```
############################## TOPIC 1 - PACKAGES ################################

# 1. WHAT ARE PACKAGES?

    # In the Python Basics, we saw about modules. We create a script called -
    # 'my_module.py' and we imported it in some other python script. On REPL,

    import my_module
    type(my_module)         # Output: <class 'module'>

    # So on a very high level, we can say - if modules are files, then packages
    # are directories/folders. So a package can contain various modules and thus
    # importing a package is equivalent of importing all the modules in it.

    # For example, 'urllib' is a ready-made package available in python. And
    # 'request.py' is a module in the directory named - urllib.

    import urllib                   # Importing a package
    import urllib.request           # Importing a module

    # However if you will check their types, both of them are objects of class
    # 'module'. This is because - anything which you import in python, is an
    # object of 'module' class. So packages and modules are no different when
    # it comes to importing them and using them in your script.

    type(urllib)            # Output: <class 'module'>
    type(urllib.request)    # Output: <class 'module'>

    # The difference between a package and a module comes based on the
    # '__path__' attribute. Eg.

    urllib.__path__         # Returns the path where this directory is located
    # ['C:\\Users\\sagar\\AppData\\Local\\Programs\\Python\\Python37\\lib\\urllib']

    urllib.request.__path__ # Modules do not have a path attribute
    # AttributeError: module 'urllib.request' has no attribute '__path__'

    # Also note that - a folder can contain subfolders as well, which in turn
    # will contain several python scripts. So a main package can contain several
    # subpackages, along with modules. It is possible, but not at all promoted,
    # to have more folders inside the subfolders. Too much of nesting of folders,
    # spoils the package structure.

    # How to make and implement a package? To answer this question, we first
    # need to understand - How does python locate and import modules?

# ------------------------------------------------------------------------------
```

```
# 2. IMPORTING MODULES

    # - Understanding SYS.PATH:

    # 'sys.path' is the list of directories in which python searches for modules.
    # All these paths are stored in the 'path' attribute of the standard 'sys'
    # module. (Since packages and modules are both the objects of class 'module',
    # we won't treat them as different. We will highlight the packages and modules
    # differently, only when we will have to show the difference between them.)

    import sys
    sys.path        # This will output a long list of directories.

    # So whenever we ask python for particular module, python checks into the
    # 1st directory at sys.path and checks for the appropiate file. If no match
    # is found in 1st directory, python checks the next directory. This goes on
    # until the match is found or python runs out of directories in sys.path -
    # in which case, an ImportError is raised.

    # Let's create a simple directory with the name - 'test' and inside that
    # create a script named 'path_test.py'. So the folder structure should look
    # like:

        # test
        # --> path_test.py

    # Add the following contents in your path_test.py:

    def found():
        print('Python found me!')

    # Now come on REPL:

    import path_test
    # ImportError: No module named 'path_test'

    # This is because the path of folder 'test' is not available in sys.path.
    # To make path_test importable, we need to add the path of 'test' in
    # sys.path. Since sys.path returns a list of directories, all operations
    # valid on list are valid here as well.

    import sys
    sys.path.append('test')
    import path_test
    path_test.found()       # Output: Python found me!
```

```
# - Understanding PYTHONPATH:

    # If we close and run the REPL again, the sys.path list will revert back
    # to normal and path of 'test' folder will be removed automatically.
    # So whenever we want to import our 'path_test' module, we have to manually
    # append the path of 'test' to sys.path. However if we wish that path of
    # 'test' should remain in sys.path, we need to add the path to 'PYTHONPATH'.

    # PYTHONPATH - is an environment variable listing paths added to sys.path
    # when python starts. To add the path in PYTHONPATH, use following commands
    # in CMD/Terminal (not on REPL):

    set PYTHONPATH = test       # For Windows
    export PYTHONPATH = test    # For MAC/Ubuntu/Bash shell

    # Now if you start the REPL, you can find that 'test' indeed is in sys.path.

    import sys
    [p for p in sys.path if 'test' in p]    # Outputs the path of 'test'

    # Now we can directly import path_test without manually editing sys.path.

# ------------------------------------------------------------------------

# 3. IMPLEMENTING A PACKAGE:

    # To create a module, you create a python file and make sure that it's
    # folder's location is available in sys.path. The process of creating a
    # package is not much differnt. To create a package, we create a root
    # directory, whose path must be in sys.path. And inside this root directory,
    # we have a directory for our package. In this package directory, there is
    # a special file '__init__.py' which makes packages importable as modules.

        #  root_folder/
        #  --> my_package/
        #      --> __init__.py

    # In the following discussion, we assume that you are working inside the
    # root_folder and the path of the root_folder is available in sys.path.

    # Create a package folder - 'reader' and inside that create a file
    # '__init__.py'. The code inside this file gets executed when the package
    # is imported. Congrats! You have just made a package! (Though it does
    # nothing.) On REPL,

    import reader
    type(reader)            # Output: <class 'module'>

    # We will create a simple 'reader' script, which can read text from a file.
```

```
    # reader/
        # --> __init__.py
        # --> reader.py

    # Contents of reader.py:

    class Reader:
        def __init__(self, filename):
            self.filename = filename
            self.f = open(self.filename, mode='rt', encoding='utf-8')

        def close(self):
            self.f.close()

        def read(self):
            return self.f.read()

    # On REPL, let's import our reader.py module:

    import reader.reader                            # Imports reader.py module
    r = reader.reader.Reader('reader/reader.py')    # Let's read the reader.py file
    r.read()                                        # Outputs the file contents
    r.close()

    # What we don't like in above implementation is - we have to type
    # 'reader.reader' to access the class of our module. Instead we wish our
    # 'Reader' class to be at the top level of reader package. To do so,
    # in __init__.py file, add the following line:

    from reader.reader import Reader

    # Restart the REPL, and check that the Reader class is at the top level
    # reader namespace.

    import reader
    r = reader.Reader('reader/__init__.py')
    r.read()
    r.close()

# ------------------------------------------------------------------------

# 4. ABSOLUTE and RELATIVE IMPORTS

    # In the __init__.py file of our reader package, we wrote an import
    # statement:

    from reader.reader import Reader        # Absoulte Import

    # This is an 'absolute import'. Such import statements use a full path to
    # a module.
```

```
    # There is nothing wrong with this type of import statement and is
    # recommended to use absolute imports, but sometimes writing such long
    # import statements become tedious. So, we can use - 'relative imports'
    # instead. Such imports use a relative path to modules in the same package.
    # For eg. we could have written following import statement in our __init__.py
    # file as well:

    from .reader import Reader              # Relative Import

    # Consider a hypothetical package shown below:

        # my_package/
        # --> __init__.py
        # --> a.py
        # --> nested/
        #     --> __init__.py
        #     --> b.py
        #     --> c.py

    # Now let's say we have following statements in the __init__.py file of
    # 'nested' folder:

    from ..a import A           # A can be class/function inside a.py
    from .b import B            # B can be class/function inside b.py

    # 2 dots preceding a module means the search is to be performed in parent
    # directory; and 1 dot preceding a module means the search in same directory.

# ------------------------------------------------------------------------

# 5. EXECUTABLE DIRECTORIES

    # When you import a package on REPL or any python script, the code inside
    # __init__.py gets executed. But on command line, when we use -

    python3 <package_name>

    # a special file inside that directory - __main__.py gets executed.
    # For example, consider the following folder structure:

        # reader/                       (notice that this is root folder)
        # --> __main__.py
        # --> reader/                   (and this is package folder)
        #     --> __init__.py
        #     --> reader.py

    # Contents of __main__.py:

    print('Executing __main__.py with name {}'.format(__name__))
```

```
    # Now on CMD/Terminal:

    python3 reader
    # Output: Executing __main__.py with name __main__

