Makefiles en Python
Version 3.0

Pierre Molinaro

30 mai 2016

Table des matières

1 Versions

Version	Date	Commentaire
1.0	4 mars 2015	Version initiale
2.0	2 octobre 2015	Définition de plusieurs cibles pour une règle (exemple page 14)
2.1	5 octobre 2015	Ajout de la vérification dynamique des types des arguments Ajout de la simulation du but `clean` (page 13)
2.2	24 octobre 2015	Remplacement de `subprocess.Popen` en `subprocess.call` dans `runCommand` Ajout de la vérification de l'existence de l'utilitaire appelé dans les commandes par la fonction `find_executable` (voir la note page 4) Ajout d'un argument booléen optionnel à l'instanciation de la classe `Make` pour afficher le chemin complet de l'utilitaire appelé dans les commandes (voir les notes pages 3 et 9)
2.3	16 avril 2016	Ajout de l'affichage du pourcentage de la progression (voir page 6)
3.0	30 mai 2016	Compatibilité avec Python 3 : le script peut être utilisé aussi bien en Python 2 qu'en Python 3

La version 2.0 apporte une incompatiblité avec les scripts de la version précédente. Le premier argument de `makefile.Rule` doit être une liste. Les lignes modifiées des listings sont complétées par le commentaire « `# Release 2` ».

2 Introduction

Le module `makefile` permet de construire facilement des scripts Python qui ont toutes les facilités des fichiers *makefile* exécutés par l'utilitaire `make`. Les avantages principaux de la classe `makefile` sont :

— la description des règles de dépendances est beaucoup plus simple ;
— les espaces dans un chemin sont pris en compte naturellement ;
— le langage Python offre des possibilités que les makefiles n'ont pas.

Dans la suite, nous allons donner un exemple d'utilisation de cette classe.

3 L'exemple

L'exemple consiste en deux fichiers C et un fichier d'en-tête.
Le fichier `main.c` :

```
#include "myRoutine.h"

int main (int argc, char * argv []) {
  myRoutine () ;
  return 0 ;
}
```

Le fichier `myRoutine.h` :

```
#ifndef MYROUTINE_DEFINED
#define MYROUTINE_DEFINED

void myRoutine (void) ;
```

```
#endif
```

Le fichier `myRoutine.c` :

```
#include "myRoutine.h"
#include <stdio.h>

void myRoutine (void) {
  printf ("Hello world!\n") ;
}
```

La compilation séparée s'effectue par les commandes :

```
mkdir -p objects
gcc -c main.c -o objects/main.c.o
gcc -c myRoutine.c -o objects/myRoutine.c.o
gcc objects/main.c.o objects/myRoutine.c.o -o myRoutine
```

Les fichiers objets sont rangés dans un répertoire `objects`, qu'il faut créer si il n'existe pas.

4 Construction du script Python

Ce script est contenu dans un fichier exécutable `build.py`, situé dans le même répertoire que les fichiers sources C. Sa composition est décrite pas-à-pas.

La première opération est d'importer le module `makefile`, ainsi que les autres modules utiles.

```
import sys, os, makefile
```

Pour pouvoir travailler avec des chemins relatifs, on fixe le répertoire courant, obtenu à partir de l'argument 0.

```
scriptDir = os.path.dirname (os.path.abspath (sys.argv[0]))
os.chdir (scriptDir)
```

Ensuite, on crée un objet `make`, qui est l'objet qui centralise la mise en œuvre de la construction. Un argument chaîne de caractères doit être fourni, il représente le but qui sera construit. La définition précise de ce but est faite plus loin.

```
make = makefile.Make ("all")
```

Note. À partir de la version 2.2, la création de l'objet `make` présente un second argument booléen optionnel. S'il est vrai, l'appel de chaque commande affiche le chemin complet de l'utilitaire appelé. Par défaut, cet argument optionnel est faux.

```
make = makefile.Make ("all", True) # Affiche le chemin de chaque commande appelee
```

Pour présenter une solution générale, on construit deux listes, l'une contenant la liste des fichiers C à compiler, l'autre la liste des fichiers objets à lier. La seconde est initialement vide, elle est construite au fur et à mesure en bouclant sur la liste des fichiers source.

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  ...
```

Dans le corps de la boucle, on construit chaque règle correspondant à la compilation d'un fichier source. On commence par définir le fichier *cible* de la règle, qui est ici le fichier objet construit par la compilation du source C. On en profite pour accumuler dans `objectList` la liste des fichiers objet.

