Sorbonne Université
Master ANDROIDE, 2020-2021

Projet ANDROIDE

Benchtrack

Auteurs :
Alexis LENOIR
Xin HE
Zhuangzhuang YANG

Encadrant :
Pierre-Henri WUILLEMIN

Table des matières

1 Introduction

1.1 Présentation du projet

L'origine de ce projet part du constat qu'il est difficile de comparer des implémentations différentes d'algorithmes avancés. En effet, celles-ci dépendent de nombreux facteurs comme une grande diversité dans les formats d'entrée et de sortie, des conditions d'utilisation et plus simplement du langage de programmation utilisé.

Il existe déjà plusieurs frameworks qui s'occupe de gèrer des tests sur des implémentations, comme le module *pytest* et la solution logicielle *Jenkins*. Cependant ils s'attachent à tester pour s'assurer de la validité du code, ils interviennent dans son élaboration, l'objectif du projet se place en aval. En effet, le but du projet n'est pas de développer des implémentations, mais en comparer des préexistantes. Le maître mot est donc comparaison.

L'objectif du projet est donc d'élaborer un framework, le plus générique possible, de comparaison d'implémentation d'algorithmes complexes qui permet d'observer et de comparer les perfomances des différentes implémentations sur différents critères. Le framework génére ainsi un *benchmark*, c'est-à-dire un banc d'essai qui permet d'évaluer la pertinence d'une implémentation à l'aide de comparaisons. Ce benchmark est ici un site statique qui contient tous les résultats des comparaisons.

1.2 Les problématiques

Le problème est formalisé de la manière suivante. Le but est de générer un *benchmark* pour une *infrastructure de benchmark*. Cette infrastructure est caractérisée par un ensemble d'implémentations d'algorithmes, que l'on appelera *target*. Chaque *target* résolvera au moins une tâche (*task*). Les tasks sont eux-même regroupés par thème (*theme*). Le regroupement des *targets* est assez structuré mais il sera toujours possible d'indiquer un nom par défaut. De ce formalisme découle naturelement une arborescence *infrastructure_de_test/theme/task*. Une problématique majeure est le formalisme adopté : il doit être clair et efficace, tout en restant le plus générique possible.

L'intérêt de ce benchmark dépend fortement des critères pris en compte. Le plus trivial est la vitesse d'exécution. Cependant, il n'est pas chose aisée de la calcule (précision, extraction des temps de chargement ect).

Afin d'optimiser le développement, sera adoptée une approche modulaire : le projet est séparé en deux. La première partie consiste à générer les résultats dans un fichier *output_date.csv*. La deuxième partie produit le site benchmark à l'aide de ce *output_date.csv*. Le projet est développé en python.

2 La génération des résultats

2.1 Introduction

Cette partie est la première partie de l'ensemble du framework. Cette partie comprend la présentation de la structure, la gestion des flags pour intéragir avec le framework,la gestion des chemins, la gestion des fichiers de configuration, le calcul des résultats et l'empaquetage de l'ensemble du framework. Le programme calcule le temps d'exécution moyen de l'algorithme en fonction de l'algorithme (target), de la tâche et des paramètres donnés par l'utilisateur, et stockera le résultat dans un fichier csv.

2.2 La structure

Pour programmer plus intuitivement, dans cette partie, on choisit une structure de données orientée objet, cette structure est en ligne avec nos besoins, car nos données contiennent des targets, éventuellement des themes, et des tasks, on choisit donc d'utiliser une structure de données orientée objet. En accédant aux targets créées par l'utilisateur selon le format, un type de variable est construit comme un objet de Benchtrack. De plus, la classe Benchtrack peut exécuter et enregistrer le temps d'exécution des targets, elle peut également personnaliser le contenu de l'exécution ou afficher les informations pertinentes sur les targets en ajoutant des indicateurs (flags). Benchtrack est la plus grande structure du programme, elle contient les tasks de tous les themes. Nous créons un objet de type theme pour chaque theme, et enregistrons les tasks dans cette objet de la même manière, puis enregistrons chaque plus petit objet plus petit target dans une task. Chaque fois que le programme est exécuté, il instanciera un objet benchtrack correspondant, et il s'assurera en premier lieu que tout est conforme aux standards du framework. Une fois la génération terminée, le programme appellera les fonctions associées aux indicateurs (flags) demandés.

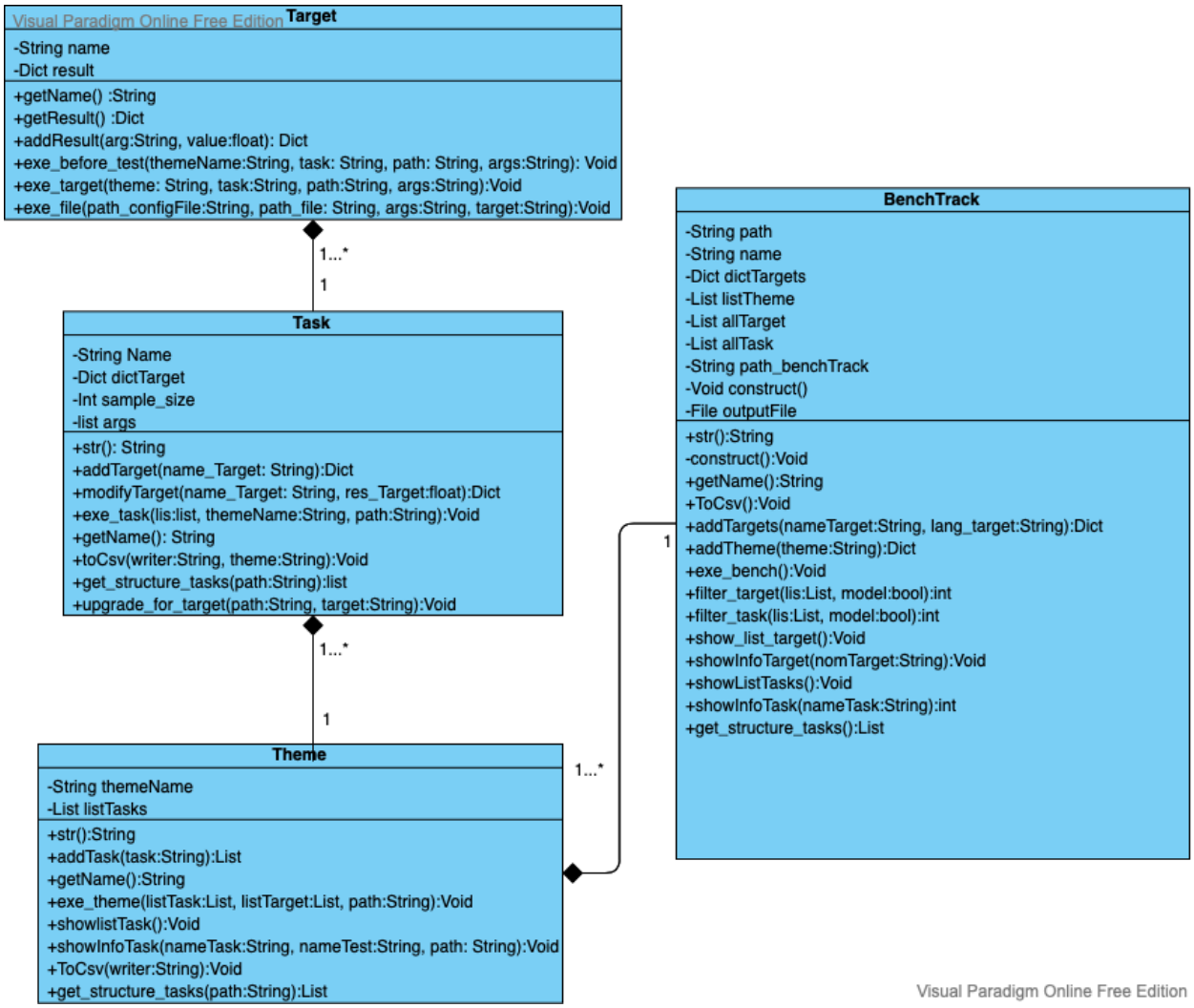
Afin de montrer plus clairement la structure de la programmation de cette partie, nous avons créé un diagramme UML. Comme indiqué ci-dessous, il s'agit de la structure de cette partie.

