Débuter en python - Partie 2

Présenté par Guillaume Mousnier

Plan

1. Les erreurs
2. Les modules
3. Les namespaces
4. Les fichiers
5. Structure d'un projet
6. (Post-projet) Les classes
7. (Post-projet) Fonctions de haut niveau

Les erreurs

Les types d'erreurs / exceptions

Python est très expressif et n'hésite pas à lever des erreurs:

```
SyntaxError # Erreur de parsing
IndentationError # Mauvaise indentation
NameError # Variable non définie
TypeError # Erreur de typage
IndexError # Index inexistant
KeyError # Clef inexistante .....
```

Python en a beaucoup déjà définies, ce sont les built-in exceptions:

https://docs.python.org/3/library/exceptions.html

Les erreurs

Les types d'erreurs / exceptions (2)

Tout ces types d'erreurs ont un type commun: `Exception`. On dit que ces erreurs héritent de la classe ou du type `Exception`. (cf partie 2).

Il est également possible d'en créer de nouvelles, mais également des dérivés (cf partie 2).

Ainsi les modules et les librairies (modules externes) amènent leurs lots de nouveaux types d'erreurs.

Les erreurs

Gérer les erreurs

Lorsqu'une erreur est levée, elle met fin à l'éxecution du programme. Il faut les gérer pour garantir la continuité de notre programme.

On utilise alors les déclarations `try` et `except` pour les gérer.

Le `try` entoure du code à risque (susceptible de lever une exception).
Si il y a une erreur (attendue), `except` va éxecuter du code qui lui a été fourni. Le programme continue.

Les erreurs

Gérer les erreurs (2)

Programme sans try / except:

```
dict_students = {"John": 18.65, "Jacques": 10.00}
print(dict_students["Martine"])
# KeyError; Le programme s'arrête.
# Le code dessous ne s'exécute pas.
print(dict_students["John"])
```

Programme avec try / except gérant les KeyError:

```
try:
    dict_students = {"John": 18.65, "Jacques": 10.00}
    print(dict_students["Martine"])
except KeyError:
    # Le programme continue.
    print(dict_students["John"])
```

Gérer les erreurs (3)

On peut chaîner les `except` pour gérer différents types d'erreurs:

```
students_name = ["John", "Gilles", "Jacques", "Jean Charle
students_notes = {"John": 18.65, "Jacques": 10.00, "Gilles
for student in students_name:
    try:
        students_notes[student] += 1.00
    except TypeError:
        print(student, "'s score is wrong!")
        del students_notes[student]
    except KeyError:
        print(student, " doesn't have score!")

print(students_notes)
```

```
Gilles 's note is wrong!
Jean Charles  doesn't have score!
{'John': 19.65, 'Jacques': 11.0}
```

Gérer les erreurs (4)

/! Evitez d'utiliser le type Exception:

```
for student in students_name:
    try:
        students_notes[student] += 1.00
    except Exception: # Chaque erreur est une exception
         print("Il y a eu une erreur !!")
    except TypeError: # On ne passe plus ici
        print(student, "'s score is wrong!")
        del students_notes[student]
    except KeyError:  # On ne passe plus ici
        print(student, " doesn't have score!")

print(students_notes)
```

```
Il y a eu une erreur !!
Il y a eu une erreur !!
{'John': 19.65, 'Jacques': 11.0}
```

Les erreurs

Lever une erreur

Lorsque l'on écrit un programme, il peut être intéressant de lever des erreurs pour communiquer avec l'utilisateur (développeur ou autres).

On peut utiliser `raise` pour lever une erreur:

```
if value < 0:
    # on choisit de lever une erreur avec un message perso
    raise ValueError("The value can't be negative!")
```

Les modules

Les modules

Parfois on veut utiliser du code prêt à l'emploi autre que les built-ins. Ce sont les modules (1 module ~= 1 fichier python).

On peut importer différents modules dans son programme pour ajouter des fonctionnalités.

Certains modules sont préinstallés avec python: https://docs.python.org/3/py-modindex.html

Les modules

Importer un module

On peut importer un module dans son intégralité:

```
import math
print(math.pi)
```

Ou importer quelques parties du module:

```
from math import pi
print(pi)
```

Ou même tout importer d'un module: **/!**

```
from math import * # charge tout dans le namespace
print(pi)
```

Les modules

Importer un module (2)

On peut aussi utiliser le système d'import pour découper son programme en plusieurs fichiers:

```
from .my_folder.my_file import my_function
# On importe my_function du sous-module my_file
```

On peut alors créer une arborescence de fichier `.py` contenant chacun une petite partie du code, que l'on importera au besoin.