```
  object = "objects/" + source + ".o"
  objectList.append (object)
```

Ensuite, on instancie un objet `rule` en précisant en premier argument la liste des fichiers cibles de la règle. Le second argument est optionnel, il s'agit du titre qui est affiché lors de l'exécution de la règle. Par défaut, en l'absence du second argument, le texte `Building` suivi du nom des cibles est affiché.

```
  rule = makefile.Rule ([object], "Compiling " + source)  # Release 2
```

Attention, la cible (ici « `[object]` ») doit être obligatoirement une liste de fichiers. On ne peut pas mettre un nom tel que `all` ou `clean` ici : ces noms sont des *buts* et leur définition est décrite plus loin dans ce document.

On ajoute ensuite les dépendances. Ici, il n'y a qu'une seule dépendance, le fichier source. La variable `rule.mDependences` est une liste de chaînes de caractères, initialement vide.

```
  rule.mDependences.append (source)
```

Attention, ne pas ajouter ici les dépendances envers les fichiers d'en-tête. On verra comment en tenir compte par la suite.

On construit maintenant la commande qui sera exécutée pour construire la cible. La variable « `rule.mCommand` » est une liste de chaînes de caractères, initialement vide.

```
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
```

Il est impératif de définir une chaîne par argument. Si par exemple on veut ajouter l'option « `-O2` », écrire `rule.mCommand.append ("gcc -O2")` est une erreur, il faut obligatoirement écrire `rule.mCommand += ["gcc", "-O2"]`.

Note. À partir de la version 2.2, l'existence de l'utilitaire appelé est vérifiée. Un message d'erreur est affiché si l'utilitaire n'est par trouvé ; par exemple, si on écrit `rule.mCommand.append ("gccc")`, alors le message d'erreur suivant s'affiche :

```
*** Cannot find 'gccc' executable ***
```

La construction de la règle est terminée ; on peut maintenant l'ajouter au makefile.

```
  make.addRule (rule)
```

Récapitulation. La construction des règles relatives à la compilation des sources C est :

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  objectList.append (object)
```

```
rule = makefile.Rule ([object], "Compiling " + source) # Release 2
rule.mDependences.append (source)
rule.mCommand.append ("gcc")
rule.mCommand += ["-c", source]
rule.mCommand += ["-o", object]
make.addRule (rule)
```

Remarquez que les fichiers objets sont rangés dans le répertoire `objects`. Or celui-ci n'existe peut-être pas, et `gcc` ne le créera pas, en déclenchant une erreur si il n'existe pas. En fait, il est inutile de s'en soucier. En effet, quand une règle doit être exécutée, notre makefile crée implicitement le répertoire du fichier cible, si ce répertoire n'existe pas.

On va maintenant construire la règle relative à l'édition des liens. Le nom du fichier exécutable est défini par la variable `product`, et les dépendances sont les fichiers objets dont la liste est définie par la variable `objectList`.

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product) # Release 2
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
make.addRule (rule)
```

À titre de vérification, il est possible d'afficher les règles.

```
make.printRules ()
```

Les trois règles que nous avons définies s'affichent :

```
--- Print 3 rules ---
Target: "objects/main.c.o"
  Dependence: "main.c"
  Command: "gcc" "-c" "main.c" "-o" "objects/main.c.o"
  Title: "Compiling main.c"
Target: "objects/myRoutine.c.o"
  Dependence: "myRoutine.c"
  Command: "gcc" "-c" "myRoutine.c" "-o" "objects/myRoutine.c.o"
  Title: "Compiling myRoutine.c"
Target: "myRoutine"
  Dependence: "objects/main.c.o"
  Dependence: "objects/myRoutine.c.o"
  Command: "gcc" "objects/main.c.o" "objects/myRoutine.c.o" "-o" "myRoutine"
  Title: "Linking myRoutine"
--- End of print rule ---
```

Nous allons maintenant définir les buts (« *goals* »). Par défaut, il y a un but nommé `clean`, qui n'a par défaut aucun effet (on verra plus loin comment le configurer). Dans le cadre de notre exemple, nous allons définir deux buts :

- `all`, qui commande la construction de l'exécutable (celui qui a été mentionné dans l'instanciation de la classe `Make`) ;
- `compile`, qui commande la compilation des sources, sans l'édition des liens (inutilisé pour l'instant).

