Nose2

August-08-17 1:30 PM

Installation

Il est recommandé d'installer **nose2** via l'utilitaire **pip**. En effet, pip s'assure que toutes les dépendances nécessaires à l'utilisation de nose2 sont satisfaites. Toutefois, il est possible de télécharger la distribution source (pypi) de nose2 ou la version de développement sur github (git://github.com/nose-devs/nose2.git*#egg=nose2*), la décompresser et exécuter la commande : ***python setup.py install***.
Seulement, dans ce cas, il faudra ensuite, si distribute ou setuptools ne sont pas installés, installer manuellement les dépendances de nose2.

Options d'exécution des tests

Option 1

Pour exécuter les tests d'un projet, on peut utiliser le script *nose2*, installé avec nose2 :

```
nose2
```

Ce script recherche et exécute les tests présents dans tous les packages du répertoire courant et de ses sous-répertoires dont le nom commence par *'test'*. Pour trouver les tests, nose2 cherche les modules dont le nom commence par *'test'* puis charge tous les tests qui sont des sous-classes de **unittest.Testcase** ainsi que les fonctions dont le nom commence par *'test'*. Le script nose2 supporte l'ajout d'options via la ligne de commande ainsi que les configurations supplémentaires via les fichiers de configuration. Les options disponibles en ligne de commande sont les suivantes :

```
nose2 -h
```

```
usage: nose2 [-s START_DIR] [-t TOP_LEVEL_DIRECTORY] [--config [CONFIG]]
             [--no-user-config] [--no-plugins] [--verbose] [--quiet] [-B] [-D]
             [--collect-only] [--log-capture] [-P] [-h]
             [testNames [testNames ...]]

positional arguments:
  testNames

optional arguments:
  -s START_DIR, --start-dir START_DIR
                        Directory to start discovery ('.' default)
  -t TOP_LEVEL_DIRECTORY, --top-level-directory TOP_LEVEL_DIRECTORY, --project-directory TOP_LEVEL_DIRE(
                        Top level directory of project (defaults to start dir)
  --config [CONFIG], -c [CONFIG]
                        Config files to load, if they exist. ('unittest.cfg'
                        and 'nose2.cfg' in start directory default)
  --no-user-config      Do not load user config files
  --no-plugins          Do not load any plugins. Warning: nose2 does not do
                        anything if no plugins are loaded
  --verbose, -v
  --quiet
  -h, --help            Show this help message and exit

plugin arguments:
  Command-line arguments added by plugins:

  -B, --output-buffer   Enable output buffer
  -D, --debugger        Enter pdb on test fail or error
  --collect-only        Collect and output test names, do not run any tests
  --log-capture         Enable log capture
  -P, --print-hooks     Print names of hooks in order of execution
```

Les options de ligne de commande permettent par exemple de préciser le répertoire à partir duquel commencer la découverte de tests (start-dir). C'est également dans ce répertoire qu'on recherche les fichiers de configuration nose2.cfg et/ou unittest.cfg.
L'option --config permet de préciser le chemin exact vers le fichier de configuration à utiliser.

Option 2

Une autre option pour exécuter les tests est via la programmation. Par exemple, pour exécuter les tests présents dans un seul module, il suffit de mettre un bloc comme suit à la fin du module:

```
if __name__ == '__main__':
    import nose2
    nose2.main()
```

Ensuite, il suffit d'exécuter *'directement'* le module sans utiliser le script nose2. On peut passer plusieurs types de paramètres à nose2.main(); certains de ces paramètres seront en réalité destinés à unittest.main() :

