This programming assignment is designed to show how Python functions can define other Python code (in this case, a class) in an unexpected way: the function can build a huge string that represents the definition of a Python class and then call **exec** on it, which causes Python to define that class just as if it were written in a file and imported (in which case Python reads the file as a big string and does the same thing). Your code will heavily rely on string formatting operations: I suggest using the **str.format** method to do the replacements: now is a good time to learn about this function if you don't already know it; but you are free to use whatever string processing tool(s) you want.

I suggest that you first read the description below and define/test/debug as much of the **Point** class as you can, writing it directly in Eclipse (especially the **__getitem__**, **__eq__**, **_asdict**, **_make**, and **_replace** methods). You might want to write a small batch file to help you test this class.

Once you have written/debugged the code for the **Point** class, define the general **pnamedtuple** function, which when given the appropriate arguments (for the **Point** class: **pnamedtuple('Point', 'x y')**) constructs a huge string that contains exactly the same code as the **Point** class that you wrote. Much of the code from your **Point** class will be turned into strings and made generic: generalized for calls to **pnamedtuple** with different arguments. Use the **.format** method to replace the generic parts with the actual strings needed for the class being defined.

Download the program3 project folder and use it to create an Eclipse project. Read and run the **miniexample.py** module, which performs a similar but simpler task: it illustrates how to write the **keep** function as a small string. All the elements needed to write the **pnamedtuple** function appear in here in a simplfied form (see especially the call to the **join** function). Put your **pnamedtuple** in the **pcollections.py** module, which can be tested in the standard driver or by writing code in the script at the bottom of the **pcollections.py** module.

pnamedtuple

Problem Summary:

Write a function named **pnamedtuple** that is passed information about a named tuple: it returns a reference to a class object from which we can construct instances of the specified named tuple. We might use this class as follows:

```
from pcollections import pnamedtuple
Point = pnamedtuple('Point', 'x y')
p = Point(0,0)
...perform operations on p using methods defined in the Point class
```

Please note that although many of the examples in this description use the **Point** class, your **pnamedtuple** function must work for all legal calls to **pnamedtuple**. For example the batch-self-check file uses descriptions (some legal, some not) of the **Triple** class.

I created six templates (one big, two medium, three small), which are each strings that have parts to fill in using the **format** method; all but the small strings are triple-quoted, multi-line strings, that look like large chunks of Python code (see **miniexample.py** in the download to help understand this paragraph, because it has similar templates).

Note calling the following **format** method on the string

```
'{name} from {country} tells you
{rest}'.format(name='Rich',country='USA',rest='blah..blah..blah'
)
```

returns the string result

```
'Rich from USA tells you blah..blah..blah'
```

If we have already bound the names **name='Rich'** and **country='USA'** and **rest='blah..blah..blah')** then we could write

```
f'{name} from {country} tells you {rest}'
```

which computes the same string result.

In many cases, the arguments I passed to the **format** calls were computed by list comprehensions turned into strings by calling the **.join** method (the opposite of the **.split** method). See the **miniexample** for an example of everything working together to define a function by filling in a template with **.format**. Finally, my solution is about 150 lines (including blank lines and comments, and a solution to the extra credit part), and that is divided between Python code (50% of the lines) and string templates that specify Python code (50% of the lines).

Details

- Define a function named **pnamedtuple** in a module named **pcollections.py** (that is the only name defined in the module, but this function can define local functions: I wrote **show_listing** and 5 other short ones). Its header is

  def pnamedtuple(type_name, field_names, mutable=False, defaults={}):

  an example call to this function is

  Point = pnamedtuple('Point', ['x','y'], mutable=False)

  which is equivalent to writing **Point = pnamedtuple('Point', 'x y')** or **Point = pnamedtuple('Point', 'x,y')**. Once we have defined **Point** in this way, we can then write code like **origin = Point(0,0)**.

Generally, a **pnamedtuple** can have an arbitrary number of field names; **the order of these field names is important and should be retained in the later code** (for example see the header of **__init__** below). So **Point = pnamedtuple('Point', 'x,y')** has a different meaning than **Point = pnamedtuple('Point', 'y,x')** even though both have the same field names: their order is different, and some methods depend on this order (again, see the header of **__init__** below, which would be **def __init__(self, y, x)** in the second case).

A **legal name** for the type and fields must start with a letter which can be followed by 0 or more letters, digits, or underscore characters (hint: I used a simple regular expression to verify legal names); also it must not be a Python keyword. Hint: the name **kwlist** is importable from the **keyword** module: it is bound to a list of all Python keywords.

The parameters must have the following structure.

- **type_name** must be a legal name (see above).
- **field_names** must be a list of legal names (see above), or a string in which spaces or commas (or some mixture of the two) separate legal names. So, we can specify **field_names** like **['x','y']** or **'x y'**, or **'x, y'**. If a name is duplicated, just ignore all but its first appearance (hint: I used the **unique** generator to filter out duplicates, which is written in the course notes).

