

```

PPOptional.add_argument('-v', dest='verbose', action='count', default=0, help='increase verbosity level')
PPOptional.add_argument('--version', version='Version ' + str(constant.CURRENT_VERSION), help='laZagne version')

PWrite = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
PWrite._optionals.title = 'output'
PWrite.add_argument('-w', dest='write', action='store_true', help='write a text file on the current directory')

all_subparser = []
for c in category.keys():
    category[c]['parser'] = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
    category[c]['parser']._optionals.title = category[c]['help']

    category[c]['subparser'] = []
    for module in modules[c].keys():
        m = modules[c][module]
        category[c]['parser'].add_argument(m.options['command'], dest=m.options['dest'], help=m.options['help'])

        if m.suboptions and m.name != 'thunderbolt':
            tmp = []
            for sub in m.suboptions:
                tmp_subparser = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
                tmp_subparser._optionals.title = sub['title']
                if 'type' in sub:
                    tmp_subparser.add_argument(sub['command'], dest=sub['dest'], help=sub['help'], action=sub['action'], dest=sub['dest'], help=sub['help'])
                else:
                    tmp_subparser.add_argument(sub['command'], dest=sub['dest'], help=sub['help'], action=sub['action'], dest=sub['dest'], help=sub['help'])
                tmp.append(tmp_subparser)
            all_subparser.append(tmp_subparser)
        category[c]['subparser'] += tmp

parents = [PPOptional] + all_subparser + [PWrite]
dic = {'all': {'parents': parents, 'help': 'Run all modules', 'func': 'run_all_modules'}}
for c in category.keys():
    parser_tab = [PPOptional, category[c]['parser']]
    if 'subparser' in category[c]:
        if category[c]['subparser']:
            parser_tab += category[c]['subparser']
    parser_tab += [PWrite]
    dic_tmp = {c: {'parents': parser_tab, 'help': 'Run %s module %s' % (c, module), 'func': 'run_module', 'auditType': 'module'}}
    dic = dict(dic.items() + dic_tmp.items())

subparsers = parser.add_subparsers(help='Choose a module to run')
for d in dic.keys():
    subparsers.add_parser(d, parents=dic[d]['parents'], help=dic[d]['help']).set_defaults(func=dic[d]['func'], auditType=d)

```

FUNCIONES

PROGRAMACIÓN DE COMPUTADORAS III

Abdel G. Martínez L.

CONCEPTO

- ⦿ Bloque de código organizado y reutilizable
- ⦿ Sirve para realizar una tarea específica
- ⦿ Proveen modularidad a la aplicación
- ⦿ Python cuenta con sus funciones internas, como `print()`
- ⦿ Sin embargo, el programador puede crear sus propias funciones

REGLAS DE DEFINICIÓN

- ⦿ Los bloques de función inician con la palabra reservada `def` seguido por el nombre de la función y paréntesis `(())`
- ⦿ Cualquier parámetro de entrada, o argumento, debe ser definidos entre los paréntesis
- ⦿ El bloque de código de la función inicia luego de los dos puntos `(:)` y debe ser debidamente sangrado
- ⦿ Debe existir una expresión de retorno, `return`

EJEMPLO

- ⦿ El siguiente ejemplo toma un texto como argumento e imprime su valor en pantalla, sin retornar nada

```
def imprime(texto):  
    "Esta función imprime un texto"  
    print texto  
    return
```

LLAMANDO UNA FUNCIÓN

- ◉ Definir una función solo da un nombre, argumentos y estructura del bloque de código a ejecutar
- ◉ El programador puede llamar una función desde otra función o bien globalmente
- ◉ Debe indicar el nombre de la función, seguido de paréntesis
- ◉ En caso de que tenga argumentos, también debe indicarlos

```
>>> imprime("Curso")
```

FUNCIONES ANÓNIMAS

- ⦿ No son declaradas de la manera estándar
- ⦿ Se utiliza la sentencia `lambda`
- ⦿ Contienen una única expresión
- ⦿ Acceden únicamente a sus variables definidas, no las globales
- ⦿ No confundir con una función de una línea

EJEMPLO

```
>>> sum = lambda arg1, arg2: arg1 + arg2;  
>>> print("Valor total: ", sum( 10, 20 ))  
Valor total: 30
```

CONSIDERACIONES ESPECIALES

- ⦿ Utilizar las funciones anónimas para tareas específicas que se cumplan en una sola sentencia y sean manejadores de llamados
- ⦿ Utilizar las funciones tradicionales para tareas generales que serán reutilizables y modulares
- ⦿ Si defino una variable dentro del bloque de código de la función sería una variable local a la función