Pour que python sache reconnaître un fichier comme un module, le dossier doit contenir un fichier `__init__.py` !

Les modules internes indispensables

- `datetime` : gestion des dates (formatage, types, calculs...)
- `math` : fonctions de calculs et constantes
- `os` , `sys` , `suprocess` , `shutil` : dialoguer avec l'os, avoir des infos du système, éxecuter des commandes, gérer les fichiers
- `random` : génération aléatoire (nombres, mélange...)
- `collection` : structures de données additionnelles
- `logging` : gestionnaire de log / print
- `json` , `csv` : lecture, écriture de fichiers (json, csv)

Les modules internes indispensables (2)

- `argparse` : gestionnaire de paramètres de script
- `threading` , `multiprocessing` : parallélisation de calculs
- `itertools` , `functools` : programmation fonctionnelle
- `string` , `re` : templating, formattage et regexp
- `operator` : tout les opérateurs en fonctions
- `configparser` : gestionnaire de configuration
- `unittest` : utilitaires pour tests unitaires

Appartée - pip

`pip` est un gestionnaire de paquets ultra complet pour python. Il permet d'installer et de gérer des modules / librairies externes pour python (dépôt `pypa`).
Sur votre système vous pouvez par exemple installer le module `requests` :

```
pip3 install requests
pip3 install requests==2.18.4
```

Mais il permet également de gérer les paquets:

```
pip3 install --upgrade requests
pip3 uninstall requests
```

Il est normalement installé par défaut. Si ce n'est pas le cas: installez le !

Quelques modules / librairies externes

- **INDISPENSABLE** `requests` : requêtes HTTP (intéractions avec apis web...)
- `sqlalchemy` : ORM pour bases de données sql (et autres)
- `numpy` : structures (Array) et calculs scientifiques
- `pandas` : structures (DataFrame) et calculs scientifiques
- `matplotlib` : visualisation de données
- `scikit-learn` : machine learning pour python
- `flask` : création d'api HTTP (rest et autres)

Les namespaces

Qu'est ce qu'un namespace

En python, chaque variable, fonctions ou modules utilisables sont enregistrées dans un registre de noms: un *namespace*.

Ainsi, quand une fonction, une variable, ou un module est utilisable dans python, elle est présente dans le *namespace* courant.

Exemple de *namespace*: `['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'math', 'students']`

Les built-ins par exemple sont présents par défaut dans le namespace global de python.

Les namespaces

Namespace global et namespaces locaux

On a deux types de namespaces:

- global (son contenu est accessible partout)
- local (son contenu diffère suivant la position dans le code)

On utilise les fonctions `globals` ou `locals` (ou `dir`) qui liste tout ce qui est dans le namespace global et local:

```
print(globals())
print(locals()) # ou dir()
```

```
['__annotations__', '__builtins__', '__doc__',
'__loader__', '__name__', '__package__', '__spec__']
```

Les namespaces

Namespace global et namespaces locaux (2)

Lorsque l'on importe un module, ou que l'on déclare une variable, python l'ajoute alors au namespace courant (local):

```
from math import pi
print(pi)
print(locals()) # Ici idem que print(globals())
```

Ce qui rend l'accés à une variable ou à une fonction possible:

```
3.141592653589793
[..., 'pi']
```

Si on est à la racine du code, la variable ou module est aussi dans le namespace global.

Les namespaces

Namespaces et fonctions

Une fonction (ou une classe ou tout autre partie ou *block* de code) a son propre namespace local.
Depuis l'intérieur d'une fonction, on peut accéder au namespace global.
Quand on déclare une variable dans une fonction, elle est ajoutée au namespace local de cette dernière.

Une variable déclarée dans la fonction n'est accessible que dans celle ci.

On dit alors qu'elle a une portée (scope) locale.

Les namespaces

La portée d'une variable

On définit la portée (scope) d'une variable comme ceci: toute variable définie dans un *block* (et présent dans son namespace) a comme *portée* ce *block* et tout ce qu'il contient.

Tant que la variable est à portée, elle est accessible.

Une variable déclarée dans la fonction n'est accessible que dans celle ci.

Une variable peut donc avoir 2 portées différentes:

- globale si elle est atteignable de partout.
- locale si elle est atteignable dans une partie (*block*) du programme uniquement.