    # We now say that - 'reader' has become an 'executable directory'.
    # But what is the use of having 'executable directories'?

    # When you make directories executable, we no more look the directories as
    # 'packages'. We now look at them as 'projects'. Usually we intend to have a
    # '__main__.py' file which can act as starting point of execution of our
    # project. Inside this project folder, we can have many other files. Here
    # is the 'recommended project structure':

        # project_name/                     (root folder)
        # --> __main__.py
        # --> project_name/
        #     --> __init__.py
        #     --> more_source_files.py
        #     --> subpackage/
        #         --> __init__.py
        #     --> test
        #         --> __init__.py
        #         --> testing_code.py
        # --> setup.py

    # We will be more familiar with such a project structure when we will be
    # creating some python projects. This is recommended structure, as it keeps
    # every file in its appropriate location.

# ------------------------------------------------------------------------
```

```
#################### TOPIC 2 - STATIC AND CLASS METHODS ########################

# 1. CLASS ATTRIBUTES

    # Create a file - 'shipping.py' and add the following contents:

    class ShippingContainer:
        def __init__(self, owner_code, contents):
            self.owner_code = owner_code
            self.contents = contents

    # This class is simple enough to run from REPL.

    from shipping import *
    c1 = ShippingContainer("YML", "books")
    c1.owner_code                    # Output: YML
    c1.contents                      # Output: books

    # If we create another shipping container, it will have its own owner_code
    # and contents as you would expect.

    c2 = ShippingContainer("MAE", "clothes")
    c2.owner_code                    # Output: MAE
    c2.contents                      # Output: clothes

    # Sometimes, we would wish to have an attribute which is associated with the
    # class and not with the instance of each of the class. In other words,
    # we would like to have an attribute whose value is shared between all
    # instances of the class. For example, let's say we want to assign each new
    # shipping container a serial number. Let this serial number start from an
    # arbitrary value 1337. We modify our class as follows:

    class ShippingContainer:
        next_serial = 1337
        def __init__(self, owner_code, contents):
            self.owner_code = owner_code
            self.contents = contents
            self.serial = ShippingContainer.next_serial
            ShippingContainer.next_serial += 1

# ------------------------------------------------------------------------------

# 2. STATIC METHOD

    # Let's make a small change in our code. Let's put the logic of getting
    # next serial number into a class method, instead of putting it in
    # initializer.
```

```
    class ShippingContainer:
        next_serial = 1337

        def _get_next_serial(self):
            result = ShippingContainer.next_serial
            ShippingContainer.next_serial += 1
            return result

        def __init__(self, owner_code, contents):
            self.owner_code = owner_code
            self.contents = contents
            self.serial = self._get_next_serial()

    # You can see that - inside _get_next_serial(), we are not using 'self'
    # at all. And moreover, in __init__() function, self._get_next_serial()
    # creates a misconception that _get_next_serial() is bound to a instance
    # instead of a class. So we modify the code a little bit, as:

    class ShippingContainer:
        next_serial = 1337

        @staticmethod
        def _get_next_serial():
            result = ShippingContainer.next_serial
            ShippingContainer.next_serial += 1
            return result

        def __init__(self, owner_code, contents):
            self.owner_code = owner_code
            self.contents = contents
            self.serial = ShippingContainer._get_next_serial()

    # When you define a method as static, it gets associated with the class,
    # and could be called using class name, instead of calling it on instance.
    # We can omit - 'self' in static methods.

# ------------------------------------------------------------------------

# 3. CLASS METHOD

    # As an alternative to static method, we can use - 'class method'.
    # Class methods require first argument as - 'cls', which stands for 'class'.
    # 'class' is already a keyword used to define classes, we are provided with
    # the word - 'cls'.

    # The 'cls' argument performs the analogous role in the class methods, what
    # 'self' performs in instance methods.
```

```python
class ShippingContainer:
    next_serial = 1337

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    def __init__(self, owner_code, contents):
        self.owner_code = owner_code
        self.contents = contents
        self.serial = ShippingContainer._get_next_serial()


# So the question is - when to use which function?

# a) If you want to call a method on each instance, you use normal
#    function with 'self' as first argument.

# b) If you want to call a method on class, you have two options -
#    use @staticmethod or @classmethod. @staticmethod receives no arguments.
#    It is like a normal function, like the ones you define outside classes,

    def some_function_name():
        # few statements

#    But this function is now restricted in the scope of the class. Apart
#    from the scope, it is similar to normal functions. @classmethod receives
#    the class object. Most of the times - @classmethod and @staticmethods
#    can be used interchangably, but it is often recommended to use
#    @classmethod if you are using the class attributes or methods inside it.

#    A common use of @classmethod is to define different type of initializers.
#    For example, in the following code - create_empty() creates an empty
#    container and create_with_items() creates a container with items.

class ShippingContainer:
    next_serial = 1337

    @classmethod
    def _get_next_serial(cls):
        result = cls.next_serial
        cls.next_serial += 1
        return result

    @classmethod
    def create_empty(cls, owner_code):
        return cls(owner_code, contents=None)
```

```
        @classmethod
        def create_with_items(cls, owner_code, items):
            return cls(owner_code, contents=list(items))

        def __init__(self, owner_code, contents):
            self.owner_code = owner_code
            self.contents = contents
            self.serial = ShippingContainer._get_next_serial()

    # On REPL,

    from shipping import *
    c3 = ShippingContainer.create_empty("YML")
    c3.serial                   # Output: 1337
    c3.owner_code               # Output: YML
    c3.contents                 # No Output, meaning None.

    c4 = ShippingContainer.create_with_items("MAE", ['food', 'oil', 'minerals'])
    c4.serial                   # Output: 1338
    c4.owner_code               # Output: MAE
    c4.contents                 # Output: ['food', 'oil', 'minerals']

# --------------------------------------------------------------------------
```

```
############################## TOPIC 3 - INHERITANCE ##############################

# 1. SINGLE INHERITANCE:

    # The following discussion assumes that you know a little bit about
    # Inheritance, from your experience in C++ or Java.

    # The syntax of single inheritance is:

    class SubClass (BaseClass):
        # methods and attributes of the class

    # The SubClass will have all the methods and attributes of the BaseClass
    # and can override them if re-defined in the BaseClass. A simple example
    # of single inheritance is given below:

    # Create a file - 'base.py' and add the following content:

    class Base:
        def __init__(self):
            print('Base class initializer')

        def f(self):
            print('Base.f()')

    class Sub(Base):
        pass

    # On REPL,

    from base import *
    b = Base()              # Output: Base class initializer
    b.f()                   # Output: Base.f()
    s = Sub()               # Output: Base class initializer
    s.f()                   # Output: Base.f()

    # Now let's override the __init__() and f() function in sub class.

    class Sub(Base):
        def __init__(self):
            print('Sub class initializer')

        def f(self):
            print('Sub.f()')

    # On REPL,

    from base import *
    s = Sub()               # Output: Sub class initializer
    s.f()                   # Output: Sub.f()
```

```
    # One important thing to notice is that, when creating the object of the
    # derived subclass, the initializer of base class is not executed. Other
    # programming languages like C++ or Java, also call initializer of base
    # class automatically when creating the object of sub class. So base class
    # __init__() is not called if overridden. If you want to call base class
    # __init__(), use super() to do so.

    class Sub(Base):
        def __init__(self):
            super().__init__()
            print('Sub class initializer')

        def f(self):
            print('Sub.f()')

    # On REPL,

    from base import *
    s = Sub()
    # Output: Base class initializer
    #         Sub class initializer

# ------------------------------------------------------------------------

# 2. EXAMPLE - WORKING WITH LISTS

    # Create a file - 'sorted_list.py' and add the following contents:

    class SimpleList:
        def __init__(self, items):
            self._items = list(items)

        def add(self, item):
            self._items.append(item)

        def __getitem__(self, index):
            return self._items[index]

        def sort(self):
            self._items.sort()

        def __len__(self):
            return len(self._items)

        def __repr__(self):
            return "SimpleList({!r})".format(self._items)

