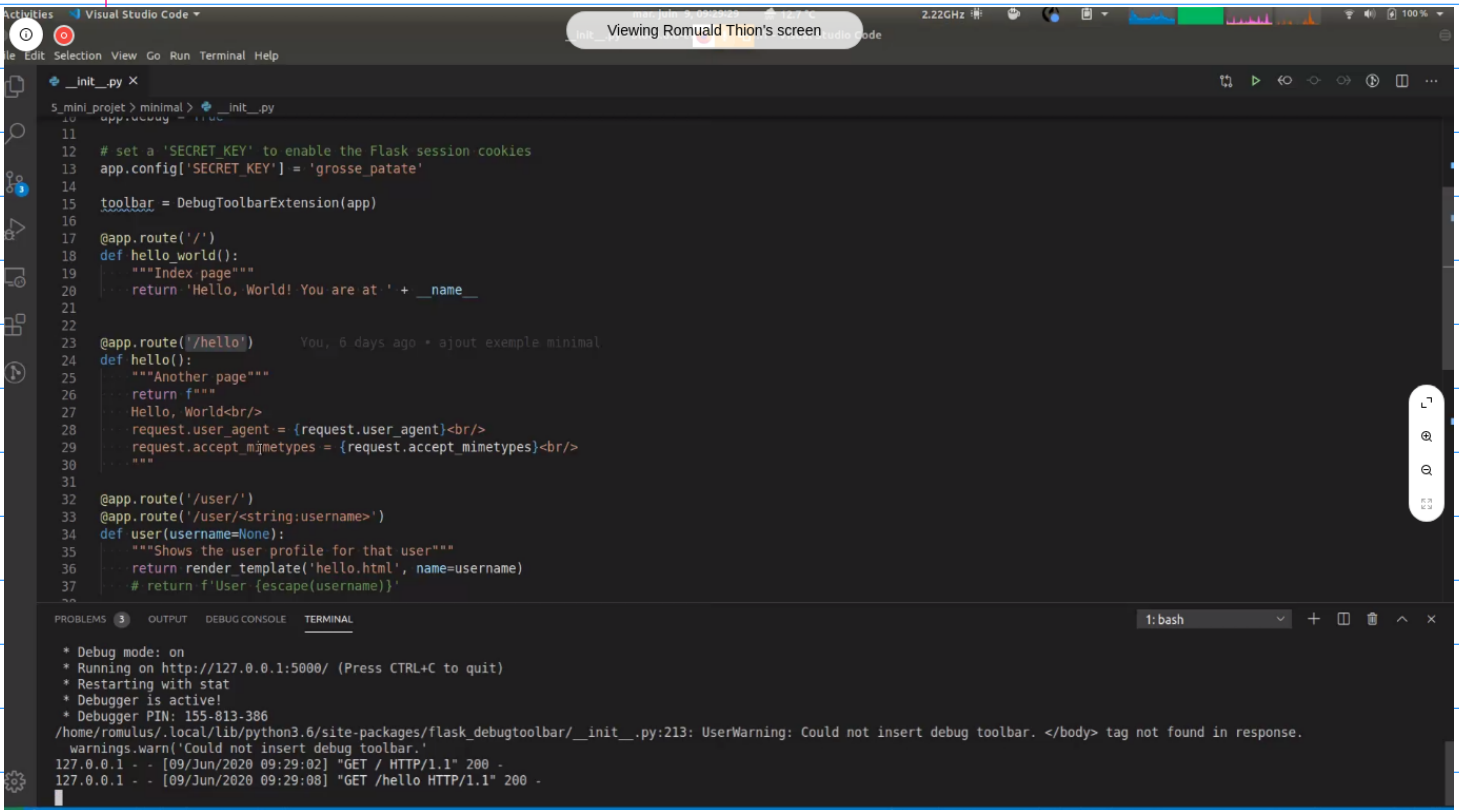
\hookrightarrow URL qui permettent
d'accéder à des

En programmation objet, l'API
est la partie publique

Par exemple en C++ et en Java certains

privés par un underscore.