1.Benchtrack : Benchtrack en tant que plus grande structure enregistre toutes les targets et tous les themes

2. target : Objet qui contient la structure d'une cible(target)

3. theme : enregistre la structure d'un theme

4. task : enregistre la structure d'une task


Visual Paradigm Online Free Edition
Target
-String name
-Dict result
+getName() :String
+getResult() :Dict
+addResult(arg:String, value:float): Dict
+exe_before_test(themeName:String, task: String, path: String, args:String): Void
+exe_target(theme: String, task:String, path:String, args:String):Void
+exe_file(path_configFile:String, path_file: String, args:String, target:String):Void
1...*
1
Task
-String Name
-Dict dictTarget
-Int sample_size
-list args
+str(): String
+addTarget(name_Target: String):Dict
+modifyTarget(name_Target: String, res_Target:float):Dict
+exe_task(lis:list, themeName:String, path:String):Void
+getName(): String
+toCsv(writer:String, theme:String):Void
+get_structure_tasks(path:String):list
+upgrade_for_target(path:String, target:String):Void
1...*
1
Theme
-String themeName
-List listTasks
+str():String
+addTask(task:String):List
+getName():String
+exe_theme(listTask:List, listTarget:List, path:String):Void
+showlistTask():Void
+showInfoTask(nameTask:String, nameTest:String, path: String):Void
+ToCsv(writer:String):Void
+get_structure_tasks(path:String):List
1...*
1
BenchTrack
-String path
-String name
-Dict dictTargets
-List listTheme
-List allTarget
-List allTask
-String path_benchTrack
-Void construct()
-File outputFile
+str():String
-construct():Void
+getName():String
+ToCsv():Void
+addTargets(nameTarget:String, lang_target:String):Dict
+addTheme(theme:String):Dict
+exe_bench():Void
+filter_target(lis:List, model:bool):int
+filter_task(lis:List, model:bool):int
+show_list_target():Void
+showInfoTarget(nomTarget:String):Void
+showListTasks():Void
+showInfoTask(nameTask:String):int
+get_structure_tasks():List
Visual Paradigm Online Free Edition

2.3 Le format de l'infrastructure

Pour utiliser notre framework, les utilisateurs doivent construire l'infrastructure selon un certain format :

```
/nom_infrastructure_benchmark
    /README.rst
    /targets
        /target1
            /README.rst
            /config.ini
        /target2
            /README.rst
            /config.ini
    /tasks
        /theme1
            /task1
                /README.rst
                /config.ini
                /data
                /target1.py
                /target2.r
            /task2
                /README.rst
                /config.ini
                /data
                    data_task2.csv
                /target2.r
        /theme2
            /task3
                /README.rst
                /config.ini
                /data
                /target1.py
                /target2.r
```

Une *infrastructure de benchmark* est un répertoire. On trouve un fichier *README.rst* qui présente l'infrastructure et deux sous-répertoires : *targets* et *tasks*.

Dans le répertoire *targets*, on trouve des sous-répertoires pour chaque *target*. Ils contiennent tous un *README.rst* les présentant et un fichier de configuration *config.ini*.

Celui de *tasks* contient les repertoires *themes*, qui eux-même contiendront une répertoire pour chaque *task* de ce theme. Chaque *target* qui résout une *task*, a son son fichier code présent dans ce sous-répertoire. On trouve aussi de même dans ce dernier, un fichier de présentation *README.rst*, un fichier de configuration *config.ini* mais également un répertoire *data* qui contient les données éventuelles d'entrée des targets.

2.4 La gestion des flags

En gérant les indicateurs (flags), les utilisateurs peuvent avoir plus de choix pour effectuer des tests. Par exemple, des utilisateurs peuvent afficher la liste des targets et des tasks, plus d'information à propos des targets et des tasks. Ils peuvent également utiliser l'exclusion (exclude) et l'inclusion (include) pour gérer les targets inclus dans la task. Voici quelques exemples :
–targets-list :Montrer la liste des targets
–tasks-list :Montrer la liste des tasks
–target-info TOTO :Montrer le fichier readme.rst de target
–task-info A :Montrer le fichier readme.rst de task
–targets-include TOTO,TITI :Exécuter que les targets que l'on choisis
–tasks-include A,B,C :Exécuter que les tasks que l'on choisis
–targets-exclude TOTO,TITI :Exécuter sauf les targets que l'on choisis
–tasks-exclude A,B,C :Exécuter sauf les tasks que l'on choisis
–check :Verifier le format de l'infrastructure

2.5 Le fichier tools.py

? Toutes les autres fonctions sont stockées dans le fichier tools.py, telles que la gestion des fichiers de configuration, la gestion des chemins, la lecture des fichiers et la gestion des indicateurs, et d'autres fichiers appellent les fonctions requises via import tools.py Pour l'exécution équation cible(target) exécutée, au début, nous avons utilisé le chemin absolu pour exécuter la cible(target), mais cela peut provoquer des erreurs dans différents systèmes d'exploitation de bureau, nous choisissons donc d'entrer d'abord dans le répertoire des tâches(task), puis d'exécuter la cible(target).