  If any of the names are not legal, raise a **SyntaxError** with an appropriate message.

- **defaults** can specify a dictionary of **field_names** and their default values: the connection between these two features will be described further below in the definition of **__init__**. Meanwhile, if any keys in the **defaults** dictionary do not appear as **field_names**, raise a **SyntaxError** with an appropriate message.

The resulting class that is written should have the following functionality. Note that the main job of **pnamedtuple** is to compute a large string that describes the class and then return the class object it represents (by using Python's **exec** function; code I have supplied). We could define the class equivalently by writing the string into a **.py** file and then importing that file.

- Define the class name to be **type_name**. After than, define two class attributes (information stored in the class itself, not in the objects constructed from the class): **_fields** and **_mutable**, which are bound to a **list** of all the field names and the **bool** parameter respectively. See the **_replace** and extra credit **__setattr** methods for how they use **_mutable**. For **Point** described above, the class would start as

```
    class Point:
        _fields = ['x','y']
        _mutable = False
```

- Define an **__init__** method that has all the field names as parameters (**in the order they appear in the second argument to pnamedtuple**) and initializes every instance name (using these same names) with the value bound to its parameter. For **Point** described above, the **__init__** method would be

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

The interesting problem here, and throughout many other parts of this is assignment, is writing a functon like **gen_init** such that **gen_init(['x','y'])** returns the a string representing the **__init__** function above (including correct indentation and a **\n** at the end of each line).

```
  "    def __init__(self, x, y):\n        self.x = x\n        self.y = y\n"
```

Finally, for every **field_name** that is specified as a key in the **defaults** parameter, include it in **__init__**'s parameter list with its associated default value. For example, in **Point = pnamedtuple('Point', 'x y', defaults={'y':0})** the header for the defining **__init__** should be **def __init__(self, x, y=0):**

- Define a **__repr__** method that returns a string, which when passed to **eval** returns a newly constructed object that has all the same instance names and values(==) as the object **__repr__** was called on. For **Point**, if we defined **origin = Point(0,0)** then calling **repr(origin)** would return **'Point(x=0,y=0)'**. We can write thie **__repr__** for **Point** using the **format** method of **f-strings**. It would appear as

```
    def __repr__(self):
        return
'Point(x={x},y={y})'.format(x=self.x,y=self.y)
```

or

```
    def __repr__(self):
        return f'Point(x={self.x},y={self.y})'
```

- Define simple query/accessor methods for each of the field names. Each method name should start as **get_** followed by the name of a field. For **Point**, there would be two query/accessor methods.

- ```
      def get_x(self):
  ```
- ```
          return self.x
  ```
-
- ```
      def get_y(self):
          return self.y
  ```

  Note that with these methods, if we had a list of **Point** named **lp**, we could call **lp.sort(key= Point.get_x)** to sort the list by their **x** coordinates. Python's builtin **namedtuple** does not have this general ability, trading it for code that retrieves these values a bit more quickly.

- Define the **__getitem__** method to overload the **[]** (indexing operator) for this class: an index of **0** returns the value of the first field name in the **field_names** list; an index of **1** returns the value of the second field name in the **field_names** list, etc. Also, the index can be a string with the named field. So, for **p = Point(1,2)** writing **p.get_x()**, or **p[0])**, or **p['x']** returns a result of **1**. Raise an **IndexError** with an appropriate message if the index is out of bounds **int** or a string that does not name a field.

  Note that this method can be used by Python to iterate through any class produced by **pnamedtuple** one index after another. It is also useful for writing the **__eq__** method: see below.

  Hint: for an **int index** parameter, combine the **self._fields**, instance name, the **get_** methods, and the **eval** function to write a short solution to this problem; in the case of **origin = Point(0,0)**, calling **origin[1]** should construct the string **'self.get_y()'** and return **eval('self.get_y()')**.

  Another way to solve this problem is to use a cascded **if** with an an arbitrary number or **elif** clauses (in this case just 2):

  ```
  if index == 0:
      returns self.get_x()
  elif index == 1:
      returns self.get_y()
  ```

  For named tuples with N fields, you'd need N-1 **elif** clauses.