* Flask par exemple propose une API avec décorateurs `@app.route` qui permettent de router / diriger les URL vers un traitement par une fonction.



```
__init__.py
10 app.config.from_object(__name__)
11
12 # set a 'SECRET_KEY' to enable the Flask session cookies
13 app.config['SECRET_KEY'] = 'grosse patate'
14
15 toolbar = DebugToolbarExtension(app)
16
17 @app.route('/')
18 def hello_world():
19     """Index page"""
20     return 'Hello, World! You are at ' + __name__
21
22
23 @app.route('/hello')
24 def hello():
25     """Another page"""
26     return f"""
27     Hello, World<br/>
28     request.user_agent = {request.user_agent}<br/>
29     request.accept_mimetypes = {request.accept_mimetypes}<br/>
30     """
31
32 @app.route('/user/')
33 @app.route('/user/<string:username>')
34 def user(username=None):
35     """Shows the user profile for that user"""
36     return render_template('hello.html', name=username)
37     # return f'User {escape(username)}'
```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL

1: bash

```
* Debug mode: on
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 155-813-386
/home/romulus/.local/lib/python3.6/site-packages/flask_debugtoolbar/__init__.py:213: UserWarning: Could not insert debug toolbar. </body> tag not found in response.
warnings.warn('Could not insert debug toolbar.')
127.0.0.1 - - [09/Jun/2020 09:29:02] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [09/Jun/2020 09:29:08] "GET /hello HTTP/1.1" 200 -
```

* Décorateur `@wraps`

↳ permet de retrouver la fonction d'origine avec l'attribut `-- wrapped --`

```
TP_func.py TP_func_test.py temp.py X
1_ paradigmes_de_programmation > TP > profs > temp.py
1  from functools import wraps
2
3  def counter(fn):
4      """Un décorateur pour compter le nombre d'appels à une fonction"""
5      @wraps(fn):
6      def wrapped(*args, **kwargs):
7          wrapped.invocations += 1
8          return fn(*args, **kwargs)
9          wrapped.invocations = 0
10         return wrapped
11
12     @counter
13     def test(x):
14         return x

PROBLEMS 56 OUTPUT DEBUG CONSOLE TERMINAL
2
>>> test
<function test at 0x7f20c4381950>
>>> test.__wrapped__
<function test at 0x7f20c43818c8>
>>>
romulus romulus-elite profs ? : 1 profs python3 -i temp.py
>>> test
<function counter.<locals>.wrapped at 0x7fbfcd225950>
>>> []
```

Un décorateur est une fonction qui modifie le comportement d'une autre fonction.
Plus précisément, c'est une fonction d qui prend une fonction f (unaire) en argument
telle que d(f) est la fonction f dont le comportement est modifié.

* le décorateur n'est exécuté qu'une seule fois (par l'appel à @decorator) par exemple lors de l'import du module

```
TP_funct.py  temp.py
paradigmes_de_programmation > TP > profs > temp.py
1  from functools import wraps
2
3  def debug(fn):
4      print("Debug")
5      """Un décorateur pour compter le nombre d'appels à une fonction"""
6      @wraps(fn)
7      def wrapped(*args, **kwargs):
8          wrapped.patate = wrapped.patate + 1
9          print(f'{fn.__name__}({args}, {kwargs})')
10         return fn(*args, **kwargs)
11     wrapped.patate = 0
12     return wrapped
13
14 @debug
15 def test(x):
16     return x
17
PROBLEMS 7  OUTPUT  DEBUG CONSOLE  TERMINAL
>>> test(3)
test((3,), {})
3
>>> test(3)
```

```
Créer Debug
Créer test
Décorer test =>
  - appeler debug
  - remplacer test par wrapped
Appelle test
```

Brigitte Ordre complet :1-2-3-14-15-4-5-7-6-11-12-8-9-10-12

on définit test puis on applique
le décorateur.

```

1_ paradigmes_de_programmation > TP > profs > temp.py
1  from functools import wraps
2
3  def debug(fn):
4      print("Debug")
5      """Un décorateur pour compter le nombre d'appels à une fonction"""
6      @wraps(fn)
7      def res(*args, **kwargs):
8          res.patate = res.patate + 1
9          print(f'{fn.__name__}({args}, {kwargs})')
10         return fn(*args, **kwargs)
11     res.patate = 0
12     return res
13
14     I
15 @debug
16 def test(x):
17     return x
18
19 @debug
20 def test2(x):
21     return x

```

La syntaxe @decorateur est-elle utilisable avec un décorateur qui aurait 2 paramètres comme le maybe du TP ?

```

# @debug
def test(x):
    return x

test = debug(test)

```

Comment passer plusieurs paramètres au décorateur ?