Les namespaces

La portée d'une variable (2)

Si on essaye d'atteindre une variable qui n'est pas présent ni dans le namespace local courant, ni dans le global:

```
print(pi)
```

Elle n'est donc pas à portée (ou alors non définie...):

```
NameError: name 'pi' is not defined
```

```python
one = 1 # variable globale

print(globals(), locals())

def add_one(number):
    result = one + number # bariable locale de addOne
    print(globals(), locals())
    return result


print(add_one(4))
print(globals(), locals())
print(result) # result n'est pas à portée !
```

```
{..., 'one': 1} {..., 'one': 1}
{..., 'one': 1, 'addOne': <function addOne>}
   {'result': 5, 'number': 4}
5
{..., 'one': 1, 'addOne': <function addOne>}
   {..., 'one': 1, 'addOne': <function addOne>}
NameError: name 'result' is not defined
```

Quand on déclare une fonction dans une fonction, la portée des variables déclarée dans la fonction parente s'étend à l'enfant:

```
def add_one_to_numbers(*numbers):
    one = 1
    def add_one(number): # Fonction closure !
        return one + number # Accés à la variable one
    results = []
    for number in numbers:
        results.append(add_one(number))
    return results
```

`add_one` a accés aux variables du scope global mais aussi à celles déclarées dans `add_one_to_numbers` (qui se retrouvent dans son namespace local).

Modifier une variable globale

Implicitement, on ne peut pas modifier une variable globale dans une fonction:

```
number = 24

def set_number_to_one():
   number = 1 # Celà crée une variable number locale
   print("In the function:", number) # locale

set_number_to_one() # NOPE
print("Outside the function:", number) # globale
```

```
In the function: 1   # variable number de la fonction
Outside the function 24 # variable number globale
```

En effet, si on essaye d'assigner une variable dans une fonction, python part du principe que c'est une nouvelle variable locale.

Modifier une variable globale (2)

Si on veut par exemple incrémenter une variable:

```
number = 24

def add_one_to_number():
    number = number + 1 # python est perdu

add_one_to_number()
```

Python nous fait bien comprendre qu'il ne compte pas utiliser la globale:

```
UnboundLocalError: local variable 'number' referenced
before assignment
```

Modifier une variable globale (3)

On peut quand même le faire en utilisant le mot clef `global` :

```
number = 24

def add_one_to_number():
    global number # On utilise la globale number
    number += 1


add_one_to_number() # MAGIC
print(number)
```

```
25
```

MAIS NE LE FAITES PAS !!!

Modifier une variable globale (4)

Modifier une variable globale comme ceci peut être dangereux:

```
rabbit = "rabbit"

a_function()
another_function()

print(rabbit) # MAGIC
```

On peut se retrouver avec des résultats non attendus et compliqués à retracer:

```
"GORILLA" # WHAAAAAAAAAAT
```

Modifier une variable globale (5)

Préférez expliciter une réassignation (ou même changer le nom de variable). N'utilisez pas `global` et utilisez un `return` :

```
number = 24

def add_one(num):
    return num + 1

number = add_one(number)
print(number)
```

On comprend mieux comment on arrive à un résultat donné:

```
25
```

Les fichiers

La fonction open

Pour lire ou écrire un fichier, on utilise la fonction open (qui ouvre un flux vers ce fichier).

Un flux s'ouvre et se ferme (quand on a terminé avec lui):

```
file = open("myfile.txt")
print(file)
print(file.read())
file.close()
print(file.read())
```

```
<_io.TextIOWrapper name='myfile.txt' mode='r'
                   encoding='UTF-8'>
'HELLO WORLD!'
ValueError: I/O operation on closed file.
```

Appartée - La déclaration with

Si on peut automatiser la fermeture d'un fichier (ou de tout flux) on peut utiliser la déclaration `with` :

```
with open("myfile.txt") as file:
    print(file.read())
```

Le flux se fermera de lui même une fois le bloc de code terminé.

Il est conseillé d'utiliser cette notation plutôt que la méthode `.close`.