- module – permet de préciser le module dans lequel exécuter les tests; par défaut il s'agit du **__main__()** dans lequel est appelé nose2.main()
- defaultTest – permet de préciser le nom d'un ou plusieurs tests si aucun nom n'est précisé via argv. S'il est à None, tous les tests du module sont exécutés.
- argv – permet de passer les options de ligne de commande. S'il est à None, par défaut on lui attribue les valeurs de sys.argv.
- testRunner – ce paramètre peut être soit une classe ou une instance de Test Runner. Toutefois, ici il est **IGNORE**; unittest.main() l'initie par défaut à None. Pour créer un test Runner, il faut instancier la classe *nose2.runner.PluggableTestRunner* via l'attribut runnerClass de nose2.main().
- testLoader – ce paramètre doit être une instance de Test Loader. Toutefois, ici il est **IGNORE**; unittest.main() l'initie par défaut à defaultTestLoader. Pour créer un test Loader, il faut instancier la classe *nose2.runner.PluggableTestLoader* via l'attribut loaderClass de nose2.main().
- exit – booléen permettant de décider entre sortir du programme (**True**) ou rester(**False**) après exécution des tests. Pour quitter le programme, on appelle *sys.exit()*.
- verbosity – entier désignant le niveau de verbosité de base utilise lors de l'exécution du programme.
- failfast – ce paramètre permet de décider s'il faut arrêter l'exécution des tests dès le premier échec ou la première erreur. Ici, il es t **IGNORE**; pour éditer cette valeur, il faut la passer en ligne de commande via argv.
- catchbreak – ce paramètre permet d'interrompre l'exécution des tests en terminant le test courant et en reportant les résultats accumulés jusque-là. Cette interruption se fait via un CTRL + C. Ici, il est **IGNORE**; pour éditer cette valeur, il faut la passer en ligne de commande via argv.
- buffer – les flux de sortie standard et d'erreur standard sont mis en cache durant l'exécution des tests. En cas d'échec ou d'erreur d 'exécution des tests, ces flux sont transmis aux sys.stdout et sys.stderr et rajoutés aussi aux messages d'erreur.
- plugins – désigne la liste des modules de plugin supplémentaires à charger (en plus des modules chargés par défaut).
- excludePlugins – désigne la liste des modules de plugin à exclure.
- extraHooks – désigne la liste de points d'entrée et d'instances de plugins à inscrire dans le système de points de démarrage de la session . Chaque élément de la liste doit être un couple(2-tuple) de (point d'entrée, instance de plugin).

Tous ces paramètres à l'exception de **plugins**, **excludePlugins** et **extraHooks** sont en réalité destinés à unittest.main() et édités via la fonction *__init__* de celui-ci.

main (unittest)

```
71
72    def __init__(self, module='__main__', defaultTest=None, argv=None,
73                 testRunner=None, testLoader=loader.defaultTestLoader,
74                 exit=True, verbosity=1, failfast=None, catchbreak=None,
75                 buffer=None):
```

Exemple d'utilisation des paramètres de nose2.main() :

As-tu essayé l'option *--collect-only* ? Ce pourrait être pratique pour lister les tests sans prendre le temps de les exécuter.
Je crois que l'option --collect-only n'a pas encore été implémenté pour nose2 mais je la reconnais de chez Nose, quand je fais nose2 -h en ligne de commande cette option n'existe pas.
D'ailleurs dans la documentation il est précisé que nose2 n'a pas encore tous les plugins de Nose et que ce sera disponible dans les prochains release
Également, il faudrait voir ce que --log-capture fait. Est-ce que le log d'exécution contient plus d'infos si on l'active ?
J'ai utilisé cette option en ligne de commande, en effet, il permet d'afficher plus d'informations. Doit-on l'utiliser dans nos programmes?
Pas nécessairement, mais il faudrait savoir c'est quoi qui est afficher en plus ? STDOUT ? STDERR ? les deux ? Est-ce des messages de nose2 ou bien ce sont des prints ou des logger.info() fait dans les tests comme tels ?

testSingleton

```
130         self.failUnlessEqual(globalVar, 7, "Global variable not at expected value of 7.  Currently at {0}.".format(globalVar))
131         o1.v = 19
132         self.failUnlessEqual(o2.v, 19, "Changing Singleton object with one reference does not reflect on other references.")
133
134
135 if __name__ == '__main__':
136     #print "Return code =", optest.main()
137     cfg_path=os.path.normpath(os.path.join(os.environ['OPTOOLBOX'], '..', 'tests', 'unittest.cfg'))
138     argv =[sys.argv[0], '--config', cfg_path]
139     nose2.main(module='testSingleton', argv=argv)
140
```