2.6 Les fichiers de configuration

Dans le choix du format pour les fichiers de configuration, nous avons comparé les types ini, xml et json, et finalement nous avons choisi ini, qui est plus facile à comprendre et pratique à écrire pour les utilisateurs. La faiblesse du fichier de configuration ini est qu'il ne peut pas gérer des structures de données complexes, mais dans ce cadre, il n'est pas nécessaire d'utiliser des structures de données complexes, donc utiliser le fichier de configuration ini est un bon choix. Ce qui suit est la structure et l'explication des deux fichiers de configuration ini que nous utiliserons :
1.Comme le montre la figure ci-dessous, dans la section exécution :
'langage' :le langage de programmation du code.
'run' :la commande pour l'exécution.

```
[execution]
language = python
run =python {script} {arg}
```

2. Fichier de configuration de la tâche : comme le montre la figure ci-dessous, dans la section en cours d'exécution :

‘sample_size’ :le nombre de fois que la tâche doit être exécutée.
‘args’ :le paramètre de la cible à exécuter, où * est le produit cartésien du paramètre à exécuter.
‘display’ :le format pour montrer le résultat sur web.

```
[running]
sample_size=5
args=(10,20,30,50,70,100) * (1:5:2)
display_mode=line
```

2.7 Unittest

Pour gagner en rigueur et en efficacité dans le développement de notre programme, nous utilisons le framework *unittest*. Cela nous permet d’écrire un programme de test, afin de nous assurer de l’intégrité de notre code chaque fois que nous ajoutons ou modifions quelque chose, et que cela n’affectera pas aussi les fonctions précédemment terminées.

2.8 Packaging

Dans le but de vouloir rendre plus accessible et plus professionnel notre projet, nous l’inscrivons dans un package qui sera téléchargable sur pypi. Nous avons à cette intention organiser les fichiers du projet dans le format suivant :

.code/
. src/
. BenchTrack/
. ___init___.py
. tous les autre fichier en python pour BenchTrack
. site/
. ___init___.py
. tous les autre fichier en python et les ressources
. benchTrack.py
. test/
. testBenchTrack.py
. LICENSE.md
. pyproject.toml
. setup.py

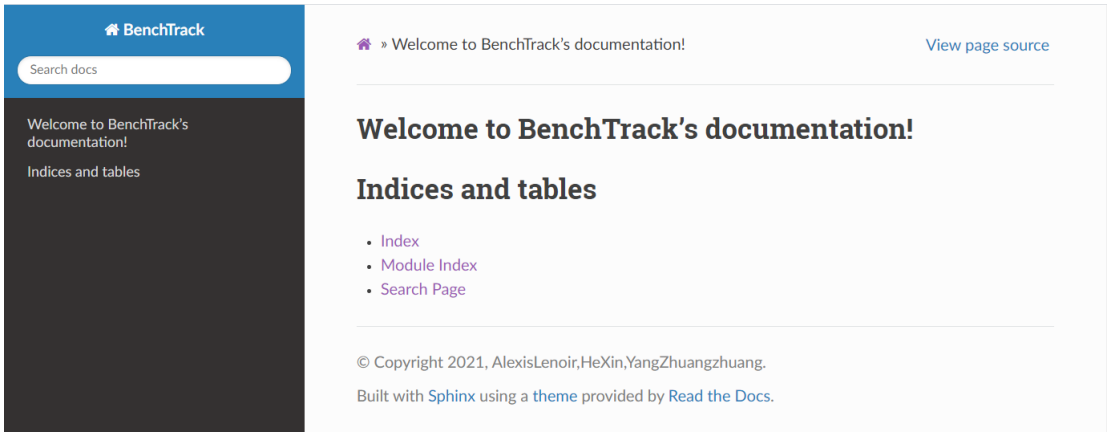
2.9 Stocker les résultats

Nous sauvegardons toutes les données des résultats d’exécution, à l’aide du module csv dans un fichier *output_date.csv*, C’est un fichier intermédiaire qui est utilisé pour générer le site statique.

	A	B	C	D	E
1	theme	task	target	args	run_time
2	inference	randomBN	pyAgrum	10 1	0.237978
3	inference	randomBN	pyAgrum	20 1	0.252804
4	inference	randomBN	pyAgrum	30 1	0.250955
5	inference	randomBN	pyAgrum	50 1	0.289621
6	inference	randomBN	pyAgrum	70 1	0.290224
7	inference	randomBN	pyAgrum	100 1	0.323798
8	Miscellena	BIFreading	pyAgrum	asia.bif 1	0.237922
9	Miscellena	BIFreading	pyAgrum	alarm.bif 2	0.308393
10	Miscellena	BIFreading	pyAgrum	Mildew.bif 3	18.31133
11	Miscellena	BIFreading	pyAgrum	Diabetes.bif 4	16.7662

2.10 Documentation automatique avec Sphinx

Pour chaque framework mature, il existe un document utilisé pour expliquer le contenu du framework, afin que les utilisateurs puissent utiliser ce framework plus commodément, et notre framework ne fait pas exception. Lorsque nous écrivons du code, nous écrivons des commentaires au format requis par sphinx, ce qui nous permet d'utiliser sphinx pour générer automatiquement un document d'explication. Le document peut être un web, fichier en latex ou un pdf.



3 La présentation du benchmark avec un site statique

Nous étudierons dans cette partie tout ce qui concerne l'affichage des résultats à l'aide d'un site statique. Ce site est conçu avec le générateur de site statique python pelican. Nous allons voir dans un premier temps la production du site avant d'aborder dans un second temps, la présentation du site.

3.1 La génération du site

Pour générer le site, nous devons passer par un certain nombre d'étapes, comme par exemple de la récupération des résultats d'éxécution à l'instauration de notre propre thème pelican, qui correpond au template de notre site. Nous commençons par récupérer certaines informations à l'aide de l'objet *Benchtrack* instancié dans la partie précédente.