```
make.addGoal ("all", [product], "Building all")
make.addGoal ("compile", objectList, "Compile C files")
```

Chaque but est défini par trois arguments :
- son nom, qui sert par la suite pour le désigner ;
- la liste des fichiers qu'il construit ;
- sa description, sous la forme d'une chaîne de caractères.

À titre de vérification, nous pouvons afficher la liste des buts.

```
make.printGoals ()
```

On obtient :

```
--- Print 2 goals ---
Goal: "compile"
  Target: "objects/main.c.o"
  Target: "objects/myRoutine.c.o"
  Message: "Compile C files"
Goal: "all"
  Target: "myRoutine"
  Message: "Building all"
--- End of print goal ---
```

Enfin, nous allons ordonner la construction du but qui a été désigné lors de l'instanciation de la classe `Make`, c'est-à-dire le but `all` (la construction du but `compile` est présenté plus loin).

```
make.runGoal (0, False)
```

Cet appel contient deux arguments :
- une valeur entière qui précise combien de règles peuvent être exécutées en parallèle ;
  - une valeur strictement positive définit le nombre de règles qui peuvent être exécutées en parallèle ; en particulier, la valeur 1 impose une exécution séquentielle des règles ;
  - la valeur 0 indique au makefile de prendre le nombre de processeurs de la machine sur laquelle tourne le script ;
  - la valeur strictement négative indique au makefile de prendre le nombre de processeurs de la machine sur laquelle tourne le script, auquel on ajoute l'opposé de la valeur ; par exemple, -2 sur un quadri-cœur définit l'exécution de $4 - (-2) = 6$ règles en parallèle ;
- une valeur booléenne qui indique si la commande relative à chaque règle doit être affichée ou non ; le titre de chaque règle est toujours affiché.

L'exécution du script affiche (noter la création du répertoire `objects`) :

```
Making "objects" directory
[ 33%] Compiling main.c
[ 66%] Compiling myRoutine.c
[100%] Linking myRoutine
```

Nouveau dans la version 2.3. Le pourcentage qui est affiché représente le pourcentage de commandes réalisées lorsque la commande citée sur la même ligne sera terminée. Si N commandes doivent être executées, pour la i^{e} commande ($1 \leqslant i \leqslant N$) la valeur affichée est $100 \times i/N$. Aussi, la dernière commande est toujours précédée de « `[100%]` ».

Si on ne veut pas afficher cette information, il suffit d'appeller la méthode `doNotShowProgressString` de la classe `Make` avant d'appeler `runGoal` :

```
make.runGoal (0, False)
make.doNotShowProgressString ()
```

Si on relance aussitôt le script, le makefile examine les dépendances et signale qu'il n'y a rien à construire :

```
Nothing to make.
```

Appeler `runGoal` avec le dernier argument à `True` permet d'afficher les commandes qui sont exécutées :

```
Making "objects" directory
mkdir -p objects
[ 33%] Compiling main.c
gcc -c main.c -o objects/main.c.o
[ 66%] Compiling myRoutine.c
gcc -c myRoutine.c -o objects/myRoutine.c.o
[100%] Linking myRoutine
gcc objects/main.c.o objects/myRoutine.c.o -o myRoutine
```

Si une erreur est détectée, par exemple si l'exécution d'une règle échoue par suite d'une erreur de compilation, l'exécution de `runGoal` attend la fin de toutes les exécutions parallèles pour terminer. Pour afficher les erreurs et abandonner l'exécution du script en retournant un code d'erreur, on peut appeler :

```
make.printErrorCountAndExitOnError ()
```

L'appel de cette méthode n'a aucun effet si il n'y a pas d'erreur.

5 Dépendance des fichiers d'en-tête

Dans les écritures précédentes, nous n'avions pas pris en compte les dépendances envers les fichiers d'en-tête pour la compilation des sources C. Dans l'exemple, ces dépendances sont très simples, il n'y a qu'un seul fichier d'en-tête utilisateur : `myRoutine.h`. On pourrait ajouter ce fichier dans la liste de dépendances définie par `rule.mDependences`.

Mais dans un cas réel, cela impose de maintenir soi-même la liste des dépendances de chaque fichier source. Or, en fonction de l'évolution du programme, cette liste peut évoluer : des fichiers d'en-tête peuvent s'avérer inutiles, d'autres peuvent être oubliés. Il est donc beaucoup plus simple et plus sûr de confier cette tâche à GCC et au makefile Python.

Maintenir automatiquement les dépendances envers les fichiers d'en-tête s'effectue en trois étapes.

D'abord nommer le fichier de fichier de dépendance : on choisit le nom du fichier objet, auquel on ajoute l'extension `dep`. Ce fichier sera placé dans le répertoire `objects`.