On peut prendre plus de contrôle sur l'exécuteur des tests (test runner) en remplaçant le script nose2 précédemment utilisé par un qui nous est propre. Pour cela, il suffit d'écrire un script qui appelle **nose2.discover** :

```
if __name__ == '__main__':
  import nose2
  nose2.discover()
```

nose2.discover prend les mêmes paramètres que nose2.main() mais assigne la valeur None au nom du module pour forcer la découverte de tests.

```
def discover(*args, **kwargs):
    """Main entry point for test discovery.

    Running discover calls :class:`nose2.main.PluggableTestProgram`,
    passing through all arguments and keyword arguments **except module**:
    ``module`` is discarded, to force test discovery.

    """
    kwargs['module'] = None
    return main(*args, **kwargs)
```

Exemple d'utilisation de nose2.discover() :
Dans notre module runTests qui prend en compte tous les tests de tous les packages, nous utilisons nose2.discover() pour la découverte de tests. Nous n'avons pas besoin de lui passer comme paramètre notre plugin de nose2 dénommé Plugnose2 et le plugin junit-xml car ils sont définis dans le fichier de configuration unittest.cfg qui est dans le même répertoire que le module runTests.

runTests

```
4 import nose2
5
6 if __name__ == '__main__':
7     from optoolbox import OpLog
8     OpLog.configureLogging()
9     nose2.discover()
10
```

Configuration

La majorité de la configuration de nose2 se fait via les fichiers de configuration. Ces fichiers sont des fichiers de configuration standard de style .ini avec des sections identifiées par des crochets ("[unittest]") et contenant des paires clé/valeur (key = value). Par défaut, nose2 recherche un fichier nommé unittest.cfg ou nose2.cfg (l'un ou l'autre) dans le répertoire d'exécution afin de charger d'éventuelles configurations.

Exemples:

```
[unittest]
start-dir = tests
code-directories = source
                   more_source
test-file-pattern = *_test.py
test-method-prefix = t
```

```
[unittest]
plugins = myproject.plugins.frobulate
          otherproject.contrib.plugins.derper

exclude-plugins = nose2.plugins.loader.functions
                  nose2.plugins.outcomes
```

- ✓ start-dir
  Cette option configure le répertoire à partir duquel on doit commencer la recherche des tests. La valeur par défaut est "." (pour désigner le répertoire courant dans lequel nose2 est exécuté).
- ✓ code-directories
  Cette option configure nose2 pour ajouter les répertoires cités à sys.path et discovery path(chemin de recherche des tests). Elle peut être utilisée lorsqu'un projet possède des morceaux de code dans d'autres endroits que le répertoire de base de ce projet, ou les répertoir es lib ou src. Les valeurs de cette option peuvent être sous forme de tableaux, dans ce cas mettre chacune sur une ligne différente dans le fich ier de configuration.
- ✓ test-file-pattern
  Cette option permet de configurer la façon(patron, expression régulière) dont nose2 détecte les modules de tests.
- ✓ test-method-prefix
  Cette option permet de configurer la façon dont nose2 détecte les fonctions et méthodes de test. Le préfixe désigné ici va être comparé (à travers une simple comparaison de chaines de caractères) au début des noms de chaque méthode dans les tests et fonctions présents dan s les modules.

Pour spécifier les plugins à charger au-delà des plugins intégrés et automatiquement chargés, il faut ajouter une entrée **plugins** dans la section [unittest] des fichiers de configuration. Pour exclure certains plugins chargés par défaut, il faut ajouter une entrée **exclude-plugins**.

- ✓ plugins
  Liste des plugins à charger. Mettre un module par ligne.
- ✓ exclude-plugins
  Liste des plugins à exclure. Mettre un module par ligne.