Les fichiers

Lire un fichier

On peut lire un fichier en précisant un mode d'ouverture read `r` (mode par défaut):

```
with open("myfile.txt", mode="r") as file:
   print(file.read())
```

Différentes méthodes sont utilisable sur fichier pour le lire:

```
file.read() # Lit tout le fichier
file.readline() # Retourne la première ligne du flux
file.readlines() # Retourne une liste avec chaque ligne
```

Les fichiers

Lire un fichier (2)

On ne peut lire un fichier inexistant:

```
with open("nanananana.txt", mode="r") as file:
    print(file.read())
```

Python nous l'indique par une erreur:

```
FileNotFoundError: [Errno 2] No such file or directory: 'n
```

Lire un fichier (3)

Un flux se consomme (on ne le lit qu'une fois):

```
with open("myotherfile") as file:
    print(file.read())
    print(file.read()) # On a déjà consommé le flux
```

```
"Hello\nMy Name is Guillaume\nNice to meet you"
```

```
with open("myotherfile") as file:
    print(file.readline()) # la premère ligne
    print(file.readline()) # la deuxième ligne
    print(file.readlines()) # on consomme le reste
```

```
"Hello\n"
"My Name is Guillaume\n"
["Nice to meet you"]
```

Les fichiers

Ecrire un fichier

On peut écrire un fichier et écraser son contenu avec le mode overwrite `w` :

```
with open("myotherfile", "w") as file:
   file.write("I overwrite ")
   file.write("what I want !!") # Ecrit à la suite
   file.write("\nWhen I want !!") # Ecrit à la suite


with open("myotherfile", "r") as file:
   print(file.readlines())
```

```
['I overwrite what I want !!\n', 'When I want !!']
```

Les fichiers

Ecrire un fichier (2)

On peut écrire un fichier et compléter son contenu avec le mode append `a` :

```
with open("myotherfile", "a") as file:
   file.write("I append something")
   file.write("\nWhen I want !!") # Ecrit à la suite


with open("myotherfile", "r") as file:
   print(file.readlines())
```

```
['I overwrite what I want !!\n',
'When I want !!I append something\n',
'When I want !!']
```

Les fichiers

Les fichiers binaires

On peut lire ou écrire un fichier en binaire en ajoutant le mode binaire `b` au modes existants:

- `rb`
- `wb`
- `ab`

...

Les fichiers

Ouvrir un csv

On peut utiliser module `csv` et la fonction `open` pour lire ou écrire du csv:

```
import csv
with open('eggs.csv') as file:
    eggs = list(csv.reader(file, delimiter=','))

with open('eggs.csv', 'w') as csvfile:
    spamwriter = csv.writer(csvfile, delimiter=',')
    spamwriter.writerow(['Spam', 'Baked Beans'])
    spamwriter.writerow(['Spam', 'Wonderful Spam'])
```

Les fichiers

Ouvrir un json

On peut utiliser module `json` et la fonction `open` pour lire ou écrire du csv:

```
import json
with open('students.json') as file:
    students = json.load(file)

students_scores = {"jean michel": 19.5,
                   "jean gilles": 10.5}
with open('students_scores.json', 'w') as file:
    json.dump(students_scores, file)
```

Structure d'un projet

Les conditions d'un bon projet

Un projet python (lib ou app) doit avoir:

- Un `README` (**md** or rst) pour expliquer le projet
- Un système de versionnement (**git** ou mercurial ou svn)
- Un `CHANGELOG` (**md** ou rst) pour expliquer les changements
- Un point d'entrée (un fichier .py ou un raccourci système)
- Un code source avec une arborescence clair
- Un fichier contenant les versions des modules externes
- **Idéalement** des tests unitaires
- **Idéalement** une configuration de linter

Structure d'un projet

Structure basique

Rendez vous sur https://git.io/vbY68 dans `projects/simple-project` pour voir un exemple de projet basique.

`README.md` contient la documentation pour installer et lancer le project.
`CHANGELOG.md` contient les différentes versions et modifications.
`simple_project.py` contient le point d'entrée du projet.
`src` contient le code source utilisé par le point d'entrée. Le fichier `__init__.py` permet à python d'importer le contenu du dossier comme des modules.
`requirements.txt` contient la liste des dépendances externes utilisées (et leurs versions).

Structure d'un projet

Structure avancée

On peut créer une structure de projet plus élaborée qui permet d'installer son applicatif via `pip` (et le publier sur le dépôt `pypa`) voir d'y ajouter des test unitaires et un linter (via `tox`).

Un bon exemple est le projet fourni par `pypa` et disponible ici: https://github.com/pypa/sampleproject.

Il peut servir de base à tout projet python (application ou librairie).

Il regroupe toutes les bonnes pratiques.

La suite après le projet...