```
  depObject = object + ".dep"
```

Ensuite, indiquer à GCC de le construire ; on ajoute les options de compilations suivantes :

```
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
```

Enfin, on ajoute à la règle la dépendance en ce fichier (il faut placer l'instance de la classe `Make` en second argument) :

```
  rule.enterSecondaryDependanceFile (depObject, make)
```

Récapitulation. Les règles de compilation des sources C deviennent donc :

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
  rule.enterSecondaryDependanceFile (depObject, make)
  make.addRule (rule)
```

Maintenant, toute modification d'un fichier d'en-tête provoquera la compilation des fichiers sources qui l'incluent directement ou indirectement. C'est une dépendance dite *secondaire* car l'absence du fichier n'empêche pas l'exécution de la règle. A contrario, l'absence du fichier source entraîne une erreur, car il n'y a aucun moyen de le construire.

L'exécution de la méthode `enterSecondaryDependanceFile` peut prendre un temps important, puisque la date de modification de tous les fichiers cités dans le fichier `depObject` est vérifiée. Toutefois, si le but `clean` est invoqué, cette interrogation n'est pas réalisée car inutile : la durée d'exécution de la méthode est alors négligeable.

6 Priorité entre les règles

Par défaut, les règles qui sont prêtes à être exécutées le sont dans leur ordre d'ajout par la méthode `addRule`. Ainsi, dans notre exemple, on a définit la liste des fichiers sources par :

```
sourceList = ["main.c", "myRoutine.c"]
```

Si on exécute les règles séquentiellement (appel de `runGoal` avec le deuxième argument égal à `1`), `main.c` sera toujours compilé avant `myRoutine.c`.

On peut changer cet ordre en affectant une priorité à une règle : il suffit de changer la valeur du champ entier `mPriority` de la règle. Par défaut, ce champ est à zéro. On peut affecter une valeur positive, négative ou nulle, du moment qu'elle est entière. Plus ce champ a une valeur importante, plus la règle est prioritaire. À priorité égale, c'est l'ordre d'insertion des règles (par la méthode `addRule`) qui définit l'ordre : la première insérée sera exécutée d'abord.

Par exemple, on peut demander à compiler les fichiers dans l'ordre décroissant de leur taille. Pour cela on écrira (juste avant l'appel de `addRule`) :

```
  rule.mPriority = os.path.getsize (scriptDir + "/" + source)
```

7 Prise en compte de plusieurs buts

Nous avons défini le but `compile`, mais sans l'avoir appelé jusqu'à présent. Nous allons voir comment nous pouvons écrire un nouveau script Python qui va appeler ce but.

D'abord, nous modifions le début du script `build.py` de la façon suivante :

```
#--- Get goal as first argument
goal = "all"
if len (sys.argv) > 1 :
  goal = sys.argv [1]
#--- Get max parallel jobs
maxParallelJobs = 0 # 0 means use host processor count
if len (sys.argv) > 2 :
  maxParallelJobs = int (sys.argv [2])
#--- Build python makefile
make = makefile.Make (goal, maxParallelJobs == 1)
```

Note. À partir de la version 2.2, `makefile.Make` présente un second argument booléen optionnel. Si il est vrai, l'appel de chaque commande affiche le chemin complet de l'utilitaire appelé. Par défaut, cet argument optionnel est faux. En plaçant `maxParallelJobs == 1` comme second argument, on affiche ce chemin si la construction des buts est réalisée séquentiellement, sans parallélisme.

Ensuite, nous modifions la fin du script `build.py` de la façon suivante :

```
make.runGoal (maxParallelJobs, maxParallelJobs == 1)
```

Ces modifications permettent d'appeler ce script avec différents arguments, ce qui permet de préciser le but et le parallélisme.

Par exemple, le script `compile.py`, qui effectue uniquement la compilation, s'écrit :

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "compile"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

Autre exemple : le script `verbose-build.py` effectue une compilation fichier par fichier, ce qui clarifie l'affichage des erreurs, en effet, la compilation parallèle peut provoquer l'affichage entrelacé des messages d'erreurs, les rendant difficilement compréhensibles.