Le chargement du fichier résultat

La première chose que l'on doit faire pour générer le site c'est de charger les résultats d'exécution. Ils sont contenus dans un fichier csv de la forme *output_date.csv*. Le format des données est `theme,task,target,args,run_time`. Nous sauvegardons ces données dans un dictionnaire à trois dimensions *structure_run_time* (un dictionnaire de dictionnaires de dictionnaires). Les clés de la première dimension sont l'ensemble des tasks, puis l'ensemble des targets pour la deuxième dimension et enfin pour la dernière ce sont les arguments de chaque target : *structure_run_time[task][target][arg] = run_time*.

La génération du répertoire content

Le générateur pelican produit un site statique en transformant des fichiers rst en fichiers html à l'aide d'un template (un thème) et de fichiers de configuration. Plus précisément, pelican utilise la solution docutils qui parse un fichier rst en noeuds, ce qui lui permet de le convertir ensuite en html. Pelican récupère ces fichiers rst qui sont le contenu du site dans un répertoire nommé *content*.

Nous générons des fichiers rst pour présenter l'infrastructure, les tasks, les targets et les targets par task. Pour cela, nous partons des fichiers readme disponibles que nous complétons avec des métadonnées, les résultats des exécutions et parfois le code source. Toutes les fonctions associées se trouvent dans le fichier *generateRst.py*.

Nous nous attendons à trouver un titre dans les fichiers rst readme, mais si cela n'est pas le cas, nous le générons automatiquement.

La personnalisation d'un thème pelican

Comme nous l'avons vu précédement le template du site statique s'appelle pour pelican un thème. C'est un répertoire qui contient deux principaux sous-répertoires. Le premier *static* contient les fichiers css qui ne changent pas selon l'exécution. Le second *templates* contient des fichiers html modèles, qui sont paramétrés avec le moteur de template Jinja. Ces fichiers sont donc complétés avec les données de *content* pour produire le site, qu'on récupère à l'aide de variables d'environnement avec Jinja.

Nous nous sommes basés sur le thème *flex*, nous l'avons modifiés pour l'adapter à nos

besoin : modications des fichiers html modèles et css.

Les fichiers de configuration pelican

Il est indispensable de paramétrer le fichier de configuration python *pelicanconf.py* pour engendrer le site. On y indique le nom du site, l'auteur, l'url, le theme et d'autres paramètres. Nous l'utilisons aussi pour savoir si le site aura un logo et un favicon. Il existe un autre fichier *publishconf.py* pour simplifier la publication du site.

L'exportation du site

On peut exporter le site de la forme *nomInfra_site* où on veut. On entend par là un répertoire qui contient seulement les fichiers html et css. Par défaut, il se trouve dans le répertoire courant. Le répertoire *nomInfra_pelican* pour produire le site avec le thème, le répertoire content et les fichiers de configurations sera lui par défaut un répertoire temporaire qui sera détruit à la fin de l'exécution, sauf si l'utilisateur précise qu'il veut le récupérer. Si on veut lancer le site sur un serveur local, il suffit de taper la commande *pelican –listen* dans le répertoire *nomInfra_pelican*.

3.2 L'apparence du site

Nous allons voir maintenant à quoi ressemble le site.

La structure de base

La structure du site se compose en trois parties. Une barre latérale (sidebar) pour pouvoir accèder rapidement aux principales pages. Elle contient un lien vers la page d'accueil, une liste des targets et une liste des tasks. On trouve ensuite le corps de la page ainsi qu'un bas de page (footer) sommaire. La navigation du site repose aussi sur les liens depuis les tableaux.

Les différents types de page

La page d'accueil est une page présentant l'infrastructure à partir du readme fourni. Elle contient aussi un tableau à deux entrées task X target, chaque case contient un rectangle dont la couleur indique si l'exécution s'est bien déroulée. Donc pour chaque target d'une task, on affiche un rectangle vert si toutes les exécutions se déroulées sans encombre, un rouge s'il y a une erreur et un orange s'il n'y a pas de résultats connus.

Nous avions voulu dans en premier lieu, changer la couleur des cases du tableau avec css à la place d'afficher des rectangles colorés. Mais cela s'est avéré plus complexe que prévu, car il est difficile d'acéder au contenu des cases du tableau rst depuis le template, les variables d'environnement étant assez limités. Une autre solution aurait été de travailler directement sur le fichier html généré et de modifier des balises pour changer l'affichage des cases avec css.

Le rectangle est aussi un lien vers la page task X target associée.

La page target contient un texte de présentation provenant de son readme, puis la présentation des résultats par task et éventuellement par argument. De même la page task est très similaire, on trouve un texte de présentation puis les résultats par target. Dans les résultats de chacunes des deux pages on trouve des liens vers les pages task X target.