On peut également configurer les différents plugins à travers les fichiers de configuration. En effet, la plupart des plugins ont un attribut configSection qui correspond à une section dans les fichiers de configuration; exemple pour un plugin intitulé my-plugin:

```
[my-plugin]
always-on = True
```

Les plugins peuvent prendre toute sorte de valeurs de configuration qui peuvent être des booléens, chaines de caractères, entiers ou tableaux.

Exemple de configuration :
Dans notre cas, nous utilisons un unique fichier de configuration, localisé dans le dossier src et désigné via argv avec l'option de ligne de commande ***--config***.

```
if __name__ == '__main__':
    argv =[sys.argv[0], '--config', 'C:\\Dev\\Nose2\\OpToolbox\\src\\optoolbox\\unittest.cfg']
    nose2.discover(plugins=[('optoolbox.optest.Plugnose2'), ('nose2.plugins.junitxml')], argv=argv)
```

Notre fichier de configuration unittest.cfg est le suivant :

unittest.cfg

```
[unittest]
start-dir = .
plugins = optoolbox.optest.Plugnose2
          nose2.plugins.junitxml
code-directories = comm
                   environment
                   executor
                   git
                   logs
                   matlab
                   opbuild
                   rtlabtools
                   rtxsgtools
                   spreadsheets
                   system
                   utils
[junit-xml]
always-on = True
```

Ici nous précisons :

→ dans la section unittest :
- le répertoire à partir duquel la découverte des tests est entamée, *start-dir*
- les plugins utilisés lors de l'exécution, *plugins*
- les répertoires (contenant des tests) à ajouter dans sys.path et discovery.path (code), *code-directories*

→ dans la section du plugin junit-xml :
- si celui-ci doit être actif ou pas lors de l'exécution de ces tests, *always-on*

Créer des Plugins avec Nose2

Nose2 supporte l'utilisation de plugins dans la collection des tests, leur sélection, leur observation, le report des résultats, etcetera. L'API des plugins de nose2 est basé sur l'API des plugins de unittest2(actuellement en développement). Elle diffère de celle de nose.

Il y a 2 règles de base concernant les plugins :

- Les classes de Plugin doivent surclasser *nose2.events.Plugin*.
- Les Plugin peuvent implémenter toute méthode définie dans la référence de Hook.

Pour que nose2 retrouve un plugin, il faut que celui-ci soit défini dans un module importable, lequel est listé sous la section unittest dans un fichier de configuration :

```
[unittest]
plugins = mypackage.someplugin
          otherpackage.thatplugin
          thirdpackage.plugins.metoo
```

Par la suite, pour activer ces plugins préalablement chargés, on peut ajouter dans le fichier de configuration une section portant le nom de la section de configuration (configSection) en précisant always-on = True. On peut également éditer l'attribut du même nom (alwaysOn = True) dans le module où est défini le plugin.

Dans notre cas, pour le plugin TestLoader défini dans le module Plugnose2, nous avons édité l'attribut alwaysOn:

Plugnose2

```
class TestLoader(events.Plugin):

    """Loader plugin that loads test functions"""
    alwaysOn = True
    configSection = 'loadtestplug'


    # This unittest TestLoader parameter is not used. We have our own sorting mechanism.
    sortTestMethodsUsing = None
```

Ici, le nom de la section de configuration dans le fichier unittest.cfg ou nose2.cfg serait "loadtestplug" suivi des différentes valeurs de configuration désirées. On peut également éditer l'attribut commandLineSwitch afin d'ajouter automatiquement une option de ligne de commande permettant de désigner le plugin; par exemple pour le plugin OutputBufferPlugin, son utilisation en ligne de commande est définie via l'attribut commandLineSwitch comme suit :

```
class OutputBufferPlugin(events.Plugin):                                    [docs]

    """Buffer output during test execution"""
    commandLineSwitch = ('B', 'output-buffer', 'Enable output buffer')
    configSection = 'output-buffer'
```

Les events, hooks et sessions

Pour passer des arguments à ses différents plugins, nose2 utilise un système d'events et hooks (évènements et gestionnaires d'évènements). Il existe différents types d'events qui traduisent l'occurrence de différentes actions. La liste des évènements est donnée dans la référence d'évènements (Event reference), nous en citons quelques ici :

> CommandLineArgsEvent est un évènement déclenché après analyse des arguments de ligne de commande; peut être utilisé par les plugins pour modifier les dits arguments.