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-
```

```
import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "all", "1"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

8 Post commandes

Il est possible d'ajouter à une règle des *post commandes* qui sont exécutées les unes après les autres à la suite l'exécution de la commande associée. Un exemple typique est l'application de l'utilitaire `strip` sur l'exécutable produit par le linker.

On ne peut pas écrire une règle ordinaire, car le fichier de dépendance est le même que le fichier cible.

Pour définir une *post commande*, on commence par instancier un nouvel objet de type `PostCommand` :

```
postCommand = makefile.PostCommand ("Stripping " + product)
```

En argument, figure le titre qui sera affiché lors de l'exécution de la post commande.

On ajoute ensuite la commande qui doit être exécutée :

```
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
```

Enfin, on ajoute la *post commande* à la liste des post commandes de la règle :

```
rule.mPostCommands.append (postCommand)
```

Récapitulation. La règle qui définit l'édition de liens devient donc :

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product) # Release 2
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

L'exécution du script affiche maintenant :

```
Making "objects" directory
[ 33%] Compiling main.c
[ 66%] Compiling myRoutine.c
[100%] Linking myRoutine
Stripping myRoutine
```

9 Suppression des cibles en cas d'erreur

En cas d'erreur d'exécution d'une commande (par exemple due à une erreur de compilation), le fichier exécutable, si il existe, n'est pas supprimé : on a alors un exécutable qui ne correspond pas à l'état de compilation du programme.

Il est très simple d'ordonner la suppression de l'exécutable lors d'une erreur d'exécution d'une commande. Il suffit de mettre l'attribut `mDeleteTargetOnError` de la règle qui définit l'édition de liens à `True` :

```
rule.mDeleteTargetOnError = True
```

Cette technique peut être utilisée pour toute cible.

Récapitulation. La règle qui définit l'édition de liens devient donc :

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product)  # Release 2
rule.mDeleteTargetOnError = True
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

10 Le but `clean`

Par défaut, le but `clean` est défini mais n'a aucun effet. Il est facile d'ajouter la suppression de fichiers et de répertoires lorsque le but `clean` est exécuté.

Dans notre exemple, nous voulons que l'exécution du but `clean` supprime :

— le fichier exécutable ;

— le répertoire contenant les fichiers objets engendrés.

Pour supprimer le fichier exécutable lors de l'exécution du but `clean`, il suffit d'ajouter à la règle définissant l'édition des liens :

```
rule.deleteTargetFileOnClean ()
```

Récapitulation. La règle qui définit l'édition de liens devient donc :

```
product = "myRoutine"
rule = makefile.Rule ([product], "Linking " + product)  # Release 2
rule.deleteTargetFileOnClean ()
rule.mDeleteTargetOnError = True
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

Pour supprimer le répertoire `objects` lors de l'exécution du but `clean`, il suffit d'ajouter à la règle définissant la compilation des sources C :

```
  rule.deleteTargetDirectoryOnClean ()
```

Récapitulation. Les règles de compilation des sources C deviennent donc :

```
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.deleteTargetDirectoryOnClean ()
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
  rule.enterSecondaryDependanceFile (depObject, make)
  rule.mPriority = os.path.getsize (scriptDir + "/" + source)
  make.addRule (rule)
```

Il suffit maintenant d'ajouter le script `clean.py`, qui appelle le but `clean` :

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os, subprocess, atexit

def cleanup():
  if childProcess.poll () == None :
    childProcess.kill ()

#--- Register a function for killing subprocess
atexit.register (cleanup)
#--- Get script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
```

```
os.chdir (scriptDir)
#---
childProcess = subprocess.Popen (["python", "build.py", "clean"])
#--- Wait for subprocess termination
if childProcess.poll () == None :
  childProcess.wait ()
if childProcess.returncode != 0 :
  sys.exit (childProcess.returncode)
```

10.1 Simulation du but `clean`

À partir de la version 2.1, il est possible d'afficher sans les exécuter les commandes lancées par le but `clean`. Pour cela, il suffit d'appeler la méthode `simulateClean`.

```
  make.simulateClean ()
```

Cette méthode doit être appelée avant l'appel de `make.runGoal` pour être prise en compte.