> Il faut faire de debug une fonction d'ordre supérieur qui renvoie un décorateur
 → comme pour la fonction

```

1 _paradigmes_de_programmation > TP > profs > temp.py
2
3 from functools import wraps
4
5 def debug(msg_debug):
6     def debug_avec_msg(fn):
7         """Un décorateur pour compter le nombre d'appels à une fonction"""
8         @wraps(fn)
9         def fn_decoree(*args, **kwargs):
10             fn_decoree.patate = fn_decoree.patate + 1
11             print(f'{msg_debug} : {fn.__name__}({args}, {kwargs})')
12             return fn(*args, **kwargs)
13         fn_decoree.patate = 0
14         return fn_decoree
15     return debug_avec_msg
16
17 @debug
18 def test(x):
19     return x
20 # test = debug(test)
21 # test = debug("mon debug", test)
22

```

• Et avec les élèves ?

Il peuvent utiliser sans mettre les mains dans le cambouis :
on jette un "voile pudique" sur le fonctionnement ?

Je trouve ça au final plutôt sympa pour comprendre et détailler la récursivité.
Je trouve que c'est chaud à bien comprendre l'empilage
et le dépilement lors d'appels récursifs et là, ça aide

les élèves ne vont pas écrire
des décorateurs mais
ils vont en utiliser
(pour des appels web)
et on pourra en écrire
pour instrumenter les
codes des élèves

Par exemple un décorateur pour compter les appels dans un fichier de test.

L > il vaut mieux rester toujours en Python.

Différence entre instruction et expression

Instruction / Statement	Expression
<pre>def f(x): return x**2</pre>	<pre>lambda x: x**2</pre> <p>(pas de docstring pas de nom)</p> <p>> on peut l'affecter à une variable</p>

functions are first class citizen

Calculabilité

2 théories

lambda-calcul
(Alonzo)
↳ programmation
fonctionnelle

Machine de Turing
· séquentiel
· mémoire \Rightarrow ruban
· UC \Rightarrow automatique

Thèse de Church-Turing
Tout ce qu'on peut calculer
avec 1 modèle, on peut le calculer
avec l'autre

Autre paradigme équivalent:
fonctions récursives ?? défini
par Gödel

↳ notion de calcul universel

Que garder dans le TP paradigmes?

* Les caractères

\hookrightarrow pour comparer les
algos de tri

* Le parall \Rightarrow applicable
à la class

On peut garder les versions itératives
et récursives :

Définition d'une liste en program-
-mation fonctionnelle :

```
LISTE = LISTE_VIDE ou X suivi de LISTE
```

Discussion finale sur la
bonne façon d'écrire du code

* Δ multiplication des variables

* \hookrightarrow exercice : corrections croisées

Commentaire de Bruce Lapaglia sur Mattemos.

Dans un monde parfait :

- définition du paradigme fonctionnel

- exemple de curryfication simple (pur fonctionnel)

- la récursivité simple

- différence entre fonctionnel et iter (traduction de récursif en iter)

- illustration par le prof de décorateurs simple avec la récursivité

- Plus tard, paradigme POO

- comparaison POO et fonctionnel avec la création d'attribut d'une fonction "à la volée"

- utilisation avec liste chaînée, arbre

Dans la réalité :

- définition du paradigme fonctionnel

- exemple de curryfication simple (pur fonctionnel)

- la récursivité simple

- différence entre fonctionnel et iter (traduction de récursif en iter)

- illustration par le prof de décorateurs simple avec la récursivité

- Plus tard, définition du paradigme POO

- comparaison POO et fonctionnel avec la création d'attribut d'une fonction "à la volée"

- utilisation avec liste chaînée, arbre

Discussions sur l'organisation de l'année de terminal

<https://www.education.gouv.fr/bo/20/Special2/MENE2001797N.htm>

pour la partie machine, exo1: Le premier exercice consiste à programmer un algorithme figurant explicitement au programme, ne présentant pas de difficulté particulière, dont on fournit une spécification. Il s'agit donc de restituer un algorithme rencontré et travaillé à plusieurs reprises en cours de formation.

Programmation événementielle

- * Thinker

- * Turtle => avantage d'être plus simple

* aller voir sur les
ressources d'autres DTU
(Strasbourg)