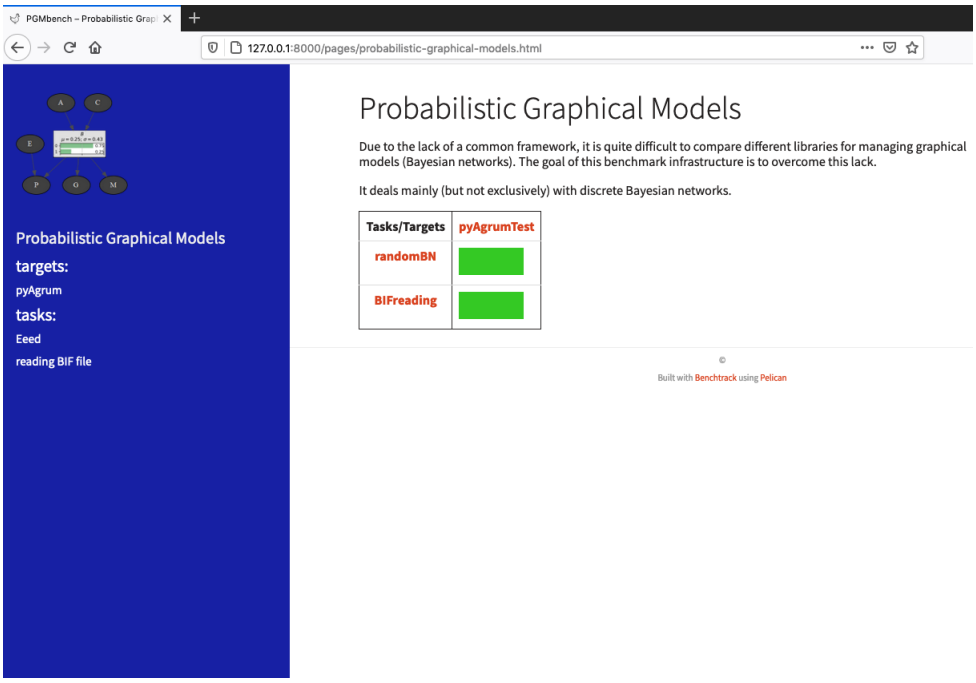


FIGURE 1 – Page d'accueil du site

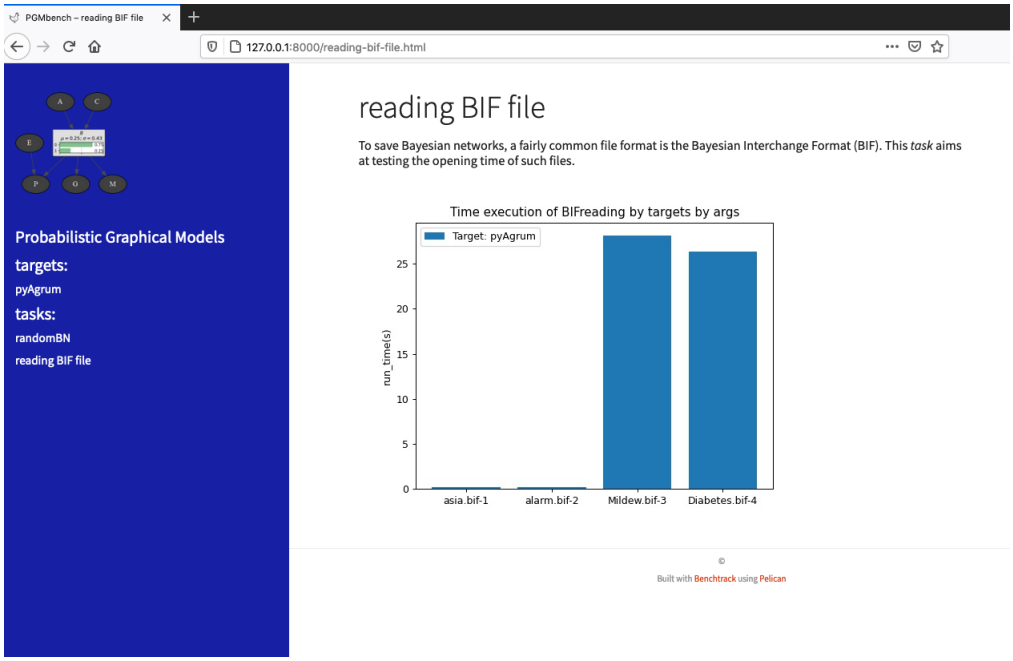


FIGURE 2 – Page présentant une task

La page task X target présente une target d'une task particulière. Elle comprends des liens vers les pages de la target et la task. Puis elle affiche les résultats (éventuellement par argument). Enfin elle montre aussi le code source de cette target précise.

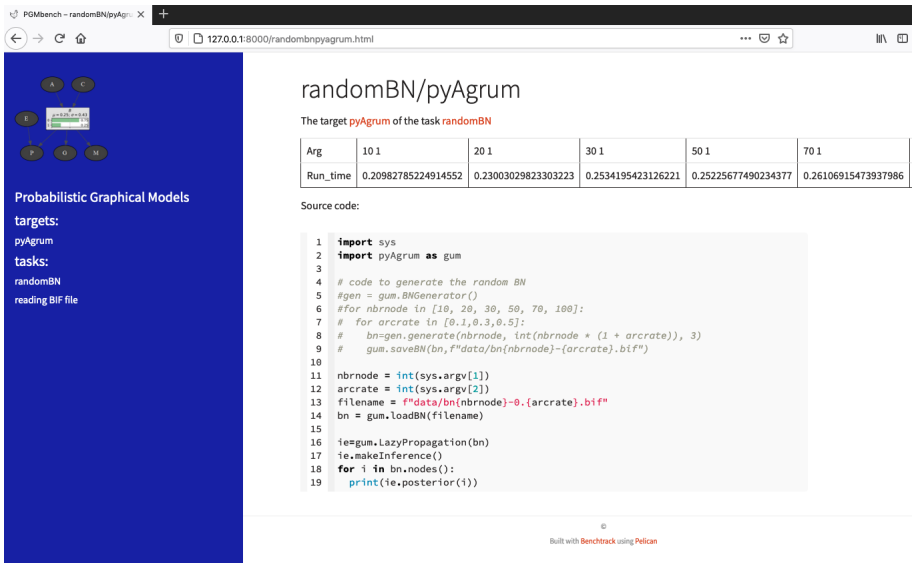


FIGURE 3 – Page task X target

L'affichage des résultats

Nous avons vu que nous avons affichons les résultats dans les pages task, target, task X target. Nous offrons la possibilité à l'utilisateur de choisir le type d'affichage de ces résultats. En effet, il faut le renseigner dans le fichier config.ini de la task. C'est le paramètre *display*, il peut prendre trois valeurs, *tabular* pour un tableau, *line* pour un courbe, *bar* pour un diagramme à barres.

Par défaut, on affiche des tableaux, on s'adapte alors au nombre d'arguments, si cela a un sens (au moins une task/target a plus d'un argument) on affiche un tableau à deux entrées (task ou target) avec les arguments.

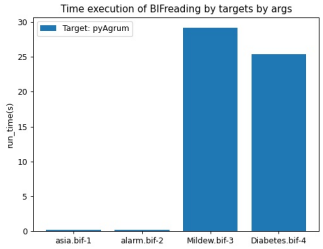
Les diagrammes à barres et courbes ne sont disponibles que pour les pages task et task X target. Dans le fichier task, on a par exemple un courbe pour chaque target de la task, avec les arguments en abscisse et le temps d'exécution en ordonnée. Les graphes sont produit à l'aide de matplotlib.