> CreateTestsEvent est déclenché avant le chargement des tests; les plugins qui gèrent le chargement de tests peuvent répondre à cet évènement e n retournant une suite de tests et en éditant l'attribut **handled** de l'event à **True** (event.handled = True)

> GetTestCaseNamesEvent qui prend en paramètre un loader, un testCase, la fonction isTestMethod et essaye de retrouver les noms de tests présents dans le testCase en question. Les plugins peuvent le taguer comme **handled** et retourner une liste de noms de tests.

> LoadFromModuleEvent est déclenché lorsqu'un module de test est chargé, prend en paramètre un loader et le module a partir duquel on doit charger des tests. Il dispose de l'attribut extratests permettant de charger les tests du module sans interférer avec les autres plugins qui répondraient au même évènement à savoir LoadFromModuleEvent.

> LoadFromNameEvent et LoadFromNamesEvent sont déclenchés pour charger des tests à partir de noms de tests. Le premier prend en paramètre un seul nom, le second en prend plusieurs.

> CreatedTestSuiteEvent est déclenché après le chargement des tests et PluginsLoadedEvent après que toutes les classes de plugin soient chargées.

> RunnerCreatedEvent, déclenché lorsque l'exécuteur (runner) de tests est créé, StartTestRunEvent, déclenché lorsqu'un test s'apprête à s'exécuter.

> StartTestEvent et StopTestEvent déclenchés respectivement avant et après l'exécution d'un test, TestOutComeEvent déclenché lorsqu'un test achève son exécution.

> ResultCreatedEvent est déclenché lorsqu'un gestionnaire de résultats est créé, ResultSuccessEvent indique un succès à la fin d'un test.

> ReportTestEvent est déclenché pour rapporter l'exécution d'un test et ReportSummaryEvent, déclenché avant et après le résumé des résultats de tests.

Le programme principal (main) de nose2 (PluggableTestProgram) qui surclasse unittest.TestProgram s'occupe de créer les **premiers** events opportuns au début de l'exécution du code. Ainsi l'évènement CommandLineArgsEvent sera créé en premier pour gérer les arguments de la ligne de commande, suivi de l'évènement CreateTestsEvents, LoadFromNamesEvent pour charger les tests, etcetera. Tous les évènements ne sont pas créés dans ce programme principal, pour plusieurs d'entre eux le soin est laissé aux autres composants(programmes) de nose2 tels que loader, runner, session, result, suite, etcetera. Après chaque création d'event, on recherche les hooks qui prennent en charge ce type d'event; pour cela on utilise les sessions.

Dans Nose2, toute la configuration qui régit l'exécution d'un test est encapsulé dans une instance de la classe Session. Les plugins ont toujours accès à l'objet session via **self.session**. Une session compte les attributs de configuration suivants:

◇ **argparse**: peut être utilisé par les plugins pour ajouter des arguments et groupes d'arguments.
◇ **pluginargs**: argument de argparse dans lequel les plugins placent par défaut leurs arguments de ligne de commande.
◇ **hooks**: instance de nose2.events.PluginInterface contenant toutes les méthodes et hooks de plugins disponibles.
◇ **plugins**: liste des plugins chargés mais pas forcément actif.
◇ **verbosity**: niveau de verbosité.
◇ **startDir**: répertoire à partir duquel démarre la recherche(découverte) de tests, par défaut répertoire courant.
◇ **toplevelDir**: répertoire racine de l'exécution des tests, par défaut startDir.
◇ **libDirs**: noms des répertoires de code relatif au répertoire startDir, par défaut lib et src.
◇ **testFilepattern**: patron à utiliser pour la recherche des modules de tests, par défaut *.py pour tous les fichiers python.
◇ **testMethodPrefix**: préfixe à utiliser pour la recherche des méthodes et fonctions de tests, par défaut test.
◇ **unittest**: section de configuration pour nose2.
◇ **configClass**: alias de Config.
◇ **get(section)**: pour récupérer une section de configuration.