    # We now create another list - SortedList, which keeps the items of list
    # sorted.
```

```
    class SortedList(SimpleList):
        def __init__(self, items=()):
            super().__init__(items)
            self.sort()

        def add(self, items):
            super().add(item)
            self.sort()

        def __repr__(self):
            return "SortedList({!r})".format(list(self))

    # On REPL, test SortedList.

    from sorted_list import SortedList
    sl = SortedList([4, 3, 78, 11])
    sl                                  # SortedList([3, 4, 11, 78])
    len(sl)                             # 4
    sl.add(-42)                         # SortedList([-42, 3, 4, 11, 78])

# ----------------------------------------------------------------------------

# 3. isinstance() and issubclass() methods:

    # Along with the SortedList class we defined earlier, we want to define
    # another class IntList, which only allows integer contents. This list
    # subclass will prevent the insertion of non-integer elements. To do this,
    # we need to check type of the items that are inserted. And the tool which
    # we will use for this is - isinstance() method.

    # isinstance() determines if an object is of a specified type. Use this
    # function for runtime type checking. It takes first argument as object,
    # and its type as second. It returns either true or false. isinstance() also
    # returns true if an object belongs to subclass and is checked for type of
    # the base class. For example, on REPL,

    isinstance(3, int)                  # True
    isinstance("sagar", str)            # True
    isinstance(4.567, bytes)            # False
    isinstance(sl, SortedList)          # True
    isinstance(sl, SimpleList)          # True

    # You can also pass a tuple of types and isinstance() returns true if the
    # type of an object matches with any of the type mentioned in the tuple.
    x = [3, 4]
    isinstance(x, (float, list, dict))  # True

    # Now add the implentation of IntList in your sorted_list.py
```

```
    class IntList(SimpleList):
        def __init__(self, items=()):
            for x in items:self._validate(x)
            super().__init__(items)

        @staticmethod
        def _validate(x):
            if not isinstance(x, int):
                raise TypeError('IntList only supports integer values.')

        def add(self, item):
            self._validate(item)
            super.add(item)

        def __repr__(self):
            return "IntList({!r})".format(list(self))

    # Let's test this on REPL.

    from sorted_list import IntList
    il = IntList([1, 2, 9, 4])
    il.add(19)
    il                          # Output: IntList([1, 2, 9, 4, 19])
    il.add('5')
    # TypeError: IntList only supports integer values.


    # Another function which does type-checking is - issubclass(). This function
    # determines if one type is a subclass of another. It takes two arguments,
    # and both of them need to be type arguments. It returns true if a type
    # is direct or indirect subclass of another. For example,

    from sorted_list import *
    issubclass(IntList, SimpleList)             # True
    issubclass(SortedList, SimpleList)          # True
    issubclass(IntList, SortedList)             # False

# ------------------------------------------------------------------------

# 4. MULTIPLE INHERITANCE:

    # Multiple inheritance means defining a class with more than one base class.
    # This feature is not universal in OOP Languages. For example, C++ supports
    # multiple inheritance while Java does not. Multiple Inheritance can lead
    # to certain complex situations. For eg. deciding what to do when more than
    # one base class defines a particular method. But Python has a simple and
    # understandable system for handling such cases in multiple inheritance.
```

```
# The syntax for defining multiple inheritance is:

class SubClass(Base1, Base2, ...):
    # methods and attributes of subclass

# Subclasses inherit methods of all bases. If there are no name conflicts,
# names resolve in obvious way. But if there are any name conflicts, MRO
# (Multiple Resolution Order) is used to lookup for names. We will look into
# MRO shortly.

# Let's define SortedIntList class.

class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))

# Here we have just overridden __repr__ method, and on REPL, we can check
# that the class works as expected.

from sorted_list import SortedIntList
sil = SortedIntList([42, 23, 2])
sil                          # Output: SortedIntList([2, 23, 42])
SortedIntList([3, 2, '1'])
# TypeError: IntList only supports integer values.
sil.add(-123)
sil                          # Output: SortedIntList([-123, 2, 23, 42])
sil.add("Smallest Number")
# TypeError: IntList only supports integer values.

# It may not be apparent as of now, how all of this works together.
# We yet don't know - How does Python know which 'add' to call?
# Or more importantly, How does Python maintain both constraints?

# Before answering these questions, there are few more details related to
# multiple inheritance which we need to cover.

# Point 1: __bases__ attribute
# This attribute returns a tuple of types of base classes.

SortedIntList.__bases__
# Output: (<class 'sorted_list.IntList'>, <class 'sorted_list.SortedList'>)
IntList.__bases__
# Output: (<class 'sorted_list.SimpleList'>, )

# Point 2: Initializer of Base Class
# If a class has multiple base classes and defines no initializer of its
# own, then only the initializer of the first base class is automatically
# called. For example:
```

```
    class Base1:
        def __init__(self):
            print('Base1.__init__')

    class Base2:
        def __init__(self):
            print('Base2.__init__')

    class Sub(Base1, Base2):
        pass

    s = Sub()               # Output: Base1.__init__

# ----------------------------------------------------------------------------

# 5. METHOD RESOLUTION ORDER (MRO)

    # MRO is the order that determines the method name lookup.

    # First lets look where MRO of a class is stored. To get MRO of a class,
    # we use an attribute __mro__.

    from sorted_list import SortedIntList
    SortedIntList.__mro__
    # Output: (<class 'sorted_list.SortedIntList'>, <class 'sorted_list.IntList'>,
    # <class 'sorted_list.SortedList'>, <class 'sorted_list.SimpleList'>,
    # <class 'object'>)

    # We can also call mro as a function to get the output in a list, than a tuple.
    SortedIntList.mro()
    # Output: [<class 'sorted_list.SortedIntList'>, <class 'sorted_list.IntList'>,
    # <class 'sorted_list.SortedList'>, <class 'sorted_list.SimpleList'>,
    # <class 'object'>]

    # When an object calls a function, python looks at the MRO of that class
    # and then starting from the first entry, python starts searching for that
    # function. If any class has the requested method, it uses that method and
    # the search stops.

    # The question is - If you look at the MRO of the SortedIntList, IntList
    # comes before SortedList and thus when we call for add() function with
    # proper integer value, add() of SortedList was called and not of IntList.
    # In this way, SortedIntList maintained both the constraints. So how is
    # IntList.add() deferring to SortedList.add() ?