Arg/Tasks	randomBN	BIFreading
10 1	0.2594728469848633	
20 1	0.2206024169921875	
30 1	0.24495162963867187	
50 1	0.24579596519470215	
70 1	0.2509302139282227	
100 1	0.2790544033050537	
10 3	0.1962007999420166	

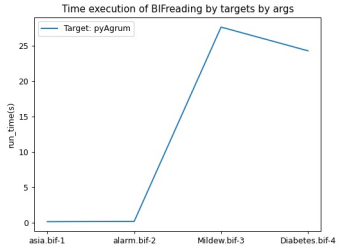
Tabular with args

Tasks	charger_employe	charger_employe_coordonnees
Run_time	0.040700531005859374	0.03851861953735351

Tabular without args



Bar



Line

FIGURE 4 – Les différents affichages des résultats

La possibilité de personnaliser le site

L'utilisateur a enfin la possibilité de personnaliser le site à son infrastructure. Il peut choisir le favicon du site, et un logo s'affichant au haut de la sidebar. Pour cela il devra mettre ces images dans un répertoire *img*, lui même placé dans le répertoire de l'infrastructure.

4 Conclusion

Après avoir terminé l'ensemble du framework, afin de faciliter l'utilisation de ce framework pour les utilisateurs, nous le conditionnons et le téléchargeons en ligne sur PyPi. On peut également utiliser *benchTrack* comme une commande pour l'exécuter directement dans le terminal. De plus, le comportement de l'empaquetage peut faciliter la maintenance du framework et les autres utilisateurs peuvent plus facilement comprendre le code, car notre code est organisé dans le format requis pour l'empaquetage.

À travers les deux parties ci-dessus, nous avons coopéré pour compléter un framework complet, des tâches de gestion à l'affichage des résultats, C'est au final un framework open source, utilisable et complet. Grâce à ce projet, notre capacité de programmation python a été améliorée, et nous avons également appris à programmer un framework, y compris l'achèvement et l'itération de versions après l'achèvement du framework, et également appris à utiliser les connaissances et les bibliothèques utilisées par de nombreux frameworks de développement, par exemple unitest et pelican. Ce framework aide donc les utilisateurs à comparer le temps d'exécution de différentes implémentations. Ce framework comporte certains domaines qui peuvent être améliorés. Par exemple, proposer plus de langages de programmation, en ajoutant plus de critères pour l'évaluation des implémentations, tels que l'ajout le taux de mémoire vive consommée par l'algorithme . Ensuite, nous continuerons à améliorer ce cadre, en espérant qu'à l'avenir, davantage de personnes pourront utiliser ce cadre pour résoudre des problèmes.

Nous tenons particulièrement à remercier le professeur Pierre-Henri Wuillemin pour ses conseils. Nous avons accumulé une expérience très précieuse et significative dans la production collaborative de frameworks.

5 Annexes

5.1 Des exemples d'infrastructures

L'*infrastructure* : *ConfigFichier*

On voudrait comparer différents formats de fichier de configuration (ini/json/xml) et leurs implémentations en python. Il s'agit donc de comparer la lecture de quelques exemples de fichier de configuration (ici deux) dans les différents formats. Il n'y a pas besoin d'une structuration en theme des deux seuls tasks.

```
/ConfigFichier
    /README.rst
    /targets
        /ini_demo
            /README.rst
            /config.ini
        /json_demo
            /README.rst
            /config.ini
        /xml_demo
            /README.rst
            /config.ini
    /tasks
        /default
            /charger_employe
                /README.rst
                /config.ini
                /data
                    /employe.ini
                    /employe.json
                    /employe.xml
                /ini_demo.py
                /json_demo.py
                /xml_demo.py
            /charger_employe_coordonnees
                /README.rst
                /config.ini
                /data
                    /employe_coordonnees.ini
                    /employe_coordonnees.json
                    /employe_coordonnees.xml
                /ini_demo.py
                /json_demo.py
                /xml_demo.py
```

L'*infrastructure PGM*

Pour les modèles probabilistes graphiques (PGM), on voudrait comparer quelques librairies (pyAgrum, pyPGM, BNLearn) sur un ensemble de tâches parfois spécifiques traitant de l'apprentissage, de l'ingérence probabiliste et de la modélisation.

On a ainsi l'arborescence suivante :

```
/PGM
    /README.rst
    /targets
        /pyAgrum
            /config.ini
            /README.rst
        /pyPGM
            /config.ini
            /README.rst
        /BNLearn
            /config.ini
            /README.rst
```

```
/tasks
    /learning
        /parameterLearning
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
        /parameterLearningWithMissingValues
            /README.rst
            /data
            /pyAgrum.py
            /BNLearn.r
        /structuralLearning
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
    /inference
        /exactInference
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
        /multipleExactInference
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
        /approximatedInferenceGibbsSampling
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
    /modelisation
        /modelingAsia
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
        /modelingAlarm
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
        /modelingPRM
            /README.rst
            /data
            /pyAgrum.py
            /pyPGM.py
            /BNLearn.r
```

5.2 Le manuel d'utilisateur

Voici un manuel d'utilisateur récapitulant les commandes les plus importantes à connaître pour utiliser notre framework :

- ☐ Pour installer le package :

  ```
  pip install -i https://test.pypi.org/simple/ BenchTrack
  ```

- ☐ Pour avoir plus d'information avec la commande *benchTrack* :

  ```
  benchTrack --help
  ```

- ☐ Pour lancer le framework simplement :

  ```
  benchTrack relativePath_ofInfrastructure
  ```

  Le chemin de l'infrastructure est relatif au répertoire courant de l'appel de la commande *benchTrack*. Le site est généré par défaut dans le répertoire courant.

- ☐ Pour exporter ailleur le site où on veut :

  ```
  benchTrack --output pathOutput relativePath_ofInfrastructure
  ```

  pathOutput est un chemin absolu.

- ☐ Pour sauvegarder l'archive pelican :

  ```
  benchTrack --pelican relativePath_ofInfrastructure
  ```

  L'archive *infrastructureName_pelican* est exporté suivant le même principe que le site : dans le path renseigné par *–ouptut* ou le répertoire courant par défaut.