- **isPluginLoaded(pluginName)**: retourne True si le plugin donné en paramètre est chargé.
- **loadConfigFiles(*filenames)**: charge les fichiers de configuration donnés en paramètre.
- **loadPlugins(modules=None, exclude=None)**: charge les plugins.
- **loadPluginsFromModule(module)**: charge les plugins à partir d'un module spécifique.
- **prepareSysPath()**: ajoute les répertoires de code à sys.path.
- **registerPlugin(plugin)**: enregistre(inscrit) un plugin.

Via l'attribut hooks de session, on recherche les différents plugins qui ont implémenté des hooks répondant à l'event créé. Par exemple, pour l'event CreateTestsEvent, dans PluggableTestProgram, on a :

main (nose2)

```
247
248     def createTests(self):
249         """Create top-level test suite"""
250         event = events.CreateTestsEvent(
251             self.testLoader, self.testNames, self.module)
252         result = self.session.hooks.createTests(event)
253         if event.handled:
254             test = result
255         else:
256             log.debug("Create tests from %s/%s", self.testNames, self.module)
257             test = self.testLoader.loadTestsFromNames(
258                 self.testNames, self.module)
```

Ici, après avoir créé l'évènement, on appelle **self.session.hooks.createTests(event)** afin de retrouver les différents plugins qui ont implémenté des gestionnaires d'évènements(hooks) pour cet évènement(event) spécifique. Si aucun plugin ne gère ce type d'évènement, aucune action n'a lieu, dans le cas contraire, le hook en question est déclenché et exécute les instructions prévues à la suite d'un tel event.

Le principe de base des évènements (events) est que leurs attributs sont accessibles en lecture et écriture. Sauf déclaration contraire dans la documentation d'un point d'entrée (hook), on peut modifier la valeur de tout attribut d'évènement (event). Les plugins et autres objets de nose2 verront cette nouvelle valeur et pourront l'utiliser soit pour remplacer des valeurs qui étaient initialement édités dans l'event (exemple : le flux de report de résultats ou exécuteur de test), soit pour rajouter quelque chose qu'ils ont trouvé ailleurs(exemple: les extraTests dans un évènement de test loading).

Les hooks offrent aux plugins la possibilité d'avoir une gestion exclusive d'un évènement, contournant les autres plugins qui pourraient vouloir répondre au même évènement. Pour ce faire, il suffit au plugin d'attribuer la valeur **True** à **event.handled** et généralement, retourner une valeur appropriée à partir de la méthode hook. La valeur appropriée varie selon le hook(par exemple il peut s'agir d'une suite de tests comme pour loadTestsFromModule et loadTestsFromNames, etcetera) et certains hooks ne peuvent être gérés de cette façon. Mais même pour les hooks dans lesquels le changement à **True** de l'attribut **handled** d'event n'arrête pas tous les traitements, cela va empêcher les plugins chargés par la suite de voir l'évènement.

Ce pourrait être dangereux de "bloquer" des plugins ainsi. Il faut faire attention au choix des plugins...

Illustration via notre Plugin, la méthode loadTestsFromNames:

Plugnose2

```
185
186     def loadTestsFromNames(self, event):
187         """Discover tests if no test names specified"""
188         log.debug("Received event %s", event)
189         if event.names or event.module:
190             return
191         event.handled = True  # I will handle discovery
192         return self._discover(event)
```

Ici, en mettant **event.handled** à **True**, les autres plugins qui implémentent des gestionnaires pour un tel évènement ne seront pas utilisés et seul notre gestionnaire pour le chargement des tests sera utilisé avec les fonctions souhaitées, ici **_discover(event)**.

Références

http://nose2.readthedocs.io/en/latest/index.html
https://github.com/nose-devs/nose2