    # The answer to this mystery contains in the way, how super() actually works.

# ----------------------------------------------------------------------------
```

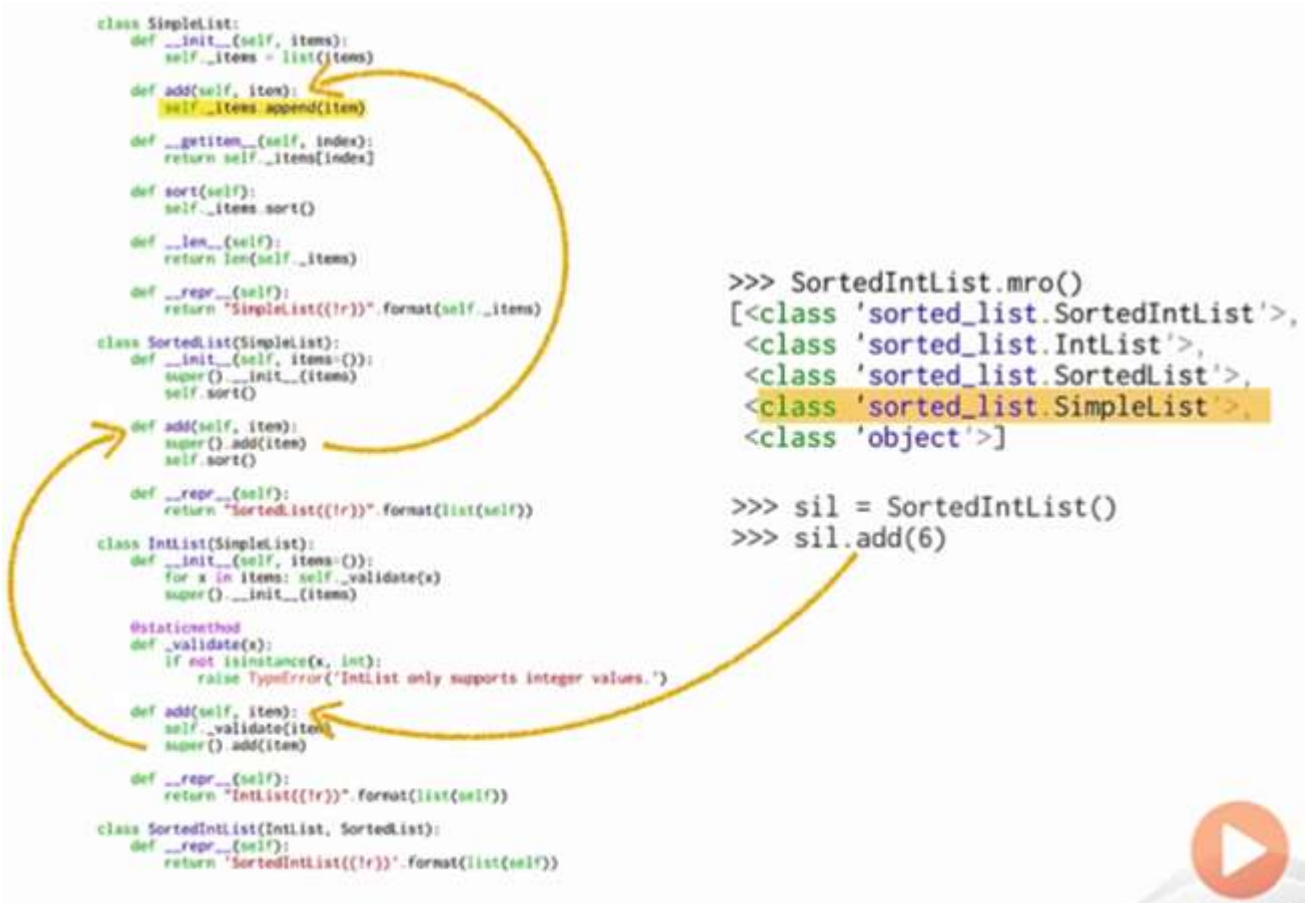
```
# 6. super() [Advanced Topic*]

    # So far we have been using super() to actually invoke the functions in the
    # base class. Recall, in SortedList class, we called super().add() in add()
    # function, to call the SimpleList's add() method.

    def add(self, item):
        super().add(item)
        self.sort()

    # From this example, you might conclude, super() somehow returns base-class
    # of the method's class and you can than invoke the method in the base-class.
    # But this is only partly true. It's hard to sum up what super() does in a
    # single sentence, but here is an attempt:

    # "Given a method resolution order and a class C, super() gives you an
    # object which resolves methods using only the part of the MRO which comes
    # after C."

    # Don't worry, if this statement doesn't make any sense to you right now.
    # We will understand this statement bit by bit. And the moment we understood
    # this statement - we are done understanding the mysteries of Multiple
    # Inheritance in Python.

    # super can be called in two ways:
    # a) super(base-class, derived-class)
    # b) super(class, instance-of-class)

    # By now, you must have concluded that super returns some temporary object
    # of the base class, on which we can call methods of the base class. We call
    # this temporary return object as 'proxy'.

    # These proxies can bound to - classes or instances. If a proxy is bound to
    # a class, it can only call 'staticmethods' and 'classmethods' i.e. methods
    # which are bound to classes. If a proxy is bound to an instance, it can
    # call the methods associated with the instance i.e. all methods containing
    # 'self'.

    # So when we call super() in first way i.e. super(base-class, derived-class),
    # we get a proxy which is bound to class (class-bound proxies). Now we must
    # find - in case of multiple inheritance, proxy belongs to which of the base
    # class. For that, following steps are followed:

    # - Python finds MRO of the derived class
    # - It then finds base-class in that MRO
    # - It takes everything after base-class in that MRO, and finds the first
    #   class in that sequence with a matching method name

    # Let's see in action, whatever we discussed above:
```

```
from sorted_list import *
SortedIntList.mro()
# [<class 'sorted_list.SortedIntList'>,
#  <class 'sorted_list.IntList'>,
#  <class 'sorted_list.SortedList'>,
#  <class 'sorted_list.SimpleList'>,
#  <class 'object'>]

super(SortedList, SortedIntList)
# <super: <class 'SortedList'>, <SortedIntList object>>

# Let's use the 3 steps mentioned above to validate the output we got by
# executing above python statement.

super(SortedList, SortedIntList)
# 1. Find the MRO of the derived class. We have already found that above.
# 2. Find 'SortedList' in that MRO. It's the third entry in the list.
# 3. It takes everything after 'SortedList'
#    i.e. [<class 'sorted_list.SimpleList>,
#          <class 'object'>]
#    and returns this MRO containing only 2 entries.

# So if we call add over this super(), add must belong to SimpleList.
# Let's see if we are right.

super(SortedList, SortedIntList).add
# Output: <function SimpleList.add at 0x10436a050>

# However we can't call this add() method because add() is bound to object
# and not to class.
super(SortedList, SortedIntList).add(4)
# TypeError: add() missing 1 required positional argument: 'item'

# We can only call 'staticmethod' and 'classmethod'. We can see this as:
super(SortedIntList, SortedIntList)._validate(5)
# No output. Statement executed successfully.
super(SortedIntList, SortedIntList)._validate("hello")
# TypeError: IntList only supports integer values.


# Now lets call super() in second way. To find the instance to which proxy
# is bound to (instance-bound proxies), we must follow the following steps:

# - Find the MRO for the type of the second argument
# - Find the location of the first argument in the MRO
# - Uses everything after the first argument's location for resolving
#   methods.

# Let's see in action, whatever we discussed above:
```

```
from sorted_list import *
SortedIntList.mro()
# [<class 'sorted_list.SortedIntList'>,
#  <class 'sorted_list.IntList'>,
#  <class 'sorted_list.SortedList'>,
#  <class 'sorted_list.SimpleList'>,
#  <class 'object'>]

sil = SortedIntList([5, 15, 10])
sil             # Output: SortedIntList([5, 10, 15])

super(SortedList, sil)
# Output: <super: <class 'SortedList'>, <SortedIntList object>>

# Like discussed in case 1, the above super() method now works on the
# remaining MRO with 2 entries only:
#   [<class 'sorted_list.SimpleList>,
#    <class 'object'>]

# So the above super proxy will directly use the SimpleList's methods,
# by-passing our constraint checks.

super(SortedList, sil).add(6)
sil                 # Output: SortedIntList([5, 10, 15, 6])
super(SortedList, sil).add("I m not a Int")
sil                 # Output: SortedIntList([5, 10, 15, 6, "I m not a Int"])

# So our SortedIntList isn't even sorted and also contains elements apart
# from ints. This means - we need to use super() with care. It does give you
# some super-powers XD.

# Examples we used above, had arguments in super(), while the examples we
# used in our sorted_list.py did not have any arguments. If no arguments are
# passed in super(), python will sort out the arguments for us.

# For class-bound proxies, super() is equivalent to:
#   super(class-of-method, class)

# For instance-bound proxies, super() is equivalent to:
#   super(class-of-method, class)


# Now that you have absorbed so much of information, you know that how
# super() uses everything after a specific class in an MRO to resolve
# method calls.

# Coming back to our mystery - How does SortedIntList work?
```

```python
# The key is - Both classes (SortedList and IntList) use super() instead of
# direct base classes references. But as we now know, super() doesn't just
# let us access the base classes, but rather let's access the complete
# MRO of the class. Let's look at the entire code again:

class SimpleList:
    def __init__(self, items):
        self._items = list(items)

    def add(self, item):
        self._items.append(item)

    def __getitem__(self, index):
        return self._items[index]

    def sort(self):
        self._items.sort()

    def __len__(self):
        return len(self._items)

    def __repr__(self):
        return "SimpleList({!r})".format(self._items)


class SortedList(SimpleList):
    def __init__(self, items=()):
        super().__init__(items)
        self.sort()

    def add(self, items):
        super().add(item)
        self.sort()

    def __repr__(self):
        return "SortedList({!r})".format(list(self))


class IntList(SimpleList):
    def __init__(self, items=()):
        for x in items:self._validate(x)
        super().__init__(items)

    @staticmethod
    def _validate(x):
        if not isinstance(x, int):
            raise TypeError('IntList only supports integer values.')