- ☐ Pour lancer le site en local, dans le répertoire *infrastructureName_pelican*

  ```
  pelican --listen
  ```

5.3 La documentation sphynx

BenchTrack package

Submodules

BenchTrack.bench2site module

————————————————bench2site———————————————— This module generates the site, It can be used by calling directly the functions, You can also call in a shell as follows: python bench2site.py path_infrastructure path_csv path_output

BenchTrack.bench2site.bench2site(*path_infra*, *path_benchTrack*, *file_csv*, *path_output*, *save_pelican*) [source]

Main functions to generate the site which call the functions: csv2content, content2html, by default the site will be in the benchtrack directory

path_infra: String, absolute path of the infrastructure path_benchTrack: String, absolute path of the package benchTrack file_csv: String, absolute path of the framework Benchtrack path_output: String, absolute path of the output site save_pelican: Boolean, indicate if we have to save pelican archive

Nothing

BenchTrack.bench2site.content2html(*path_site_infra*, *path_infra*, *name_infra*, *path_output*) [source]

This function generates the static site with pelican

path_site_infra : String, absolute path of the infrastructure site path_infra : String, absolute path of the infrastructure name_infra: String, name of the current infrastructure path_output: String, absolute path of the output

Nothing

BenchTrack.bench2site.csv2content(*path_infra*, *path_benchTrack*, *file_csv*) [source]

This function generates a content directory for the static site

path_infra : String, absolute path of the infrastructure path_benchTrack: String, absolute path of the framework Benchtrack file_csv: String, name of the csv file used

path_site_infra: String, absolute path of temporary repository for generate site name_infra: String, infrastructure name

BenchTrack.bench2site.getDate() [source]

BenchTrack.bench2site.load_csv_results(*path_infra_csv*, *structure_run_time*) [source]

Loads the csv file of the run results

path_infra_csv : String, path of the csv file structure_run_time: Dictionary 3D [task][target][arg] which has been initialized

structure_run_time: The completed dictionary [task][target][arg]

BenchTrack.generateRst module

————————————————generateRst———————————————— This module contains all functions to generate rst files for the content directory

BenchTrack.generateRst.create_graph(*name_task*, *structure_run_time*, *list_target*, *path_images*, *display*, *name_target='targets'*) [source]

Create graph of run time for task page, either barchart or curve depends of display

name_task : String, name of the task structure_run_time: Dictionary 3D [task][target][arg] list_targets: String, list of the targets path_images: String, absolute path of images display: String, the type of display name_target:

path_relative_file: String, relative path to content of graph file

BenchTrack.generateRst.create_infra_rst(*name_infra*, *path_content*, *path_readme*, *structure_run_time*, *list_targets*) [source]

Write the rst file of the infrastructure presentation and the summary of results

name_infra : String, name of the infrastructure path_content: String, absolute path of the content directory path_readme: String, absolute path of the corresponding readme file structure_run_time: Dictionary 3D [task][target][arg] list_targets: String, list of the targets

Nothing

BenchTrack.generateRst.create_rst_base(*readme*, *title_rst*) [source]

This function generates the beginning of the rst files, it manages the titles

readme : String, the content of a readme file title_rst: String, title of the rst file

base_rst: String, the beginning of the rst file

BenchTrack.generateRst.create_targetXtask_rst(*name_target*, *name_task*, *path_code*,

path_targetsXtasks, structure_run_time, path_images, display) [source]

Write the rst file for a target of a task (result and source code)

name_target: String, name of the target name_tasks: String, name of the task path_code: String, absolute path of the code file path_targetsXtasks: String, absolute path of the targetsXtasks directory structure_run_time: Dictionary 3D [task][target][arg]

Nothing

BenchTrack.generateRst.create_target_rst(*name_target, path_readme, path_targets, structure_run_time, list_tasks*) [source]

Write the rst file for a target (presentation and result)

name_target : String, name of the target path_readme: String, absolute path of the corresponding readme file path_targets: String, absolute path of the targets directory structure_run_time: Dictionary 3D [task][target][arg] list_tasks: String, list of the tasks

Nothing

BenchTrack.generateRst.create_task_rst(*name_task, path_readme, path_tasks, structure_run_time, list_targets, path_images, display*) [source]

Write the rst file for a task (presentation and result)

name_task : String, name of the task path_readme: String, absolute path of the corresponding readme file path_tasks: String, absolute path of the tasks directory structure_run_time: Dictionary 3D [task][target][arg] list_targets: String, list of the targets path_images: String, absolute path of images display: String, the type of display

Nothing

BenchTrack.generateRst.meta_data() [source]

This function writes the string that contains all the metadata

Nothing

Nothing

BenchTrack.structureBench module

---

class **BenchTrack.structureBench.BenchTrack**(*path_inf, path_benchTrack*) [source]

Bases: `object`

This class contains the structure of the BenchTrack

:cvar str __path:the relative path that contains the infrastructure.which is also the first parameter of input :cvar str __name:name of bench :cvar dict __dictTargets:a dictionnaire contains keys of target's name and values of object Target :cvar list __listThemes:a list contains of object theme :cvar list __allTarget:a list of all targets must be tested :cvar list __allTask:a list of all tasks must be tested :cvar str __path_benchTrack:path of BenchTrack.py :cvar str __outputFile:path of output.csv

ToCsv() [source]

write infos of the infrastructure to csv file

:return:no return

addTargets(*name_target*, *lang_target*) [source]

Add targets at the dictionay dictTargets

Param: str name_target:targets of the infrastructure.

Param: str lang_target:programming languague of the target.

:return:no return

exe_bench() [source]

Execution of the programme

Returns: no return

filter_target(*lis*, *model*) [source]

delet the target has been excluded from the command of user

:param:list lis:list of targets will be excluded/included. :param bool model :if true, all the targets selected will be added in the list of execution.

else, delet those targets excluded

Returns: The number of targets to be tested

filter_task(*lis*, *model*) [source]

delet the task has been excluded from the command of user

:param list lis:list of tasks will be excluded/included. :param bool model:if true, all the tasks selected will be added in the list of execution.

else, delet those tasks excluded

Returns: no returnThe number of targets to be tested

getDisplay() [source]

getName() [source]

Getter of parameter __name

:return:name of Bench

getPathInf() [source]

getPathOutputFile() [source]

getPathOutputHtml() [source]

get_structure_tasks() [source]

get all infos of targets,tasks

Returns: the structure of the tasks

isPelican() [source]

setPathOutputHtml(*path*) [source]

setPelican(*b*) [source]

showInfoTarget(*nomTarget*) [source]

Output infos (readme) of the target

:param str nomTarget :target's name.

Returns: no return

showInfoTask(*nameTask*) [source]

output info(readme) of the task

:param str nameTask :task's name.