11 Ouverture automatique d'un source sur erreur

Il est possible de programmer l'ouverture automatique avec un éditeur externe d'un fichier source quand une erreur se déclenche. Il suffit d'ajouter aux règles concernées la ligne :

```
  rule.mOpenSourceOnError = True
```

Nous allons commencer par provoquer une erreur de compilation en insérant un caractère « §» à la 3e ligne du fichier `myRoutine.h` :

```
#ifndef MYROUTINE_DEFINED
#define MYROUTINE_DEFINED
§
void myRoutine (void) ;

#endif
```

Si on relance la compilation, l'erreur est détectée (sur un multi-cœur, les deux compilations ont lieu en parallèle) :

```
[ 33%] Compiling myRoutine.c
[ 66%] Compiling main.c
In file included from main.c:1:
In file included from myRoutine.c:1:
./myRoutine.h:3:1: error: non-ASCII characters are not allowed outside of literals and iden-
tifiers
§
^
1 error generated.
./myRoutine.h:3:1: error: non-ASCII characters are not allowed outside of literals and iden-
tifiers
§
^
1 error generated.
```

```
Return code: 1
Wait for job termination...
1 error.
```

Quand la propriété `mOpenSourceOnError` est vraie et en cas d'erreur, le makefile recherche dans le flux de sortie des commandes ayant provoqué une erreur un nom de fichier en début de ligne. Plus précisément, on se base sur la chaîne qui précède la première occurrence du caractère « : ». Cette recherche fournit quatre réponses :

— `In file included from main.c`
— `Compiling main.c`
— `./myRoutine.h`
— `./myRoutine.h`

Les deux premières réponses ne correspondent pas à un fichier existant, elles sont ignorées. Les deux suivantes provoquent l'ouverture du fichier `./myRoutine.h` avec un éditeur externe, qui est par défaut :

— sur Mac : `TextEdit` ;
— sur Linux : `gEdit`.

Il est facile de changer l'éditeur de texte par défaut, il suffit d'affecter les propriétés suivantes de l'objet `make` :

— pour Mac : `mMacTextEditor` ;
— pour Linux : `mLinuxTextEditor`.

Par exemple, pour que l'éditeur `TextWrangler` soit utilisé sur Mac :

```
make.mMacTextEditor = "TextWrangler"
```

12 Définition de plusieurs cibles à une règle

Avec la version 2, plusieurs fichiers cibles peuvent être construis par l'exécution d'une règle. À titre d'exemple, nous allons modifier la règle d'édition de liens pour construire un fichier texte « map » qui contient la carte mémoire de l'exécutable. Trois modifications sont apportées dans le listing ci-dessus (notées `note1`, `note2` et `note3`).

```
product = "myRoutine"
mapFile = product + ".map" # note1
rule = makefile.Rule ([product, mapFile], "Linking " + product) # Release 2, note2
rule.mDeleteTargetOnError = True
rule.deleteTargetFileOnClean ()
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
rule.mCommand += ["-Wl,-map," + mapFile] # note3
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
```

note1. Le nouveau fichier cible `mapFile` est défini ici.
note2. Il est ajouté à la liste des cibles construites par la règle.

note3. La commande est complétée pour construire le nouveau fichier cible.

En exécutant le but `all`, les deux fichiers `myRoutine` et `myRoutine.map` sont construits. Si on supprime l'un des deux fichiers, exécuter le but `all` le reconstruit. Si on commente la construction du fichier `myRoutine.map` (ligne commentée par `note3`), exécuter le but `all` affiche une alerte indiquant que le fichier n'a pas été construit.