    def add(self, item):
        self._validate(item)
        super.add(item)

    def __repr__(self):
        return "IntList({!r})".format(list(self))
```

```
class SortedIntList(IntList, SortedList):
    def __repr__(self):
        return 'SortedIntList({!r})'.format(list(self))


# Look again at the MRO of the SortedIntList:

# [<class 'sorted_list.SortedIntList'>,
#  <class 'sorted_list.IntList'>,
#  <class 'sorted_list.SortedList'>,
#  <class 'sorted_list.SimpleList'>,
#  <class 'object'>]

sil = SortedIntList()
sil.add(6)

# A call to add() on SortedIntList, resolves to the add() of the IntList.
# The add() of the IntList calls super().add(). This super() uses the MRO
# after IntList class, to resolve for add() and thus refers to the add()
# of the SortedList, instead of SimpleList (what we thought earlier).
# This is how add() on SortedIntList maintains two constraints, without
# manually combining them. The super().add() in SortedList resolves to add()
# of the SimpleList.

# This is a fairly deep result and if you understand how SortedIntList works,
# then you have a good grasp on super().
```




```
# ------------------------------------------------------------------------------
```

```
####################### TOPIC 4 - FUCNTION DECORATORS ##########################

# 1. INTRO TO DECORATORS:

    # Decorators can be used with functions, as well as with classes. However,
    # in this topic, we will restrict our discussion to functions only.
    # Ensure that you have read about 'Scopes' from 'Python Basics' pdf.
    # This topic is little tricky. So, bear with the information given to you
    # till the end - to get few 'Aha...' moments!

    # Look at the following functions given below.

    def decorator(func):
        def new_func():
            original_result = func()
            modified_result = original_result.upper()
            return modified_result
        return new_func

    @decorator
    def my_function():
        greeting = "Hello World!"
        return greeting

    # On REPL, when we call this function, we get the output with all letters
    # turned into uppercase.

    my_function()           # Output: 'HELLO WORLD!'

    # Let's understand the flow of what's happening here.

    # STEP 1 - Python sees the function decorator() and stores it like any other
    # normal function.

    # STEP 2 - Python sees a new syntax - @<name of the decorator function>.
    # This is the syntax for assigining a decorator to a function, my_function()
    # in this case. Whenever python sees a decorator like this, it first compiles
    # the base function i.e. my_function() and creates an object of that function.

    # STEP 3 - Python passes this base function's object to the decorator
    # function, which accepts this object in it's argument 'func'. Now remember,
    # a function decorator should always accept a function object as argument
    # and must also return a function object, usually a modified one. Inside
    # this decorator function, we define another function - which modifies the
    # result of the original function (which is now stored in 'func') and returns
    # an object of it's own.

    # STEP 4 - So when we call my_function() with a decorator, it is not executed
    # immediately. What is actually executed is - decorator(my_function). So,
    # an object of my_function() is created which is passed to the decorator,
    # a new function is defined, inside which our original function is called.
```

```
    # So 'original_result' now stores 'Hello World!' and then the result is
    # modified.This modified result will be returned when new_func() will be
    # called.

    # STEP 5 - Now decorator(my_function) internally calls new_func() and the
    # value returned by new_func() is returned by decorator(), i.e. the modified
    # one.

    # I know the flow is little bit complicated, but you must understand this
    # complicacy, if you want to simplify alot of your tasks. For example,
    # imagine you have 10 functions in a program and you want to calculate
    # time taken by each function to complete its task.

    # One way to do this is - In every function, start the timer clock at the
    # first line, then stop the timer and print the time before the
    # return statement. Or else, you can write one simple decorator as follows:

    import time

    def time_calc(func):
        def wrapper_func(*arg, **kwargs):
            t = time.time()
            result = func(*arg, **kwargs)
            print("Time taken by {} is {} seconds".format(func.__name__,
time.time()-t))
            return result
        return wrapper_func

    @time_calc
    def my_function(n):
        # do something here

    # All you now need to do is, just put @time_calc before the function
    # definition, whose execution time you want to calculate.

    # Decorators have many use cases. Suppose 10 functions are communicating with
    # each other i.e. the return value of one function, goes as the argument of
    # the other and so on. And you want to print the values of arguments recieved
    # and the values returned for every function in order. Either you will add
    # bunch of print statements in every function and comment them out, or you
    # will make a simple decorator from the next time!

# ------------------------------------------------------------------------
```

```
# 2. MULTIPLE DECORATORS

    # It is possible to have multiple decorators for a function. Eg.

    @decorator1
    @decorator2
    @decorator3
    def my_function():
        # Some steps here

    # In this case, the object of my_function() is passed to decorator3(),
    # the return function of decorator3() is passed to decorator2() and so on.
    # This means, the decorators are evaluated in the order: 3 -> 2 -> 1.

    # So calling my_function() would be equivalent of calling:

    decorator1(decorator2(decorator3(my_function)))

    # Now again you must ask - What is the use of multiple decorators?

    # What if you want to print values of arguments for some functions,
    # print time taken by function to execute for some other functions and
    # print both for some functions. Managing print() statements in each of the
    # functions is quite tedious. You can make two separate decorators and use
    # them.

    # If this use of multiple decorators doesn't convince you, then here is my
    # favourite use of multiple decorators:

    # In python, type-checking is not done. If you define a function,

    def my_func(a, b):
        # some statements

    # 'a' and 'b' can be anything! They can be lists, integers, floats, etc.
    # What if we want 'a' and 'b' to be integers only, or strings only. If this
    # function was defined only for strings, even if you pass lists, there is a
    # possibility that the function will execute and return you some value,
    # without giving you an error. So at this moment, I want to do Type-checking.

    # Here is my 'types' decorator:

    def types(arg_name, *allowed_types):
        def make_wrapper(f):
            if hasattr(f, "wrapped_args"):
                wrapped_args = getattr(f, "wrapped_args")
            else:
                code = f.__code__
                wrapped_args = list(code.co_varnames[:code.co_argcount])
```

```
            try:
                arg_index = wrapped_args.index(arg_name)
            except ValueError:
                raise NameError(arg_name)

            def wrapper(*args, **kwargs):
                if len(args) > arg_index:
                    arg = args[arg_index]
                    if not isinstance(arg, allowed_types):
                        type_list = " or ".join(str(allowed_type) for allowed_type
                                    in allowed_types)
                        raise TypeError("Expected '%s' to be %s; was %s."
                                        % (arg_name, type_list, type(arg)))
                else:
                    if arg_name in kwargs:
                        arg = kwargs[arg_name]
                        if not isinstance(arg, allowed_types):
                            type_list = " or ".join(str(allowed_type) for
                            allowed_type in allowed_types)
                            raise TypeError("Expected '%s' to be %s; was %s."