Returns: no return

showListTasks() [source]

output all the tasks that will be executed

Returns: no return

show_list_target() [source]

Output the list of targets that will be executed

Returns: no return

BenchTrack.target module

class **BenchTrack.target.Target**(*name*) [source]

Bases: `object`

This class contains the structure of the Target

:cvar str __Name :name of the target. :cvar dict __result:a dictionnaire contains keys of parameter and values of result

addResult(*arg*, *value*) [source]

add a result of test

:param str arg:the parameter of test :param float value:the result

exe_before_test(*themeName*, *task*, *path*, *args*) [source]

execute some before-task file like imports

:param str themeName:name of theme :param str task:name of task :param str path :path of the infrastructure :param str args:args of execution

exe_file(*path_configFile*, *path_file*, *args*, *target*) [source]

execution of a target

:param str path_configFile:path of file config of target :param str path_file:file to execute :param str args:args of execution :param str target :name of target

exe_target(*themeName*, *task*, *path*, *args*) [source]

drive infos from configure file of target and transforme these infos

:param str themeName:name of theme :param str task:name of task :param str path :path of the infrastructure :param str args:args of execution

getName() [source]

get name of the task

:return:name of the task.

getResult() [source]

get result of the target

:return:the dictionnaire of result.

BenchTrack.task module

class **BenchTrack.task.Task**(*name*, *args*, *sample_size=20*) [source]

Bases: `object`

This class contains the structure of the Task

:cvar str __Name :name of the task. :cvar str __args :parameters of the task. :cvar int __sample_size :optional,the times of execution. The default is 20. :cvar dict __dictTargets:a dictionnaire contains keys of target's name and values of object Target

ToCsv(*writer*, *theme*) [source]

genereto the csv file which contains some infos about results of execution

addTarget(*name_Target*) [source]

Add targets to targets dictionary of the task

param str name_Target :name of target.

exe_task(*lis*, *themeName*, *path*) [source]

Execution of the tasks

:param list lis :list of targets. :param str themeName : name of theme of the task. :param str path :the path contains the theme

Returns: no return

getName() [source]

get name of the task

:return:name of the task.

get_structure_tasks(*path*) [source]

get targets of tasks

Returns: list_target : TYPE DESCRIPTION.

modifyTarget(*name_Target*, *res_Target*) [source]

Add execution time of targets of the task

:param str name_Target :the target. :param float res_Target : execution time.

Returns: no return

upgrade_for_target(*path*, *target*) [source]

BenchTrack.theme module

class **BenchTrack.theme.Theme**(*name*) [source]

Bases: `object`

This class contains the structure of the theme of Task

:cvar str __themeName:name of the theme :cvar list __listTasks: list of object task in the theme

ToCsv(*writer*) [source]

To write list of tasks of the theme in csv file

None.

addTask(*task*) [source]

add task to the listTasks of the theme

:param Object_Task task:the task to add

:return:no return

exe_theme(*listTask*, *listTarget*, *path*) [source]

execution of the theme

:param list listTask :all the tasks of the theme. :param list listTarget :all the targets of the task in the theme. :param str path :the path contains the theme.

Returns: no return

getName() [source]

getter of parameter Name of the theme

Returns: string:name of the theme

get_structure_tasks(*path*) [source]

Get the structure of every task

:param str path :the path of the infrastructure

Returns: a dictionnaire

showInfoTask(*nameTask*, *nameTest*, *path*) [source]

Output the Info(readme) file of the task in the theme

:param str nameTask :name of task :param str nameTest : name of the Bench :param str path : the path contains the theme.

Returns: no return

showlistTasks() [source]

Output the list of tasks of the theme

Returns: no return

BenchTrack.tools module

BenchTrack.tools.ConfigFileTarget(*path_target*) [source]

This method trait configure file and return their command and programming language of the target.

:param str path_target:path of the config.ini for the target

:return:run:the command of executing the target

language:the programming language of the target

BenchTrack.tools.ConfigFileTask(*file*) [source]

This method trait configure file of task and return their sample_size and parameters of the task.

:param str file:path of the the config.ini for the task

:return:sample_size:sample size for execute all targets in this task

:arg:all args of the target

BenchTrack.tools.checkInfrastructure(*path_infras*) [source]

BenchTrack.tools.exeCmd(*path*, *parameter*, *cmd*, *language*, *target*) [source]

exeCmd is a fonction that switch the task to other specific fonction to execute them

:param str path : the path that contains the task need to be execute. :param str parameter : the parameter of the task. :param str cmd : command of executing based on terminal. :param str language : the programming language of task . :param str target: the name of target

BenchTrack.tools.exe_python(*target_name*) [source]

for calculating time of executing of python file Parameters :

target_name:path et nom pour target

:returns :temp d'execution

BenchTrack.tools.execute(*path, cmd*) [source]

This fonction is made to run files

:param str path :the path that contains the task need to be execute. :param str cmd :command of executing based on terminal.

BenchTrack.tools.existFile(*fileName, path*) [source]

To test if a Path contain a file

:param str fileName:name of the file :param str path:the path that contains the file

:returns:bool:True for Found a file name filename in path

esle False

BenchTrack.tools.file_read(*nameTest, nameTarget, typeF*) [source]

To read readme file and print the contents

nameTest : string

name of task.

nameTarget : string

name of target.

typeF : string

task/target.

:return:int:succeded print the content of readme.

BenchTrack.tools.find_all_file(*base*) [source]

Find all files in base

:param str base : the system path.

:yield str f : all files found in base.

BenchTrack.tools.flagTargets(*argv, bench, flag*) [source]

gestion of flags of the targets

:param str argv : name of the benchmark. :param str bench : name of bench for testing. :param str flag : type of flags.

BenchTrack.tools.flagTasks(*argv, bench, flag*) [source]

gestion of flags of the tasks

:param str argv :name of the benchmark. :param str bench :name of bench for testing. :param str flag : type of flags.

BenchTrack.tools.generateAgrs2D(*list2D*) [source]

BenchTrack.tools.generateArgsIter(*listIter*) [source]

BenchTrack.tools.generateArgsList(*string*) [source]

gernerate the list of args from a string

:param str string : string of args from config.

:return list:list of all args

BenchTrack.tools.getDate() [source]

BenchTrack.tools.get_suffixe(*language*) [source]

This method return the suffixe of programming language

:param str language:language to get a suffixe

Returns: suffixe of the fichier in language input

BenchTrack.tools.help() [source]

BenchTrack.tools.manage_flag(*argv, bench*) [source]

This function manage all flags, with a flag for show a list or the information,this function call a function to show that with a flag include or exclude,this function change the object bench without flag,return 1

:param argv:args Bench:l'object BenchTrack

BenchTrack.tools.to_txt(*list_res*) [source]

Turns the results to txt file

list_res : float list

time of executing.