13 Le script complet `build.py`

```
#! /usr/bin/env python
# -*- coding: UTF-8 -*-

import sys, os
import makefile

#--- Change dir to script absolute path
scriptDir = os.path.dirname (os.path.abspath (sys.argv [0]))
os.chdir (scriptDir)
#--- Get goal as first argument
goal = "all"
if len (sys.argv) > 1 :
  goal = sys.argv [1]
#--- Get max parallel jobs as second argument
maxParallelJobs = 0 # 0 means use host processor count
if len (sys.argv) > 2 :
  maxParallelJobs = int (sys.argv [2])
#--- Build python makefile
make = makefile.Make (goal, maxParallelJobs == 1) # Display executable if sequential build
make.mMacTextEditor = "TextWrangler"
#--- Add C files compile rule
sourceList = ["main.c", "myRoutine.c"]
objectList = []
for source in sourceList:
#--- Add compile rules
  object = "objects/" + source + ".o"
  depObject = object + ".dep"
  objectList.append (object)
  rule = makefile.Rule ([object], "Compiling " + source) # Release 2
  rule.deleteTargetDirectoryOnClean ()
  rule.mDependences.append (source)
  rule.mCommand.append ("gcc")
  rule.mCommand += ["-c", source]
  rule.mCommand += ["-o", object]
  rule.mCommand += ["-MD", "-MP", "-MF", depObject]
  rule.enterSecondaryDependanceFile (depObject, make)
  rule.mPriority = os.path.getsize (scriptDir + "/" + source)
  rule.mOpenSourceOnError = True
  make.addRule (rule)
#--- Add linker rule
product = "myRoutine"
mapFile = product + ".map"
rule = makefile.Rule ([product, mapFile], "Linking " + product) # Release 2
```

```
rule.mDeleteTargetOnError = True
rule.deleteTargetFileOnClean ()
rule.mDependences += objectList
rule.mCommand += ["gcc"]
rule.mCommand += objectList
rule.mCommand += ["-o", product]
rule.mCommand += ["-Wl,-map," + mapFile]
postCommand = makefile.PostCommand ("Stripping " + product)
postCommand.mCommand += ["strip", "-A", "-n", "-r", "-u", product]
rule.mPostCommands.append (postCommand)
make.addRule (rule)
#--- Print rules
# make.printRules ()
# make.writeRuleDependancesInDotFile ("make-deps.dot")
make.checkRules ()
#--- Add goals
make.addGoal ("all", [product, mapFile], "Building all")
make.addGoal ("compile", objectList, "Compile C files")
#make.simulateClean ()
#make.printGoals ()
#make.doNotShowProgressString ()
make.runGoal (maxParallelJobs, maxParallelJobs == 1)
#--- Build Ok ?
make.printErrorCountAndExitOnError ()
```

14 Autres fonctions

14.1 Impression en couleur

Le module `makefile` définit les fonctions suivantes qui permettent d'effectuer des impressions en couleur :

- `BLACK()`, `RED()`, `GREEN()`, `YELLOW()`, `BLUE()`, `MAGENTA()`, `CYAN()`, `WHITE()` ;
- `BOLD()`, `BLINK()`, `UNDERLINE()` ;
- `ENDC()`, qui remet à zéro les attributs d'impression.

Toutes ces fonctions retournent une chaîne de caractères que l'on peut directement utiliser dans un `print` :

```
print makefile.RED () + makefile.BOLD () + "Erreur !" + makefile.ENDC ()
```

14.2 La méthode `enterError`

La méthode `enterError` de la classe `Make` permet d'afficher un message d'erreur et d'incrémenter le compteur d'erreurs maintenu par le récepteur. Elle présente un argument chaîne de caractères, qui est imprimé en rouge et en gras sur le terminal.

```
make.enterError ("il y a une erreur quelque part !")
```

Rappelons qu'un compteur d'erreur non nul inhibe l'exécution des règles lorsque `runGoal` est invoqué.

14.3 La méthode `errorCount`

La méthode `errorCount` de la classe `Make` retourne la valeur courante du compteur d'erreurs maintenu par le récepteur.

```
nombre_erreurs = make.errorCount ()
```

14.4 La méthode `printErrorCount`

La méthode `printErrorCount` de la classe `Make` imprime la valeur courante du compteur d'erreurs maintenu par le récepteur si celle-ci est non nulle. Si elle est nulle, rien n'est imprimé.

```
make.printErrorCount ()
```

14.5 Recherche d'un fichier dans une arborescence

La méthode `searchFileInDirectories` de la classe `Make` recherche un fichier dans une liste de répertoires :

```
chemin = make.searchFileInDirectories (fichier, liste_de_repertoires)
```

Où :

- `fichier` est une chaîne de caractères qui définit le nom du fichier recherché ;
- `liste_de_repertoires` est une liste de chaînes de caractères qui définit la liste des répertoires dans lesquels le fichier sera recherché ;

La fonction retourne :

- si `fichier` est trouvé exactement une fois, le nom du fichier préfixé par le nom du répertoire qui contient le fichier ;
- la chaîne vide si le fichier n'est pas trouvé, ou si il est trouvé plusieurs fois ; un message d'erreur est alors affiché, et le compteur d'erreur maintenu par le récepteur est incrémenté.

Rappelons qu'un compteur d'erreur non nul inhibe l'exécution des règles lorsque `runGoal` est invoqué.