                                            % (arg_name, type_list, type(arg)))
                return f(*args, **kwargs)

            wrapper.wrapped_args = wrapped_args
            return wrapper

        return make_wrapper


    @types("x", int, float)
    @types("y", float)
    def foo(x, y):
        return x+y

    foo(1, 2.5)         # Output: 3.5
    foo(2.0, 2.5)       # Output: 4.5
    foo('asdf', 2.5)    # Raises TypeError exception
    foo(1, 2)           # Raises TypeError exception

    # Although it is not expected to check types in Python and we should try
    # to avoid such upfront type checks as much as possible. The sole purpose
    # of above code was to show you that - Type Checking is possible in Python
    # as well, using decorators!

# ------------------------------------------------------------------------
```

```
####################### TOPIC 5 - OBJECT INTERNALS ############################

# 1. ATTRIBUTES and __dict__

    # Let's start with a basic class to represent a basic 2D vector.

    class Vector:
        def __init__(self, x, y):
            self.x = x
            self.y = y

        def __repr__(self):
            return "{}({}, {})".format(self.__class__.__name__, self.x, self.y)

    # We now test few things on the REPL.

    v = Vector(5, 3)
    v                       # Output: Vector(5, 3)
    dir(v)
    # ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    #  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
    #  '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
    #  '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    #  '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
    #  'x', 'y']

    # This is the list of all the attributes which could be called over v.
    # For our further discussion, we are interested in one attribute - '__dict__'.

    v.__dict__              # Output: {'x': 5, 'y': 3}
    v.__dict__['x']         # Output: 5
    v.__dict__['x'] = 17
    v.x                     # Output: 17
    del v.__dict__['x']
    v.x
    # AttributeError: 'Vector' object has no attribute 'x'
    'x' in v.__dict__       # Output: False
    'y' in v.__dict__       # Output: True
    v.__dict__['z'] = 3
    v.z                     # Output: 3

    # Even though such manipulations with attributes of an object are possible
    # using dictionary, you should use methods:

    getattr(v, 'y')         # Output: 3
    hasattr(v, 'x')         # Output: False
    delattr(v, 'z')
    setattr(v, 'x', 9)
    v.x                     # Output: 9
```

```
# So there are two issues in above code:
# a) The Notation: Here we use 'x' and 'y' as labels of the coordinates in
#    2D plane. What if the user wants to have 'u' and 'v' as the coordinate
#    labels. This means, we want to give user the freedom to make attributes
#    of the instance as per their desire.
# b) Scope: If you see the output of dir(v), two new attributes have come in
#    the list: 'x' and 'y'. Thus, we can say - the attributes are public
#    and could be changed. We don't want these attributes to be visible to
#    the user and object should be immutable.

# Let's solve the first issue:

class Vector:
    def __init__(self, **coords):
        self.__dict__.update(coords)

    def __repr__(self):
        return "{}({})".format(
            self.__class__.__name__,
            ', '.join("{k}={v}".format(
            k = k,
            v = self.__dict__[k])
            for k in sorted(self.__dict__.keys())))

# On REPL,

v = Vector(p = 3, q = 7)
v                           # Output: Vector(p=3, q=7)
dir(v)
# ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
#  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
#  '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',

#  '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
#  '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'p', 'q']

# Let's make a simple change, so that user should not be allowed to access
# attribute names directly.

class Vector:
    def __init__(self, **coords):
        private_coords = {'_' + k:v for k, v in coords.items()}
        self.__dict__.update(private_coords)

    def __repr__(self):
        return "{}({})".format(
            self.__class__.__name__,
            ', '.join("{k}={v}".format(
            k = k[1:],
            v = self.__dict__[k])
            for k in sorted(self.__dict__.keys())))
```

```
    v = Vector(p = 9, q = 7)
    v                           # Output: Vector(p=9, q=7)
    dir(v)
    # ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
    #  '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
    #  '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',

    #  '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
    #  '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_p', '_q']

    v.p
    # AttributeError: 'Vector' object has no attribute 'p'

    # What we want to do here is - We should allow the read access to the
    # attributes, and the error should be raised only when attempt to modify it
    # is made. The read access could be made by overriding __getattr__, and
    # avoid the write access by overriding __setattr__.

    class Vector:
        def __init__(self, **coords):
            private_coords = {'_' + k:v for k, v in coords.items()}
            self.__dict__.update(private_coords)

        def __getattr__(self, name):
            private_name = '_' + name
            try:
                return self.__dict__[private_name]
            except KeyError:
                raise AttributeError('{!r} object has no attribute {!r}'.format(
                self.__class__, name))

        def __setattr__(self, name, value):
            raise AttributeError("Can't set attribute {!r}".format(name))

        def __delattr__(self, name):
            raise AttributeError("Can't delete attribute {!r}".format(name))

        def __repr__(self):
            return "{}({})".format(
                self.__class__.__name__,
                ', '.join("{k}={v}".format(
                k = k[1:],
                v = self.__dict__[k])
                for k in sorted(self.__dict__.keys())))

# --------------------------------------------------------------------------
```

```
# 2. INSTANCE CREATION

    # __new__ is one of the most easily abused features in Python.
    # It's obscure, riddled with pitfalls, and almost every use case I've
    # found for it has been better served by another of Python's many tools.
    # However, when you do need __new__, it's incredibly powerful and invaluable
    # to understand.

    # The predominant use case for __new__ is in metaclasses. Metaclasses are
    # complex enough to merit their own article, so I don't touch on them here.
    # If you already understand metaclasses, great. If not, don't worry;
    # understanding how Python creates objects is valuable regardless.

    # - Constructors

    # With the proliferation of class-based languages, constructors are likely
    # the most popular method for instantiating objects.

    # a) JAVA
    class StandardClass {
        private int x;
        public StandardClass() {
            this.x = 5;
        }

        public int getX() {
            return this.x;
        }
    }

    # b) PYTHON
    class StandardClass(object):
        def __init__(self, x):
            self.x = x

    # c) JAVASCRIPT
    function StandardClass(x) {
        this.x = x;
    }
    var standard = new StandardClass(5);

    # In Python, as well as many other languages, there are two steps to object
    # instantiation:

    # STEP 1 - The New Step
    # Before you can access an object, it must first be created. This is not
    # the constructor. In the above examples, we use this or self to reference
    # an object in the constructor; the object had already been created by then.
    # The New Step creates the object before it is passed to the constructor.
    # This generally involves allocating space in memory and/or whatever
    # language specific actions newing-up an object requires.
```

```
# STEP 2 - The Constructor Step
# Here, the newed-up object is passed to the constructor. In Python, this is
# when __init__ is called.

# This is the normal way to instantiate a StandardClass object:
standard = StandardClass(5)

# StandardClass(5) is the normal instance creation syntax for Python. It
# performs the New Step followed by the Constructor Step for us. Python also
# allows us to deconstruct this process:

# The New Step
newed_up_standard = object.__new__(StandardClass)
type(newed_up_standard) is StandardClass
hasattr(newed_up_standard,'x') is False

# The Constructor Step
StandardClass.__init__(newed_up_standard, 5)
newed_up_standard.x == 5

# object.__new__ is the default New Step for object instantiation. It's what
# creates an instance from a class. This happens implicitly as the first
# part of StandardClass(5).

# Notice, x is not set until after newed_up_standard is run through __init__.
# This is because object.__new__ doesn't call __init__. They are disparate
# functions. If we wanted to perform checks on newed_up_standard or
# manipulate it before the constructor is run, we could. However, explicitly
# calling the New Step followed by Constructor Step is neither clean nor
# scalable. Fortunately, there is an easy way.

# Python allows us to override the New Step of any object via the __new__
# magic method.

class NewedBaseCheck(object):
    def __new__(cls):
        obj = super(NewedBaseCheck,cls).__new__(cls)
        obj._from_base_class = type(obj) == NewedBaseCheck
        return obj
    def __init__(self):
        self.x = 5

newed = NewedBaseCheck()
newed.x == 5
newed._from_base_class is True