- Overload the **==** operator so that it returns **True** when the two named tuples come from the same class and have all their name fields bound to equal values. Hint: use **__getitem__** for each name to check for equality.
- Define the **_asdict** method, which takes no arguments; it returns the **pnamedtuple** as a **dict** of names associated with their values. In the case of **p1= Point(0,1)**, calling **p1._asdict()** should return **{'x': 0, 'y': 1}**
  ```

- Define the **_make** method, which takes one iterable argument (and no **self** argument: the purpose of **_make** is to make a new object; see how it is called below); it returns a new object whose fields (in the order they were specified) are bound to the values in the interable (in that same order). For example, if we called **Point._make((0,1))** the result returned is a new **Point** object whose **x** attribute is bound to **0** and whose **y** attribute is bound to **1**.
- Define a **_replace** method, which takes ****kargs** as a parameter (keyword args). This allows the name **kargs** to be used in the method as a **dict** of parameter names and their matching argument values. The semantics of the **_replace** method depends on the value stored in the instance name **self._mutable**:
    - If **True**, the instance namess of the object it is called on are changed and the method returns **None**. So, if **origin = Point(0,0)** and we call **origin._replace(y=5)**, then **print(origin)** would display as **Point(x=0,y=5)** because **origin** is mutated.
    - If **False**, it returns a new object of the same class, whose instance name's values are the same, except for those specified in **kargs**. So, if **origin = Point(0,0)** and we call **new_origin = origin._replace(y=5)**, then **print(origin,new_origin)** would display as **Point(x=0,y=0) Point(x=0,y=5)** because **origin** is not mutated.

  If any of the ****kargs** names are not **field_names** raise a **TypeError** Exception.

  Define this method to look like

  ```
  def _replace(self,**kargs):
      check for all legal field names in **kargs
      if self._mutable:
          ...
      else:
          ...
  ```

  In both **...** we iterate (through **kargs.items()** or **self._fields**) and refer to **self.__dict__** to retrieve the current values bound to the instance names: this is a bit tricky. Use the notes or web resources to learn more about ****kargs** in general; feel free to post specific question on the forum not relating to their actual use in **_replace** and also, not, "Could someone please explain **kargs to me").

  The **kargexample.py** module has a little ****kargs** demo in it.

- **Extra credit**: Define the **__setattr__** method so after **__init__** finishes, if the **mutable** parameter is **False**, the named tuple will not allow any instance names to be changed: it will raise an **AttributeError** with an appropriate message.

Of course, our **pnamedtuple** function should work for **Point** as illustrated above, but should also work for any other legal call to create a named tuple. The actual **namedtuple** class in Python is specified and implemented differently than the requirements of this assignment. You may not use Python's actual **namedtuple** in this assignment.

Testing

The **pcollections.py** module includes a script that calls **driver.driver()**. The project folder contains a **bsc.txt** file (examine it) to use for batch-self-checking your function. These are rigorous but not exhaustive tests.

Note that when exceptions are raised, they are printed by the driver but the **Command:** prompt sometimes appears misplaced.

You can write other code at the bottom of your **pcollections.py** module to test the **pnamedtuple** function, or type code into the driver as illustrated below. Notice the default for each command is the command previously entered.

```
  Driver started
  Command[!]: from pcollections import pnamedtuple as pnt
  Command[from pcollections import pnamedtuple as pnt]: Point = 
pnt('Point', 'x y')
  Command[Point = pnt('Point', 'x y')]: origin = Point(0,0)
  Command[origin = Point(0,0)]: p1 = Point(5,2)
  Command[p1 = Point(5,2)]: print(p1)
  Point(x=5,y=2)
  Command[print(p1)]: print(p1.get_x())
  5
  Command[print(p1.get_x())]: print(p1[0])
  5
  Command[print(p1[0])]: print(p1['x'])
  5
  Command[print(p1['x'])]: print(p1['z'])
  Traceback (most recent call last):
    File "C:\Users\USER\workspace\courselib\driver.py", line 
224, in driver
      exec(old,local,globl)
    File "", line 1, in 
    File "", line 17, in __getitem__
  IndexError: Point.__getitem__: index(z) is illegal
  Command[print(p1['z'])]: print(p1._asdict())
  {'x': 5, 'y': 2}
  Command[print(p1._asdict())]: newp = Point._make([0,1])
  Command[newp = Point._make([0,1])]: print(newp)
  Point(x=0,y=1)
  Command[print(newp)]: p2 = p1._replace(x=2,y=5)
  Command[p2 = p1._replace(x=2,y=5)]: print(p1,p2)
```

```
Point(x=5,y=2) Point(x=2,y=5)
Command[print(p1,p2)]: quit
Driver stopped
```

Files for **batch_test** (see **bt.txt** for an example) just contain commands that will be executed; many are calls to the **print** function, which show the result of the print.

Remember that your **pnamedtuple** function can print on the console, for debugging purposes, the string it is about to **exec** so you can look for errors there (just eyeball whether the code correct). The **show_listing** function (defined in the **pnamedtuple** function) display a string on the console, numbering its lines (useful when **exec** finds an error: it reports a line number that **show_listing** shows).

Finally, I have also included two programs that use Python's **namedtuple** (with those names changed to **pnamedtuple**). It would be a good idea to test you **pnamedtuple** in these contexts, and in the script with other numbers/names of fields.