Modules and Packages

- Creating a Module
- Module search path
- Module vs Script
- Package Creation and importing

Creating a Module

- Any script created in python is a module and can be imported in other scripts/modules in python.
- Use the import statement to import modules

  import <mymodule>
- Use from-import syntax to import specific parts from a module

  *from <mymodule> import * #import everything*

  from <mymodule> import <someobject> #import only someobject

trainer.cpp@gmail.com

Module search path

- When running a script, all modules are searched from the current execution directory
- Use the environment variable **PYTHONPATH** to add paths to modules other than current working directory.
- Use the sys.path variable to add the current path.

Module vs Script

- There is a special attribute **__name__** available in modules/scripts.
- When a python script is executed as main script, it has the attribute **__name__** set to the value **'__main__'**
- The __name__ check

```
if __name__ == '__main__':
        .....
```

Can be used to identify when a script is being run as a top-level script or is being used as a module.

Package creation and importing

- When using modules in nested sub-directories, the individual directories are treated as modules, and each module in the import statement is separated by dot (.)

  Ex: **modules/dir1/dir2/add.py** to access from a script present in **modules**

  import dir1.dir2.add

Package creation and importing

- When using modules in nested sub-directories, the individual directories are treated as modules, and each module in the import statement is separated by dot (.)

  Ex: **modules/dir1/dir2/***add.py*

  /*diff.py*

  /*testScript1.py*

  /dir1/*testScript2.py*

Iterators and Generators

- Iterator vs Iterable
- Understanding with list example
- Iterable Requirements
- Iterator Requirements
- Generators and iterator behavior

Iterator vs Iterable

- An iterator is an object that allows the next method to be called upon it and returns values.
- In iterable is an object that has the __iter__ method, which returns an iterator.
- Ex: **list** is an **iterable**

  calling the **__iter__** method return an iterator

trainer.cpp@gmail.com

Iterable Requirements

- Should support an **__iter__** method which returns an iterator object upon calling.

- Example:

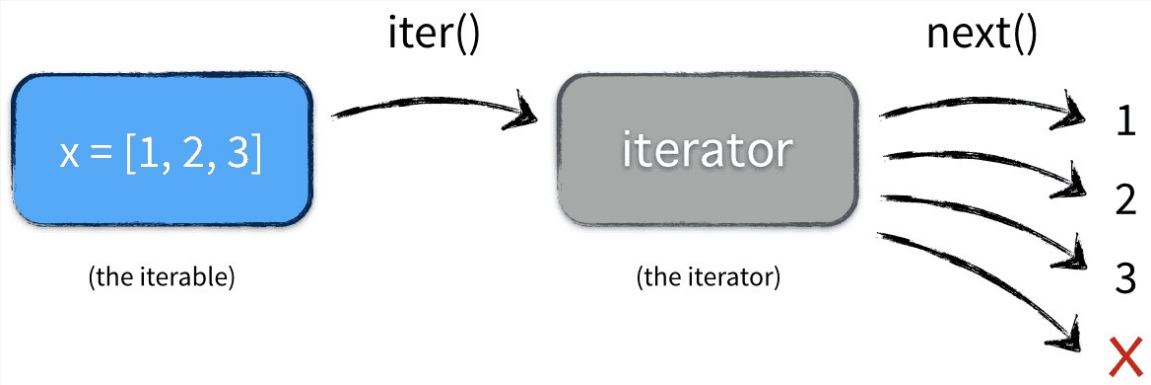
```
l = [1, 2, 3]
dir(l)
it1 = l.__iter__()
it2 = iter(l)
```


Iterator requirements

- An iterator should support the **__next__** method.
- Should raise a **StopIteration** exception upon reaching the last element to be iterated.
- Example:

```
l = [1, 2, 3]
itr = iter(l)
itr.__next__()
next(itr)
```

Understanding with list example



Generator and Iterator behavior

- Generator objects also support iterator protocol.
- They have the method __next__ to allow iteration
- Example:

```
def my_range():
    for value in range(10):
        yield(value)
itr = my_range()
dir(itr)
```