# __new__ takes a class instead of an instance as the first argument. Since
# it creates an instance, that makes sense.
# super(NewedClass, cls).__new__(cls) is very important. We don't want to
# call object.__new__ directly; again, you'll see why later.
```

```
# Why is from_base_class defined in __new__ instead of __init__?
# It's metadata about object creation, which makes more semantic sense in
# __new__. However, if you really wanted to, you could place define
# _from_base_class:

class StandardBaseCheck(object):
    def __init__(self):
        self.x = 5
        self._from_base_class == type(self) == StandardBaseCheck

standard_base_check = StandardBaseCheck()
standard_base_check.x == 5
standard_base_check._from_base_class is True

# There is a major behavioral difference between NewBaseCheck and
# StandardBaseCheck in how they handle inheritance:

class SubNewedBaseCheck(NewedBaseCheck):
    def __init__(self):
        self.x = 9

subnewed = SubNewedBaseCheck()
subnewed.x == 9
subnewed._from_base_class is False

class SubStandardBaseCheck(StandardBaseCheck):
    def __init__(self):
        self.x = 9

substandard_base_check = SubStandardBaseCheck()
substandard_base_check.x == 9
hasattr(substandard_base_check,"_from_base_class") is False

# Because we failed to call super(...).__init__ in the constructors,
# _from_base_class is never set.

# Up until now, classes defining both __init__ and __new__ had no-argument
# constructors. Adding arguments has a few pitfalls to watch out for.
# We'll modify NewBaseCheck:

class NewedBaseCheck(object):
    def __new__(cls):
        obj = super(NewedBaseCheck,cls).__new__(cls)
        obj._from_base_class = type(obj) == NewedBaseCheck
        return obj

    def __init__(self, x):
        self.x = x
```

```
try:
    NewedBaseCheck(5)
except TypeError:
    print True

# Instantiating a new NewedBaseCheck throws a TypeError. NewedBaseCheck(5)
# first calls NewBaseCheck.__new__(NewBaseCheck, 5). Since __new__ takes
# only one argument, Python complains. Let's fix this:

class NewedBaseCheck(object):
    def __new__(cls, x):
        obj = super(NewedBaseCheck,cls).__new__(cls)
        obj._from_base_class = type(obj) == NewedBaseCheck
        return obj

    def __init__(self, x):
        self.x = x

newed = NewedBaseCheck(5)
newed.x == 5

# There are still problems with subclassing:

class SubNewedBaseCheck(NewedBaseCheck):
    def __init__(self, x, y):
        self.x = x
        self.y = y

try:
    SubNewedBaseCheck(5,6)
except TypeError:
    print True

# We get the same TypeError as above; __new__ takes cls and x, and we're
# trying to pass in cls, x, and y. The generic fix is fairly simple:

class NewedBaseCheck(object):
    def __new__(cls, *args, **kwargs):
        obj = super(NewedBaseCheck,cls).__new__(cls)
        obj._from_base_class = type(obj) == NewedBaseCheck
        return obj

    def __init__(self, x):
        self.x = x

newed = NewedBaseCheck(5)
newed.x == 5

subnewed = SubNewedBaseCheck(5,6)
subnewed.x == 5
subnewed.y == 6
```

```
# Unless you have a good reason otherwise, always define __new__ with
# *args and **kwargs.


# __new__ is incredibly powerful (and dangerous) because you manually
# return an object. There are no limitations to the type of object you
# return.

class GimmeFive(object):
    def __new__(cls, *args, **kwargs)):
        return 5

GimmeFive() == 5

# If __new__ doesn't return an instance of the class it's bound to
# (e.g. GimmeFive), it skips the Constructor Step entirely:

class GimmeFive(object):
    def __new__(cls, *args, **kwargs):
        return 5

    def __init__(self,x):
        self.x = x

five = GimmeFive()
five == 5
isinstance(five, int) is True
hasattr(five, "x") is False

# That makes sense: __init__ will throw an error if passed anything but an
# instance of GimmeFive, or a subclass, for self. Knowing all this, we can
# easily define Python's object creation process:

def instantiate(cls, *args, **kwargs):
    obj = cls.__new__(cls, *args, **kwargs)
    if isinstance(obj,cls):
        cls.__init__(obj, *args, **kwargs)
    return obj

instantiate(GimmeFive) == 5
newed = instantiate(NewedBaseCheck, 5)
type(newed) == NewedBaseCheck
newed.x == 5

# Note: DON'T DO THIS EVER:

# While experimenting for this post I created a monster that, like
# Dr. Frankenstein, I will share with the world. It is a great example of
# how horrifically __new__ can be abused. (Seriously, don't ever do this.)
```

```
    class A(object):
        def __new__(cls):
            return super(A,cls).__new__(B)
        def __init__(self):
            self.name = "A"

    class B(object):
        def __new__(cls):
            return super(B,cls).__new__(A)
        def __init__(self):
            self.name = "B"

    a = A()
    b = B()
    type(a) == B
    type(b) == A
    hasattr(a,"name") == False
    hasattr(b,"name") == False

    # The point of the above code snippet: please use __new__ responsibly;
    # everyone you code with will thank you.

    # __new__ and the new step, in the right hands and for the right task, are
    # powerful tools. Conceptually, they neatly tie together object creation.
    # Practically, they are a blessing when you need them. They also have a
    # dark side. Use them wisely.

# --------------------------------------------------------------------------
```

```
############################ TOPIC 6 - PROPERTIES ##############################

# 1. PROPERTIES

    # Let us assume that you decide to make a class that could store the
    # temperature in degree Celsius. It would also implement a method to convert
    # the temperature into degree Fahrenheit. One way of doing this is:

    class Celsius:
        def __init__(self, temperature = 0):
            self.temperature = temperature

        def to_fahrenheit(self):
            return (self.temperature * 1.8) + 32

    # We could make objects out of this class and manipulate the attribute
    # temperature as we wished. Try these on REPL.

    man = Celsius()
    man.temperature = 37
    man.temperature             # Output: 37
    man.to_fahrenheit()         # Output: 98.60000000000001

    # Whenever we assign or retrieve any object attribute like temperature,
    # Python searches it in the object's __dict__ dictionary.
    man.__dict__                # Output: {'temperature': 37}

    # Therefore, man.temperature internally becomes man.__dict__['temperature'].

    # Now, let's further assume that our class got popular among clients and
    # they started using it in their programs. They did all kinds of assignments
    # to the object. One fateful day, a trusted client came to us and suggested
    # that temperatures cannot go below -273 degree Celsius (students of
    # thermodynamics might argue that it's actually -273.15), also called the
    # absolute zero. He further asked us to implement this value constraint.

    # An obvious solution to the above constraint will be to hide the attribute
    # temperature (make it private) and define new getter and setter interfaces
    # to manipulate it. This can be done as follows.

    class Celsius:
        def __init__(self, temperature = 0):
            self.set_temperature(temperature)

        def to_fahrenheit(self):
            return (self.get_temperature() * 1.8) + 32

        # new update
        def get_temperature(self):
            return self._temperature
```

```
    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value

# We can see above that new methods get_temperature() and set_temperature()
# were defined and furthermore, temperature was replaced with _temperature.
# An underscore (_) at the beginning is used to denote private variables in
# Python.

c = Celsius(-277)
# ValueError: Temperature below -273 is not possible

# This update successfully implemented the new restriction. We are no longer
# allowed to set temperature below -273.

# Now please note that private variables don't exist in Python. There are
# simply norms to be followed. Language itself doesn't apply any restrictions.

c._temperature = -300
c.get_temperature()             # Output: -300

# But this is not of great concern. The big problem with the above update is
# that, all the clients who implemented our previous class in their program
# have to modify their code from obj.temperature to obj.get_temperature()
# and all assignments like obj.temperature = val to obj.set_temperature(val).
# This refactoring can cause headaches to the clients with hundreds of
# thousands of lines of codes. All in all, our new update was not backward
# compatible. This is where property comes to rescue.

# The pythonic way to deal with the above problem is to use property.
# Here is how we could have achieved it.

class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature,set_temperature)
```

```
# We added a print() function inside get_temperature() and set_temperature()
# to clearly observe that they are being executed.

# The last line of the code, makes a property object temperature. Simply put,
# property attaches some code (get_temperature and set_temperature) to the
# member attribute accesses (temperature).

# Any code that retrieves the value of temperature will automatically call
# get_temperature() instead of a dictionary (__dict__) look-up. Similarly,
# any code that assigns a value to temperature will automatically call
# set_temperature(). This is one cool feature in Python.

# We can see above that set_temperature() was called even when we created
# an object. Can you guess why?

# The reason is that when an object is created, __init__() method gets
# called. This method has the line self.temperature = temperature.
# This assignment automatically called set_temperature().

c = Celsius()
c.temperature
# Output: Getting value
#         0

# Similarly, any access like c.temperature automatically calls
# get_temperature(). This is what property does. Here are a few more
# examples.

c.temperature = 37          # Output: Setting value
c.to_fahrenheit()
# Output: Getting value
#         98.60000000000001

# By using property, we can see that, we modified our class and implemented
# the value constraint without any change required to the client code.
# Thus our implementation was backward compatible and everybody is happy.

# Finally note that, the actual temperature value is stored in the private
# variable _temperature. The attribute temperature is a property object
# which provides interface to this private variable.

# In Python, property() is a built-in function that creates and returns a
# property object. The signature of this function is

property(fget=None, fset=None, fdel=None, doc=None)

# where, fget is function to get value of the attribute, fset is function
# to set value of the attribute, fdel is function to delete the attribute
# and doc is a string (like a comment). As seen from the implementation,
# these function arguments are optional.
```

```
    # So, a property object can simply be created as follows.

    property()      # Output: <property object at 0x0000000003239B38>

    # A property object has three methods, getter(), setter(), and deleter()
    # to specify fget, fset and fdel at a later point. This means, the line

    temperature = property(get_temperature,set_temperature)

    # could have been broken down as

    # make empty property
    temperature = property()
    # assign fget
    temperature = temperature.getter(get_temperature)
    # assign fset
    temperature = temperature.setter(set_temperature)

    # These two pieces of codes are equivalent.

    # Programmers familiar with decorators in Python can recognize that the
    # above construct can be implemented as decorators. We can further go on
    # and not define names get_temperature and set_temperature as they are
    # unnecessary and pollute the class namespace. For this, we reuse the
    # name temperature while defining our getter and setter functions.
    # This is how it can be done.

    class Celsius:
        def __init__(self, temperature = 0):
            self._temperature = temperature

        def to_fahrenheit(self):
            return (self.temperature * 1.8) + 32

        @property
        def temperature(self):
            print("Getting value")
            return self._temperature

        @temperature.setter
        def temperature(self, value):
            if value < -273:
                raise ValueError("Temperature below -273 is not possible")
            print("Setting value")
            self._temperature = value

# ----------------------------------------------------------------------------
```

```
####################### TOPIC 7 - CLASS DECORATORS ##########################

# There are two different ways to use decorators on classes. The first is
# by decorating the methods of a class or decorating the whole class.

# 1. BUILT-IN CLASS DECORATORS

    # Some commonly used decorators that are built into Python are @classmethod,
    # @staticmethod, and @property. The @classmethod and @staticmethod
    # decorators are used to define methods inside a class namespace that's not
    # connected to a particular instance of that class. The @property decorator
    # is used to customize getters and setters for class attributes.

    class Circle:
        def __init__(self, radius):
            self._radius = radius

        @property
        def radius(self):
            """Get value of radius"""
            return self._radius

        @radius.setter
        def radius(self, value):
            """Set radius, raise error if negative"""
            if value >= 0:
                self._radius = value
            else:
                raise ValueError("Radius must be positive")

        @property
        def area(self):
            """Calculate area inside circle"""
            return self.pi() * self.radius**2

        def cylinder_volume(self, height):
            """Calculate volume of cylinder with circle as base"""
            return self.area * height

        @classmethod
        def unit_circle(cls):
            """Factory method creating a circle with radius 1"""
            return cls(1)

        @staticmethod
        def pi():
            """Value of π, could use math.pi instead though"""
            return 3.1415926535
```

```
    # In this class:

    # - .cylinder_volume() is a regular method.

    # - .radius is a mutable property: it can be set to a different value.
    #    However, by defining a setter method, we can do some error testing to
    #    make sure it's not set to a nonsensical negative number. Properties
    #    are accessed as attributes without parentheses.

    # - .area is an immutable property: properties without .setter() methods
    #    can't be changed. Even though it is defined as a method, it can be
    #    retrieved as an attribute without parentheses.

    # - .unit_circle() is a class method. It's not bound to a particular
    #    instance of Circle. Class methods are often used as factory methods
    #    that can create specific instances of the class.

    # - .pi() is a static method. It's not really dependent on the Circle class,
    #    except that it's part of its namespace. Static methods can be called on
    #    either an instance or the class.

    c = Circle(5)
    c.radius                        # Output: 5
    c.area                          # Output: 78.5398163375
    c.area = 100
    # AttributeError: can't set attribute
    c.cylinder_volume(height=4)     # Output: 50.265482456
    c.radius = -1
    # ValueError: Radius must be positive
    c = Circle.unit_circle()
    c.pi()                          # Output: 3.1415926535
    Circle.pi()                     # Output: 3.1415926535

# ----------------------------------------------------------------------------

# 2. DECORATING A CLASS METHOD

    # Recall our decorator function which can be used to calculate the time
    # taken by a function to execute. Here is another function, which does the
    # similar task.

    def timer(func):
        def wrapper(*args, **kwargs):
            start_time = time.perf_counter()
            value = func(*args, **kwargs)
            end_time = time.perf_counter()
            run_time = end_time - start_time
            print("Finished {} in {} secs".format(repr(func.__name__),
                   round(run_time, 3)))
            return value
        return wrapper
```

```
    class Calculator:
        def __init__(self, num):
            self.num = num

        @timer
        def doubled_and_add(self):
            res = sum([i * 2 for i in range(self.num)])
            print("Result : {}".format(res))

    # Let's test this on REPL.
    c = Calculator(10000)
    c.doubled_and_add()
    # Output: Result : 99990000
    #         Finished 'doubled_and_add' in 0.001 secs

# ------------------------------------------------------------------------

# 3. DECORATE ENTIRE CLASS

    # Decorating a class does not decorate its methods. Here, @timer only
    # measures the time it takes to instantiate the class.

    @timer
    class Calculator:
        def __init__(self, num):
            self.num = num
            import time
            time.sleep(2)

        def doubled_and_add(self):
            res = sum([i * 2 for i in range(self.num)])
            print("Result : {}".format(res))

    c = Calculator(10000)
    # Output: Finished 'Calculator' in 2.001 secs

# ------------------------------------------------------------------------

# 4. CLASSES AS DECORATORS

    # The best way to maintain state is by using classes. If we want to use
    # class as a decorator it needs to take func as an argument in its
    # .__init__() method. Furthermore, the class needs to be callable so that
    # it can stand in for the decorated function. For a class to be callable,
    # you implement the special .__call__() method.

    class CountCalls:
        def __init__(self, func):
            functools.update_wrapper(self, func)
            self.func = func
            self.num_calls = 0
```

```
        def __call__(self, *args, **kwargs):
            self.num_calls += 1
            print(f"Call {self.num_calls} of {self.func.__name__!r}")
            return self.func(*args, **kwargs)


    @CountCalls
    def say():
        print("Hello!")

    say()
    say()
    say()
    say()
    print(say.num_calls)


    # Output:
    #   Call 1 of 'say'
    #   Hello!
    #   Call 2 of 'say'
    #   Hello!
    #   Call 3 of 'say'
    #   Hello!
    #   Call 4 of 'say'
    #   Hello!
    #   4

# ------------------------------------------------------------------------
```